Price $50

ORION
INSTRUMENTS

Reference Guide for the
Unilab 8620

UniLab
analyzer-emulator

ORION Instruments, Inc.
180 Independence Drive
Menlo Park, California
94025

ORION

INSTRUMENTS

Reference Manual for the
UniLab 8620

UniLab
Reference Manual

ORION Instruments, Inc.
180 Independence Drive
Menlo Park, California
94025

CONTENTS

GLOSSARY CONTENTS

Alphabetical list with page numbers.

GUIDE TO DOCUMENTATION

Reference Manual
This *Reference Manual* contains information mainly for advanced users. It describes the UniLab's macro-programming features; the complete UniLab command set, including advanced and rarely used commands; and appendices about custom cables (for those without an Emulation Module), debugging interrupt routines, object and symbol formats, etc.

These are not needed for most UniLab operations. We suggest you rely on the separate *User's Guide* and On-Line Help until necessity requires details found only in this volume, or until experience has prepared you for the "graduate school" it represents.

User's Guide
The *User's Guide* describes the features and operation of the UniLab 8620. It contains everything most users will need to know. Further elaboration can be found easily via the On-Line Help facilities, described therein.

Target Application Notes
Your optional Disassembler/DEBUG (DDB) software package is shipped with *Target Application Notes* containing processor-specific details and cautionary notes. Refer to it before connecting the UniLab to any target system other than an Orion MicroTarget™.

Glossary of Commands

The glossary which comprises the bulk of this volume documents the complete, generalized set of UniLab commands. The explanations are essentially the same as those provided by On-Line Help. You can display the glossary entry for any command by typing:

```
HELP <command>
```

Processor-Specific Commands

Every disassembler/DEBUG (DDB) software package includes commands specific to that package, which are not documented in this manual. To learn more about processor-specific words, consult your *Target Application Notes* and the "Processor specific" option of the Help PopUp (press `Alt-F1` for recent notes about your DDB).

Glossary Format

The first line of each glossary entry is the command. The second line shows the command with its parameters, if any, inside <angle brackets>.

Some commands have a final entry on the second line, e.g.:

- *PPA* Part of the optional Program Performance Analyzer.
- *F#* Assigned to the indicated function key when pressed at the `Command>` prompt. (Such default assignments can be changed by the user.) Commands assigned to `F8` are usually accessed via the mode panels or the Configuration PopUp rather than executed by name.
- *Macro System* Only a valid command when used in the macro mode of UniLab operation (see `MACRO`).
- *Rarely Used*

Definition

The first block of text tells what the command does.

Usage

The next block of text tells how and when to use the command.

Example

Most glossary entries includes annotated examples.

Comments

This includes warnings, historical notes, and other bits of information (not included for every command).

COMMANDS LISTED BY FUNCTION

Each UniLab command is shown below, logically grouped by function, along with any alternate methods of invoking it. A full description of each is provided in the alphabetical glossary.

Analyzer Triggers

1AFTER	
2AFTER	
3AFTER	
=SAMP	
ADR	Trigger Dialog (F6)
ADR?	Analyzer PopUp
AFTER	Trigger Dialog (F6)
ALSO	Trigger Dialog (F6)
ANY	
AS	Analyzer PopUp
CONT	Trigger Dialog (F6)
CONTROL	
CYCLES?	Analyzer PopUp
DATA	Trigger Dialog (F6)
DCYCLES	Trigger Dialog (F6)
DEVENTS	Trigger Dialog (F6)
EVENTS?	
FETCH	Trigger Dialog (F6)
FILTER	Trigger Dialog (F6)
FIND-ADR	
FIND-CONT	
FIND-DATA	
FIND-HADR	
FIND-HDATA	
FIND-LADR	
FIND-MISC	
FIND-WDATA	
FIND-xxxx	
HADR	Trigger Dialog (F6)
HDAT	
HDATA	Trigger Dialog (F6)
INFINITE	
INT	
INT'	
LADR	Trigger Dialog (F6)
MASK	Trigger Dialog (F6)
MISC	Trigger Dialog (F6)
MISC'	
NDATA	
NO	
NORMB	Trigger Dialog (F6)
NORMM	Trigger Dialog (F6)
NORMT	Trigger Dialog (F6)
NOT	Trigger Dialog (F6)
NOW?	Analyzer PopUp
NX	
ONLY	Trigger Dialog (F6)
PCYCLES	
PEVENTS	
Q1	Trigger Dialog (F6)
Q2	Trigger Dialog (F6)
Q3	Trigger Dialog (F6)
QUALIFIERS	Trigger Dialog (F6)
READ	Trigger Dialog (F6)
RES-	Alt-F5
RES-'	
RESET	Analyzer PopUp
RESET'	Analyzer PopUp
S	Trigger Dialog (F6)
S+	Ctrl-F9
SAMP	Analyzer PopUp
SET-TRIG	F6
SR	
SST	
STARTUP	F9
T	
TD	
TN	
TNT	
TO	Trigger Dialog (F6)
TRIG	Trigger Dialog (F6)
TS	
TSTAT	Trigger Dialog (F6)
USEC?	Analyzer PopUp

Configuration	
16BIT	
8BIT	
=EMSEG	Configuration PopUp
=HISTORY	
=MBASE	Mode Panel (F8)
=OVERLAY	Configuration PopUp
=SYMBOLS	
=WAIT	
DEBUG	Mode Panel (F8)
DEBUG'	Mode Panel (F8)
EMCLR	Configuration PopUp
EMENABLE	Configuration PopUp
EMSTAT	Configuration PopUp
ESTAT	Configuration PopUp
HARDWARE	Mode Panel (F8)
INIT	Ctrl-F6
MODE	F8
PATCH	DDB
SAVE-SYS	Configuration PopUp
SET-COLOR	Configuration PopUp
SET-EM	Configuration PopUp
SET-GRAPH-COLOR	
SOFT	Configuration PopUp
SOFTWARE	Mode Panel (F8)

Macros	
:	
;	
<TST>	
ASCII	
BPEX	
BPEX2	
EDIT_MACROS	Shift-F10
EMIT	
MACRO	
MACRO_2	Shift-F2
MACRO_3	Shift-F3
MACRO_4	Shift-F4
MACRO_5	Shift-F5
MACRO_6	Shift-F6
MACRO_7	Shift-F7
MACRO_8	Shift-F8
MACRO_9	Shift-F9
MAKE-OPERATOR	
MS	
OPERATOR	
SC	
SHOW_MACROS	Shift-F1

InSight	
=STROBE	
INSIGHT	Ctrl-F3
RESIGHT	Ctrl-F4

DEBUG	
=BC	
CLRMBP	
DMBP	
G	Debug PopUp
GB	Debug PopUp
GW	Debug PopUp
LP	Debug PopUp
R	
RB	Debug PopUp
RI	Trigger Dialog (F6)
RMBP	
RZ	Debug PopUp
SI	Trigger Dialog (F6)
SMBP	
STEP-INTO	F4
STEP-LINE	Alt-F4
STEP-OVER	F3

DOS and User Interface

`?FREE`	
`ALT-FKEY`	
`ASC`	Special PopUp
`BYE`	Configuration PopUp
`CATALOG`	Special PopUp
`CTRL-FKEY`	
`B#`	
`B.`	
`D#`	
`DOS`	Special PopUp
`FKEY`	
`FKEY?`	Alt-F3
`HELP`	F1
`HELP+`	Ctrl-F1
`HELP_DEBUG`	Alt-F1
`LOADER`	
`MEMO`	Special PopUp
`MESSAGE`	
`PINOUT`	Special PopUp
`POP`	Mode Panel (F8)
`POP'`	Mode Panel (F8)
`POPUPS`	F10
`PRINT`	Mode Panel (F8)
`PRINT'`	Mode Panel (F8)
`T.`	
`WORDS`	Help PopUp

Memory

`ASM`	Memory PopUp
`ASM-FILE`	
`CKSUM`	Memory PopUp
`DM`	Memory PopUp
`DN`	Ctrl-F2
`EMMOVE0`	
`EMMOVE1`	
`M`	
`M!`	Memory PopUp
`M?`	
`MCOMP`	Memory PopUp
`MDUMP`	Memory PopUp
`MFILL`	Memory PopUp
`MM`	
`MM!`	Memory PopUp
`MM?`	
`MMOVE`	Memory PopUp
`MODIFY`	Memory PopUp
`ORG`	
`PAGE0`	
`PAGE1`	
`TRAM`	
`TRAM'`	

PPA

`AHIST`	Orion PopUp
`MHIST`	Orion PopUp
`PPA`	Alt-F10
`THIST`	Orion PopUp

EPROM Programmer & Stimulus Generator

`PROM`	
`RCOMP`	Eprom PopUp
`READ-ROM`	Eprom PopUp
`RES`	
`SET`	
`STIMULUS`	
`WRITE-EPROM`	Eprom PopUp

Symbols

`CLRSYM`	
`IS`	
`ISMODULE`	
`ISOFFSET`	
`ISSEG`	
`SYMB`	Mode Panel (F8)
`SYMB'`	Mode Panel (F8)
`SYMDEL`	
`SYMLIST`	Special PopUp

Files	
BINLOAD	Files PopUp (F5)
BINLOADL	Files PopUp (F5)
BINSAVE	Files PopUp (F5)
BINSAVEL	Files PopUp (F5)
HEXLOAD	Files PopUp (F5)
HLOAD	Files PopUp (F5)
HSAVE	Files PopUp (F5)
LOADER	
LTARG	Files PopUp (F5)
MAPSYM	
MAPSYM+	
SSAVE	Alt-F9
SYMFILE	Files PopUp (F5)
SYMFIX	LOADER
SYMLOAD	Files PopUp (F5)
SYMSAVE	Files PopUp (F5)
TCOMP	Files PopUp (F5)
TEXTFILE	Files PopUp (F5)
TMASK	
TOFILE	Mode Panel (F8)
TOFILE'	Mode Panel (F8)
TSAVE	Files PopUp (F5)
TSHOW	Files PopUp (F5)
TX	
TXH	

Display	
BPDM	Mode Panel (F8)
BPT	Mode Panel (F8)
COLOR	Mode Panel (F8)
DASM	Mode Panel (F8)
DASM'	Mode Panel (F8)
PAGINATE	Mode Panel (F8)
PAGINATE'	Mode Panel (F8)
SHOWC	Mode Panel (F8)
SHOWC'	Mode Panel (F8)
SHOWM	Mode Panel (F8)
SHOWM'	Mode Panel (F8)
SOURCE	
SOURCE'	
SPLIT	F2
TOP/BOT	End
TRACK	
TRACK'	
WSIZE	

16BIT

no parameters *Macro System*

Selects 16-bit mode for memory emulation, trace display, and PROM reading/writing.

Usage

You probably won't use this command. It sets the UniLab to work with processors that have a 16-bit data bus. If you have purchased a disassembler, either this command or `8BIT` is built into the software.

Comments

`16BIT` is one word (i.e., no space after the 16). This command changes both the trace display and the signals put onto the target's bus by the UniLab. If you are not using an Emulation Module, you will need a 16-bit ROM cable (which terminates in two ROM plugs rather than one).

The `HL` and `LH` commands determine the order in which the trace shows the bytes (also preset by your UniLab DDB software).

1AFTER

`1AFTER <trigger_spec>`

Clears the current trigger specification and enables trace filtering. Instructs the analyzer to record bus cycles that match the trigger specification, plus the first cycle that follows each such occurrence.

Usage

The UniLab stores the trigger cycle and the following one, every time it sees bus conditions that match the trigger spec. This is similar to ONLY, but it also stores the cycle immediately after the trigger cycle.

The disassembler will not work properly on code fragments, and should be disabled via F8 or with DASM' while you are using the xAFTER commands. After setting the trigger spec, use S to start the analyzer. The "trigger status display line" shows how many cycles have been recorded. The UniLab automatically displays the trace after the entire trace buffer is full.

Checking the Trace

If you want to see the trace without waiting for the buffer to fill completely, press Esc to stop the analyzer, and type T to display the data that has been collected thus far.

The trace buffer fills from the bottom, and each new cycle pushes up the already recorded data. Thus, if you do interrupt the process of recording bus cycles, what you want to see is probably in the last part of the buffer.

Examples

```
1AFTER 1200 ADR S
```

Records each cycle with 1200 on the address lines, and the following cycle.

```
1AFTER 235 TO 560 ADR S
```

Records two cycles every time a cycle has an address from 235 – 560.

Comments

Don't 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, to show the address of the cycle after the one that caused the memory access.

xAFTER initializes all trigger features, so NORMx is unnecessary before using these commands.

2AFTER

```
2AFTER <trigger_spec>
```

Same as 1AFTER, but two following cycles are recorded with each trigger cycle.

Usage

Sets the analyzer to record the trigger cycle and the two following cycles, every time it sees conditions that match the trigger specification.

Comments

See 1AFTER.

3AFTER

```
3AFTER <trigger_spec>
```

Like `1AFTER`, but three following cycles are recorded with each trigger cycle.

Comments

See `1AFTER`. This type of filtered trace will sometimes contain enough information to provide a useful disassembly.

8BIT

no parameters *Macro System*

Selects eight-bit mode for the trace display and memory emulation, and for PROM burning and reading.

Usage

You probably won't use this command. It sets the UniLab to work with processors that make use of eight-bit data. If you have purchased a disassembler, either this command or `16BIT` is built into the software.

Comments

`8BIT` is one word, with no space between the numeral 8 and the rest of the command. If you are not using an Emulation Module, you will need a 8-bit ROM cable (which terminates in one ROM plug rather than two).

: (colon)

no parameters

Macro System

The colon character begins a macro definition. The word following the colon is the name of the macro. Primarily for advanced users and for programmers who write automated test routines.

Usage

An easier way to define simple macros is to use EDIT_MACROS (Shift-F10). This lets you assign short (one-line) command sequences to the shifted function keys, and doesn't require MACRO mode operation.

Once a macro is defined, you can execute any lengthy series of commands just by invoking its name. See the appendix "Writing Macros" for further information. See also BPEX.

What a Macro Does

If you repeatedly execute a particular sequence of commands, or use the same file frequently, consider taking the time to define a one-word macro. It can save time and trouble, even in the short term.

A macro is a new command, created from pre-existing commands and their usual parameters. For example,

```
: LOADUP     0 3FFF BINLOAD  A:MYPROG  ;
```

creates a macro called LOADUP that uses the UniLab command BINLOAD to load a file called "myprog" from drive A:. You can see that typing LOADUP is easier and less error-prone than using the parameters-BINLOAD-filename sequence to load this file.

How to Define Macros

A macro's definition begins with a colon and ends with a semicolon (;). The first word after : is the macro's name, and the rest is its definition.

There must be at least one space after the colon, and another before the semicolon. You can use any commands and parameters, even other macros you have already defined, as long as one or more spaces separate each item:

```
: NAME  FIRSTWORD SECONDWORD VALUE THIRDWORD ;
```

Forth

When you define a macro, you are using the Forth programming language. With it, you can define new UniLab commands that use conditional statements, loops, and more. The best introduction to this powerful language is Leo Brodie's *Starting FORTH* (Prentice-Hall).

Why Macros

The example below defines a macro called READRAM. After the macro has been defined, typing READRAM will set a trigger to record cycles that read from the address range 1000 – 1FFF.

Example

```
: READRAM  ONLY READ  1000 TO 1FFF ADR  S ;
```

Defines a macro called READRAM.

Comments

Whenever the word immediately following : is typed, the result is the same as if the rest of the words up to ; were typed. After entering the above example, typing the word READRAM will have the same effect as typing ONLY READ 1000 TO 1FFF ADR S.

To save a macro definition, use the "save conFiguration" option of the Configuration PopUp (or SAVE-SYS) before leaving the UniLab program.

See the appendix "Writing Macros" for more information.

;   (semicolon)
no parameters

Macro System

Ends a macro definition started by : (colon).

Usage

See the preceding discussion of : (colon), and the appendix "Writing Macros."

<TST>

`<value> ' <TST> !` *Macro System*

Set this constant to one to turn off the output of some messages, and to leave on the stack many results which are normally printed on the screen.

Usage

Used during testing procedures, and for sophisticated macros. When <TST> is set to one, words like MM? leave their results on the stack rather than displaying them. When setting this constant to a value, the apostrophe must precede the command.

Example

```
: NEWMM?   ( addr -- val )
  1 ' <TST> !   MM? 0 ' <TST> ! ;
```

This macro will act the same as MM? but will leave the word value it finds on the stack instead of displaying it.

Comments

<TST> is mainly used for writing advanced, automated test macros. See the appendix "Writing Macros" for more information.

Remember to set <TST> back to zero when you are done.

=BC

`<n> =BC`

Changes the contents of the BC register to n.

Usage

An example of register-control commands available with a DEBUG package. This command addresses the Z80's internal register BC. Consult your *Target Application Notes* for commands to alter your particular processor's registers.

Example

```
1234  =BC
```

Puts 1234 in the BC register.

Comments

You can use the register commands only while the processor is at a DEBUG breakpoint. (See `STEP-INTO` or `RB` for more about breakpoints.)

This is a typical register-changing-instruction format. A similar command is provided for each of the processor's internal registers, except `SP`.

No space appears between the = and the register name.

Advanced Users

Related commands will copy the contents of any register to the UniLab operating system's stack. The names for those commands are similar to the ones used to change the register's contents, but instead of starting with an = sign, they end with a ?.

For instance, in the Z80 DDB, you would use `BC?` to get the last value of the BC register pair. You could then use this value in conditional loops, or could mask off and automatically change only the C register at a breakpoint. See the appendix "Writing Macros."

Other DDBs will have commands like:

Change register contents	*Put register contents on stack*
`=R1`	`R1?`
`=X`	`X?`
`=DPTR`	`DPTR?`

=EMSEG

`<hex_digit> =EMSEG` *Rarely Used*

Sets the A16 – A19 context for subsequent EMENABLE statement(s). This determines which 64K segment of memory the emulated ROM will occupy. You can also set this value from within the SET-EM panel.

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 preset for each DDB software package. If you do need to change the value of EMSEG, you will find SET-EM easier to use.

Why It May Matter
Although the upper four bits of our 20-bit address bus are meaningful only with processors that can address more than 64K of memory, =EMSEG is always set to some value by the processor-specific DDB software.

On some microprocessors, those four lines are floating high; on others, several of the lines are pulled low. And on processors with more than 16 bits of address, these inputs to the UniLab are connected to the upper bits of the address bus.

How It Works
This command just sets a variable. EMENABLE is the command that actually enables memory.

When It Matters
The UniLab looks at the upper four bits of address (A16 – A19) during fetch and read cycles, to see if your microprocessor wants to fetch an instruction from emulation ROM. If the upper four bits that the UniLab sees don't match the =EMSEG parameter, the UniLab will not respond to the microprocessor's request.

Use ESTAT to see how this command affects the settings of emulated memory.

Examples

```
7  =EMSEG
```

Tells emulation memory to come onto the bus only when the upper four bits of the 20-bit address bus have the value 7 hex (0111 binary: A19 is 0, and A16 – A18 each are one.)

```
F  =EMSEG  0 TO 1FFF EMENABLE
```

Enables addresses F0000 to F1FFF.

```
E  =EMSEG 0 EMENABLE ALSO F =EMSEG 0 EMENABLE
```

Enables addresses E0000 – E07FF and F0000 – F07FF.

Comments

The four most significant bits of the UniLab's 20-bit, memory-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 processor refers to emulation memory only when:

- A16 – A19 agree with the =EMSEG value, *and*
- A11 – A15 indicate an enabled 2K block.

Inputs A16 – A19 (displayed in the trace as the right-hand digit of the CONT column) are the values seen by the emulation-enable logic. If they aren't connected, they will "float" as all 1s (hex F).

The =EMSEG command has no effect until an EMENABLE or INIT sends the data to the UniLab.

=HISTORY

```
<#Kbytes> =HISTORY
```

Selects the amount (in hex) of memory allocated to screen history.

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 3C Kbytes (decimal 60).

The new setting does not take effect until you use the "save conFiguration" option of the Configuration PopUp, exit from the UniLab software, and restart it.

Use ?FREE to find the current allocation.

Why Change It

You might want to have a longer record of on-screen events, or you might want to free some of the host RAM for other purposes.

Example

```
3C =HISTORY
```

Allocates the maximum space allowed for line history.

=MBASE

`<n> =MBASE` *F8*

Sets the number base in which the MISC inputs (M0 – M7) to the UniLab will be shown in the trace display.

Usage

The miscellaneous inputs (MISC) to the UniLab are usually displayed in binary. This allows you to see, directly, which inputs are receiving a high signal, and which are receiving a low signal. But you may have a different use for these inputs—like reading data from onboard RAM—for which a hex or decimal display would be more useful.

Normally, you will use the mode panel (F8) to change this feature, but the panel only toggles the display between binary and hex. For other bases, you must type the command explicitly.

If you are working with an eight-bit processor, this command also changes the number base of the HDATA column.

Examples

```
10  =MBASE
```

Sets a hexadecimal display, the most space efficient. Note that A, not 10, specifies a decimal display.

```
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 target signals you like.

=OVERLAY

`<address> =OVERLAY`

Changes the emulation ROM addresses used by the DEBUG software.

Usage

Use this command when your target software must use the memory normally reserved for DEBUG code, or if InSight requires more overlay area than DEBUG had previously used.

When DEBUG is enabled, the reserved area (which is just two bytes on most processors) cannot contain user code. The overlay area, located above the reserved area, is used to swap DEBUG routines into memory. Target code can reside there and be executed, but DEBUG routines cannot be used there. (See the appendix, "How DEBUG Works.")

This command changes the location of both the overlay and the reserved area. You must use the "save conFiguration" option of the Configuration PopUp (or `SAVE-SYS`) to make the change permanent.

To find the current address of the reserved area, use `Alt-F1` to display the `HELP_DEBUG` options, then choose topic 1.

Example

```
2310  =OVERLAY
```

Moves the reserved area to start at 2310, and puts the overlay area above there.

Comments

The overlay area must be entirely within a bank of emulation memory. Be careful when changing its location—be sure it does not extend beyond your current emulation memory boundaries.

You can, instead, disable the DEBUG features for completely transparent operation. See `DEBUG'`.

=PORT

```
<address> =PORT
```

Changes the address where the UniLab software expects to find the host's parallel interface card (default = 0220H).

Usage

Used to change the parallel card's port address. Only needed if you wish to connect multiple UniLab 8620s to your host computer, or if you have another peripheral device that uses port addresses 0220H – 023FH.

Acceptable values for =PORT are:

```
200   220   240   260   280   2A0   2C0   2E0
300   320   340   360   380   3A0   3C0   3E0
```

You will need to change the jumpers on the slot card to reflect any change in address. See the *User's Guide* appendix "Parallel-Board Configuration."

You must use the "save conFiguration" option of the Configuration PopUp (or SAVE-SYS) to make the change permanent.

Example

```
240  =PORT
```

Tells the software to look for the UniLab at host port address 240.

=register

```
<word> =register
```

This is the format of cpu-specific commands that change the contents of a register to n. (No space between the = and the register name.)

Usage

Represents one kind of processor-specific, register-control commands available with a DEBUG package. Consult the *Target Application Notes* for commands that will alter the contents of your processor's registers.

Example

```
1234   =BC
```

Puts 1234 in the BC register.

Comments

You can use register commands only while the processor is at a DEBUG breakpoint. See `STEP-INTO` or `RB` for more about breakpoints.

This is a typical register-changing instruction format. A similar command is provided for each of the processor's internal registers, except `SP`.

Advanced Users

There are equivalent commands which will get the contents of any register, after a breakpoint display, and copy the value onto the stack. The names for these commands are the same as those for changing the register's contents but they *do not* start with an equals sign, they end with a question mark.

For example, in the Z80 DDB, you would use `BC?` to retrieve the last value of the BC register pair. You could then use this value in conditional loops; or you could mask off and automatically change only the C register at a breakpoint. See the *Reference Manual* appendix "Writing Macros."

Other DDB's will have commands like these:

Change register contents	*Put register contents on stack*
`=R1`	`R1?`
`=X`	`X?`
`=DPTR`	`DPTR?`

=SAMP

```
<#milliseconds> =SAMP
```

Adjusts the delay between cycles collected by the UniLab commands SAMP and ADR?. The default is 100H milliseconds (about 1/4 second).

Usage

Changes the length of the delay in order to see more or fewer samples per second. Set it to zero for the fastest possible sampling rate.

You must use the "save conFiguration" option of the Configuration PopUp (or SAVE-SYS) in order to make the change permanent.

Example

```
400  =SAMP
```

Slows the sampling rate to approximately one per second (400H millisecond delay between samples).

=STROBE

```
<#milliseconds> =STROBE
```

Changes the length of the stimulus pulse that can be sent from the InSight display (default = 30H). See INSIGHT for more information.

Usage

Use this to change the length of the stimulus pulse generated by InSight. You must use the "save conFiguration" option of the Configuration PopUp (or SAVE-SYS) to make the change permanent.

Example

```
200  =STROBE
```

Sets the InSight display's stimulus strobe to 200H milliseconds duration.

=SYMBOLS

```
<#Kbytes> =SYMBOLS
```

Sets the amount of space allowed for UniLab symbol tables.

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 use the Configuration PopUp to "save conFiguration," exit from the UniLab software, and restart the saved system.

Use ?FREE to view the current allocation.

Why Change It

You might want a larger symbol table, or you may need to conserve host RAM for other purposes.

Example

```
80 =SYMBOLS
```

Makes the symbol table the maximum possible size.

=WAIT

```
<#milliseconds> =WAIT
```

Determines how long after resetting the target processor the UniLab will wait before checking for a working processor clock.

Usage

Used when the target system needs a longer interval after reset to start functioning. The default value is 140 (hex). Use the "save conFiguration" option of the Configuration PopUp (or SAVE-SYS) to make this change permanent.

Example

```
280  =WAIT
```

Sets the wait time to twice the default value.

?FREE

no parameters

Displays the amount of host RAM allocated to screen history and to the symbol table. Also shows how much host RAM is currently free.

Usage

Used to find how much you can increase the amount of space dedicated to history or to the symbol table, or whether you need to reduce it.

See =HISTORY and =SYMBOLS.

ADR

```
          <word> ADR
<word> TO <word> ADR
```

Sets the trigger specification for analyzer inputs A0 – A15. (Sets trigger for A0 – A19 if `<word>` is a five-digit address ending in a period.)

Usage

Determines which 16-bit addresses will trigger the analyzer. Can also trigger on 20-bit addresses. With `TO`, the trigger will occur on any value in the address range. If `NOT` precedes the address value(s), the UniLab will trigger outside the specified address (or range).

All previous entries to the address trigger spec are erased unless you precede this spec with the word `ALSO`.

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 on address outside the range 120 – 455.

```
12345. ADR S
```

Trigger on 20-bit address 12345. The 1 will appear in the right digit of the CONT column.

Warning—You can inadvertently produce "cross-products" when using more than one `ALSO` statement with `ADR`:

```
1200 ADR  ALSO 8 ADR  ALSO 1503 ADR
```

Sets the analyzer to trigger when the address is 1200, 0008 or 1503. But because of cross-products, it will also trigger on address 0003 and 1508.

Comments

`AS` is a convenient abbreviation for `NORMT ADR S`.

Generally, you can use multiple `ALSO`s without cross-products if the high-order byte of the previous spec and the new one match. To avoid this problem entirely, specify the two bytes of the address separately with `HADR` and `LADR`.

ADR?

no parameters

Displays addresses seen on the bus—approximately ten every second.

Usage

This command displays ten of the addresses that appear on the bus each second. This is useful for getting a rough idea of how a program behaves. (To change the rate of sampling, see =SAMP.)

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 if the target program gets stuck in a loop. ADR? turns RESET off and sets a trigger spec of its own, so use one of the NORMx words before starting a new trigger spec.

AFTER

`AFTER <qualifier_specification>`

Precedes the description of a qualifying event. Qualifying events are bus states that must be seen before the analyzer will search for the trigger.

Usage

Qualifiers are used to postpone the analyzer's search for a trigger until an event that matches the qualifier specification has been seen on the target bus. Each qualifier must begin with AFTER. In other regards, qualifiers are defined like triggers.

You can specify up to three qualifiers, a sequence of bus cycles that must appear on the target bus *with no intervening cycles* before the analyzer will begin to watch for the trigger event. The trigger itself can appear any time after the qualifier, or sequence of qualifiers, has occurred.

You cannot use MISC inputs as qualifiers.

Delays and Repetitions

You can specify how many bus cycles must follow the last qualifier before the UniLab will look for the trigger. (See PCYCLES. Default = 0 PCYCLES.)

You can specify how many complete repetitions of the qualifier sequence must occur before the UniLab will look for the trigger. (See PEVENTS. Default = 1 PEVENTS.)

Examples

```
NORMT 100 ADR AFTER 535 ADR S
```

Triggers on address 100, any time after address 535 is seen on the bus.

```
AFTER 3F DATA S
```

Adds a second qualifying event, which must occur before the first. Now, address 535 must be immediately preceded by data 3F 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 immediately follows 344.

Comments

Equivalent results can be obtained with <n> QUALIFIERS to set the number of qualifiers; the four related commands TRIG, Q1, Q2, and Q3 can then be used to define the various triggers. But for most users, AFTER is the more natural way to do it. You will find the other commands handy when you want to change the description of an qualifier sequence without starting from scratch.

AHIST

no parameters *PPA*

AHIST, the address-domain histogram, invokes the optional Program Performance Analyzer (PPA) to display the activity of your target program in up to 15 user-specified address ranges. See also MHIST and THIST.

Usage

Allows you to examine the performance of your target software, to see where the program spends its time. Press Esc to exit from this menu-driven feature.

Before you use the PPA the first time, you must issue the command SOFT to install this optional feature.[1] SOFT performs a SAVE-SYS, then exits to DOS. The next time you boot the UniLab software, the PPA will be installed.

Start the Histogram—To produce a histogram, first specify the upper and lower limits of each address "bin" you want displayed, then start the histogram.

When you issue the command AHIST, you get the histogram screen with the cursor positioned on the first bin. You can then type a lower and upper limit for each bin. Press return, tab, or an arrow key to move to the next field.

Press F1 to start displaying the histogram.

Save to a File—You can save a histogram's setup (i.e., bin limits, title, and any labels) in a file, along with any collected data, after you exit from the histogram screen. Just type HSAVE <filename> at the Command> prompt. Or, use the Files PopUp instead.

Load From a File—You can load a previously saved histogram by typing HLOAD <filename> at the Command> prompt, or via the Files PopUp. Loading a histogram file also invokes the PPA in the correct mode.

Example

```
AHIST
```

This command is never used in combination with anything else.

[1] *Only the first time you use the PPA.*

ALSO

no parameters

Used with EMENABLE and with trigger-specification commands. Adds the following input-group specification to the current trigger, instead of overwriting any existing value for that input group.

Usage

The trigger-spec commands CONT, ADR, DATA, HDATA, HADR, LADR, and MISC, normally cause the UniLab to replace any existing parameters with the new ones. By using ALSO, you can instruct the UniLab to trigger on the old *or* the new conditions.

EMENABLE normally clears any previous settings when it enables the new addresses in memory. By using ALSO, you can add a new range of emulation memory without losing the current settings. (See the second example below.)

In trigger specifications, ALSO is only necessary when you expand the existing specification of an input group. The UniLab automatically ANDs the specs of different groups, unless you explicitly clear the current trigger (e.g., with NORMx).

Note: You can inadvertently produce "cross products" when using more than one ALSO statement 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 be 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

```
<key#>  ALT-FKEY  <command>
```

Assigns a command to an Alt-function key combination.

Usage

Reassigns the function keys on IBM PCs and compatibles. Press Alt-F1 (see the FKEY? command) to view all the current function-key assignments.

The function keys allow you to execute a command or macro with a single keystroke. The default assignments represent many common functions, but you can change them to suit your particular needs and working habits.

To make such assignments permanent, use the "save conFiguration" option of the Configuration PopUp (or SAVE-SYS) before quitting.

Example

```
2 ALT-FKEY WSIZE
```

Assigns the command WSIZE to Alt-F2.

Comments

To execute a sequence of commands in this way, even including parameters, define a macro (see MACRO) and then assign the macro to a function key. (See EDIT_MACROS for information about Easy Macros.)

See also FKEY and CTRL-FKEY.

ANY

```
ANY <input_group>
```

Sets a trigger on any value in the specified input group.

Usage

Provides a way to clear the trigger specification for a single input group. This saves the trouble of re-typing a trigger spec from scratch when you just want to clear one input group's specs.

Example

```
ANY CONT
```

Triggers when any value appears on the CONT input lines (although the rest of the trigger spec remains unchanged and in effect).

Comments

The macro definition of this command looks like this:

```
: ANY   0 TO FFFF ;
```

AS

```
<addr> AS
```

An abbreviation for `NORMT <addr> ADR S`.

Usage

Defines an analyzer trigger spec, and starts the analyzer. The trigger event appears near the top of the trace as cycle zero. A useful abbreviation—saves keystrokes when you just want to trigger on an address.

Won't work on address ranges (i.e., with `TO`) or with `NOT`.

Example

```
1234 AS
```

Triggers on an address value of 1234.

Comments

The macro definition of this command:

```
: AS NORMT ADR S ;
```

ASC

no parameters

Alt-F4

Displays an on-screen chart of the ASCII characters.

Usage

Shows each character with its decimal and hex value.

Example

```
ASC
```

This command is never used in combination with anything else.

Comments

This saves the trouble of hunting for a printed ASCII table. (See `MODIFY`, if you want to type ASCII characters directly into emulation memory or target RAM.

ASCII

```
ASCII <character>
```

Converts the following character into the ASCII code for that character. See also `EMIT`.

Usage

A quick way to get the ASCII code for a single character. See also `ASC`.

Example

```
ASCII z
```

Leaves on the stack the ASCII code for lower-case z.

ASM

`<address> ASM <instruction>`

Invokes the processor-specific, line-by-line assembler.

Usage

Patches assembly language code to the given address in emulation ROM. Allows you to overwrite target program addresses in the UniLab's emulation ROM, to quickly fix simple bugs when you find them. The assembler overwrites 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, ASM will display the address to which it is assembling, and wait for an instruction followed by a carriage return.

The assembler will continue to prompt you with consecutive addresses, patching the hand-entered code into memory until you press End on a blank line.

Conventions

The line-by-line assembler will only accept assembly language instructions, not ORIGIN or EQU statements. (Use the UniLab command IS if you need to define symbols.)

Only type one instruction per line.

The normal conventions of assembly language apply, e.g., at least one space between the instruction and the operands. The *Target Application Notes* list the instruction set recognized by the assembler.

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 displays a blank line beginning with that address, and waits 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 from the screens in 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 may want to edit and re-load. ASM-FILE follows the same input conventions as ASM.

Include comments on a screen by preceding them with a semicolon (;). The assembler will ignore everything after the semicolon on that line. The semicolon can be the first character on a line, or it must be preceded by a space.

Forth Files and the Editor

You will want to put your code into a file of its own if you want to save or archive it. You must make a macro-level system before you can use the file commands. (See MACRO.)

If you only have a few lines of code to write, you can use the MEMO screen. See MEMO for advice about the Forth screen editor.

Opening a New File

Create a new file with OPEN <filename>. It will be created with three screens; increase its size with <#screens> SCREENS (1K is allocated per screen).

Type <screen#> EDIT to get into the file. Use screen zero only for comments, if at all, because code cannot be loaded from it later.

When you are finished assembling from anything other than the default system file, press F10 twice: once to get into the PopUp display, and again to exit. This reopens the UniLab system file. If you don't do this, some On-Line Help and error messages may not work.

Examples

```
1200  1D 1F  ASM-FILE
```

Loads assembly code into memory, starting at address 1200, from screens 1D – 1F of the currently open Forth file.

```
1 4 ASM-FILE
```

Loads code from screens 1 – 4, starting at the current value of ORG.

B#

`B# <binary_number>` *Rarely Used*

Interprets the following number as a binary number.

Usage

Useful when you want to input a number in 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
```

The same as entering 0A89H.

```
NORMT  B#  1111110  MISC  S
```

Triggers when the `MISC` inputs are 11111110.

Comments

Changes the base to binary, *just for the next number.* Allows input of individual binary numbers, just as `D#` allows decimal input.

B.

`<hex_number>  B.` *Rarely Used*

Displays the preceding hex number as a binary number.

Usage

When you want to find the binary equivalent of a hex number, this will save pencil and paper.

Example

```
A89 B.
```

Displays the binary equivalent of A89, which is 101010001001.

BINLOAD

```
<from_addr> <to_addr> BINLOAD <filename>
```

Loads a binary file from disk into emulation memory. Prompts for the filename if you don't include it on the command line.

Usage

Starts loading a binary file into memory at the from_addr, and stops at the to_addr or at the end-of-file, whichever comes first. The binary file should contain only executable code. This command fully supports DOS pathnames, and can be used to load the product of a cross-compiler into emulation memory.

Use with .COM, .BIN, or .TSK files. See HEXLOAD for Intel hex files, LOADER for other formats. See also BINSAVE.

Example

```
0 400 BINLOAD \ASM\MAIN.BIN
```

Loads the binary DOS file MAIN.BIN from the \ASM directory, starting at location 0 and ending at location 400.

Comments

Loads the exact binary contents of a file until DOS indicates end-of-file or the to_addr is reached. If you don't know the ending address, just enter FFFF as to_addr, and loading will stop at the end-of-file.

The Orion software can load to target RAM as well. As with all memory-writing commands, don't write into your stack area when loading into RAM. See STEP-INTO and RB.

Operator Shortcuts

You can load binary files via the Files PopUp, or press F5 at the Command> prompt.

For Advanced Users

When using BINLOAD in an advanced macro, you have two options:

1. Include in the macro's definition the name of the file you wish it to load. This is useful in macros that are used with the same files—or with files of the same name—every time. For example:

```
        : MY_LOAD 0 7FF BINLOAD MY.BIN ;
```

2. When the macro is executed, have it ask the user to supply the name of the desired symbol file. As shown below, [COMPILE] defers the following command's action until the macro is executed. For example:

```
        : MY_LOAD 0 7FF [COMPILE] BINLOAD ;
```

BINSAVE

`<from_addr> <to_addr> BINSAVE <filename>`

Saves the specified range of memory as a binary file, prompting for the filename if you didn't include it. Or use the Files PopUp instead.

Usage

This command saves a range of program memory to a disk file, fully supporting DOS pathnames. Often used after loading a program via HEXLOAD, since the new binary file will load faster than the hex version.

Example

```
100 4FF BINSAVE
```

Saves target locations 100 – 4FF to a disk file. Since a filename was not entered after BINSAVE, the user will be prompted it.

Comments

Saves the binary contents of a range of target memory as a named file. This file can later be reloaded with the BINLOAD command.

You can use this command to save the contents of target RAM as well.

See also STEP-INTO and RB.

Operator Shortcuts

You can save binary files via the Files PopUp (F5).

Advanced Usage in Macros

When using BINSAVE in an advanced macro, you have two options:

1. Include in the macro's definition the name of the file you wish it to save. This is useful in macros that generate the same files—that is, files of the same name—every time. For example:

```
: MY_LOAD 0 7FF BINSAVE MY.BIN ;
```

2. When the macro is executed, have it ask the user to supply a filename for the output file. As shown below, [COMPILE] defers the following command's action until the macro is executed. For example:

```
: MY_LOAD 0 7FF [COMPILE] BINSAVE ;
```

BPEX

`BPEX <macro_name>` *Macro System*

Executes the specified command or macro at every breakpoint, after displaying the register contents.

Usage

Allows you to automatically execute a command or macro at every breakpoint.

BPEX will not accept a string of commands, just the first word that follows. This means no parameters can be used directly with BPEX, but there is a way to work around this apparent limitation. In the example below, the macro called SEE-RAM invokes MDUMP, which requires two parameters; but the same values are to be used each time it is executed, so they are included within the macro's definition. The one-word macro name is then assigned to BPEX normally, the required parameters implicit in the macro's definition.

See MACRO, : (colon), and the appendix "Writing Macros."

Turn It Off

To turn off automatic execution of a command at every breakpoint, simply type: BPEX NOOP.

Example

```
: SEE-RAM  8000 8080 MDUMP  ;
```

Defines a macro called SEE-RAM that dumps 80 memory locations.

```
BPEX SEE-RAM
```

Executes the newly defined macro by dumping the above-specified range of memory at every subsequent breakpoint.

Comments

Available only with DEBUG packages. This is useful if you want, for example, to watch a memory window as you single-step through the program, as above.

BPEX2

`BPEX2 <macro_name>`

Macro System

Executes a second macro at each breakpoint.

Usage

See BPEX.

BYE

no parameters

Exits from UniLab program.

Usage

Returns to DOS. Use the "save conFiguration" option of the Configuration PopUp to save the current state of the system.

Use the UniLab command `DOS` instead, if you just want to execute a few DOS commands and return to the UniLab.

Example

```
BYE
```

This command is never used in combination with anything else.

CATALOG

no parameters

Displays a directory of all the available pinouts. The proper cable hookups for each microprocessor are shown along with the pinouts.

Usage

Once this word is entered, select the number of the microprocessor diagram you'd like on the screen. You can continue looking at pinouts until you press `Esc` to exit.

Comments

The pinout information is contained in three library files on the Orion Utilities diskette. The UniLab system can run without them, but `CATALOG` and `PINOUT` will not work unless these files are in the \ORION directory on your disk.

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 by 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, use `CKSUM` to calculate the checksum. Last, patch in the complement of the sum.

For example, if the sum is 1234, use the command `-1234 FFE MM!`. The resulting PROM will have a checksum of 0.

CLRMBP

no parameters

Clears any multiple breakpoints that have been set.

Usage

Used to wipe the slate clean and start setting multiple breakpoints again. Use SMBP to set the breakpoints.

Example

```
CLRMBP
```

This command is never used in combination with anything else.

Comments

Clears all the numbered breakpoints set with SMBP. To clear one at a time, see RMBP.

CLRSYM

no parameters

Clears the current symbol table.

Usage

Used to get rid of any symbols currently defined for your program. It's a good idea to save the symbol table first, in case you decide you want them after all. (See SYMSAVE.)

You could instead use SYMB', to turn off the effects of the symbol table without erasing it.

Example

```
CLRSYM
```

This command is never used in combination with anything else.

Comments

SYMLOAD clears the symbol table automatically before it loads the new symbols.

COLOR

no parameters *F8 – Rarely Used*

Displays in color. Only effective when used with a color monitor.

Usage

Turns on the color display. You have to use the "save conFiguration" option of the Configuration PopUp (or `SAVE-SYS`) afterward if you want the UniLab program to start up with the color display.

Changing Colors

Use the UniLab command `SET-COLOR` to show the settings as you change them. You will have to save the system configuration if you want to preserve the new colors.

Example

```
COLOR
```

This command is never used in combination with anything else.

CONT

```
             <byte>  CONT
  <byte> TO <byte>  CONT
<byte> MASK <byte>  CONT
```

Rarely Used

Sets 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 – 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 in the range. NOT causes the UniLab to trigger when the value falls outside the specified range or value.

You can use <k> MASK <m> CONT to examine any subset of the eight input lines. See below for details.

Unless you use ALSO, the previous trigger spec is cleared when you use this command.

Examples

```
B# 00011111 CONT
```

Specifies a cycle in which inputs C7, C6, and C5 are zero, and C4, A19, A18, A17, and A16 are one.

```
70 TO 7F CONT
```

Specifies a cycle in which input C7 is zero and C6 – C4 are one (A19 – A16 can be any value).

```
F MASK 3 CONT
```

Specifies a cycle in which inputs A19 and A18 are zero, and A17 and A16 are one (C7 – C4 can be any value).

Comments

The low four bits of the CONT lines refer to the highest four bits of the address, the segment-address bits that are set by =EMSEG.

When you use the command <k> MASK <m> CONT, the bits in <k> with a value of one represent the inputs the UniLab will examine. The <m> indicates what value those lines must have in order to trigger.

For example, F0 MASK A0 CONT tells the UniLab to only look at the upper four bits of the CONT lines. The A0 tells the UniLab to trigger when bits 7 and 5 are high while bits 6 and 4 are low. The UniLab will ignore the value of the lower four bits.

CONTROL

`CONTROL FILTER` *Rarely Used*

Used before `FILTER` to set a filter specification on just the CONT inputs.

Usage

Rarely used. You probably will never use this command, which triggers on the full specification, but filters based only on the eight bits of the CONT inputs.

The filter mechanism of the UniLab gets turned on by the `xAFTER` macros. Those commands set the filter to `MISC' FILTER`, which allows you to filter all inputs except for the MISC wires.

See also `HDAT`, `MISC'`, and `NO`.

The CONT Inputs

The upper four CONT bits identify the processor cycle type, while the lower four bits identify 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 write cycles. You must use `DEVENTS` to get a trace buffer full of cycles that match the filter spec. (The UniLab 8620's trace buffer holds AA8 cycles. The first filtered cycle in a trace is always non-valid.)

CTRL-FKEY

```
<key#>  CTRL-FKEY  <command>
```

Assigns a command or macro to a `Ctrl`-function key combination.

Usage

Reassigns the function keys on IBM PCs and compatibles. `Alt-F1` (see the `FKEY?` command) shows the current assignments.

Function keys allow you to execute a single command or macro with one keystroke. The default assignments execute many common functions, but you might want to change them to suit your particular needs and working habits.

To make your assignments permanent, use the "save conFiguration" option of the Configuration PopUp (or `SAVE-SYS`).

Example

```
5 CTRL-FKEY DOS
```

Assigns the UniLab command `DOS` to `Ctrl-F5`.

Comments

To execute a string of commands in this manner, perhaps even including parameters, define a macro (with `:`) and assign the macro's name to a function key. (See `EDIT_MACROS` for information about easy macros.)

See also `FKEY` and `Alt-FKEY`.

CYCLES?

`<from_addr> <to_addr>  CYCLES?`

Counts the number of bus cycles between two addresses.

Usage

Used to count the number of bus cycles required by a loop, as in the first example below, or by the code between any two addresses. See USEC? to count the milliseconds between two addresses.

Bus Cycle Count

Not the number of machine cycles or the number of instructions fetched, but the number of *bus cycles* that occur between appearances of the first address and the second on the bus.

Example

```
123 123 CYCLES?
```

Counts the cycles between two occurrences of the address 123, as when this address is inside a loop

```
123 456 CYCLES?
```

Counts the cycles between address 123 and address 456.

Comments

Useful for checking quickly whether a loop works as you intended. CYCLES? sets its own trigger spec, which you will have to clear with a NORMx command before proceeding with other trigger specifications.

D#

`D# <decimal_number>` *Rarely Used*

Treats the number that follows as a decimal value, rather than hexadecimal, which is the default.

Usage

Saves the trouble of converting a number to hex before using it as input.

Examples

```
D# 16 ADR
```

Equivalent to entering 10 ADR.

```
D# 32 .
```

Displays 20 (the hex equivalent of 32 decimal).

```
D# 135 B.
```

Converts a number from decimal to binary.

```
D# 1000 MS
```

Pause 1 second.

Comments

See also B# for entering binary numbers.

DASM

no parameters *F8*

Enables the trace disassembler.

Usage

Turns on disassembly of machine code. You will usually want this on, turning it off for functions like `xAFTER` and `SAMP`. To turn off the disassembler, use `DASM'`.

Normally, you will use the mode panel (`F8`) to toggle this feature on/off.

Example

```
DASM
```

Turns on the disassembly in the trace display.

Comments

Works only if you have installed an optional disassembler.

DASM'

no parameters *F8*

Disables the trace disassembler.

Usage

Turns off disassembly of machine code. See `DASM`.

Normally, you will use the mode panel (`F8`) to toggle this feature.

Example

```
DASM'
```

Removes the trace display's column for disassembled instructions.

DATA

```
            <byte> DATA
  <byte> TO <byte> DATA
<byte> MASK <byte> DATA
```

Changes the analyzer trigger for the data inputs (D0 – D7).

The Data Inputs

The UniLab gets an address and data from the bus during each memory read and write. The "data" that appears on the bus could be 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 a trigger on a single data value. The UniLab will trigger when it sees that hex number on the bus as data. (See the first example below.)

Ranges of Data

`TO` lets you set 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 on the DATA inputs is outside the specified range or value.

Masking

You can use `<k> MASK <m> DATA` to examine any subset of the eight data lines. The high bits of `<k>` that have a value of one represent the inputs the UniLab will examine, while the bit configuration of byte `<m>` indicates what value those lines must have in order to trigger.

For example, `80 MASK FF DATA` selects only the highest data bit for examination (with binary value 1000 0000), and tells the UniLab to trigger when this bit has a high value. (The instruction `80 MASK 80 DATA` would have the same effect.)

Examples

```
NORMT 12 DATA S
```

Clears previous trigger settings with NORMT, then sets a trigger on data input 12, and uses S to start the analyzer.

```
12 TO 34 DATA
```

Requires a data value between 12 and 34 hex in order to trigger.

```
F0 MASK 30 DATA
```

Instructs the UniLab to examine the four highest data bits, and to trigger when it finds a three on those lines.

```
23 DATA ALSO 45 DATA
```

Will trigger on a cycle that has a data value of 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 the data bus, and HDATA is the other half.

If you need to use a large number of ALSO terms, see NDATA.

DCYCLES

`<#cycles> DCYCLES`

Sets how many cycles the UniLab will record after finding a cycle that matches the trigger specification.

Usage

When the UniLab sees the trigger event on the target board, it consults this variable to determine how many more cycles to record. Each time a new cycle enters the trace buffer, the oldest recorded cycle is lost. After the UniLab records the specified number of post-trigger cycles, it displays the trace buffer's contents.

Why You Don't Need to Bother—This command is executed by a number of other commands. `NORMT`, for example, sets the delay value so that the trigger event is near the top of the trace buffer (ten cycles from the top).

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 goes at that time. The `DCYCLES` command does the job perfectly.

Example

```
104 DCYCLES
```

Specifies 104 hex delay cycles (260 decimal)

Comments

`NORMT`, `NORMM`, and `NORMB` set `DCYCLES` so that the trigger is at the top, middle, and bottom of the trace, respectively. `S+` increases the number of delay cycles by AAAH cycles, so you can see what follows the events in the current trace. The maximum possible delay count is FFFF.

DEBUG

no parameters *F8*

Re-enables DEBUG after it has been disabled by `DEBUG'` or `EMCLR`.

Usage

Use this when you have turned off the DEBUG features and now want to use them again. `STEP-INTO` will not work until this and the hardware `STEP-INTO` are enabled. (`RB` automatically executes this command and re-enables DEBUG.)

Normally, you will use the mode panel (`F8`) to enable/disable this feature. Press `Alt-F1` or use the Help PopUp for On-Line Help with DEBUG and information about processor-specific features.

Additional Information

Use `HELP <command>` for details about the following, which represent the most common DEBUG functions:

```
G            GB           GW
LP           R            RB
RZ           STEP-INTO    STEP-LINE
STEP-OVER    SOFTWARE     HARDWARE
```

DEBUG'

no parameters *F8*

Turns off the DEBUG features.

Usage

This command is used for complete transparency—emulation memory is not affected by the UniLab operation. (DEBUG is disabled automatically by EMCLR.) It allows your program to use areas otherwise reserved for DEBUG vectors and overlays. Press `Ctrl-F1` and press 2 to show the reserved areas for your processor.

Normally, you will use the mode panel (`F8`) to toggle this feature on/off.

Comments

The command `RESET <addr> RB` re-enables DEBUG automatically, if it was turned off.

DEVENTS

`<#events> DEVENTS` *Rarely Used*

Sets the number of trigger events that must occur before the UniLab freezes the trace buffer. Also forces 1 DCYCLES.

Usage

Used to see the nth target bus cycle that matches the trigger specification. Rarely useful. The nth occurrence shows as the last cycle in the trace buffer.

See also PEVENTS.

Example

```
NORMT 1800 TO 18FF ADR 30 DEVENTS
```

Sets 30 delay events—the last cycle in the trace will be the thirtieth access to memory between addresses 1800 and 18FF.

DM

`<start_addr> <count> DM`

Disassembles <count> number of lines, starting at the given address.

Usage

Very useful for examining memory. Allows you to see what instructions are in emulation memory, and in RAM as well. (See also DN.)

Can give misleading results if start_addr isn't the first address of an opcode; but even then, it 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. (See STEP-INTO and RB.)

DMBP

no parameters

Displays the settings of all eight multiple breakpoints.

Usage

Used to show the settings of your multiple breakpoints. Automatically executed whenever you set one of the eight multiple breakpoints with `SMBP`.

Example

```
DMBP
```

This command is never used in combination with anything else.

DN

no parameters

Displays a special disassembly window on the right-hand side of the screen. Prompts for the starting address of the disassembly. Displays one instruction per line and fills the dedicated window.

Usage

This command is similar to DM, except it writes into a portion of the screen used only for this feature. Use it to keep a disassembly on screen while performing other operations.

The disassembly produced by DN will stay on the screen if you exit from the display with End. If you use Esc to exit, it will not stay on the screen.

You can also remove the display by using F2 to get rid of the split screen and then scrolling (or, if currently looking at an unsplit screen, press F2 twice).

Example

```
DN
```

This command is never used in combination with anything else.

Comments

Due to limited screen space, DN doesn't display symbols.

DOS

```
DOS   <DOS_command>
```

Execute a DOS command from the UniLab program. Or, use with no parameters to exit to DOS temporarily. Return to the UniLab by typing `EXIT`.

Usage

Use this when you have reason to use a DOS utility. You can execute a single command, or go to DOS and execute a series of commands. If you go to DOS, you can 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 will have to start it again normally at the DOS prompt.

Examples

```
DOS DIR /w
```

Executes the DOS command `DIR /w`.

```
DOS
```

Allows you to execute any series of DOS commands and return to the UniLab program.

```
DOS  RENAME MY.BIN OLD.BIN
```

Changes the name of a file called MY.BIN to OLD.BIN.

For Advanced Users

You cannot use `DOS` in advanced macros. In order to access DOS functions in a macro, you use the lower level command `<DOS>`. `<DOS>` expects the command to precede it in quotes. For example, if you wanted to change the name of a file with a macro, you would do it like this:

```
: CHANGE_FILE " RENAME ORION.BIN PLAN9.BIN" <DOS> ;
```

Or if you wanted a listing of all binary files in a directory called \WORK, you would enter:

```
: WORK_DIR " DIR \WORK\*.BIN" <DOS> ;
```

Note that the first quote must have a leading and a trailing space.

EDIT_MACROS

no parameters *Shift-F10*

Invokes the macro view/edit display.

Usage

EDIT_MACROS provides an easy way to assign a series of commands to a function key.

Once within the EDIT_MACROS display, choose a macro number and type any series of commands that you can execute from the keyboard. This group of commands is assigned to a shifted function key.

For example, if you press 3 when prompted for the macro number, your macro will be assigned to function key Shift-F3, and can also be invoked by typing the command MACRO_3.

Exit the EDIT-MACRO display by pressing Esc.

Example

```
EDIT_MACROS
```

After you type EDIT_MACROS, you will be prompted for the number of the macro you wish to define or edit.

EMCLR

no parameters

Tells the UniLab not to emulate ROM—clears the emulation memory settings.

Usage

Tells the UniLab not to respond to any microprocessor requests for data or instructions. Used only when you want to run a program from on-board ROM.

Instead of running the program from a chip, you can use the EPROM PopUp to read a program into emulation memory from most ROM chips.

Side Effect

This word also disables all DEBUG features. To turn them on again, use the DEBUG toggle in the mode panel (F8), or type DEBUG.

Example

```
EMCLR
```

This command is never used in combination with anything else.

EMENABLE

```
              <address> EMENABLE
<from_addr> TO <to_addr> EMENABLE
```

Enables emulation memory; needed before you can load a program. See also SET-EM.

Usage

Used to enable memory from within macros. Otherwise, the SET-EM interface is easier to use.

You must first set =EMSEG properly when you use EMENABLE.

Given a single address, EMENABLE enables the 2K memory region that includes the address. Given a range, EMENABLE enables the group of 2K blocks that contain the range.

You can use the "save conFiguration" option of the Configuration PopUp (or type SAVE-SYS) to make the current settings permanent.

Enable a Range of Addresses

TO enables emulation memory from the beginning of the 2K segment that includes the <from_addr> to the end of the 2K segment that includes the <to_addr>.

Clear Previous Settings

Unless you precede an emulation statement with ALSO, it clears any previous EMENABLE settings.

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 refer to the same physical location in the UniLab. Of course, you can enable separate areas which do not overlap; e.g., 0 TO 3FFF and C000 TO FFFF would not conflict.

Examples

```
F  =EMSEG   O EMENABLE
```

Enables target addresses 0 – 7FF with A16 – A19 set high.

```
0 TO 1FFF EMENABLE ALSO FFFF EMENABLE
```

Enables 0 – 1FFF and F800 – FFFF.

```
F =EMSEG 0 EMENABLE ALSO E =EMSEG 0 EMENABLE
```

Enables locations F0000 – F07FF and E0000 – E07FF

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 in which each entry corresponds to a 2K segment of memory. When the segment *and* the 2K block are enabled, the UniLab accepts the address, and puts its data on the bus.

EMIT

`<code> EMIT`

Types the character represented by the ASCII code. See also ASCII, ASC.

Usage

Used as a quick way to get the character that corresponds to an ASCII code.

Example

```
03 EMIT
```

Types a heart-shaped graphic character.

EMMOVE0

`<start_addr> <end_addr> <dest> EMMOVE0`

Moves a block of memory from the current page of emulation memory to PAGE0.

Usage

Moves data from one 64K page of emulation memory to the other. See also MMOVE (for moves within a page), and EMMOVE1 (for moving to PAGE1).

Examples

```
PAGE1 1000 2000 1000 EMMOVE0
```

After making PAGE1 the current page, copies the data from 1000 – 2000 to addresses beginning at 1000 in PAGE0.

```
PAGE1 200 2FF FF00 EMMOVE0
```

Copies the data from 200 – 2FF in PAGE1 to FF00 – FFFF in PAGE0.

Comments

Make certain the code you move is relocatable. If it is not, you may have to patch some of the absolute address references. In general, exercise caution—use DM on the moved memory to see if the instructions still do what you intend.

EMMOVE1

`<start_addr> <end_addr> <dest> EMMOVE1`

Moves a block of memory from the current page of emulation memory to PAGE1.

Usage
Moves data from one 64K page of emulation memory to the other. (See also MMOVE, for moves within a page.)

See EMMOVE0.

EMSTAT

no parameters

Graphically displays the emulation memory settings. See also `SET-EM`, `ESTAT`.

Usage

Used to see which 2K blocks of target ROM are being emulated by the UniLab.

Example

```
EMSTAT
```

This command is never used in combination with anything else.

ESTAT

no parameters

Provides a text-only description of the status of emulation memory. See also `EMSTAT`, `SET-EM`.

EVENTS?

no parameters

Starts the analyzer and counts the occurrences of the currently defined trigger event.

Usage

Useful for monitoring events for which you don't need a trace. An excellent way to see whether the program does what it should. This command will reveal whether a routine messes up spectacularly or performs flawlessly.

Examples

```
NORMT  123 ADR  EVENTS?
```
Counts the occurrences of address 123.

```
NORMT  123 ADR FF DATA  EVENTS?
```
Counts the occurrences of cycles containing data value FF and address 123.

```
NORMT  WRITE  78 TO FF DATA  1210 ADR   EVENTS?
```
Counts how often a data value greater than 78 is written to location 1210.

Comments

You can also count such things as error conditions or system usage.

Use this command to sync a scope on the UniLab's test-point output.

FETCH

no parameters

Tells the UniLab to look for the trigger event only during fetch cycles.

Usage

To search for a particular opcode. When you use this command in a trigger spec, the UniLab will not look for the trigger during read or write cycles. Similar commands are used to recognize read and write cycles.

This command is not available for all processors; check your *Target Application Notes*.

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 – 7FF range.

Comments

This command, included with most processor-specific DDB packages, specifies a range of `CONT` values corresponding to fetch cycles.

FILTER

`<filter_type> FILTER` *Rarely Used*

Selects the type of trace filter, according to the preceding command (i.e., CONTROL, HDAT, MISC', or NO).

Why You Don't Need to Bother—For most trace filtering, you will use the commands ONLY or xAFTER, which automatically select the MISC' filter. The NORMx words turn off filtering.

Use this command to set a filter spec that is different from your trigger spec. This is sometimes very useful.

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.

FIND-ADR

`<16-bit_value> FIND-ADR` *See:* `FIND-xxxx`

FIND-CONT

`<8-bit_value> FIND-CONT` *See:* `FIND-xxxx`

FIND-DATA

`<8-bit_value> FIND-DATA` *See:* `FIND-xxxx`

FIND-HADR

`<8-bit_value> FIND-HADR` *See:* `FIND-xxxx`

FIND-HDATA

`<8-bit_value> FIND-HDATA` *See:* `FIND-xxxx`

FIND-LADR

`<8-bit_value> FIND-LADR` *See:* `FIND-xxxx`

FIND-MISC

`<8-bit_value> FIND-MISC` *See:* `FIND-xxxx`

FIND-WDATA

`<16-bit_value> FIND-WDATA` *See:* `FIND-xxxx`

FIND-xxxx

<value> FIND-xxxx

The FIND- family of words will search the trace buffer for a particular value.

Usage

Used to search the trace buffer *from the start* for an address, data, MISC, or CONT value. After finding the first such occurrence, NX will find the *next* match.

The FIND-xxxx family of words:

FIND-ADR
FIND-WDATA
FIND-MISC
FIND-LADR
FIND-DATA
FIND-CONT
FIND-HADR
FIND-HDATA

The FIND-xxxx words search for 8-bit values, except FIND-ADR and FIND-WDATA, which search for 16-bit values.

Example

3289 FIND-ADR
Finds and displays the cycle containing address 3289 in the current trace buffer.

FKEY

```
<key#> FKEY  <command>
```

Assigns a command to a function key. See also ALT-FKEY, CTRL-FKEY, and EDIT_MACROS.

Usage

Reassigns the function keys on IBM PCs and compatibles. Alt-F1 (see the FKEY? command) shows the current function-key assignments.

The function keys allow you to execute a single command or macro with a single keystroke. The default assignments represent most common functions, but you can change them to suit your particular needs and working habits.

Use key number A (hex) to assign a command to F10. To make your reassignments permanent, use the "save conFiguration" option of the Configuration PopUp.

Example

```
2 FKEY STARTUP
```

Assigns the command STARTUP to the F9 key.

Comments

To execute a string of commands in this manner, perhaps even including parameters, define a macro (with :) and assign the macro's name to a function key. For example,

```
: DUMP100    0 100 MDUMP   ;

6 FKEY DUMP100
```

allows you to dump locations 0 – 100 by pressing F6.

You must be in the macro-level system to define a macro with the : (colon) and ; (semi-colon). Alternatively, you could enter the same command as an Easy Macro and execute it with one of the shifted function keys.

FKEY?

no parameters *Alt-F3*

Displays the current function key assignments.

Usage

Used to see the command or macro name that each function key will execute when pressed.

See FKEY, ALT-FKEY, CTRL-FKEY, and EDIT_MACROS to reassign keys.

Example

```
FKEY?
```

This command is never used in combination with anything else.

G

`<addr> G`

Goes to the indicated address. Releases the microprocessor from the DEBUG breakpoint and lets it continue executing code from <address>. This is equivalent to changing the program counter.

Usage

Used at a DEBUG breakpoint, when you want to change the program counter and release the target from the breakpoint. G is one of several ways to do this.

G sets the program counter to addr and lets the target microprocessor resume executing code there. (Reset was turned off automatically when the DEBUG breakpoint was established.) After that, you can enter a trigger spec and restart the analyzer, or you can use one of the "big picture" words: ADR?, SAMP, and NOW?.

You could, instead, use STARTUP to restart the analyzer and the board at the same time. Or use NORMx followed by a trigger specification and S to restart the analyzer and give a trace of the event you describe.

Example

```
1030 G
```

Exits from a DEBUG breakpoint, and resumes execution of the target program at location 1030.

Comments

Appropriate if you have a DEBUG package and have run to a breakpoint by entering RESET <addr> RB or STEP-INTO. You can let the microprocessor resume at any point in the program; the DEBUG breakpoint will be released.

Use GB if you wish to resume the program at an address different from the one you are stopped at, but with a new breakpoint set.

GB

```
<go_to_addr> <breakpoint_addr> GB
```

Goes to the first address and starts executing code, with a breakpoint set at the second address.

Usage

When you want the microprocessor to resume the program at a particular address and run to a new breakpoint.

Example

```
1200 330 GB
```

Resumes the program at address 1200, with a breakpoint set at 330.

Comments

Available if you have a DEBUG package and are stopped at a breakpoint. See `RB` or `STEP-INTO` for ways to set breakpoints.

GW

```
<addr> GW
```

Goes to the indicated address and waits until the analyzer is started. Releases the target board from the DEBUG breakpoint.

Usage

Used to continue executing the program, starting at the given address, after a new trigger spec has been defined. A specialized but very useful command.

Example

```
1100 GW  NORMT 1200 ADR S
```

Sets the microprocessor's program counter to 1100 and waits for the analyzer to be started. The trigger spec sets the analyzer to capture a trace at address 1200. Then the analyzer is started with the `S` command and the microprocessor is released from the breakpoint. The analyzer will trigger when address 1200 shows up on the bus.

Why This is Useful—This command utilizes a combination of DEBUG breakpoints and the UniLab analyzer. If you were to use `RZ` to release the microprocessor from the DEBUG breakpoint, address 1200 might be executed before you could type the trigger spec. Be sure to use one of the `NORMx` words to clear any trigger spec DEBUG may have set.

Note that reset is always automatically turned off after any DEBUG operation.

HADR

Rarely Used

```
         <byte> HADR
  <byte> TO <byte> HADR
<byte> MASK <byte> HADR
```

Sets the trigger value for the high byte (A8 – A15) of a 16-bit address.

Usage

You can use this trigger-spec command in the same way as DATA, CONT, etc. However, it is most frequently used when setting an address trigger that uses many calls to ALSO.

Address Inputs—You will normally use ADR to set 16 or 20 bits at once, but there are limits to the use of ALSO in combination with ADR (see ADR). You can change the trigger spec of the low-order address byte with LADR, of the second byte with the HADR command, and of the high four bytes (for 20-bit addresses) with CONT or ASEG.

Example

```
NORMT 12 HADR  ALSO 34 LADR  ALSO 10 LADR  ALSO 5 LADR
```

Sets the analyzer to trigger on any of the addresses 1234, 1210 or 1205.

Comments

Makes it possible to treat the first two bytes of an address separateiy. LADR is the lower byte, HADR the higher. The UniLab works with up to 20-bit addresses.

HARDWARE

no parameters *F8*

Enables installation of the non-maskable interrupt vector.

Usage

This command re-enables the UniLab's use of the NMI (or IRQ) pin and vector to execute `STEP-INTO`. It also allows you to get an "instant breakpoint" by toggling the NMI pin of the target microprocessor. If InSight is implemented in your DDB package, this will be automatically enabled when you enter InSight.

Only disable this feature for more transparent operation, when you don't need all the DEBUG features. See also `SOFTWARE`. (See `DEBUG'` for complete transparency.)

Normally, you will use the mode panels (`F8`) to change this feature.

HDAT

`HDAT FILTER` *Rarely Used*

Used before `FILTER` to set a filter spec based only on the high byte of the DATA inputs (D8 – D15).

Usage

You will probably never use this command. It triggers on the full specification, but filters based only on the eight bits D8 – D15.

While deciding whether to include the current cycle in a filtered trace, the UniLab will check only these eight bits of the 48 inputs. This is a good way to see all bus cycles that have a certain value in the upper data byte.

The filter mechanism of the UniLab gets turned on by `ONLY` and `xAFTER`. Those commands set the filter to `MISC' FILTER`, which allows you to define a trigger based on all inputs except the HDATA and MISC wires.

See also `CONTROL`, `MISC'`, and `NO`.

High Data Inputs

On 16-bit processors, these lines represent the high byte of the 16-bit data path. On 8-bit processors, the lines can float or be connected to other signals on the target board.

Example

```
NORMT   HDAT FILTER  80 TO FF HDATA
3410 ADR   A7 DEVENTS  S
```

This captures a trace showing only cycles with D15 high, beginning with the first bus cycle that has D15 high and address 3410. `DEVENTS` tells the analyzer how many filter-matching events to record (up to the total size of the trace buffer).

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 – D15). On eight-bit processors, these are spare inputs.

Usage

The simplest use sets 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 by looking at just the high-order byte, the UniLab doesn't care about the low-order byte—*so it actually searches for a range.* (See the first example below.) To specify a single, 16-bit wide data value, you must use both HDATA and DATA.

The Data Inputs—The UniLab gets both address and data during each bus cycle. The "data" that appears on the bus may be either a value or a machine code instruction. On eight-bit processors, inputs D8 – D15 can be connected to anything you like.

Ranges of Data—TO lets you set a trigger on any data in a range 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 <k> MASK <m> HDATA to examine any subset of the eight most significant data lines. The high bits of <k> tell the analyzer which bits to examine, while the bit configuration of byte <m> indicates what values the lines must have for a trigger to occur.

For example, 01 MASK FF HDATA selects only the least significant data bit for examination (with binary value 0000 0001), and tells the UniLab to trigger when that bit has a high value. (The instruction 01 MASK 01 HDATA would have the same effect.)

Examples

```
NORMT 12 HDATA S
```

Clears all previous trigger settings with NORMT, sets a trigger for data input 12xx (actually 1200 – 12FF), then uses S to start the analyzer.

```
12 HDATA 80 DATA
```

Sets a trigger on the 16-bit data value 1280.

```
12 TO 34 HDATA
```

Sets a trigger on data values from 12xx and 34xx hex (i.e., 1200 – 34FF).

```
F0 MASK 00 HDATA
```

Sets a trigger based on the four highest bits of data, and looks for a zero on those lines.

```
12 TO 23 HDATA ALSO 45 HDATA
```

Sets a trigger on cycles in which the high data byte is from 12 – 23, or is 45.

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, 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 extra MISC inputs, typically to look at target I/O.

HELP

```
HELP <command>                                        F1
HELP
```

Finds the reference information for a command. With no word following, displays a screen about On-Line Help. Press `Ctrl-F1` for additional help by category, `Alt-F1` for help with processor-specific functions. `Shift-F1` displays the current Easy Macros.

Usage

Displays information about a command from the On-Line Help version of the glossary of commands. (See also `WORDS`.)

Examples

```
HELP
```

Displays the main help screen.

```
HELP BYE
```

Provides information about the `BYE` command.

HELP+

no parameters `Ctrl-F1`

Provides On-Line Help, by category, about a variety of UniLab functions.

HELP_DEBUG

no parameters *Alt-F1*

Offers On-line Help for DEBUG operations and processor-specific features.

HEXLOAD

`HEXLOAD <filename>`

Loads an Intel hex format object file into the UniLab's emulation memory. Prompts for the filename if you don't include it. Or, use the Files PopUp.

Usage

Loads into emulation memory a program that was stored in Intel hex format. You can then run, debug, and alter that program like any other.

Binary format files are more compact, and load two to three times faster. You may want your assembler to produce binary format files, if it can. Or, after loading your program with HEXLOAD, save it with BINSAVE to create a new binary file for future use. Binary format files are loaded with BINLOAD or the Files PopUp.

Intel hex format files contain information about where each opcode should be stored. Be sure the proper sections of emulation memory are enabled before loading the file. See EMENABLE.

Loading Into RAM

The UniLab will not load a file into RAM unless you have stopped the processor at a DEBUG breakpoint. A program (LTARG, for example) must already have been loaded into emulation memory and run to a breakpoint (e.g., with RESET <address> RB).

If your target processor is not at a DEBUG breakpoint, attempts to load into memory that is not enabled will generate an "auto-breakpoint." You will see
"-step-into-" and "Accessing target ram or I/O" messages.

Example

```
HEXLOAD MYPROG.HEX
```

Loads an Intel hex format file called MYPROG.HEX.

Comments

Only record types 0 – 3 are supported. Bytes 7 and 8 of each line in the file tell what record type that line uses. See also the appendix "Object- and Symbol-File Formats."

16-bit processor note: If the UniLab detects a type 2 record (extended address), then address bits A16 – A19 will be compared to the current =EMSEG; data won't be loaded if it is intended for a segment other than the current one.

Operator Shortcuts

You can load hex files via the Files PopUp.

HLOAD

`HLOAD <filename>` *PPA*

Loads from a disk file the setup data for a histogram. This is only used with the optional Program Performance Analyzer.

Usage

Loads into memory a PPA template—or an entire run you saved previously—then invokes the correct histogram: `AHIST`, `MHIST`, or `THIST`.

Example

```
HLOAD AUG28.HST
```

Loads into memory the information in the file AUG28.HST that had been saved with `HSAVE`.

Operator Shortcuts

You can load histogram files via the Files PopUp or from the PPA.

HSAVE

`HSAVE <filename>` *PPA*

Saves to a file the data describing a histogram. This is only used with the optional Program Performance Analyzer, after exiting from THIST, MHIST, or AHIST. Or use the Files PopUp.

Usage
This is handy when you periodically run 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, including the collected data, for later review or comparison.

Load the information back into memory with HLOAD or use the Files PopUp.

Example

```
HSAVE AUG28.HST
```

Saves as a file the data that describes the last histogram you saw before exiting from AHIST, THIST, or MHIST.

Operator Shortcuts
You can save histograms via the PPA menu or the Files PopUp.

INFINITE

`INFINITE PEVENTS` *Rarely Used*

Used only before `PEVENTS`, instead of a count, to indicate that the trigger event must immediately follow the qualifiers.

Usage

Used with a trigger specification (see `ADR`, `DATA`, `READ`, `WRITE`, etc.) and a qualifier specification (see `AFTER` and `QUALIFIERS`), when you are only interested in the trigger event if it occurs during the bus cycle that immediately follows the qualifying cycle(s).

Background

By default, the UniLab searches just for the first occurrence of the qualifying sequence. After the sequence has been found once, the UniLab proceeds to look for the trigger.

With `<count> PEVENTS`, the UniLab searches for the qualifying events until it finds them `<count>` number of times; then it looks for the trigger.

What It Really Does—`INFINITE` causes the UniLab to search for the qualifying sequence and, when it finds it, to see if the very next cycle matches the trigger spec. If not, the UniLab starts looking for the *qualifiers* again.

Example

```
123 ADR  AFTER 345 ADR  INFINITE PEVENTS
```

Triggers if address 123 immediately follows address 345.

INIT

no parameters

Sends an initialization message to the UniLab.

Usage

If the UniLab was not properly connected when you executed the software, or if you turned off the UniLab during use, it must be re-initialized. This command re-initializes the UniLab without the need to return to DOS.

When you start the software, it initializes the instrument just after the screen clears and the UniLab version number is displayed. If you tap any key right after the screen clears, the automatic initialization won't occur. You will then have to type `INIT` before you can send commands to the instrument.

Example

```
INIT
```

This command is never used in combination with anything else.

Comments

Initializes the mode bits, baud rate, and emulation enable map. Executed automatically after PROM operations to re-initialize the analyzer modes.

INSIGHT

no parameters *Ctrl-F3*

Provides access to the Insight display/control panel.

Usage

Displays the InSight panel, ready to poll the processor at any address. See also `RESIGHT` to re-enter InSight with the prior InSight trigger intact.

Example

```
INSIGHT
```

This command is never used in combination with anything else.

INT

no parameters *Rarely Used*

Generates a low-going signal on the NMI- wire when the current qualifiers have been satisfied. The signal goes high again when the trigger is satisfied.

Usage

Useful for causing the target to execute an interrupt routine when the analyzer goes into trigger-search state (i.e., after the qualifier sequence has been found). Can also be used to prevent damage to equipment by branching control to a "soft shutdown" routine when some error condition occurs. (You must, of course, write and install your own shutdown routine.)

Orion DEBUG packages use this command internally. If you want to use it, disable the `STEP-INTO` feature with the mode panel (`F8`) or with `SOFTWARE`.

`NORMx` turns off `INT`.

Example

```
NORMT  INT  AFTER 123 ADR  S
```

Interrupts the target processor during the bus cycle after address 123 is reached, then triggers immediately.

```
NORMT  INT  12 DATA  AFTER 345 ADR  S
```

Interrupts during the bus cycle after address 345 occurs, then will trigger when 12 data occurs.

Comments

The interrupt occurs when the qualifying sequence is complete, not when the trigger event is detected. This makes it possible to trigger on something specific after the interrupt occurs.

INT'

no parameters

Rarely Used

Disables the INT mode. See INT.

Usage

Not often used, since NORMx also disables the INT mode. Only used to reissue the current trigger specification without generating a hardware interrupt.

IS

```
<value>  IS  <name>
```

Assigns a symbolic name to an address or data value.

Usage

Used to define mnemonic names for numerical information, which then can be used as command parameters and will appear in traces and disassemblies. If you already have an assembler-generated symbol table, use the symbol table features of the UniLab to load it, and just use this command to create individual labels 'on the fly,' as the need arises. (See the Files PopUp and LOADER.)

IS turns on symbol display mode.

Example

```
1234 IS MREGISTER
```

Assigns to 1234 the symbolic name MREGISTER—wherever that value occurs, it will be displayed by this label as long as the symbol display is active.

Comments

See also ISMODULE, ISOFFSET, ISSEG, SYMB, SYMB', SYMSAVE, CLRSYM and the Files PopUp.

ISMODULE

```
<module#> ISMODULE <new_name>
```

Assigns a new name to a module by its symbol-table number.

Usage

Changes the name of the file referred to by symbol table line-number. Use this command when the symbol table has assigned an inaccurate file extension to a filename.

Example

```
1 ISMODULE OUT.C
```

Assigns the name OUT.C to module #1 in the UniLab's symbol table.

```
3 ISMODULE PROM0000
```

Assigns the name PROM0000 to module #3.

Comments

See also `IS`, `ISSEG`, `SYMB`, `SYMB'`, `SYMSAVE`, `CLRSYM`, `SYMLOAD` and the Files PopUp.

ISOFFSET

`<16-bit> ISOFFSET <name>` *8088/86 DDB*

Assigns a symbolic name to an offset from the current segment. For use with the 8088/86 family DDB only.

Usage

To make mnemonic names for memory locations, which then can be used with commands and which will show in traces and disassemblies. If you already have an assembler-generated symbol table, use the symbol table features of the UniLab to import it—see `LOADER` and the Files PopUp (`F5`).

`ISOFFSET` creates a segmented symbol whose value depends on both the offset and the current segment. `ISOFFSET` defaults to the value of `CS:`, but you can override it; see the example below.

`ISOFFSET` will create an unnamed segment symbol if a segment symbol with the correct value does not yet exist.

`ISOFFSET` turns on symbol display mode.

For more information on `ISOFFSET` and the 8088/86 address space, refer to Orion's *Engineering Technical Note #24* (ETN-24).

Example

```
1234 ISOFFSET CODE-ADR
```

Gives the name `CODE-ADR` to 1234 offset from `CS:`.

```
9988: 5678 ISOFFSET MYSYMB
```

Assigns the name `MYSYMB` to 5678 offset from 9988.

Comments

See also `IS`, `ISMODULE`, `ISSEG`, `SYMB`, `SYMB'`, `SYMSAVE`, `CLRSYM`.

ISSEG

`<value> ISSEG <name>` *8088/86 DDB*

Assigns a symbolic name to a segment value. For use with 8088/86 family DDB only.

Usage

Used to assign a mnemonic name to a segment, for use with the UniLab's memory operations. If you already have an assembler-generated symbol table, you will prefer the symbol table features of the UniLab; see LOADER and the Files PopUp.

You can use ISSEG to add symbols before or after you have loaded a symbol table.

Use this to create a named offset symbol before using ISOFFSET, since ISOFFSET will create an unnamed segment symbol if a symbol with the correct value does not yet exist.

When you enter the name of a segment symbol, it overrides the default segment value of the next memory operation. For more about ISSEG and the 8088/86 address space, refer to Orion's *Engineering Technical Note #24* (ETN-24).

Example

```
9988 ISSEG MYSEG
```

Assigns the name MYSEG to segment value 9988.

Comments

See also IS, ISMODULE, ISOFFSET, SYMB, SYMB', SYMSAVE, CLRSYM, SYMLOAD.

LADR

```
             <byte> LADR
  <byte> TO <byte> LADR
<byte> MASK <byte> LADR
```

Rarely Used

Sets the trigger value for the low-order byte of the address (A0 – A7).

Usage

You can use this trigger-spec command in the same way as DATA, CONT, etc. However, it is most frequently used when setting an address trigger that uses many calls to ALSO.

LADR is also useful for setting a trigger on a Z80 port address. The ports of that processor have 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.

Address Inputs—You will normally use ADR to set 16 or 20 bits at once, but there are limits to the use of ALSO in combination with ADR (see ADR). You can change the trigger spec of the low-order address byte with LADR, of the second byte with the HADR command, and of the high four bytes (for 20-bit addresses) with CONT or ASEG.

Example

```
NORMT 12 HADR  ALSO 34 LADR  ALSO 10 LADR  ALSO 5 LADR
```

Sets 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.)

LOADER

Not a UniLab command—execute from DOS.

Loads object code files and mixed symbol-and-object files.

Usage

LOADER is a standalone application for processing OMF and other files; it produces symbol and binary files which can be loaded by the UniLab. (See SYMLOAD, BINLOAD, and HEXLOAD, or use the Files PopUp.) This utility can also process fixed-format symbol records that you describe to the loader; it usually makes two passes through such a file, informing you of the status.

Invoking the Loader—Just type LOADER at the DOS prompt to display the menu shown below. If you include all the necessary command-line arguments (see following) when you first invoke the loader, it exits to DOS after successful completion. It can be very efficient to use LOADER in a batch file with your compiler/assembler/linker, to translate the output files after each new pass, and then start the UniLab.

Formats processed by LOADER:

```
 F1)  OMF51    Intel OMF for the 8051 family.          (l o s)
 F2)  OMF86    Intel OMF for the 88/86 family.         (l o s)
 F3)  OMF80    Intel OMF for the 8080 family.          (l o s)
 F4)  TEKHEX   Tektronix Hex.                          (o)
 F5)  MOTORS   Motorola S records.                     (o)
 F6)  SYMFIX   User-defined fixed-length format.       (s)
 F7)  2500AD   Linker option M (Microtek).             (s)
 F8)  MANX     ASCII file—value, then name.            (s)
 F9)  AVOCET   ASCII file—name, then value.            (s)
F10)  SDSI     Software Development Systems Inc.       (l s)
End   EXIT from LOADER
Esc   Go to LOADER command mode              (l) = line numbers
                                             (s) = symbols
                                             (o) = object code
```

Explanation

Press the key that corresponds to the format of the file you want to load—you will be prompted for its filename.

Command-Line Options & LOADER

Several options allow you to control the loader. These ‘switches’ can follow `LOADER` on the command line or in a batch file; or they can be used after invoking the loader menu, by entering its command mode (`F9`). Command mode also allows you to review a record of the loading process (see below). Which *load command* you use determines what kind of file the loader will process—this is always the last command-line option:

```
LOADER first_opt ... last_opt load_command filename
```

Command-line options:

`-D`	Eliminates symbols with same value; saves space and speeds file translation.
`-E`	Reports all errors and warnings. (Usually only fatal errors cause the loader to terminate.)
`-O filename`	Changes output filename. (Defaults are ORION.BIN and ORION.SYM.)
`-U`	Suppresses upper-case conversion. Permits case-sensitive symbols, *but* all commands must then be in upper case.[1]
`-W`	Waits for a keypress before exiting `LOADER`; mostly used in batch files to see file-size messages before exiting from the loader.

Load commands (descriptions on previous page):

```
OMF51      OMF86       OMF80
TEKHEX     MOTORS      SYMFIX
2500AD     MANX        AVOCET
SDSI
```

Example

```
C> LOADER  -O VERS02  -D  OMF51  SIEVE.A51
```

Processes the file SIEVE.A51, creating output files VERS02.BIN and VERS02.SYM, and eliminating duplicate symbols.

[1] *To make the UniLab case-sensitive as well, go to* `MACRO` *mode, type* `L/U? OFF` *and enter all subsequent UniLab commands in upper case.*

Review Loading Process—To review a record of the loading process, go to the loader's command mode and press F5 repeatedly. This moves back through a record of actions taken by the loader. Move forward again with F6.

Loader Output—If the file you are processing contains symbols, the loader will create a symbol file named ORION.SYM. (Use the -O option to specify a base filename other than ORION.)

If the file is of a format that can only address a single 64K segment (e.g., OMF51 and TEKHEX), the object code will be named ORION.BIN. (Again, use the -O option to specify a base filename other than ORION.)

If the format can address multiple 64K segments, a separate binary file is created for each 64K segment addressed by the application. Each of these files has a name in the form xORION.BIN, where x is the hex value of the file's destination =EMSEG. For example, FORION.BIN should be loaded into F =EMSEG.

Load the New Files
After the loader creates the output files, it tells you where in memory to load each of them. Start the UniLab software and enable memory in those segment(s). Then use BINLOAD and SYMLOAD, or the Files PopUp, to load the binary file(s) and the symbol file. (The binary file—but not the symbol file—can be loaded into the UDL as well.)

Supported File Formats

LOADER directly supports most widely used file formats. Details of the formats themselves are discussed in the appendix "Object- and Symbol-File Formats." They include:

- absolute Object Module Format (OMF) for the 8051 family
- OMF for the 8086 family
- OMF for the 8080 family
- Tektronix Hex (but not, yet, extended TekHex)
- Motorola S records
- 2500AD symbol files (linker option M—Microtek)
- MANX symbol files (and others with one symbol per line, consisting of a four-digit hex value followed by the name)
- AVOCET symbol files (and others with one symbol per line consisting of the name followed by a four hex-digit value)
- Software Development Systems, Inc. files (which usually have a .OUT extension)
- Other fixed-record-length files are supported via SYMFIX, described in On-Line Help and in the appendix.

Notes:

LP

no parameters

Executes a loop once and stops.

Usage

You must already have stopped the microprocessor at a DEBUG breakpoint (see `RB`) and be at an address that will be executed again (i.e., within a loop). This command allows the program to run around the loop once and up to the current address, displaying the registers when it stops.

Watch Out—Won't 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 Notes* for your processor about any additional restrictions.

Example

```
LP
```

This command is never used in combination with anything else.

Comments

`LP` works by saving the current breakpoint address, executing `STEP-OVER` (single-steps without following branches), and then `<saved_address> RB`. Processors with multiple-byte breakpoint opcodes will execute `STEP-OVER` 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. Packaged with the optional 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 as it executes this 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, it will not run without some patching.

The *Target Application Notes* for each DDB includes an `LTARG` sample session.

Example
```
LTARG
```
This command is never used in combination with anything else.

M

```
<byte>  M
```

Stores a byte in ROM or RAM, and increments the reference address.

Usage

A prior ORG statement must be used to set the initial reference address. If that address is in emulation ROM, M simply stores the specified byte there and increments the ORG value for further use by M or another memory-patching command.

M can change the contents of target RAM just as easily, if the microprocessor is at a breakpoint. But if the target program is still running when the M command is issued *and* the ORG address is not in emulation memory, first STEP-INTO gets an invisible breakpoint, then the memory operation is performed, and normal execution resumes. (See STEP-INTO for details; if it isn't enabled, this won't work in RAM.)

As with all memory-patching commands, be careful not to overwrite your stack area when writing into RAM.

Examples

```
3000  ORG  12  M
```

Stores a 12 at 3000.

```
150  ORG   5  M  10  M
```

Stores a 5 at location 150, and a 10 at 151,

Comments

Used for entering data tables, program patches, etc. (See also MM, MM!, and M!.)

M!

```
<byte> <addr>  M!
```

Stores a byte of data at the specified address.

Usage

Used to patch program memory. Similar to M but doesn't require a previous ORG statement—instead requires an address parameter. (See also M.)

M! can change the contents of target RAM just as easily, if the microprocessor is at a breakpoint. But if the target program is still running when the M! command is issued *and* the address is not in emulation memory, first STEP-INTO gets an invisible breakpoint, then the memory operation is performed, and normal execution resumes. (See STEP-INTO for details; if it isn't enabled, this operation won't work.)

As with all memory-patching commands, be careful not to overwrite your stack area when writing into RAM.

Examples

```
12 3000 M!
```

Stores a 12 at 3000.

```
5 150  M!    10 150  M!
```

Stores a 5 at location 150, and a 10 at 151.

Comments

Used for entering small patches—areas larger than one byte can be changed with fewer keystrokes by other memory commands. (See also MM, MM!, and M.)

M?

```
<addr>   M?
```

Displays the byte stored at the specified address.

Usage

Used to find out what is stored at a single memory location, in either ROM or RAM (see below). Use `MM?` for looking at words, and `MDUMP` or `DM` to see larger areas of memory.

Example

```
1210 M?
```

Displays the byte stored at 1210.

Comments

If the address is in emulation memory, its contents will be displayed; otherwise, the UniLab will attempt to use DEBUG features to display target RAM. (See `STEP-INTO` for details; if it isn't enabled, this won't work in RAM.)

MACRO

no parameters

Switches the UniLab software to a macro-level system.

Usage

Only necessary to write macros or for access to the "internal" words of the UniLab software. For information about macros, refer to the glossary entry for : (colon), and the appendix "Writing Macros." See also EDIT_MACROS for information about simple macros that do not require the macro system.

Several otherwise-unused files must be in the UniLab directory when you switch the software to the macro-level system. Included on your UniLab diskettes are files with an .OPR extension for the normal, operator-level system; and others with an .MCR extension for the macro system. You should have one .OPR and one .MCR file for every .EXE or .OVL file. When you switch to the macro system, the UniLab software will search for an .MCR file whose name matches that of the current .EXE file.

You can save the UniLab software in its macro-level state any time after using the MACRO command: SAVE-SYS (normally invoked via the "save configuration" option of the Configuration PopUp) will save an .EXE file and a matching .MCR file to the new filename you specify.

Example

MACRO
Converts the UniLab software to a macro system.

Comments

In the operator system (see OPERATOR and MAKE-OPERATOR), you have access only to the commands in the UniLab glossary and to the somewhat limited Easy Macros. However, these features suit many users admirably.

MACRO_2

no parameters *Shift-F2*
(See EDIT_MACROS.)

MACRO_3

no parameters *Shift-F3*
(See EDIT_MACROS.)

MACRO_4

no parameters *Shift-F4*
(See EDIT_MACROS.)

MACRO_5

no parameters *Shift-F5*
(See EDIT_MACROS.)

MACRO_6

no parameters *Shift-F6*
(See EDIT_MACROS.)

MACRO_7

no parameters *Shift-F7*
(See EDIT_MACROS.)

MACRO_8

no parameters *Shift-F8*
(See EDIT_MACROS.)

MACRO_9

no parameters *Shift-F9*
(See EDIT_MACROS.)

MAKE-OPERATOR

`MAKE-OPERATOR <filename>` *Macro System*

Use this command to create an operator-level system (i.e., restricted access to UniLab commands) with access to words you have defined in the MACRO system.

Usage

This command performs four actions:

1) saves the current macro system,
2) creates a new version of the operator system,
3) saves the new operator system, and
4) exits to DOS.

The standard (default) operator system only offers access to commands documented in the On-Line Help version of this glossary. A newly created operator system *also* recognizes any macro-level commands you defined, but still restricts access to the advanced features of the macro system.

Filenames—To make an operator system with this command, your UniLab directory must contain the MAKE file from the Orion Utilities diskette.

Example

```
MAKE-OPERATOR TESTER
```

Saves a macro system with the filename TESTER.MCR, then creates an operator system and saves that as TESTER.EXE, with the associated file TESTER.OPR.

MAPSYM

```
MAPSYM <filename>
```

Reads from a .MAP file the information the UniLab needs to support high-level source code. Clears the current symbol table before loading the information. See also MAPSYM+.

Usage

Makes it possible to display high-level language source files in the trace. After you issue this command, each line of your source code will be displayed just before the instructions generated by that line. (The source files must be in the current directory, or they will not be found.)

You do not need to use this command if your linker puts out an OMF51 or OMF86 file, or if you are using the SDSI 68000 or Z80 C compiler. Output files from those linkers contain all the line number information to display high-level language source files. MAPSYM is of limited use and is only needed if you have another file format and want to insert information from your text file into the trace display or disassembly listings.

Use SYMLIST to view the contents of the symbol table, save it to disk with SYMSAVE, and reload it later with SYMLOAD. You can turn off the display of high-level source files with SOURCE'.

Also see the appendix "Object- and Symbol-File Formats."

.MAP File Formats

You can use a Microsoft- or ORION-format .MAP file. The Microsoft .MAP file contains a mixture of symbol and line-number data. The Orion .MAP file is simply an ASCII file containing an unlimited number of file records, one for each source file, which makes it easier to generate a .MAP file.

In Orion-format .MAP files, the first line of every file record contains the keyword SOURCE followed by the source code's filename. Each remaining line in the file record consists of a line number from the source code and the absolute 16-bit address of the code generated by that 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).

Sample Orion .MAP File

The following is a simple example of an Orion format .MAP file. This file describes the relationship between the source files and the 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.)

Sample Orion .MAP File:

```
SOURCE SIMPLE1.C
2 0034
5 0040
6 0050
<blank line>
SOURCE SIMPLE2.C
3 0055
5 0070
<blank line>
<blank line>
```

Example

`MAPSYM TEST.MAP`

Loads into the symbol table the information in .MAP file TEST.MAP. The source files themselves are not opened until needed, e.g., while looking at a trace display or at a disassembly from memory.

MAPSYM+

```
MAPSYM+ <filename>
```

Like MAPSYM, but doesn't clear the symbol table before loading the .MAP file.

MASK

`<byte1> MASK <byte2>`

Specifies a mask for the trigger specification that immediately follows.

Usage

A modifier used with ADR, CONT, DATA, HADR, HDATA, LADR, and MISC.

The first byte describes which of the eight lines to watch (a bit set to one means pay attention, a zero means ignore that wire). The second byte tells the UniLab what value to look for on those lines.

The UniLab ignores inputs that the first byte tells it to ignore. Thus, 01 MASK 01 has the same effect as 01 MASK FF.

Example

```
NORMT 2 MASK 2 MISC S
```

Triggers if input M1 goes high.

```
NORMT B# 0010 MASK B# 0010 MISC S
```

The same effect as the first example—triggers if input M1 goes high.

```
NORMT 3 MASK 2 MISC S
```

Requires inputs M1 = 1 and M0 = 0 in order to trigger.

Comments

MASK cannot be used with TO, NOT, or ALSO.

MCOMP

```
<start_addr> <end_addr> <comp_addr> MCOMP
```

Compares two areas of memory and indicates differences.

Usage

Compares the two areas of memory, and gives a message about each discrepancy. Press any key to abort.

Example Report

```
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 EPROM PopUp to read the contents of 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 emulated ROM or target RAM. (See STEP-INTO and RB.)

MDUMP

```
<from_addr> <to_addr>  MDUMP
```

Displays the contents of an area of memory. (See also MODIFY.)

Usage

Allows you to look at any size block of memory. Press any key to freeze the scrolling of the display. Press any key again to continue. While scrolling is stopped, press any key twice quickly to stop.

Example

```
1234 1334 MDUMP
```

Displays the contents of locations 1234 – 1334 in hex and ASCII.

Comments

As with all M commands, this displays the contents of emulation memory if the address has been EMENABLEd, otherwise DEBUG features will display the contents of target RAM. (See STEP-INTO and RB.)

MEMO

no parameters *Alt-F2*

Displays, and allows editing of, the on-line memo pad.

Usage

A handy way to keep notes to yourself, and to store macros or useful series of commands.

Helpful prompts are always displayed. Press `F1` for more help. Exit the full-screen editor with `Esc` or `End`. `End` will save your work, `Esc` will allow you to exit without saving your changes to the screen.

Example

```
MEMO
```

This command is never used in combination with anything else.

Comments

This works only when EDITxx.VIR and MEMO.SCR from the Orion Utilities diskette are in the same disk directory as the UniLab program.

See the appendix “Writing Macros” for help with the editor.

MESSAGE

no parameters

Gives a screenful of information about recent updates and additions to the UniLab software.

Usage

Helps you to be sure you know the latest capabilities of the UniLab.

MFILL

```
<from_addr> <to_addr> <byte>  MFILL
```

Fills every location in an area of memory with the same byte.

Usage

A good way to see if the 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 won't work: starting at the reset address, fill a block of memory with NOP instructions and then use STARTUP. You should see a trace of consecutive addresses. This is a heavy-handed way to poke a single byte into memory: see MM, M, MM!, and M! for more elegant ways to do this.)

Like other memory-altering commands, if you use this to access RAM and you aren't stopped at a breakpoint, STEP-INTO is automatically invoked to first get an invisible breakpoint. (See STEP-INTO for details.)

Example

```
1200 1300 20 MFILL
```

Fills locations 1200 – 1300 with the value 20 hex.

Comments

As with all memory-writing commands, be careful not to write into your stack area when loading into RAM.

MHIST

no parameters *PPA*

MHIST, the multiple-pass, address-domain histogram, invokes the optional Program Performance Analyzer (PPA) in the mode which displays the execution time of the code in up to 15 user-specified address ranges. See also THIST and AHIST.

Usage

Allows you to examine the performance of your software, to see where your program spends most of its time. Press Esc to exit from this menu-driven feature.

Before you use the PPA the first time, you must issue the command SOFT to install it.[1] SOFT performs a SAVE-SYS, then causes an exit to DOS. The next time you boot the UniLab software, the PPA will be installed.

Start the Histogram—To produce a histogram, first specify the upper and lower limits of each address "bin" you want to display, then start the histogram.

When you issue the command MHIST, you get the chart screen with the cursor positioned on the first bin. You can then type a lower and upper limit for each bin. Press return, tab, or an arrow key to move to the next field. Press F1 or Alt-F1 to start the histogram.

Save to a File—You can save a histogram's setup (i.e., bin limits, title, and any labels) in a file, along with any collected data, after you exit from the histogram screen. Just type HSAVE <filename> at the Command> prompt or use the Files PopUp.

Load From a File—You can load a previously saved histogram by typing HLOAD <filename> at the Command> prompt, or via the Files PopUp. Loading a histogram file also invokes the PPA in the correct mode.

Example

```
MHIST
```

This command is never used in combination with anything else.

Operator Shortcuts

This command is available on the Orion PopUp and the PPA menu, which can be displayed by pressing Alt-F10.

[1] *Only the first time you use the PPA.*

MISC

```
              <byte> MISC
  <byte> TO <byte> MISC
<byte> MASK <byte> MISC
```

Changes the analyzer trigger for the miscellaneous inputs.

The MISC Inputs
The UniLab's 48-bit-wide trace buffer has room for eight more bits than are used for the data, address, and control lines. These eight input lines are available 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 could hook them to an output port, to trigger when a particular bit configuration gets asserted on that port.

Qualifier and filter specifications always ignore the MISC inputs.

Usage
The simplest use sets a trigger on a single value on miscellaneous inputs. The UniLab will search for the byte value and will trigger when it sees that hex number on the lines. (See the first example below.)

Ranges—`TO` lets you set a trigger on any input between two byte values, inclusive. (See the second example below.)

NOT—`NOT` causes the UniLab to trigger when a value is detected outside the specified range or value.

Masking—You can use `<k> MASK <m> MISC` to examine any subset of the eight 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 six lines, since they don't mean anything.

The high bits of <k> mark the bits to be examined, while the bit configuration of <m> indicates what values the lines must have for a trigger to occur.

For example, `03 MASK FF MISC` selects only bits M0 and M1 for examination (with binary value 0000 0011). The UniLab will trigger when both these bits have a high value. The instruction `03 MASK 03 MISC` would have the same effect.

With Filtering—All trace-filtering methods 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`

Clears all previous settings with `NORMT`, sets a trigger for miscellaneous input 12, and uses `S` to start the analyzer.

`12 TO 34 MISC`

Specifies a miscellaneous input value in the range 12 – 34 hex.

`F0 MASK 00 MISC`

Sets a trigger based on the four highest bits. The UniLab will trigger when it finds zero 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, and are often used to look at target I/O.

MISC'

MISC' FILTER *Rarely Used*

Used before FILTER to enable filtering on all inputs except the MISCellaneous (M0 – M7) and HDATA inputs (D8 – D15).

Why You Don't Need to Bother

This is taken care of by ONLY and by xAFTER, so it is unlikely that you will need to use this command directly. (See also CONTROL, HDAT, and NO.)

NORMx turns filtering off.

Example

```
MISC' FILTER
```

Enables filtering on all inputs except M0 – M7 and D8 – D15.

MM

`<word> MM`

Stores a 16-bit word into ROM or RAM, and increments the reference address.

Usage

A prior ORG statement must be used to set the initial reference address. If that address is in emulation ROM, MM simply stores the specified word there and increments the ORG value for further use by MM or another memory-patching command.

MM can change the contents of target RAM just as easily, if the microprocessor is at a breakpoint. But if the target program is still running when the MM command is issued *and* the ORG address is not in emulation memory, first STEP-INTO gets an invisible breakpoint, then the memory operation is performed, and normal execution resumes. (See STEP-INTO for details; if it isn't enabled, this won't work in RAM.)

As with all memory-patching commands, be careful not to overwrite your stack area when writing into RAM.

Examples

```
3000  ORG  1210 MM
```

Stores 1210 at 3000.

```
150  ORG   5000  MM  7001 MM
```

Stores 5000 at location 150 and 7001 at 152.

Comments

Useful for entering data tables, program patches, etc. See also M, MM!, and M!. See ASM for information about the line-by-line assembler.

If you have a disassembler, the byte order is set correctly; otherwise, you can set it with HL or LH.

MM!

```
<word> <addr>  MM!
```

Stores a 16-bit word of data at the specified address.

Usage

Used to patch program memory. Similar to MM but doesn't require a previous ORG statement—instead requires an address parameter. (See also M!.)

MM! can change the contents of target RAM just as easily, if the microprocessor is at a breakpoint. But if the target program is still running when the MM! command is issued *and* the address is not in emulation memory, first STEP-INTO gets an invisible breakpoint, then the memory operation is performed, and normal execution resumes. (See STEP-INTO for details; if it isn't enabled, this operation won't work.)

As with all memory-patching commands, be careful not to overwrite your stack area when writing into RAM.

Examples

```
1200 3000 MM!
```

Stores a 1200 at 3000

```
5000 150  MM!   1000 152  MM!
```

Stores 5000 at location 151 and 1000 at 153.

Comments

Useful for entering small patches—anything larger than one word can be done with fewer keystrokes by one of the other memory commands. (See MM and M.)

If you have a disassembler, the byte order is set correctly; otherwise, you can set it with HL or LH.

MM?

```
<addr>  MM?
```

Displays the 16-bit word that is at the specified address.

Usage

Used to find out what is stored at a single memory location, in either ROM or RAM (see below). Use MM? for looking at words, and MDUMP or DM to see larger areas of memory.

Example

```
1210 MM?
```

Displays the word stored at 1210.

Comments

If the address is in emulation memory, its contents will be displayed; otherwise, the UniLab will attempt to use DEBUG features to display target RAM. (See STEP-INTO for details; if it isn't enabled, this won't work in RAM.)

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 target memory.

Usage

A good way to make more room when you need to patch extra code into a program. You can also use this to relocate a relocatable code module.

Like other memory-altering commands, if you use this to access RAM and you aren't stopped at a breakpoint, STEP-INTO is automatically invoked to first get an invisible breakpoint. (See STEP-INTO for details.)

Smart Mover—Automatically prevents overwriting when moving the contents of one area to an overlapping area. Starts moving from the beginning or the end of the range, 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 places. Starts moving from the beginning.

Comments

Be sure the code you move is relocatable. If it isn't, you may 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.

As with all memory-writing commands, don't write into your stack area when loading into RAM.

MODE

no parameters *F8*

Invokes the mode panels, which allow you to change the display and log modes, to toggle certain features on/off, etc.

Usage

Press F8 once to display the first mode panel, which contains analyzer mode switches. The second panel contains trace display mode switches. The third panel contains log mode and DEBUG switches.

Navigating the Mode Panels—To get from one panel to another, use PgDn. Press End to exit from the panels.

You can move around a panel and select features by pressing the ↑ ↓ cursor keys. Press the → key to toggle a feature on/off. Press F1 for help with the option that is currently highlighted by the cursor.

What They Do—See the "Mode Panels" appendix in the *User's Guide* for the complete story. You can also check the glossary entry for each feature's equivalent command.

Equivalent commands for mode-panel options:

Panel One:	DASM	SYMB	POP		
Panel Two:	SHOWM	SHOWC	=MBASE	PAGINATE	COLOR
Panel Three:	TOFILE	PRINTER	HARDWARE	SOFTWARE	
	DEBUG	BPDM	BPT		

Example

```
MODE
```

This command is never used in combination with anything else.

Operator Shortcuts

It's easier to press F8, or you can use the Configuration PopUp.

MODIFY

```
<addr> MODIFY
```

Dumps a screenful of memory, like MDUMP, but lets you overwrite the value at any address with a new one.

Usage

This invokes the memory editor, which is the best way to display and alter memory. You can alter any location by typing a new hexadecimal value or ASCII character.

Press End to exit from MODIFY.

Moving Around—The cursor keys move the cursor, scrolling the display vertically as needed (see Comments, below). PgUp moves up one screenful, PgDn moves down one.

Press Ctrl-→ to move from the hexadecimal dump area to the ASCII. Ctrl-← 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 until you press End to save changes and exit, or Esc to exit without saving changes.

Comments

If you change a location in memory and cause that address to scroll off the memory editor's screen, the change will be saved in emulation memory even if you use Esc to exit from the editor.

MS

`<count> MS` *Rarely Used*

Pauses for `<count>` number of milliseconds.

Usage

Used in test programs when you need a pause. (400 hex milliseconds is one second.)

Example

```
800 MS
```

Pauses for 2 seconds.

NDATA

`<byte1> <byte2> ... <byteN> <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.

Ranges of Data—If the data falls into ranges, you can use `TO` instead. For example, `12 TO 25 DATA` sets the analyzer looking for any data from 12 – 25, inclusive.

Example

Using `ALSO` again and again is cumbersome and requires many keystrokes:

```
18 DATA ALSO 32 DATA ALSO 36 DATA ALSO 47 DATA
```

This accomplishes the same thing, and is less prone to error:

```
18 32 36 47  4 NDATA
```

Comments

This has the same effect as OR-ing the terms with `ALSO`. Any number of terms can be listed, but be sure the count is correct.

You can use `ALSO` in combination with this command to add a range of values.

NO

NO FILTER *Rarely Used*

Used before FILTER to disable the current filter.

Usage

You will probably never use this command. It is used only to turn off the filter while preserving the current trigger spec.

The filter mechanism of the UniLab is turned on by the xAFTER macros. Those commands set the filter to MISC' FILTER, which allows you to set a trigger on all inputs except for the MISC and HDATA inputs.

See also CONTROL, HDAT and MISC'.

Example

```
NO FILTER
```

Turn off the filtering of bus cycles, but leaves the rest of the trigger spec unchanged.

NORMB

no parameters

Clears (NORMalizes) all trigger descriptions and sets the trigger event near the bottom of the trace buffer.

Usage

Used to start a new trigger definition when you want to see the events that lead up to the trigger. Use `TSTAT` to see how this command changes the `DCYCLES` setting.

When you want to start from scratch with a new trigger description, begin with `NORMB`, `NORMM`, or `NORMT`. These commands vary only in where they place the trigger event in the trace buffer—at the bottom, middle, or top.

How They Work

The `NORMx` commands clear the truth tables the analyzer uses to search for the trigger event, and set the number of cycles the analyzer will wait after the trigger before freezing the buffer. (See `DCYCLES`.)

Examples

```
NORMB
```

Clears any current specifications and sets four delay cycles.

```
NORMB NOT 0 TO 1000 ADR S
```

Shows what happened before the address went outside the 0 – 1000 range.

Comments

`NORMB` should be used when you want to know what leads up to the trigger event.

See `S+` to get a trace of the cycles that immediately follow those in the current trace buffer.

Use `Home` to view the current trace from the top of the buffer, or `<n> TN` to view it from cycle #n.

NORMM

no parameters

Clears (NORMalizes) all trigger descriptions and sets the trigger near the middle of the trace buffer.

Usage

Used to start a new trigger definition when you want to see the events that lead up to the trigger, and also what follows it. Useful when you want to see the context in which the trigger occurs. Use TSTAT to see how this command changes the DCYCLES setting.

See NORMB for details.

Example

```
NORMM
```

Sets the delay cycles to 555h (1365 decimal).

Comments

Use Home to view the current trace from the top of the buffer, or <n> TN to view it from cycle #n.

NORMT

no parameters

Clears (NORMalizes) all trigger descriptions and sets the trigger near the top of the trace buffer.

Usage

Used to start a new trigger definition when you want to see the events that follow the trigger. Use `TSTAT` to look at how this command changes the `DCYCLES` setting.

See `NORMB` for details.

Example

```
NORMT
```

Sets the delay cycles to 2720 decimal.

Comments

Use `Home` to view the current trace from the top of the buffer, or `<n> TN` to view it from cycle #n.

NOT

```
NOT  <trigger_description>
```

Triggers on the first bus cycle that doesn't match the trigger description.

Usage

Used to tell the analyzer to trigger when some byte of the 48-channel input bus goes outside a certain range or value. Commonly used to trap bad data or bad addresses.

Examples

```
NORMT   NOT 00 TO 4FF ADR S
```

Triggers if the address goes outside the range 00 – 4FF.

```
ONLY 127 ADR NOT 12 DATA S
```

Shows only cycles in which the address is 127 and the data is not 12.

```
NORMM  NOT 12 DATA  ALSO NOT 34 TO 56 DATA   S
```

Triggers when the data is neither 12 nor between 34 – 56.

Comments

Sets a flag for the next trigger word (ADR, CONT, DATA, HADR, HDATA, LADR, and MISC).

NOT clears the truth table to all 1s (except when it is used with ALSO) and writes 0s into the specified areas. This is the opposite of what would happen if NOT weren't used.

NOW?

no parameters

Shows what is happening on the target board right now.

Usage

Shows the code the microprocessor executes during the next buffer-full of cycles.

Example

```
NOW?
```

This command is never used in combination with anything else.

Comments

This simple macro turns off `RESET` (so it doesn't restart the target), then sets its own trigger and captures a trace.

NX

no parameters

Finds the next match of the current `FIND-xxxx` value.

Usage

After you have issued a `FIND-xxxx` command, use `NX` to search the current trace buffer for the *next* matching cycle. For example, use this when you want to search the trace buffer for all accesses to a particular memory location.

See `FIND-xxxx`.

Example

```
NX
```

This command is never used in combination with anything else.

ONLY

```
ONLY  <trigger_description>
```

Fills the trace buffer only with cycles that match the description.

Usage

Clears the current trigger specification and enables filtering, so only bus cycles that match the trigger will be recorded. For example, only the read cycles, or only cycles that execute the code at address 0100.

Eliminate Boring Loops—This is especially useful for filtering out status and timing loops that hog space in the display. (See the second example below.)

Notice that, when filtering, you must use `AFTER` if you want to start the trace at some particular point in the program.

ONLY and the Disassembler—Partially disassembled opcodes can be confusing, so you will sometimes want to turn off the disassembler when filtering a trace. Use the mode panel (`F8`) or `DASM'` to turn off the disassembler.

Examples

```
ONLY READ
```

Searches for and records only 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 – 135.

```
ONLY 0100 ADR
```

Records only cycles containing address 0100.

Comments

The analyzer will run until the trace buffer is full, informing you of the number of matching cycles needed to fill the buffer. To stop the analyzer before it is full, press `Esc`, then enter `T` to view the trace buffer.

OPERATOR

no parameters *Macro System*

Switches the UniLab software back to an operator-level system.

Usage

This command is only used to return to the operator level after you have entered the macro-level system with the MACRO command. An operator system created this way won't recognize any macros defined in the macro system. (See MAKE-OPERATOR to make an operator system that will recognize your macros.)

The UniLab software is shipped as an operator system, which offers access only to the UniLab commands that are found in the on-line version of the glossary (accessed via HELP <command>); it has fewer commands than the macro system, but enough power for most purposes.

Filenames—In order to switch to the operator system without first exiting to DOS, your disk directory must contain an .OPR file with the same name as the current .EXE file. If you saved your macro system under a new filename, that name was used only for new .EXE and .MCR files; so you must use the DOS command COPY to copy the original .OPR file to a new file with same name as your .EXE file—except the extension, of course. (You can still call the original operator-level system from DOS as before.)

See MAKE-OPERATOR to create a non-standard operator system.

Example

```
OPERATOR
```

Converts a macro-level system back to an operator-level system.

ORG

`<addr>   ORG`

Sets the origin address for subsequent memory-altering commands.

Usage

Sets the initial address used by commands that change the information stored in several sequential bytes of memory.

You can alter emulation ROM any time and RAM, too, if you keep `STEP-INTO` enabled, because the Orion software will automatically get a breakpoint, read the RAM, and resume target execution.

If you disable `STEP-INTO`, you must run to a breakpoint before reading from RAM. (See `RB`.)

Example

```
101 ORG 12 M 3410 MM
```

Stores 12 to location 101 and 3410 to locations 102 and 103.

Comments

Useful for entering program patches. (See also `M!` and `MM!`. See `ASM` for information about the line-by-line assembler.)

PAGE0

no parameters

Only used if your UniLab has the optional 128K memory. Selects the bottom 64K page of emulation memory (the even EMSEG value).

Usage

Addresses that are four hex digits long (16-bit binary numbers) cover a 64K memory space, but the UniLab has an optional 128K memory space. You must establish a context for addresses; for example, if your =EMSEG values are E and F, this command sets the EMSEG to E (even =EMSEG), while PAGE1 sets the EMSEG to F. Thus, after executing PAGE0, address 1300 refers to location E1300. After PAGE1, address 1300 refers to location F1300.

Example

```
PAGE0
```

This command is never used in combination with anything else.

PAGE1

no parameters

Only used if your UniLab has the optional 128K memory. Selects the top 64K page of emulation memory (odd `EMSEG` value).

Usage

See `PAGE0` above.

PAGINATE

no parameters *F8*

Enables pagination of the trace display.

Usage
The default condition—the trace display stops after each screenful.

Normally, you will use the mode panel (`F8`) to toggle this feature on/off. `PAGINATE'` is the corresponding command to turn off pagination.

Comments
If you press any key while the display is scrolling, the display will stop.

PAGINATE'

no parameters *F8*

Disables pagination of the trace display.

Usage

The trace display will scroll continuously—not very useful, except to save an entire trace to a disk file. See `PAGINATE`.

Normally, you will use the mode panel (`F8`) to toggle this feature on/off.

PATCH

no parameters

Redisplays the menu of processors supported by your DDB software.

Usage

Used if you make an error selecting your processor from the introductory display, or to try a different configuration.

Example

```
PATCH
```

This command is never used in combination with anything else.

Comments

After using `PATCH` or selecting the processor from the initial menu, you can select the "save conFiguration" option of the Configuration PopUp to save the system. The selection menu will no longer be presented each time you enter the system. Later, you can use `PATCH` to reconfigure for one of the other processors supported by your DDB software.

Only used on DDBs that can support different processors or configurations.

PCYCLES

```
<count>  PCYCLES
```

Sets the number of bus cycles after the last qualifier that the analyzer will wait before searching for the trigger event.

Usage

The default value is zero—the analyzer starts its search for the trigger event immediately after finding the qualifiers. However, you will sometimes want the UniLab to wait a number of cycles after the qualifiers, before it looks for the trigger.

For example, you know a program jumps correctly from address 235 to address 1000. But you can't understand why the code at address 1000 is executed again later. To trigger on the troublesome cycle, you don't want the UniLab to search for address 1000 until several cycles after it sees address 235.

Example

```
NORMB 1000 ADR 10 PCYCLES AFTER 235 ADR S
```

Triggers if 1000 occurs 10 or more cycles after address 235.

Comments

A "pass cycle" value postpones the search for a trigger. If there are several qualifiers, the pass count starts after the complete qualifier sequence occurs.

PEVENTS

```
<n>  PEVENTS
```

Sets the number of times the UniLab must see the complete qualifier sequence before searching for the trigger event.

Usage

Useful for catching a trace after the nth iteration of a particular sequence of bus cycles. The default value is one—the UniLab starts to search for the trigger event as soon as it has seen the qualifiers once. Use PEVENTS when you don't want to search for the trigger until the qualifiers have occurred <n> times.

This command differs from PCYCLES, which waits a specified number of bus cycles after the qualifiers before searching for the trigger.

Example

```
NORMT  12 DATA 4 PEVENTS AFTER 30 DATA  S
```

Begins searching for 12 data (the trigger) after 30 data (the qualifier) 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 the pinout of your target processor.

Usage

A handy reference correlating signal names, analyzer cable connections, and pin numbers.

Example

```
PINOUT
```

This command is never used in combination with anything else.

Comments

In order for this command to work, the three library files from the Orion Utilities diskette must be in the same directory as the UniLab program.

POPUPS

no parameters F10

Allows execution of UniLab functions from the PopUp bar.

Usage

Press F10 to place the cursor in the PopUp bar; then use the ← → cursor keys to highlight the desired item, and press the Enter key to display the related PopUp panel.

Next, use the ↑ ↓ cursor keys to select the desired function and press Enter to execute it.

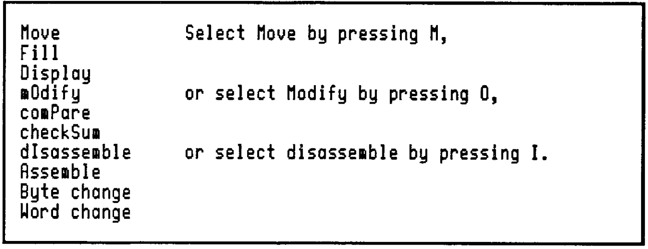
Press F1 while on any selection for On-Line Help with that function.

If you need the extra line of screen space, the PopUp bar can be disabled with the mode panels (F8), or with the command `POP'` (re-enable with `POP`).

Power Users

PopUps require fewer keystrokes than the equivalent commands, and with an added feature you can execute any PopUp function with only two keystrokes. Select any group in the PopUp bar by holding down `Alt` and pressing the key that corresponds to the capitalized letter in the bar. You can use the Memory PopUp, for example, by pressing `Alt-M` instead of scrolling along the PopUp bar. And you can then select any of the memory functions by pressing, once again, the letter that is capitalized on screen; for example, the letter "M" to select the Move option, "F" to select Fill, etc. This allows fast execution with an absolute minimum of keystrokes.

"Action" key is capitalized in PopUps:



```
Move            Select Move by pressing M,
Fill
Display
mOdify          or select Modify by pressing O,
comPare
checkSum
dIsassemble     or select disassemble by pressing I.
Assemble
Byte change
Word change
```

Keyboard shortcuts to PopUps:

`Alt-O`	Orion	`Alt-F`	Files	`Alt-E`	Eprom
`Alt-H`	Help	`Alt-A`	Analyzer	`Alt-S`	Special
`Alt-M`	Memory	`Alt-D`	Debug	`Alt-C`	Configuration

The equivalent commands are: `OPOP`, `HPOP`, `MPOP`, `FPOP`, `APOP`, `DPOP`, `EPOP`, `SPOP`, and `CPOP`. These can be used in macros or assigned to function keys.

Select "Show commands" on the Help PopUp to display the equivalent UniLab commands next to the PopUp selections.

Example

`POPUPS`

This command is never used in combination with anything else.

POP

no parameters *F8*

Enables display of the PopUp bar.

Usage
Usually done from the mode panel.

If `F10` is pressed after PopUps are disabled, they will automatically be re-enabled. Disable with `POP'`. (See `POPUPS`.)

POP'

no parameters *F8*

Disables display of the PopUp bar.

Usage

Usually done from the mode panel. Useful when you need the extra line of screen display. Enable with `POP`, or if `F10` is pressed after PopUps are disabled, they will be re-enabled automatically. (See `POPUPS`.)

PPA

no parameters *Alt-F10*

Calls up the Program Performance Analyzer menu.

Usage
Displays the menu from which you can select the PPA features.

Example

```
PPA
```

This command is 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 (`F8`) to toggle this feature on/off.

PRINT'

no parameters *F8*

Stops logging screen output to the printer.

Usage

Normally, you will use the mode panel (F8) to toggle this feature on/off.

PROM

no parameters

Rarely Used

Opens the library of EPROM read and write commands.

Usage

Advanced users can include this command in macros to burn ROMs and EPROMs or to load from them. Or you can specify the PROM on the command line, as shown below, instead of using the menu that this command invokes when used alone.

Example

```
PROM 0 7FF R2716
```

Loads the contents of a 2716 (which has been placed in the EPROM Programmer socket) into emulation memory addresses 0 – 7FF.

It is easier to use this function from the Eprom PopUp. See `READ-ROM` and `WRITE-EPROM` for more information about using EPROM commands in macros.

Q1

`Q1 <trigger_spec>` *Rarely Used*

Selects the bus-cycle description (trigger spec) that follows as qualifier one. See also QUALIFIERS and TRIG.

Usage

You will rarely use this, because AFTER—which is more natural to use —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 one qualifier without re-typing the entire trigger spec.

Example

```
Q1 15 LADR
```

Defines qualifier number one, so the UniLab will look for 15 on the low byte of the address lines.

```
Q1 ALSO 28 LADR
```

Alters qualifier one, so the UniLab will look for either 15 or 28 on the low byte of the address.

Q2

`Q2  <trigger_spec>` *Rarely Used*

Selects the bus-cycle description that follows as qualifier number two.

Usage

See `Q1`.

Q3

`Q3 <trigger_spec>` *Rarely Used*

Selects the bus-cycle description that follows as qualifier number three.

Usage

See `Q1`.

QUALIFIERS

`<1, 2, or 3> QUALIFIERS` *Rarely Used*

Selects the number of qualifying events.

Usage

Allows you to reduce the number of qualifying events. You'll probably prefer using AFTER to define the qualifiers, and will use this command only to reduce the number of qualifiers without retyping the specification. See AFTER.

Order of Qualifiers—If you have defined three qualifiers, the UniLab looks first for Q3, then for Q2, and then for Q1. It must see the qualifying events during consecutive bus cycles, or it starts over searching for Q3. (If there are only two qualifiers, the UniLab looks for Q2 followed by Q1.) Unless PEVENTS or PCYCLES has been set, the UniLab will start searching for the trigger right after it sees the entire qualifier sequence.

Example

```
2 QUALIFIERS  S
```

Changes the number of qualifiers, so that the third one is ignored.

RB

`<addr> RB`

Resumes executing the target program, with a breakpoint set at the indicated address. Must be used with RESET enabled in order to establish the first DEBUG breakpoint.

Usage

The first breakpoint must be in emulated ROM and must come after the stack pointer has been initialized. RESET must be enabled for the first breakpoint, because a software interrupt (or a call) is inserted in the target program. After the first breakpoint, RESET is automatically disabled.

See also STEP-INTO and RI ... SI.

Missed Breakpoints—If the breakpoint is not reached, the processor will continue executing code until you press any key. Pressing a key while waiting for a breakpoint causes the UniLab to try to STEP-INTO your program.

You can only set a breakpoint on an address that contains the first byte of an instruction. To be sure an address gets executed by the program, set an analyzer trigger on the same address with NORMT <address> AS.

Be sure your program initializes 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 restarts the target with a breakpoint set at address 123.

```
1007 RB
```

Without restarting the target, releases the microprocessor from the DEBUG breakpoint, and allows it to run to a breakpoint set at address 1007.

Comments

The second example above will work only if the target microprocessor is at a DEBUG breakpoint.

The first example will let the microprocessor run to a DEBUG breakpoint (alternatively, STEP-INTO would try to get an immediate breakpoint). RESET doesn't restart your target—it enables the reset flag, so that the following S or RB restarts the target.

DEBUG uses the analyzer triggers internally and may leave them in an unknown state. After using DEBUG, always use one of the NORMx words to clear the trigger spec.

See also G, GB, GW, RZ, STEP-INTO, RI, and SI.

RCOMP

no parameters *Eprom PopUp*

Compares EPROM with memory and indicates discrepancies.

Usage

Compares the contents of the device in the UniLab's EPROM Programmer socket to the contents of memory, and reports any differences. Press any key to abort the comparison.

Example

RCOMP
Presents a list of supported EPROMs, from which you indicate which device you are using. Prompts for the first and last memory addresses in memory to compare to the EPROM. A list of all discrepancies will be presented, e.g.:

```
Data is 23 at addr 0110 ..but is 67 in EPROM
Data is 15 at addr 0F6E ..but is 35 in EPROM
Data is EF at addr 211F ..but is FF in EPROM
```

If your UniLab is already configured to read the EPROM device you are using, you only need to enter two addresses—the starting and ending address of the first block of memory:

2000 3FFF RCOMP
Compares data at target addresses 2000 – 3FFF to the data in the installed EPROM.

Comments

Works on emulation ROM and target RAM. See STEP-INTO and RB.

To compare two blocks of data, use MCOMP.

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 values, not program 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
```
Triggers when the processor reads a 13.

```
NORMT READ 1000 TO 2000 ADR S
```
Triggers when the processor reads any data from addresses 1000H – 2000H.

Comments
This simple macro specifies a range of `CONT` input values, so it is only valid if the CONT column shows a unique value for target RAM read operations. Like `WRITE` and `FETCH`, this is defined in the optional disassembler software.

READ-ROM

no parameters *Eprom PopUp*

Displays a menu of ROM-reading routines.

Usage

This command provides access to the UniLab's ROM-reading routines. To copy the contents of a ROM into emulation memory, press the number of the desired ROM type. The software will ask what emulation addresses to use. (The ROM must be correctly installed in the EPROM Programmer socket, as described in the *User's Guide*.)

Operator Shortcuts

Normal access to the ROM-reading commands is from the UniLab's Eprom PopUp.

Advanced Usage in Macros

To read a ROM or EPROM from within a macro, first use the command `PROM` to "open" the library of EPROM routines. Here are the commands this makes available to read ROMs:

```
R27512      R27256
R27128      R2764
R2732       R2532
R2716
```

These commands need to know which addresses in memory to use, in the format:

```
<start address> <end address> <command>
```

Example

The following example puts an EPROM-reading command into a macro named LOAD_ROM.

Example command used in macro:

```
: LOAD_ROM
  CR ." Put PROM in socket, press any key when ready..."
  KEY DROP   PROM 1000 1FFF R2732 ;
```

Explanation:

- This macro is named LOAD_ROM (first word after the colon).
- It does a carriage return (CR), then prints (.") a message string (the string is terminated by ").
- The words KEY DROP wait for the user to press any key (then drops its value off the stack).
- PROM opens the EPROM library, and 1000 1FFF R2732 is the command to load the contents of a 2732 PROM into addresses 1000 – 1FFF.

RES

`<n> RES`

Clears bit n of the stimulus generator's output (<n> must be from 0 – 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.

Example

```
2 RES
```

Resets output S2.

```
1 SET   1 RES
```

Pulses output S1.

Comments

Used to reset individual bits of the eight stimulus outputs. See also `SET` and `STIMULUS`.

RES-

RES- <memory_access_command> *Alt-F5*

Pulls the UniLab's RES- output low and holds it low until the analyzer is started.

Usage

With some target systems, you can use this command to hold the target processor in a reset state while you access emulation memory; otherwise, the target microprocessor will see only FFs (i.e., all data lines high) when it tries to fetch from emulation memory. 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 force your target processor to reset, whether or not reset is enabled.

RES- won't work if your target has a "one-shot" in the reset circuit.

Alternate Solution

The more general solution to this problem is to get a DEBUG breakpoint before you access emulation memory. That way, your processor will be held in the idle loop while you access emulation ROM.

Example

```
RES- 10 DN
```

Pulls the reset line low, 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.

RES-'

no parameters

Disables the protracted reset pulse.

Usage

The UniLab writes into the reserved area whenever the analyzer is started with reset enabled. Therefore, to prevent unpredictable processor behavior, the UniLab pulls the analyzer cable's RES- line low. It is held low for a long time, while the analyzer is armed and started. If the protracted length of the reset pulse causes problems, disable it with `RES-'`.

RESET

no parameters *F8*

Selects automatic reset, enabling the target system to be reset by the next command you use to start the analyzer.

Usage

Along with `RESET'`, allows you to choose whether to restart the target board or just to watch a program already in operation. Automatic reset is turned on by `STARTUP`, and is turned off by `NOW`, `ADR?`, `SAMP`, and `RB`.

To run to the *first* occurrence of a breakpoint address with `RB`, you must enable reset. Always type `RESET <addr> RB` to be sure.

The status of reset is not affected by `NORMx`.

Normally, you will use the mode panel (`F8` at the `Command>` prompt) to toggle this feature on/off.

Example

```
RESET
```

Selects auto-reset

RESET'

no parameters *F8*

Turns off the automatic reset mode. See `RESET`.

Usage

Normally, you will use the mode panel (`F8` at the `Command>` prompt) to toggle this feature on/off.

RESIGHT

no parameters *Ctrl-F4*

Re-enters the InSight control/display panel, preserving any previous InSight trigger-address specification.

Usage

Provides access to the real-time InSight display. `INSIGHT` displays the same screen, but forces the `ANY ADR` (i.e., `0000 TO FFFF ADR`) address spec. Whether you use `RESIGHT` or `INSIGHT`, you can change the trigger specification from within the InSight display.

Example

```
RESIGHT
```

Re-enters the InSight display.

RI

```
RI  <trigger_spec> SI
```

Allows you get a DEBUG breakpoint when a bus cycle matches the trigger.

Usage

Used to get a DEBUG breakpoint on any cycle that can be described with the usual trigger specification commands. `RI` marks the beginning of the trigger specification, `SI` marks the end.

When the bus state you specify occurs, such as a data or control value or a range of addresses, this feature asserts a `STEP-INTO`. Generally, it takes one or two instruction cycles before DEBUG gets the breakpoint.

Examples

```
RI  450 TO 470 ADR  SI
```

Gets a DEBUG breakpoint after any address in the range 450 – 470 appears on the bus.

```
RI  WRITE  34 DATA  SI
```

Gets a DEBUG breakpoint after the value 34 is written into RAM.

Comments

This command uses the NMI or IRQ feature of the target processor to get a breakpoint, so an address trigger spec will often yield a breakpoint a few cycles *after* that address is executed.

`RI` ... `SI` uses some of the UniLab trigger logic. Avoid using `AFTER` with them.

RMBP

```
<breakpoint#> RMBP
```

Resets (clears) one of the multiple breakpoints and displays their new status.

Usage

Used to clear a breakpoint that was set with SMBP. See CLRMBP to clear all the multiple breakpoints.

Example

```
3  RMBP
```

Clears multiple breakpoint number 3.

Comments

Multiple breakpoints are used to break on any one of several addresses. Eight multiple breakpoints are available, in addition to the standard (unnumbered) breakpoint set by RB or GB.

RZ

no parameters

Resumes program execution from a breakpoint, without any new breakpoints set. The target processor will be released from the DEBUG breakpoint.

Usage

Used to run a program from the current address. Handy command for exiting DEBUG and letting the processor continue executing your program.

Example

```
RZ
```

Continues executing the program from a breakpoint.

Comments

Don't specify a trigger event before RZ—it won't work. GW waits until you start the analyzer before it releases the processor, so you can set an analyzer trigger before you leave the breakpoint.

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(s) that defines the trigger specification, though that is the usual practice. S is a separate command that gets the analyzer going, with whatever trigger spec you created already in place.

You can type TSTAT to see the current trigger status.

Examples

```
S
```

Starts the analyzer, with whatever trigger was last defined.

```
NORMT RESET 123 ADR S
```

Clears the trigger spec, turns on auto-reset, sets the analyzer to trigger on address 123, and starts the analyzer. It will run until the trigger spec (address = 123) is seen on the bus.

S+

no parameters

Identical to S, but increases the delay cycle by AA6.

Usage

S+ changes the value of DCYCLES and restarts the analyzer. It is handiest when you find that the 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 three cycles before the end of the current one.

Use this if your trigger event is regularly repeated during the program, or with RESET enabled. If your trigger is only met once in the course of program execution and RESET is disabled, the UniLab will be searching a program in progress for an event that will not occur again.

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 if the program gets stuck in an infinite loop, or if it goes far astray. But you will not be able to see much detail, only one cycle out of every several thousand.

Disassembly—You will probably want to turn off the disassembler, with the mode panel (F8) or by typing DASM'. When the disassembler is enabled, the isolated cycles will probably be disassembled incorrectly.

Example

```
SAMP
```

This command is never used in combination with anything else.

Comments

Useful when you are trying to connect analyzer inputs to something and want to continuously monitor their state. Similar to 1 SR, but it runs faster. Gives more detail on program execution than ADR?.

To change the sampling rate, see =SAMP.

Don't forget to use a NORMx command to start from scratch on trigger specs after using SAMP, because it defines its own trigger. It also turns off RESET.

SAVE-SYS

```
SAVE-SYS <filename>
```

Saves the entire UniLab system in its present state as a named DOS file. Prompts you for the filename if you don't include it on the command line.

Usage

To save a version of the system with new macros, or with default pathnames changed. Or, just to save the current emulation values or the trigger definition.

Warning: does not save the symbol table. See SYMSAVE.

Example

```
SAVE-SYS B:NEWUL
```

Saves the current state of the system to a new, executable file on the B: drive.

Comments

The target program in emulation memory is not saved by this command. See BINSAVE.

This command adds the .EXE file extension to the filename. Since the entire program image is saved, including any unintentional damage to the program, *always keep backup copies.*

Operator Shortcuts

Usually executed via the "save conFiguration" option of the Configuration PopUp.

SC

```
<max#_milliseconds> SC <filename>
```

Starts the analyzer and waits the specified, maximum number of milliseconds for the trigger. Then, when the trigger occurs, the resulting trace is compared to a previously saved trace.

Usage

When writing test programs, it is often useful to compare the current trace to a known, good trace that has been saved on disk (see TSAVE). If the traces don't match, the host computer beeps and displays a section of the saved trace and the first differing cycle of the current trace.

Hardware Checkout—SC is probably most useful during hardware checkout. To get an idea of its capabilities, save a trace by typing TSAVE test. Then pull the RAM off your target board and execute the example below. After saving the good trace, use the same trigger spec when getting the new trace. See the appendix "Writing Macros" for examples.

Example

```
400 SC test
```

Starts the analyzer with a 400H ms. trigger time limit (1 second), and compares the resulting trace to the one saved in file "test."

Comments

If the specified time limit passes before the analyzer sees the trigger cycle, the host displays a "No Trigger" message and beeps.

SET

```
<n> SET
```

Sets bit n of the stimulus generator's output (<n> must be from 0 – 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.

Example

```
7 SET
```

Sets stimulus output S7.

```
1 SET  1 RES
```

Pulses stimulus output S1.

Comments

Used to set the eight stimulus outputs separately. See also `RES` and `STIMULUS`.

SET-COLOR

no parameters

Changes the display colors when used with a color monitor.

Usage

After you have issued the command COLOR to inform the UniLab software that you have a color monitor, change the display colors with this command.

Use the cursor keys to display and choose different colors. Press End when you have completed your choices. You will need to save the system with the "save conFiguration" option of the Configuration PopUp if you want the colors to be permanent.

SET-EM

no parameters *Alt-F7*

User-friendly way to enable emulation memory.

Usage

Use the cursor keys and the space bar to enable/disable emulation memory in 2K blocks. Ctrl-← moves the cursor to the EMSEG setting, and allows you to change the 64K segment. Press End to save the new settings, or Esc to revert to the previous settings.

Comments

Eliminates the need for =EMSEG, EMENABLE, and ESTAT, except within macros.

SET-GRAPH-COLOR

no parameters *PPA*

Changes the display colors of the graph generated by the Program Performance Analyzer option (AHIST and THIST). Only appropriate when using a color monitor.

Usage

Use this—after you have issued the command COLOR to inform the UniLab software that you have a color monitor—to change the display colors of the AHIST and THIST histograms. Use the cursor keys to display and choose different colors. Press End when you have completed your choices.

You will need to save the system with the "save conFiguration" option of the Configuration PopUp if you want the colors to be permanent.

SET-TRIG

no parameters *F6*

Displays the Trigger Dialog Box.

Usage

The Trigger Dialog Box consists of a scrollable list of the most commonly useful trigger options. You can select a simple trigger from the list, or combine several selections into a more complex trigger. The system provides prompts for any necessary parameters and displays the current trigger specification at all times. See the *User's Guide* discussion of analyzer triggers for details about this feature.

Operator Shortcuts

This command is available from the Analyzer PopUp (`F6`).

Advanced Users

There are two otherwise undocumented power keys:

- `Ctrl-Q` This traverses the different qualifier levels, allowing you to modify lower-level qualifiers after higher ones have been defined. What you see is what you get; even if you only define one qualifier, if you use `Ctrl-Q` to move to `Q3`, it will set the number of qualifiers to three.

- `Ctrl-K` This "kills" the current term the next time a new value is entered, instead of ALSOing it. It allows you to clear one term or one qualifier level without deleting the entire trigger spec.

SHOWC

no parameters *F8*

Shows the control lines on the trace display (the default condition).

Usage

Turns on display of the control lines (C7 – C4) and of the high four bits of the address bus (A19 – A16).

Normally, you will use the mode panel (F8) to toggle this feature on/off.

SHOWC'

no parameters *F8*

Turns off display of the control lines on the trace display.

Usage
Turns off display of the control lines (C7 – C4) and of the high four bits of the address bus (A19 – A16).

The UniLab always monitors these wires, and sometimes they provide vital information (e.g., when the wires are hooked up wrong); but usually, you don't need to see them.

Normally, you will use the mode panel (F8) to toggle this feature on/off.

SHOWM

no parameters *F8*

Shows the miscellaneous lines and the HDATA lines on the trace display (the default condition).

Usage
Turns on display of the miscellaneous lines and—on eight-bit processors—of the high data lines. You will want to see these lines if they are hooked to your target system. Otherwise, you can ignore them.

Normally, you will use the mode panel (F8) to toggle this feature on/off.

SHOWM'

no parameters *F8*

Hides the miscellaneous lines and the HDATA lines on the trace display.

Usage

Turns off display of the miscellaneous lines and—on eight-bit processors—of the high data lines.

Normally, you will use the mode panel (F8) to toggle this feature on/off.

SHOW_MACROS

no parameters *Shift-F1*

Displays the currently defined Easy Macros. Change them with EDIT_MACROS (Shift-F10).

Usage

Easy Macros are assigned to function keys Shift-F2 to Shift-F9. These macros are definable while in the OPERATOR system.

For On-Line Help with the Easy Macros, press F1 while in the SHOW_MACROS screen.

Example

```
SHOW_MACROS
```

Normally, this function is invoked with Shift-F1.

SI

```
RI <trigger_spec> SI
```

Allows you to get DEBUG breakpoints on a wide variety of bus conditions.

Usage

Always used in combination with `RI`. (See `RI`.)

SMBP

```
<addr> <breakpoint#> SMBP
```

Sets one of the eight multiple breakpoints on the given address.

Usage

Allows you to set up to eight breakpoints, in addition to the unnumbered breakpoint set by RB or GB. The status of all eight breakpoints is displayed every time you set or clear one of them. *Set all but one of the multiple breakpoints* with SMBP. Next, use RB or GB with the remaining breakpoint address, to run the target until it executes any of the addresses you have set.

If the processor stops at an address set by SMBP, that breakpoint must be cleared (see RMBP) before the processor can run to the next breakpoint with RB or GB.

Example

```
123 4   SMBP
```

Sets breakpoint #4 at address 123.

```
4 RMBP   250 RB
```

Releases the previous multiple breakpoint (assuming the breakpoint in the above example was reached) and runs to another breakpoint, at 250 or any other address set earlier with SMBP.

Comments

Before resorting to multiple breakpoints, examine the possibility of doing the same thing more easily and quickly by using the powerful analyzer capabilities.

See also STEP-OVER, CLRMBP, RMBP.

SOFT

`SOFT <filename>` *PPA*

Enables the optional Program Performance Analyzer for the new command file it creates. Prompts for the filename if you don't include it on the command line.

Usage

Needs to be used only once. Reconfigures your software, so you can use the Program Performance Analyzer commands `AHIST`, `MHIST`, `THIST`, `HSAVE`, `HLOAD`, and `SET-GRAPH-COLOR`.

Do not use `SOFT` until you have copied the files from the PPA diskette into your \ORION directory.

Example

```
SOFT ppaZ80
```

Creates a new .EXE file, PPAZ80, which will recognize the Program Performance Analyzer commands.

SOFTWARE

no parameters *F8*

Disables the hardware STEP-INTO function (non-maskable interrupt vector installation).

Usage

This command disables the UniLab's ability to execute the hardware STEP-INTO, and enables the software single-step features.

The DEBUG features include the ability to send a non-maskable interrupt (NMI) or an interrupt request (IRQ) to the target processor. Orion software uses this to get breakpoints at any time. You can enable software single-stepping instead if your target system needs to use these signals, sacrificing only the ability to single-step into interrupts and to get immediate breakpoints.

Comments

See HARDWARE to re-enable the vector installation.

Use this command to preserve your target system's ability to use NMI or IRQ while under test. After executing SOFTWARE, the UniLab operating system will use software simulation to follow branches. (To follow program flow without using the hardware NMI or IRQ pin and vector, flags and return addresses on the stack are inspected by DEBUG.)

Operator Shortcuts

Normally, you will use the mode panels (F8) to toggle this on/off.

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 with MAPSYM.

Usage

It is necessary to use this command only if you previously disabled high-level support with SOURCE' (e.g., to see more code on screen).

About High-Level Support—Orion's high-level support shows the source code that generated each line of your assembly code. To use this feature, load your .MAP file with MAPSYM and have relevant source files in the current directory.

Source Tracking—If you use TEXTFILE to view your source code file in the upper window of the split screen, the source-tracking feature will be enabled. This updates the on-screen listing to show the source code that generated the line being shown in the other window by the analyzer, disassembler, or DEBUG. Go back to the upper window (press End) to view your original text file.

To use source tracking, you must load symbols containing high-level line number information (or use MAPSYM to load a .MAP file) and the relevant source files must be in the current disk directory. (Also see TRACK.)

See LOADER and MAPSYM for more details.

SOURCE'

no parameters

Turns off the display of source code.

Usage

See SOURCE.

SPLIT

no parameters *F2*

Toggles split-screen mode on/off.

Usage
Provides the ability to simultaneously look at two traces, or at two parts of the same trace. You can also compare a trace on screen with the assembly code (DN) and with your source text file (TEXTFILE).

The right quadrants of the screen are reserved for the output of DN and for the mode panels (F8). Text displayed via TEXTFILE and On-Line Help is shown in the upper window.

Navigating—Press End to move the cursor from one window to the other.

History—The history mechanism saves a record of what happens during your sessions with the UniLab, but it only records activity from the bottom window.

Example

```
SPLIT
```

This command is never used in combination with anything else.

Operator Shortcuts
This command is also executed by pressing F2.

SR

`<n>   SR`

Starts the analyzer repeatedly. Displays n lines each time trigger occurs.

Usage

Very useful for logging things repeatedly. You should first set the trigger and starting point of the display with `S` and `TN`. If you use `RESET`, the target system will be reset each time the analyzer starts.

Stopping—You started the infinite loop by entering `SR`. Break out by pressing any key.

Hard Copy—Use the mode panel (`F8`) or `PRINT` to log all output to the printer. The mode panel also contains a feature that allows you to log to a disk file. (See `TOFILE`.)

When to Use Something Else—If the events you want to see occur more often than once per second and you want to view them in sequence, use `xAFTER` with `AA9  SR` to log bursts of the events in filtered format.

Example

```
20 SR
```

Repeatedly displays twenty lines of the trace buffer, restarting the analyzer after each display.

SSAVE

`SSAVE <filename>` *Alt-F9*

Saves the screen image as a DOS text file.

Usage

Saves any screen image you want, e.g., a graph generated by the Program Performance Analyzer option.

Example

```
SSAVE  nice.scr
```

Saves the current screen as the disk file nice.scr.

SST

`<trigger_spec> SST`

Starts the analyzer in the standalone mode.

Usage

Use SST rather than S to start the analyzer looking for a cycle that may take a while to find. Then you can exit from the UniLab program (with BYE), or even disconnect your host PC while the UniLab searches for the trigger.

Either way, the LED on the UniLab goes out when it finds the trigger. Then, just plug in the UniLab again and call up the software with TS as a "command tail." If the trigger was found, the trace will be displayed. If it wasn't reached, you can abort and re-initialize with INIT.

If you re-enter the system without the TS command tail, press Enter before the log-on screen is displayed. This will prevent the UniLab from reinitializing the trace buffer and overwriting its contents.

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.

See also TS.

STARTUP

no parameters *F9*

Restarts the target system and shows a trace of the initial cycles of target system operation.

Usage

Very useful in the first stage of system checkout. Allows you to check the first few instructions, to be sure 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. (See your *Target Application Notes.*

The very first cycle (cy# 0) is particularly important, because if the wrong data is fetched—often due to the address not being properly `EMENABLE`d—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, consider putting a logic element into your reset circuit, such as two Schmitt triggers in a row (part number LS14). Then your system will always get a good, strong reset signal.

Example

```
STARTUP
```

This command is never used in combination with anything else.

Comments

This is a target-specific macro that, in most versions, looks for the reset vector address on the bus. If that address doesn't show up, the system will wait forever. Or, if a `HALT` instruction is fetched, it will display the message "Target Processor is Stopped." (See "General Troubleshooting" in the *User's Guide.*)

STEP-INTO

no parameters *F4*

Establishes a DEBUG breakpoint immediately or, if you are already at a breakpoint, executes a single instruction.

Usage

Supported by all Orion DEBUG packages. Uses the target processor's non-maskable interrupt (NMI) or the interrupt request (IRQ) pin. This allows you to set a DEBUG breakpoint on a running program. (Use `RB` or `RI` ... `SI` to set breakpoints if the program is already stopped.)

Auto-Breakpoint—If this feature is enabled, you can read or write RAM and I/O without first stopping at a DEBUG breakpoint. When you access a non-emulated address while the target is running, the UniLab issues a hardware `STEP-INTO` to get an invisible breakpoint (i.e., no breakpoint display), performs the requested operation, and releases the processor from the breakpoint, allowing it to resume execution.

Single-Stepping—Two separate commands allow flexibility when you single-step through code. If you are at a breakpoint, `STEP-INTO` executes the next instruction, *no matter what.* It follows jumps and branches (e.g., places in the code that always branch over a data area); another single-step command skips them (see `STEP-OVER`).

Comments

You may wish to disable `STEP-INTO` if your target uses the NMI (or IRQ) pin, or for some other reason. If so, use the mode panel (`F8`) or `SOFTWARE`. Note, however, that disabling `STEP-INTO` also disables `SI`, which uses the non-maskable interrupt. Likewise, disabling DEBUG (via the mode panel or `DEBUG'`) also disables `STEP-INTO`.

On a number of DDBs, disabling the hardware `STEP-INTO` enables a software `STEP-INTO` that will follow all program branches and returns. To use it, toggle from hardware to software in the mode panel, then use `STEP-INTO` (`F4`) normally.

STEP-OVER

no parameters *F3*

Resumes target execution, with a breakpoint set to the address after the next instruction.

Usage

Used while stopped at a breakpoint, when you want to execute only the next instruction pointed to by the program counter. With it, you can step past loops and branches, which is often very useful. For example, if the program counter is pointing at a subroutine call, use `STEP-OVER` to see the state of the processor when it returns from the routine.

Fall Through Loops—You usually won't want to single-step through loops as many times as the microprocessor executes them. 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 instruction pointed to by the program counter. So the program runs until it reaches that address.

Watch Out—If the program never reaches the breakpoint address, it will run without stopping. For example, if you don't want to use `STEP-OVER` at the last command in an infinite loop (the jump back to the beginning), the program never reaches the code that follows that last jump.

Comments

Available only when the processor is at a breakpoint.

Use `STEP-INTO` when you wish to single-step through the execution of loops and branches.

STIMULUS

```
<byte> STIMULUS
```

Changes the eight stimulus-generator outputs (S0 – S7) to correspond to the specified byte, and pulses the ST- output.

Usage

Useful for changing all the stimulus outputs at once. Use SET or RES to set/reset individual signals.

Example

```
10 STIMULUS
```

Makes all stimulus outputs zero, except S4.

Comments

The stimulus outputs originate in the PROM socket on the front of the UniLab, and are normally connected via the stimulus cable. These signals are used mainly to provide any test inputs the target system needs.

SYMB

no parameters *F8*

Enables symbol translation.

Usage

Turns symbol translation on, after it has been disabled with `SYMB'`. Symbols make a trace more readable by allowing you to replace data and addresses with symbolic names.

Normally, you will use the mode panel (`F8`) to toggle this feature on/off.

Symbols are defined with `IS` or loaded with `SYMLOAD`. Both commands enable symbol translation.

SYMB'

no parameters *F8*

Disables symbol translation.

Usage

Type this to turn off symbol translation without clearing the symbol table. See CLRSYM if you want to clear the table.

Normally, you will use the mode panel (F8) to toggle this feature on/off.

SYMDEL

```
<symlist#> SYMDEL
```

Allows you to remove a symbol from the current symbol table.

Usage

Use SYMLIST first to get a list of all the symbols (sorted according to each symbol's user-assigned value). This list is numbered. You delete a symbol *by using its list number*, not its value.

Example

```
5 SYMDEL
```

Deletes the fifth symbol in the list displayed by SYMLIST.

SYMLIST

no parameters

Displays a list of all currently defined symbols, sorted in numeric order according to each symbol's user-assigned value.

Usage

This function verifies that your symbol file has successfully loaded, and shows symbols defined with `IS`. It also gives the information you need to selectively delete symbols. (See `SYMDEL`.)

Example

```
SYMLIST
```

Lists all the current symbols.

SYMLOAD

```
SYMLOAD <filename>
```

Loads a UniLab-format symbol table from the disk file. Prompts for the filename if you don't include it on the command line.

Usage

Loads a symbol table that was saved with SYMSAVE or created by the standalone LOADER program. These are variable-length files, allowing symbols up to 255 characters long. (See the appendix "Object- and Symbol-File Formats.")

Warning: not compatible with UniLab-format symbol tables saved with SYMSAVE before version 3.0.

Example

```
SYMLOAD B:oldsyms
```

Loads a symbol table file from the B drive.

Operator Shortcuts

This function can be used via the Files PopUp.

Advanced Usage in Macros

When using SYMLOAD in an advanced macro, you have two options:

1. Include in the macro's definition the name of the symbol file you wish it to load. This is useful in macros that are used with the same files—or with files of the same name—every time.

   Example:
   ```
   : MY_LOAD 0 7FF SYMLOAD MY.BIN ;
   ```

2. When the macro is executed, have it ask the user to supply the name of the desired symbol file. As shown below, [COMPILE] defers the following command's action until the macro is executed.

   Example:
   ```
   : MY_LOAD 0 7FF [COMPILE] SYMLOAD ;
   ```

SYMSAVE

```
SYMSAVE <filename>
```

Saves the current symbol table as a named DOS file. Prompts for the filename if you don't include it with the command.

Usage

This command saves only the symbol table—the "save conFiguration" option of the Configuration PopUp saves the entire system. SYMSAVEd files can be reloaded later with SYMLOAD.

Example

```
SYMSAVE july3.sym
```

Saves the current symbol table as a disk file called july3.sym.

Operator Shortcuts

This function can be used via the Files PopUp.

Advanced Usage in Macros

When using SYMSAVE in an advanced macro, you have two options:

1. Include in the macro's definition the name of the symbol file you wish it to save. This is useful in macros that generate the same files—that is, files of the same name—every time.

   Example:

   ```
   : MY_LOAD 0 7FF SYMSAVE MY.BIN ;
   ```

2. When the macro is executed, have it ask the user to supply a filename for the output file. As shown below, [COMPILE] defers the following command's action until the macro is executed.

   Example:

   ```
   : MY_LOAD 0 7FF [COMPILE] SYMSAVE ;
   ```

T

no parameters

Displays the trace, from its current starting point, until any key is pressed.

Example

```
T
```

Displays the trace.

Comments

The starting point for the trace display is defined by the most recent `TN` command. (`STARTUP` usually sets it to -4.) If the starting cy# is not actually in the buffer, the trace is started four lines from the closest cycle number that is in the buffer.

T.

```
<hex_number> T.
```

Displays the decimal equivalent of a hex number.

Usage

Shows the decimal equivalent—compare this with `D#`, which allows you to enter a decimal number that will be used by the next command.

This word is similar to `B.` which shows the binary equivalent of a hex number.

Examples

```
10 T.
```

This will cause 16 to be displayed.

```
333 133 - T.
```

This will display 512, the decimal equivalent of 333 minus 133 (hex).

TCOMP

```
<n> TCOMP <filename>
```

Compares the contents of the current trace buffer to a trace that was previously stored in the named file. Compares the last <n> cycles. Aborts and indicates an error if any bit fails to compare.

Usage

Very useful in automated testing of target systems. Use TSAVE to save the trace of a known, good system; later, use that trace as the standard by which to judge the performance of other systems. If TCOMP finds a difference between the current trace and the one in the file, it displays five lines of the stored trace and the first bad line from the trace of the system being tested.

You can use TMASK to force TCOMP to ignore one or more inputs when comparing the traces—see TMASK for details.

You can also use SC to compare traces.

Example

```
AAA TCOMP march.2
```

Compares the entire trace to the one stored as the disk file march.2.

Operator Shortcuts

This function is easily accessed via the Files PopUp.

Comments

AAA is the hexadecimal size of the UniLab 8620's trace buffer. To compare just part of the trace, use a smaller number; then TCOMP will ignore the first part of the file. This is useful for skipping already-known discrepancies.

If TCOMP behaves in a confusing manner, try using it with the disassembler disabled (type DASM' or F8, the mode panel).

TD

no parameters

Stops the analyzer and displays the current contents of the trace buffer.

Usage

Used to see what is going on, when the trigger has not occurred or when you are producing a filtered trace that you do not think will fill the trace buffer. Normally, the trace is automatically uploaded to the host when the trigger occurs.

TD skips the first cycle in the buffer and any other empty space (all 1's) at the top of the buffer.

Example

```
TD
```

This command is never used in combination with anything else.

Comments

Since the buffer is filled with 1s before the analyzer is started, a partly full filtered trace will have good data only near the end. TD skips such empty space.

TEXTFILE

```
TEXTFILE <filename>
```

Allows you to look at a text file from within the UniLab program.

Usage

`TEXTFILE` only works in the upper window. It will take a few seconds to analyze the file, and then will display the first window of text. You can't alter the file in any way—only look at it.

This feature could replace hard-copy listings for looking at your source code while you debug it.

Moving Through the File

Press this:	**To display this:**
`PgDn`	the next screenful
↓	the next line
`PgUp`	the previous screenful
↑	the previous line
`<line#> TX`	a specific line number in the file
`Home`	the beginning of the file
`End`	toggles cursor into other window

Example

```
TEXTFILE \memo\project1
```

Opens the DOS file project1, in the directory called memo.

Operator Shortcuts

This function can be accessed via the Files PopUp.

THIST

no parameters *PPA*

THIST, the time-domain histogram, invokes the optional Program Performance Analyzer 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 see how the elapsed time between any two addresses changes, as different conditional jumps or branches are taken. You will probably want to measure the time between two addresses in your program's main loop.

Before you use the PPA the first time, you must issue the command SOFT to install it.[1] SOFT performs a SAVE-SYS, then causes an exit to DOS. The next time you boot the UniLab software, the PPA will be installed.

Start the Histogram—To produce a histogram, first specify the upper and lower limits of each address "bin" you want to display, then start the histogram.

When you issue the command THIST, the histogram screen is displayed with the cursor positioned on the first bin. You can then type a lower and upper limit for each bin. Press return, tab, or an arrow key to move to the next field. Press F1 to start the histogram.

Save to a File—You can save a histogram's setup (i.e., bin limits, title, and any labels) in a file, along with any collected data, after you exit from the histogram screen. Just type HSAVE <filename> at the Command> prompt or use the Files PopUp.

Load From a File—You can load a previously saved histogram by typing HLOAD <filename> at the Command> prompt, or via the Files PopUp. Loading a histogram file also invokes the PPA in the correct mode.

Example

```
THIST
```

This command is never used in combination with anything else.

Operator Shortcuts

This function can be used via Alt-F10, the PPA menu.

[1] *Only the first time you use the PPA.*

TMASK

```
<byte_value> TMASK
```

Sets a mask that tells TCOMP which columns of the trace display to compare.

Usage

Used when comparing traces, to filter already-known discrepancies or unimportant error messages (e.g., to ignore different MISC wiring). The lower six bits of the hex byte's value tell TCOMP which input groups to use when comparing traces. The default is 3F (0011 1111 binary), which tells TCOMP to compare all columns.

Mask Bits—Each of the six bits corresponds to one of the analyzer's input groups. If the bit is set to one, TCOMP will include that group in its comparisons:

Binary	Inputs	Hex
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 of subsequent trace displays.

Usage

For random access to the trace buffer, when you also want to change the starting point that will be used by `T`. (To see a cycle without changing the default starting point of the display, use `TNT`.)

Example

```
12 TN
```

Displays the trace, starting 12 cycles after the trigger. Subsequent traces in this session will also be displayed starting 12 cycles after the trigger.

Comments

You will usually use `TNT`. Use `TN` only when you want to display from the same cy# on later trace displays.

TNT

```
<n> TNT
```

Displays the trace buffer, starting at cycle n (default = -5).

Usage

Allows random access to the trace buffer. `TN` does the same thing, but changes the default starting point used by `T` when it displays subsequent traces.

Example

```
-7 TNT
```

Displays the trace starting 7 cycles before the trigger.

TO

```
<number> TO <number> <command>
```

Sets a flag to indicate that a range of numbers is being entered.

Usage

Used with the trigger-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-input values in the range 12 – 34, inclusive.

TOFILE

`TOFILE <filename>` *F8*

Opens a DOS text file and starts copying all screen output to it.

Usage

Logs all screen output to a disk file. You can include this command on the DOS command line as a "command tail." You will be prompted for the filename if you don't include it.

You must first type this command explicitly at the `Command>` prompt, to open the file. Then the mode panel (`F8`) can be used to toggle logging on/off. (See also `TOFILE'`.)

Example

```
C> ULZ80  TOFILE A:JUNE7.LOG
```

Comments

Files produced in this way can be edited with a word processor, or displayed with the DOS command `TYPE <filename>`.

TOFILE'

no parameters *F8*

Stops sending a copy of the screen output to a DOS text file.

Usage
Used to stop logging information to a file. Normally, you will use the mode panel (`F8`) to toggle this on/off.

See `TOFILE`.

TOP/BOT

no parameters *End*

Toggles the cursor between the top and bottom windows of the split screen.

Usage

Usually, you will just press `End`. This command is only active when the screen has been split. See `SPLIT` for details about windows.

Example

```
TOP/BOT
```

This command is never used in combination with anything else.

Operator Shortcuts

This function is normally used by pressing the `End` key.

TRACK

no parameters

Turns on the source-tracking feature.

Usage

If a `TEXTFILE` window is open, the proper source file will be opened and displayed in the upper window as you perform DEBUG functions, look at traces, or disassemble memory. This is the default, and is done automatically if high-level lines are loaded in the UniLab symbol table. Each time a new source line is shown in the breakpoint display, trace, or disassembly, the corresponding section of the source file is shown in the upper window. Disable this feature by returning to a full-screen display (`F2`) or by typing `TRACK'`.

TRACK'

no parameters

Turns off the source-tracking feature.

Usage

See TRACK.

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 RAMto occupy the same address space, or that address more than one 64K segment of memory.

Usage

Only needed when referring to RAM that occupies the same 16-bit address space as ROM, as when working with the 8051 or 8048, which have data memory and/or special function registers at the same addresses as program memory.

This flag stays reset until you use the command `TRAM'`.

Why You Don't Need to Bother—Orion DDB packages for processors with these types of memory configurations have special commands for inspecting and changing the contents of RAM and special-function registers.

Example

```
TRAM   0  F  MDUMP
```

Dumps the contents of RAM addresses 0 – F.

Comments

This can get you into trouble: if you use RB after TRAM, you will be setting a breakpoint in RAM. That is fine if you intend to do so, but can be disastrous if you meant to set the breakpoint in ROM.

TRAM'

no parameters

The default condition: sets a flag so that subsequent memory reference commands refer to ROM rather than RAM.

Usage

Only needed after using TRAM or similar processor-specific commands.

Example

```
TRAM'  30 3F MDUMP
```

Dumps the contents of ROM addresses 30 – 3F.

TRIG

```
TRIG  <specification>
```

The specification following will be used as the trigger event.

Usage

As opposed to Q1, Q2, Q3, and AFTER, which tell the analyzer to use the following specification as a qualifier. Useful to alter the trigger without altering or retyping the existing qualifiers. If you use TRIG and still want the analyzer to use the existing qualifiers, issue the appropriate Qx command before starting the analyzer.

Example

```
TRIG  123 ADR
```

Searches for 123 on the address lines.

Comments

Used to set the TRIG truth-table context after your use of Q1, Q2, Q3, or AFTER has selected another truth table. Useful if you change your mind about the trigger spec after you have defined a qualifier.

The four truth tables are TRIG, Q1, Q2, and Q3.

TS

```
ULxx   TS
```

Displays the trace buffer's contents; used after the analyzer was started with the SST (standalone) command.

Usage

When you use SST to start the analyzer, you can exit from the UniLab environment (with BYE) and use other programs, or even disconnect the UniLab from the host computer. When the analyzer finds the trigger cycle, the light next to the analyzer cable goes out. You can retrieve the trace any time after that.

To display the trace, restart the UniLab software with a TS in the "command tail." Or start the UniLab normally, but press Enter *before the initial display appears.* This prevents the UniLab from reinitializing the trace buffer, which would overwrite its contents. You can then type TS to display the data.

Example

```
C>ULZ80 TS
```

The Z80 UniLab system is executed from DOS with the TS command tail. If the system was previously exited after an SST trigger spec, and if that trigger was reached, the trace buffer data will be displayed upon re-entry.

TSAVE

```
TSAVE <filename>
```

Saves the current trace buffer as a file.

Usage

A good way to save information about a trace for later review (see TSHOW), or for automatic comparison to another trace (see TCOMP and SC).

Example

```
TSAVE good.trc
```

Saves the current trace as a disk file named good.trc.

Operator Shortcuts

This function can be accessed via the Files PopUp (F5).

TSHOW

```
TSHOW <filename>
```

Displays a previously saved trace.

Usage

A useful way to examine traces saved to disk manually or by an automatic testing program. (See TSAVE.) Use TCOMP if you want to compare the current trace buffer to a saved trace—it will serve most purposes better than studying a trace visually. (To print a trace in progress, turn on the log-to-printer switch via the mode panels before you start the analyzer.)

Example

```
TSHOW good.trc
```

Displays the trace named good.trc, which was saved as a file earlier by TSAVE.

Operator Shortcuts

This function is easily accessed via the Files PopUp.

Comments

After you use TSHOW, the host computer's memory contains the recalled trace, and you can use T, TN, or TNT to control how it is displayed.

To reload the UniLab's current trace buffer back into the host, type TD. TSHOW changes the DCYCLES setting, so the cycle numbers may be incorrect.

TSTAT

no parameters

F7

Displays the current trigger's status—the entire specification, including qualifiers, delay and pass counts, filtering, and reset.

Usage

A good way to review the current settings, especially to see that the UniLab interprets your trigger specification commands as you intend.

Example

`TSTAT`

This command is never used in combination with anything else.

Operator Shortcuts

This is the same as pressing `F7`.

Simplified Display

In `OPERATOR` mode, `TSTAT` shows a simplified version of most trigger specifications. For instance, instead of `4 DCYCLES`, it says "Trigger at Bottom"; and it displays "read cycles" or "fetch cycles" instead of the explicit filter commands. But if you switch to `MACRO` mode, or define a fairly complex trigger, `TSTAT` will show the entire specification. (See `OPERATOR` and `MACRO`.)

TX

`<line#>  TX`

Moves to the specified line number of the open text file.

Usage

A good way to move around a text file quickly. (See `TEXTFILE`.)

Example

```
300 TX
```

Moves to line 300 (decimal) of the currently open text file.

Comments

The `<line#>` is interpreted as a decimal number, because text files viewed with the `TEXTFILE` command are displayed with decimal line numbers. To move to a hex line number, type:

```
<line#> TXH
```

USEC?

```
<from_addr> <to_addr>  USEC?
```

Counts the number of microseconds between two addresses.

Usage

Used to count the execution time in microseconds of a loop (first example below) or of any segment of code (second example). See CYCLES? to count the number of bus cycles between two addresses.

Examples

```
123 123 USEC?
```

Measures the execution time of the loop that contains address 123.

```
123 456 USEC?
```

Measures the execution time of the code in addresses 123 – 456.

Comments

Useful for checking execution times. USEC? makes its own trigger spec, so you will have to start fresh on your trigger after using this command.

VER?

no parameters

Prints information about your version of the UniLab software.

Usage

Used to show the processor selected (if your software supports multiple processors), as well as emulation memory size, date of most recent DEBUG software update, and the version number of your UniLab system software.

Example

```
VER?
     128K Emulation Memory
     Configured for 80188/24Nov88/ul4.04
```

Comments

When calling for technical support, you will be asked to type this command, so Orion's staff can know what equipment and software version you are using.

WORDS

`WORDS <command>`

Displays an alphabetical list of the UniLab's commands, starting with any command or characters included with it on the command line.

Usage

Used to remind you of the names of UniLab commands. Press any key to stop the listing.

Example

```
WORDS INIT
```

Shows an alphabetical list of UniLab commands, starting with `INIT`.

WRITE-EPROM

no parameters

Displays the EPROM programming screen.

Usage

Provides access to the EPROM writing routines. This command presents the menu of supported ROMs. From this menu, program an EPROM by typing the number that labels the type of EPROM you are using. The software will prompt for the start and end addresses in emulation memory of the source.

See "EPROM Programmer" in the *User's Guide* for details.

Advanced Usage in Macros

To burn an EPROM from within a macro, first use the command PROM to "open" the library of EPROM routines. Here are the commands this makes available to burn EPROMs:

```
P27512     P27256
P27128     P2764
P2532      P2732A
P2716
```

These commands need to know which addresses in memory to use, in the format:

```
<start address> <end address> <command>
```

The following example puts an EPROM-burning command into a macro named BURN.

```
: BURN
  CR ." Put EPROM in socket, press any key when ready..."
  KEY DROP   PROM 0 7FFF   P27256 ;
```

Explanation:

- This macro is named BURN (first word after the colon).
- It does a carriage return (CR), then prints (.") a message string (the string is terminated by ").
- The words KEY DROP wait for the user to press any key (then drops its value off the stack).
- PROM opens the EPROM library, and 0 7FFF P27256 is the command to burn a 27256 EPROM with the contents of emulation memory addresses 0 – 7FFF.

WSIZE

no parameters *Alt-F8*

Allows you to adjust the size of the split-screen windows. See also `SPLIT`.

Usage

After you type this, use the up- and down-arrow keys to adjust the vertical dimensions of the windows. Press `End` to accept your changes and exit, or `Esc` to quit without changing the window sizes.

Example

```
WSIZE
```

This command is never used in combination with anything else.

APPENDICES

WRITING MACROS

Introduction

Easy Macros allow you to link UniLab commands, and to execute them with a single function key. They provide enough macro power for most users. (See EDIT-MACROS.)

If you choose to advance, the UniLab's macro-level system will provide a number of advantages over Easy Macros. You can:

- write longer macro sequences,
- extend the UniLab command language with your own custom commands,
- create complex control structures,
- rename UniLab commands—abbreviate existing commands, or make them longer and more descriptive of your application, and
- define as many macros as memory allows.

If you wish to write advanced macros or test routines, familiarize yourself with this appendix. Any macro language requires familiarity with its vocabulary and syntax. The UniLab's macro features derive their power from Forth, the underlying programming environment, so its macro capabilities are endless—and well documented elsewhere (see "About the Macro Language" later in this appendix).

It is not necessary to learn Forth to use the UniLab. And you don't need to use control structures to write useful macros—but for further reference, the UniLab control structures are explained in several books about Forth.

Operator-Level and Macro-Level Systems

The default UniLab software environment is known as the operator-level system. You can use PopUp selections, any command in the On-Line Help glossary, and Easy Macros. The `Command>` prompt is displayed at this level.

The macro-level system provides advanced UniLab commands, plus the entire Forth programming language. With it, you can add new commands to the UniLab system and then return to the normal mode of operation. The `ok` prompt is displayed at this level.

Type `MACRO` to invoke the macro-level system—the word `ok` will replace the usual `Command>` prompt.

Type `OPERATOR` to return to the operator-level system.

How to Write a Macro

A macro definition begins with : (a colon) and ends with ; (a semi-colon). These will not be recognized by the UniLab software unless you are in the macro-level system (i.e., type MACRO).

Always put a space after the colon, and another before the semicolon—*every element in a macro definition must be delimited by one or more spaces.*

The first word after the colon is the macro's name—what you will type to execute the new command. Following its name is the sequence of parameters and current commands that it will execute. As always, the macro definition concludes with a semi-colon. (See the glossary entry for :.)

Example macro definition:

```
: D10 DUP 10 + MDUMP ;
```

Creates a macro called D10, which will dump ten memory locations. This new command requires the starting address of the range to dump. E.g., in use it will be preceded by an address:

```
    <start_addr> D10
```

Explanation:

:	Begins the macro definition.
D10	Name of the new command.
DUP	Copies the last parameter that was entered (the address).
10 +	Adds 10 (hex) to it.
MDUMP	Displays the range `start_addr` to `start_addr+10`
;	Concludes the macro definition.

After defining this macro, you could type:

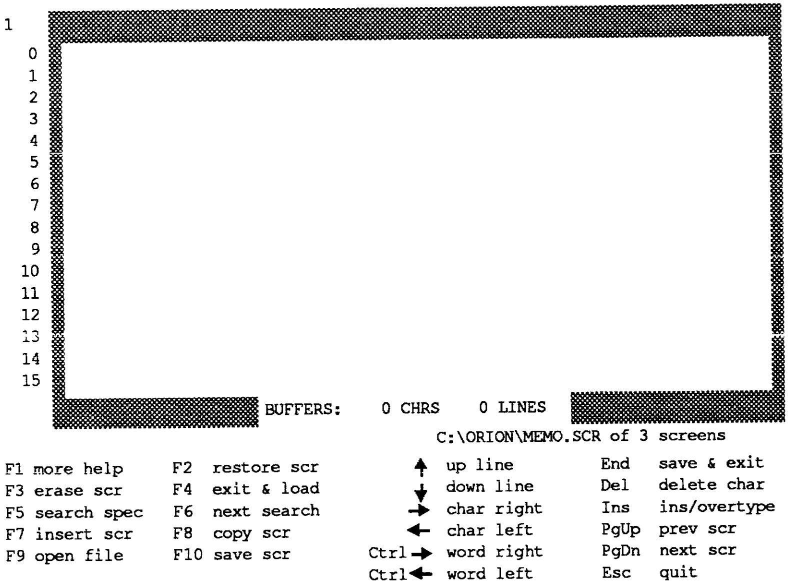
```
342  D10
```

to display the contents of addresses 342 – 352. If you then used the "save conFiguration" option of the Configuration PopUp, this new command would be a permanent part of your UniLab system. (All numbers are in hexadecimal, and all math operations use postfix notation.)

Write Macros on Forth Screens

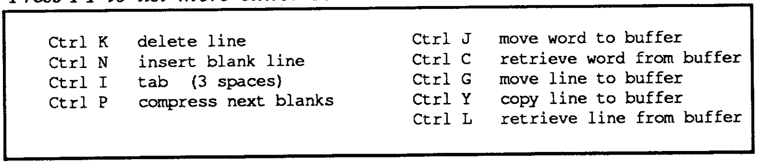
It is easiest to write, test, and alter macros in files (with the Forth-screen editor), and to then load the macros from the file.

The UniLab software normally keeps open a file named 8620.SYS for some system functions. Another file is available for writing macros or just for typing notes on its three screens. Type MEMO to display the first of those screens. You'll also see prompts for using the Forth-screen editor. Press Esc to exit from the editor.



```
1
   0
   1
   2
   3
   4
   5
   6
   7
   8
   9
  10
  11
  12
  13
  14
  15
                  BUFFERS:    0 CHRS    0 LINES
                                    C:\ORION\MEMO.SCR of 3 screens
F1 more help     F2  restore scr          ↑  up line      End   save & exit
F3 erase scr     F4  exit & load          ↓  down line    Del   delete char
F5 search spec   F6  next search          →  char right   Ins   ins/overtype
F7 insert scr    F8  copy scr             ←  char left    PgUp  prev scr
F9 open file     F10 save scr        Ctrl →  word right   PgDn  next scr
                                     Ctrl ←  word left    Esc   quit
```

Press F1 to list more editor commands:



```
Ctrl K   delete line                  Ctrl J   move word to buffer
Ctrl N   insert blank line            Ctrl C   retrieve word from buffer
Ctrl I   tab  (3 spaces)              Ctrl G   move line to buffer
Ctrl P   compress next blanks         Ctrl Y   copy line to buffer
                                      Ctrl L   retrieve line from buffer
```

Load Macros From a Screen

After you type a macro on a screen, press F4 to load the contents of that screen into memory. When they are on screens, your macros are easily available for later alteration, as the occasion demands.

Create a Forth File for Macros

After working with macros a short while, you will find it very convenient to store them in separate, loadable files.

Create a new file by typing:

```
OPEN <filename>
```

If a file by that name doesn't already exist, this creates one with three blank screens (default size). If the specified file already exists, this command just opens it.

Increase the open file's size by typing:

```
<#screens> SCREENS
```

1K is allocated per screen. Always specify at least two screens (numbered from zero, i.e., 1 SCREENS). Screen number zero can't be loaded into memory—leave it blank or just use it for comments.

```
<screen#> EDIT
```

Moves you into a file after it has been opened, invoking the Forth-screen editor used by MEMO.

To close a file of screens and return to the UniLab's macro level, type:

```
UDL.SCR
```

Closes your file and re-opens the 8620.SYS file. If you don't do this, some of the On-Line Help facilities and error messages will not work. This is done automatically if you use F10 twice (i.e., enter and then leave the PopUp Bar).

Breakpoint Macros

Suppose you need to know—every time a subroutine is executed—the value of a port and the contents of a memory buffer. You could define a macro to provide exactly what you need.

First, a macro could be written to show the port and memory-buffer contents:

```
: DISP_MEM&PORT 24 INP 7C00 7CFF MDUMP ;
```

Some text strings and a carriage return will improve the display:

```
: DISP_MEM&PORT
  ." Value at Sensor="  24 INP CR
  ." Data Buffer="  7C00 7CFF MDUMP ;
```

Now, when stopped at a breakpoint, you can execute the new macro by typing DISP_MEM&PORT. The macro could be assigned to a function key, or to an Easy Macro and executed with a shifted function key.

To execute this macro when the program reaches a subroutine labelled CALC_DELAY, type this command:

```
CALC_DELAY RB DISP_MEM&PORT
```

Using BPEX

There is a way to add any macro to a UniLab breakpoint display. You do this with BPEX, the breakpoint-execution assignment command.

To install the above macro into your system's BPEX, type:

```
BPEX DISP_MEM&PORT
```

By patching in your macro, you have customized the breakpoint display. Now, DISP_MEM&PORT is executed automatically at every breakpoint display. In this example, CALC_DELAY RB will now execute DISP_MEM&PORT after the register display and before the "next step" disassembly.

To disable the customized breakpoint display, type:

```
BPEX NOOP
```

The possibilities with BPEX are endless. You could insert a macro to turn off timers on the target board or to resume program execution immediately after the breakpoint display.

In fact, you can do many things with BPEX that are similar to the the UniLab's InSight display. In effect, you can "roll your own" custom InSight routine!

Once you have the memory displays set into BPEX, you can define a recurring breakpoint which continually monitors the register contents and the memory locations. You need to use a Forth loop structure. A simple Forth loop can be constructed with BEGIN ... UNTIL. For instance:

```
: MY_INSIGHT
  BEGIN
  CALC_DELAY RB ?TERMINAL
  UNTIL ;
```

produces a series of breakpoints until you press a key.

Everything between BEGIN and UNTIL will be repeated, until the flag before the UNTIL is true—in this example, a breakpoint will happen each time CALC_DELAY is called in the program. The Forth word ?TERMINAL is used to exit the loop by passing a true flag to UNTIL if any key on the terminal is pressed.

Now, you can just turn on the printer (or the log-to-file switch), type MY_INSIGHT, and go to lunch. On your return, there should be enough data about your program collected for a leisurely inspection of its actions.

Summary: Breakpoint Macros

You have been introduced to the basic techniques for extending the UniLab commands. For most, this is a good place to stop. Only read the rest of this appendix when you need more power or flexibility.

But once you master BPEX, you may want to create loops to test the condition of register values, or to mask and change flags. At that stage, read the last part this appendix, called "DEBUG Variables." Then learn about Forth control structures like IF ... ELSE ... THEN and DO ... LOOP.

Write Test Programs

You can use the macro capability of your UniLab system to write automatic test programs. In this section, we present some examples you can adapt to your specific needs.

The NOP Test

A good first test for a new system is simply to let it execute no-op instructions. For example, let's assume you have a Z-80 system with addresses 0 – 7FF enabled. You fill enabled memory with the `00` (nop) opcode by typing `0 7FF 0 MFILL`. Next you use the Analyzer PopUp, or type `STARTUP`, to display the trace as the target starts after reset. If the address bus has a short or is open, it will show in the trace display as a departure from the normal sequence of addresses.

The Known, Good Test

The UniLab can compare the current contents of the trace buffer with a trace saved on disk (see `TSAVE`) and will report any differences. You can, thus, automate system checkout by comparing the trace of an untested system to one saved from a known, good system.

`STARTUP` and `S` are not well suited to automated test macros, since they cause the trace to be displayed until a key is pressed. Use this instead:

`<n> SC <filename>`
Starts the analyzer, waits n milliseconds, then compares to the trace that is stored on disk as `<filename>`. This is very useful for automated test sequences. (Of course, you will have to use `TSAVE` to save the reference trace from a good system.)

Automated NOP-test macro using SC:

```
: TEST1 0 7FF 0 MFILL NORMT RESET 100 SC A:.TRC ;
```

Explanation

This macro defines a preliminary test named `TEST1`. When executed, it will fill the first 7FF memory locations with zeros (nop instructions), reset the target processor, start the analyzer, wait 100 milliseconds, and then compare the contents of the trace buffer to a trace that was saved as a file on drive A.

With such a macro, a technician can test a system simply by typing TEST1. The UniLab will reply with an OK if the current trace matches the one stored on disk.

Compare traces after triggering:

```
: TEST2 NORMT 6FF ADR 400 SC A:.TRC ;
```

If these two tests each work properly when used separately, we can easily combine them into a single test:

```
: TEST ." Test 1:"  TEST1  ." Test 2:"  TEST2 ;
```

Explanation
This defines a new word TEST to execute the two macros (above) in sequence, identifying each and displaying an OK if the system passes the test. (Note that ." <message>" will print <message> on the screen when the macro is executed). If the target system fails either test, the process stops and SC displays the faulty bus cycles.

Include Messages in Macros

You can use a macro to display a message to the UniLab's operator. Just start your message with ." and end it with " as above.

You can also use KEY to wait for a keystroke. This word also leaves the ASCII code of the pressed key on the stack. You can get rid of the code with DROP or use it as you wish. For example, you could use it to decide the next action:

```
: 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 presses "y." If any other key is pressed, it will just print the closing "Bye."

Removing Macros

If you define a test that doesn't work, erase the definition by typing:

```
FORGET <macro_name>
```

Warning: FORGET also forgets everything you defined *after* <macro_name>. Type VLIST for a list of all defined words, from last to first.)

Another method of effectively "erasing" a macro is to define another macro, using the same name. The latter-most definition supercedes any earlier ones, and is the one that FORGET operates on.

Make Custom Operator Systems

The operator-level system hides advanced macro commands and most Forth words. In this way, someone who doesn't use Forth can avoid confusion. For example, this prevents you from mistyping `WORDS` and accidentally executing `WORD` instead, a special Forth command which would cause unexpected results.

It also shields the user from accidentally entering the compiling state, and from disastrous crashes caused by unknowing use of Forth words. The `Command>` prompt denotes the operator-level system.

In the `OPERATOR` mode you can use the PopUps, all UniLab commands that are in the manuals, and On-Line Help; but you cannot make macros (except Easy Macros).

To define macros and save the resulting system *at the operator level*, use the `MAKE-OPERATOR` command. This will first save the current macro-level system, asking for a filename. Then it will reconfigure for the operator level, again asking for a filename. Use the same filename, or a different name if you want both an `OPERATOR` and a `MACRO` system available on your hard disk. (Users of floppy-disk-based systems will not have room on one disk for two systems.)

About the Macro Language

Your system's macro features, as described so far, are only the tip of the iceberg. A complete Forth system is resident within the UniLab software. This language includes control structures such as `DO ... LOOP`, `IF ... ELSE ... THEN`, and `BEGIN ... UNTIL`. See the appendix, "Forth Reference." You can define much more complex macros than the examples we have covered.

To learn more about Forth, the best book is:

Starting Forth by Leo Brodie
Prentice-Hall
Englewood Cliffs, New Jersey

To contact Forth users or to receive a catalog of books, etc.:

Forth Interest Group
P.O. Box 8231
San Jose, California 95155
(408) 277-0668

They sponsor the meetings of many local chapters and publish the bi-monthly journal *Forth Dimensions.*

Many Forth books and programs are available from:

Mountain View Press Inc.
P.O. Box 4656
Mountain View, California 94040
(415) 961-4103

The UniLab software was originally developed from the MVP-Forth Professional Application Development System (PADS; see following). The public-domain part of that system, a modified Forth-79 Standard, is included with the UniLab. Excellent documentation is provided in *All About Forth* by Glen B. Haydon, a complete glossary of Forth words.

To buy the manual for the PADS Forth system, if you plan to use the UniLab software's Forth capabilities:

ForthKIT
240 Prince Ave.
Los Gatos, CA 95030

The manual includes source code in Forth screens, and documentation for many nice utilities that are included with the system.

UniLab Forth Operating System

The UniLab operating system is a full-featured Forth compiler. It was derived from the PADS Forth system developed by Tom Wempe and distributed by Mountain View Press. PADS (the Professional Applications and Development System) was an extension of the Forth-79 Standard. Orion subsequently changed this Forth compiler's structure somewhat, and augmented it to handle larger memory areas.

The UniLab Forth is still, basically, a 16-bit Forth system (i.e., the maximum number of bits in a single number is 16 bits).

The two main changes in structure are:

- The Forth system is now multi-segment: it operates in more than one 64K block of the host PC's memory. Although PADS Forth has a way to deal with this, we have taken it further by making it more transparent. You will not have to deal with it in any explicit way.
- The "heads" of all words (the name and link fields) are separated from the "bodies" (the parameter and code fields). This allows more room for definitions in a single 64K address space.

Alterations to PADS Forth Words

`NOT` Renamed to `XOT`, in order to avoid conflict with the UniLab command `NOT`. We recommend using `0=` instead.

`EMIT` Now vectored through `TYPE`, rather than vice-versa. `TYPE` is *fast.*

`TYPE` Changed to write directly to the PC's memory-mapped video buffer.

`CFA` `CFA` is a valid hexadecimal number, so we renamed it `CFADR`.

`SCR` Most user-variable names in Forth are preceded by an asterisk. Thus, `SCR` is renamed `*SCR` in the UniLab Forth system.

UniLab's Forth Editor

While writing complex macros, you will want to use the editor. The Forth Screen Editor from the PADS system has been modified by Orion for use with the file commands `OPEN`, `OPEN-TO`, `COPYF->F`, etc., as documented in PADS.

Invoke the editor with `<screen#> EDIT`. The default file in the UniLab system is 8620.SYS, which is used for messages, certain displays, and some other functions. If you open another file and later execute UniLab commands which need this data, you may get some strange error messages. Type `UDL.SCR` or press `F10` twice at the `Command>` prompt to re-open the 8620.SYS file.

All cursor and editor functions are displayed on screen when the editor is invoked. Press `F1` at the Command> prompt or use the Help PopUp to find details about editor functions.

Stack Diagrams

As a matter of programming convention, each Forth definition includes a *stack diagram* that shows the stack effects of that word. This is just to help the readability of the code. The diagram is shown in parentheses so, like other comments, it won't be executed even when it is in-line with Forth code. Anything to the left of the double hyphens represents parameters on the stack needed by that word; to the right are the stack results of its execution. For example,

Example uses of stack diagrams:

`WORD1 ( -- )`	Requires no parameters, and returns no values to the stack.
`WORD2 ( n1 n2 -- n3 )`	Requires two parameters, and returns a third value to the stack.

File and Editor Commands

`FILE.` (--)
Prints the pathname and size of the current screen file.

`OPEN <filename>` (--)
Opens the specified file, making it the current screen file. Creates the file if it doesn't already exist. Press `F9` to open another file.

`EDIT` (n --)
Invokes the Forth editor at screen n. See the PADS editor documentation for related words.

`V` (--)
Invokes the Forth editor at the last screen edited.

`INCLUDE <filename>` (--)
Used in a Forth source file; opens the specified file, loads screen one, and returns to the previously active screen file. Supports nesting.

`SCREENS` (n --)
Extends the current screen file to <n> screens. Does nothing if n is less than the current size of the file—won't decrease the size of a file. To decrease the size of a file, create a new file with `OPEN`, use `OPEN-TO` and `COPYF->F` (see PADS documentation), delete the old file, and rename the new one.

`COPY` (n1 n2 --)
Copies screen n1 to screen n2 in the current screen file.

Decompiler

Your UniLab software includes a simple decompiler. It is useful when you want to understand any of the UniLab or Forth words, or to recall your own words' definitions. To use it, type:

```
' <word> XX
```

where <word> is any word in the system. Each time you type XX, the decompiler will print the address, contents, and name of each sequential word in the definition of <word>.

When you decompile words that are referenced in other segments, you might see rather strange decompilations beginning with the word [SEGJMP].

Character I/O

We have a function-key handler (<FUNC-KEY) in the definition of EXPECT. You may wish to defeat this key handler, to prevent a test technician from accessing the function-key assignments. Do this by, first, making a variable to save the current function-key handler. Below, it's called FUNC-SAVE.

Defining a variable to save the function-key handler:

```
VARIABLE FUNC-SAVE
' <FUNC-KEY> @ FUNC-SAVE !   ( save function key handler )
' NOOP CFADR ' <FUNC-KEY> !  ( cripple function key handler )
```

Command to restore the function-key handler:

```
FUNC-SAVE @ <FUNC-KEY !
```

Definition of a word that detects function keys and regular keys:

```
: EX-KEY? 0 0 0 0 16 INTCALL      ( ROM call for key scan)
  DUP FF AND 0= ;  ( -- asc f)    ( f is true if function key)
```

Case Sensitivity

A word called UPPER_KEY returns only upper-case values when a key is pressed. It also detects function and cursor keys, returning unique values for them (i.e., bit 7 is set). This makes the system insensitive to the case of user input. All new words in the dictionary will be converted to upper case, and all lower-case hexadecimal digits will be converted before they are interpreted as numbers.

Most user input is converted to upper case by this word:

```
: L->U     ( addr --)
  L/U? IF  <L->U> ELSE  DROP THEN ;
```

If you want a system that is case sensitive, change the constant L/U? to zero, as shown below.

Command to make your system case-sensitive:

```
' L/U? OFF
```

Macros that Accept Input

```
GET#/SYM          ( -- d f )
```

Use GET#/SYM in a macro definition to prompt the user for a literal or symbolic value during the macro's execution. It leaves a double number and a flag on the stack, which are then used by subsequent macro commands. If the user provides a symbol that is not in the current symbol table, the flag will be zero (false) and the double number meaningless. A correct response—a hex number or valid symbol—results in the flag value of one (true) and the double number on the stack.

<G#> is a number-input word. When executed, this command waits for the user to enter a number followed by Enter. <G#> leaves the number and a flag on the stack. If the input was a valid number, the flag is true.

All character output, even when using EMIT, is sent through TYPE. Output can also be routed to the printer and to a DOS file.

The following word issues a carriage return if you are within <n> characters of the right margin:

```
?WRAP  ( n -- )   ( CR if less than n chars left on line)
```

Commands that Require Filenames
The file-loading words are immediate, so you can compile a filename right into a macro. This can be done if you use the same filename every time with a macro.

Include a filename in a word's definition:

```
: MY_MACRO
    0 TO 7FFF EMENABLE
    0 TO 7FFF BINLOAD MYPRG.BIN
    STARTUP ;
```

To write a macro that doesn't compile the filename, precede the file-loading word with `[COMPILE]`. That way, `MY_MACRO` will prompt the user for a filename when it is executed.

Prompt user for filename when the macro is executed:

```
: MY_MACRO
    0 TO 7FFF EMENABLE
    0 TO 7FFF [COMPILE] BINLOAD
    STARTUP ;
```

UniLab Internals

Words that move data between PC and Emulation Memory:

```
EMREADN-MAIN   ( start_tadr end_tadr hadr --)
```

Where: `start_tadr` = start emulation memory address
`end_tadr` = end emulation memory address
`hadr` = start host address.

```
EMLOADN-MAIN   (start_hadr end_hadr tadr --)
```

Where: `start_hadr` = start host address
`end_hadr` = end host memory address
`tadr` = start emulation memory address

Read the Host's Image of the Trace

The uploaded trace buffer exists in its own host-memory region, defined by the word ABUF. (The actual data starts at ABUF plus four; the first four bytes are not valid data.) Use the words T@ and TC@ to fetch words or bytes from the trace-image area. These manage the requisite "long" fetches—just provide the offset from the start of the buffer. For example:

```
ABUF 40 + T@
```

Accesses the fortieth byte in the trace image, and leaves its value on the stack.

Data is stored in this sequence:

LADR	HADR	CONT	LDATA	HDATA	MISC

Any byte of data can be read in the trace-buffer image by specifying the cycle, the order, and the starting address of ABUF. The table below shows how to find, for cycle #n in the trace buffer, the address of each type of data:

Trace data	Label	Address of byte in cy# n
low address	A0 – A7	n * 6 + (ABUF + 4)
high address	A8 – A15	n * 6 + (ABUF + 5)
control	C0 – C7	n * 6 + (ABUF + 6)
low data	D0 – D7	n * 6 + (ABUF + 7)
high data	D8 – D15	n * 6 + (ABUF + 8)
miscellaneous	M0 – M7	n * 6 + (ABUF + 9)

Note: n is the absolute cycle number, which is not the same as the cycle numbers shown in the trace display. NORMB, for instance, numbers cycles so that cy# 0 is near the end of the trace; the absolute cycle number is directly related to a cycle's position in the trace buffer.

To see the absolute cycle numbers, filter the display by typing MISC' FILTER. This renumbers all the cycles, starting with cy# 0 at the top of the display.

Using the above table, a command to display the value in the Low Data field at cycle n could be found using the following macro.

Calculate offset into buffer, then fetch and print the data:

```
: GET_DATA       ( n -- )
      6 *        ( multiply cycle number by 6 )
      ABUF +     ( add result to start address of buffer image )
      7 +        ( add offset to low data field )
      TC@  .   ; ( fetch data and print it )
```

Now, 7C GET_DATA will print the value of the Low Data byte at cycle 7C. To leave the value on the stack (e.g., for further testing) instead of printing it, leave the period out of the definition.

Usually, the trace buffer is full when the trigger is reached, so ABUF is uploaded immediately from the UniLab to the host PC's memory. But if the trigger occurs before the buffer is full, the UniLab displays the additional number of cycles it wants before it will upload the data (Pass Count = xxx). If the buffer never fills, or if you don't need to wait for a completely full buffer, press <cr> and type the TD command; the buffer contents will be uploaded and displayed.

Manual Transmission

To start the analyzer without displaying the resulting trace, use ASTART instead of S. Use the command TRIGWAIT to wait for the current trigger and then freeze the buffer. The command ADUMP will manually upload the trace buffer into host memory, without displaying the trace.

For instance, this simple macro starts the analyzer, waits for the trigger, and then uploads the data without displaying a trace:

```
: mystart astart trigwait adump ;
```

Use <TST> to Access Memory Values in Macros

There is a way to toggle some UniLab commands to leave a number on the Forth stack, for use in macros, instead of printing them on the screen. Switch them to this mode by changing the constant <TST>.

To leave results on the stack:

```
' <TST> ON
```

To return the normal display of results:

```
' <TST> OFF
```

Use this during testing procedures or in sophisticated macros. When <TST> is "on," words that return a single value (such as MM?), and some DEBUG-specific words (such as INP) leave their results on the stack instead of printing them. The apostrophe *must* precede the <TST> when setting it on or off.

Example use of <TST>:

```
: BREAK_ON_2
     ' <TST> ON
     BEGIN
       CALC_DELAY RB 1234 MM? 27 =
     UNTIL
     ' <TST> OFF ;
```

Explanation

This will work like the loop shown in the section on BPEX, but it will stop executing breakpoints when the value at memory location 1234h is equal to 27h. Here, <TST> is set to off when the operation has finished.

Using Register Contents

All the Orion DEBUG routines have facilities for changing register contents. They vary somewhat from processor to processor, but words like =HL and =B are used to change HL and B registers to a new value.

Similarly, HL? and B? copy the values of the HL and B registers to the UniLab operating system stack, so they can be tested or compared with other values. In general, any time a register value can be changed with an =register command, it can also be retrieved with a register? command. Every =register command has an equivalent register? command in the macro-level system.

With conditional statements, you can use register contents to stop a loop. For instance, instead of using ?TERMINAL to terminate the earlier BPEX routine, we can test a register's contents for a certain value.

Execute a macro until a register holds a certain value:

```
: SHOW_ALL_CALCS
    BEGIN CALC_DELAY RB
    B? 7F = UNTIL ;
```

Explanation

Here, the B register in our hypothetical DEBUG software is tested. When it finally contains the value 7F, the routine will stop and we can inspect all data collected so far. Or, we could wait until the B register exceeds 7F.

Execute a macro until a register exceeds a certain value:

```
: SHOW_ALL_CALCS BEGIN CALC_DELAY RB
    B? 7F > UNTIL ;
```

As you get more experienced with macros, the power of the operating system will allow even more complex macros.

Change the contents of a register if a flag bit is set:

```
: ZEROB_ON_OV
  CC? 4 AND  IF 0 =B  THEN ;
```

Explanation

This routine masks off bit 2 of the value in the CC register (with the 4 AND) and, if it isn't zero, changes the contents of the B register to zero. This routine could be used in a loop that runs to a breakpoint at TEST_POINT, waits until the CC register has bit 2 set, then sets B to zero and checks to see if the A register is greater than the X register.

Run until bit 2 of CC is set and *A is greater than X:*

```
: CATCH_A>X RESET
   BEGIN
     TEST_POINT RB
     CC? 4 AND
       IF 0 =B
         A? X? >
          IF ." Routine finished" EXIT
          THEN
       THEN
   AGAIN ;
```

The same routine with comments:

```
: CATCH_A>X          ( Name of the macro is CATCH_A>X)
  RESET              ( Start with RESET enabled)
  BEGIN              ( Start loop)
   TEST_POINT RB     ( Run to breakpoint at test_point)
    CC? 4 AND        ( Check to see if CC register has bit 2 set)
      IF 0 =B        ( If CC bit 2 is set, set B register to zero,)
       A? X? >       ( then check if A register is greater than X)
        IF ." Routine finished" EXIT    ( quit if A>X)
        THEN         ( every IF conditional must end with THEN)
      THEN
  AGAIN ;            ( Go to BEGIN again if A isn't greater)
                     ( RESET is disabled after first RB command.)
```

Notes:

Notes:

FORTH REFERENCE

All Forth commands documented here are available from the UniLab's macro-level system. Stack diagrams are shown:

```
    ( input1 input2 -- output1 output2)
```

where the top of the stack is on the right. (I.e., input2 is the top item on the stack before the word is executed, output2 is the top of the stack after the word is executed.) *See the key to operands, below.*

For more information, see *All About Forth* by Glen B. Haydon (Mountain View Press). Also see "About the Macro Language" in the previous appendix. The following chart was adapted from the Forth-79 Handy Reference; it and other Forth products are available from the:

Forth Interest Group
P.O. Box 8231
San Jose, CA 95155 USA
408-277-0668

Key to Operands

d, d1	32-bit signed numbers
addr, addr1	16-bit addresses
char	7-bit ASCII character value
n, n1	16-bit signed numbers
u	unsigned 16-bit number
byte	8-bit byte
flag	Boolean flag (1 = true, 0 = false)

Stack Manipulation

DUP	(n -- n n)	Duplicate top of stack. "dupe"
DROP	(n --)	Discard top of stack.
SWAP	(n1 n2 -- n2 n1)	Exchange top two stack items.
OVER	(n1 n2 -- n1 n2 n1)	Copy second stack item to top.
ROT	(n1 n2 n3 -- n2 n3 n1)	Rotate third item to top. "rote"
PICK	(n1 -- n2)	Copy nth item to top. Thus, 1 PICK = DUP, 2 PICK = OVER.
ROLL	(n --)	Rotate nth item to top. Thus, 2 ROLL = SWAP, 3 ROLL = ROT.
?DUP	(n -- n (n))	Duplicate only if non-zero. "query-dup"
>R	(n --)	Move top item to return stack for temporary storage (use caution). "to-r"
R>	(-- n)	Retrieve item from return stack. "r-from"
R@	(-- n)	"r-fetch" Copy top of return stack onto stack.
DEPTH	(-- n)	Count number of items on stack.

Comparison

<	(n1 n2 -- flag)	True if n1 < n2. "less-than"
=	(n1 n2 -- flag)	True if top two numbers are equal. "equals"
>	(n1 n2 -- flag)	True if n1 > n2. "greater-than"
0<	(n -- flag)	True if top number is negative. "zero-less"
0=	(n -- flag)	True if top number is zero. Equivalent to NOT. "zero-equals"
0>	(n -- flag)	True if top number > zero. "zero-greater"
D<	(d1 d2 -- flag)	True if d1 < d2. "d-less-than"
U<	(un1 un2 -- flag)	Compare top two items as unsigned integers. "u-less-than"

Arithmetic and Logical

+	(n1 n2 -- sum)	Add top two numbers on stack. "plus"
D+	(d1 d2 -- sum)	Add double-precision numbers. "d-plus"
-	(n1 n2 -- diff)	Subtract (n1 – n2). "minus"
1+	(n -- n+1)	Add 1 to top number. "one-plus"
1-	(n -- n-)	Subtract 1 from top number. "one-minus"
2+	(n -- n+2)	Add 2 to top number. "two-plus"
2-	(n -- n-2)	Subtract 2 from top number. "two-minus"
*	(n1 n2 -- prod)	Multiply. "times"
/	(n1 n2 -- quot)	Divide (n1/n2). Quotient will be rounded toward zero. "divide"
MOD	(n1 n2 -- rem)	Modulo (i.e., returns the remainder from dividing n1/n2). Remainder has same sign as n1. "mod"
/MOD	(n1 n2 -- rem quot)	Divide, giving remainder and quotient. "divide-mod"
*/MOD	(n1 n2 n3 -- rem quot)	Multiply, then divide (n1*n2/n3), with double-precision intermediate. "times-divide-mod"
*/	(n1 n2 n3 -- quot)	Similar to */MOD, but gives quotient only, rounded toward zero. "times-divide"
U*	(un1 un2 -- ud)	Multiply unsigned numbers, leaving the unsigned, double-precision result. "u-times"
U/MOD	(ud un -- urem uquot)	Divide double number by single, giving unsigned remainder and quotient. "u-divide-mod"
MAX	(n1 n2 -- max)	Leave greater of two numbers. "max"

MIN	(n1 n2 -- min)	Leave lesser of two numbers. "min"
ABS	(n -- |n|)	Absolute value. "absolute"
NEGATE	(n -- n)	Leave two's complement.
DNEGATE	(d -- d)	"d-negate" Leave two's complement of double-precision number.
AND	(n1 n2 -- and)	Bitwise logical AND.
OR	(n1 n2 -- or)	Bitwise logical OR.
XOR	(n1 n2 -- xor)	Bitwise logical exclusive-OR. "x-or"

Memory

@	(addr -- n)	Replace the address on the stack by the number at that address. "fetch"
!	(n addr --)	Store n at addr. "store"
C@	(addr -- byte)	Fetch the least significant byte only. "c-fetch"
C!	(n addr --)	Store the least significant byte only. "c-store"
?	(addr --)	Display the value that is at address. "question-mark"
+!	(n addr --)	Add n to the number at addr. "plus-store"
MOVE	(addr1 addr2 n --)	Move n numbers, starting from addr1, to memory starting at addr2, if n>0.
CMOVE	(addr1 addr2 n --)	Move n bytes, starting from addr1, to memory starting at addr2, if n>0. "c-move"
FILL	(addr n byte --)	Fill n bytes in memory with byte, beginning at addr, if n>0.

Control Structures

DO ... LOOP	do:(end+1 start --)	Set up loop, given index range.
I	(-- index)	Place current loop index on data stack.
J	(-- index)	Return index of the next outer loop in the same definition.
LEAVE	(--)	Terminate loop at next LOOP or +LOOP, by setting the limit equal to index.
DO ... **+LOOP**	do:(limit start --) +LOOP: (n --)	Like DO ... LOOP, but adds stack value (instead of always 1) to index. Loop terminates when index in greater than or equal to limit (n>0), or when index is less than limit (n<0). "plus-loop"
IF ...(true)... **THEN**	If:(flag --)	If top of stack true, execute.
IF ...(true)... **ELSE** ...(false)... **THEN**	If:(flag --)	Same, but if false execute ELSE clause.
BEGIN ... **UNTIL**	until:(flag --)	Loop back to BEGIN until true at UNTIL.
BEGIN ... **WHILE** ... **REPEAT**	while:(flag --)	Loop while true at WHILE. REPEAT loops unconditionally to BEGIN. When false, continue after REPEAT.
EXIT	(--)	Terminate execution of colon definition. (May not be used within a DO ... LOOP.)
EXECUTE	(addr --)	Execute dictionary entry at compilation address (e.g., the address returned by FIND).

Terminal Input/Output

CR	(--)	Do a carriage return and line feed. "c-r"
EMIT	(char --)	Type the ASCII value from stack.
SPACE	(--)	Type one space.
SPACES	(n --)	Type n spaces, if n>0.
TYPE	(addr n --)	Type the string of n characters beginning at addr, if n>0.
COUNT	(addr -- addr+1 n)	Leave address of string (first byte at addr contains length) to TYPE from.
-TRAILING	(addr n1 -- addr n2)	Reduce character count of string at addr, to omit trailing blanks. "dash-trailing"
KEY	(-- char)	Read key and leave ASCII value on stack.
EXPECT	(addr n --)	Read n characters (or until carriage return) from terminal to address, with null(s) at end.
QUERY	(--)	Read line of up to 80 characters from terminal to input buffer.
WORD	(char -- addr)	Read next word from input stream using char as delimiter, or until null. Leave address of length byte.

Numeric Conversion

***BASE**	(-- addr)	System variable containing radix for numeric conversion.
DECIMAL	(--)	Set decimal number base.
.	(n --)	Print number with one trailing blank, and sign if negative. "dot"
U.	(un --)	Print top of stack as unsigned number with one trailing blank. "u-dot"
CONVERT	(d1 addr1 -- d2 addr2)	Convert string at addr1+1 to double number. Add to d1, leaving sum d2 and addr2 of first non-digit.
<#	(--)	Start numeric output string conversion. "less-sharp"

#	(ud1 -- ud2)	Convert next digit of the unsigned double number and add character to output string. "sharp"
#S	(ud -- 0 0)	Convert all significant digits of the unsigned double number to output string. "sharp-s"
HOLD	(char --)	Add ASCII char to output string.
SIGN	(n --)	Add minus sign to output string if n<0.
#>	(d -- addr n)	Drop d and terminate the numeric output string, leaving addr and count for TYPE. "sharp-greater"

Mass Storage Input/Output

LIST	(n --)	List screen n and set SCR to contain n.
LOAD	(n --)	Interpret screen n, then resume interpretation of the current input stream.
*SCR	(-- addr)	System variable containing screen number most recently listed.
BLOCK	(n -- addr)	Leave memory address of block, reading from mass storage if necessary.
UPDATE	(--)	Mark last block referenced as modified.
BUFFER	(n -- addr)	Leave address of a free buffer, assigned to block n; write previous contents to mass storage if UPDATEd.
SAVE-BUFFERS	(--)	Write all UPDATEd blocks to mass storage.
EMPTY-BUFFERS	(--)	Mark all block buffers as empty, without writing UPDATEd blocks to mass storage.

Defining Words

: xxx	(--)	Begin colon definition of xxx. "colon"
;	(--)	End colon definition. "semi-colon"
VARIABLE xxx	(--) xxx: (-- addr)	Create a two-byte variable named xxx; returns address when executed.
CONSTANT xxx	(n --) xxx:(-- n)	Create a constant named xxx with value n; returns value when executed.
CREATE ... **DOES>**		
	does:(-- addr)	Used to create a new defining word, with execution-time routine in high-level Forth. "does"

Dictionary

' xxx	(-- addr)	Find address of xxx in dictionary; if used in definition, compile the address. "tick"
FIND	(-- addr)	Leave compilation address of next word in input stream. If not found in CONTEXT or FORTH, leave 0.
FORGET xxx	(--)	Forget all definitions back to and including xxx.

Compiler

,	(n --)	Compile a number into the dictionary. "comma"
ALLOT	(n --)	Add two bytes to the parameter field of the most recently-defined word.
."	(--)	Print message (terminated by "). If used in definition, print when executed. "dot-quote"
IMMEDIATE	(--)	Mark last-defined word to be executed when encountered in a definition, rather than compiled.
LITERAL	(n --)	If compiling, save n in dictionary, to be returned to stack when definition is executed.
STATE	(-- addr)	System variable whose value is non-zero during compilation.
[	(--)	Stop compiling input text and begin executing. "left bracket"
]	(--)	Stop executing input text and begin compiling. "right bracket"
COMPILE	(--)	Compile the address of the next non-IMMEDIATE word into the dictionary.
[COMPILE]	(--)	Compile the following word, even if IMMEDIATE. "bracket-compile"

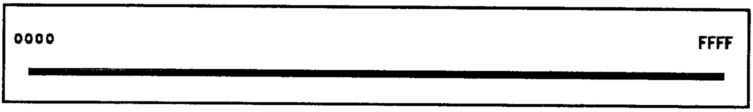
Miscellaneous

(	(--)	Begin comment, terminated by) on same line or screen; use a space after the opening parenthesis. "paren" and "close-paren"
HERE	(-- addr)	Leave address of next available dictionary location.
PAD	(-- addr)	Leave address of a scratch area of at least 64 bytes.
***>IN**	(-- addr)	System variable containing the character offset into the input buffer; used, e.g., by WORD. "to-in"
***BLK**	(-- addr)	System variable containing the block number currently being interpreted, or 0 if from terminal. "b-l-k"
ABORT	(--)	Clear the data and return stacks, set execution mode, return control to terminal.
QUIT	(--)	Like ABORT, but doesn't clear the data stack or print any message.

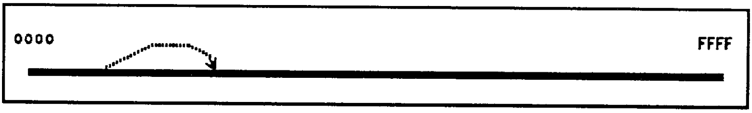
HOW DEBUG WORKS

UniLab DEBUG functions use the analyzer, emulation memory, a special idle register, and some of the target processor's resources to provide the traditional DEBUG functions of in-circuit emulators. Breakpoints can be set, the target code can be single-stepped, registers can be changed, and the contents of target RAM and ports can be displayed and modified. Because the UniLab is *not* an in-circuit emulator, it uses different techniques to do this. Following is a discussion about how this is done.

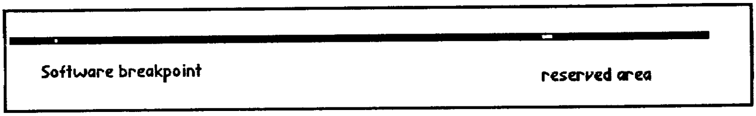
The sequence of DEBUG operations that sets a software breakpoint can be described graphically. Imagine the target memory space as a line that extends from address 0 to address FFFF:



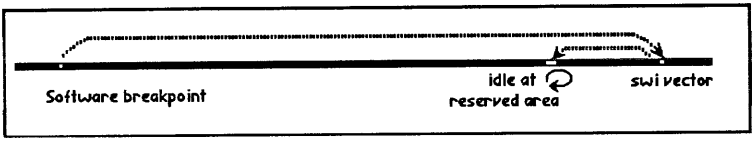
Think of your code as executing along this line, with program branches and jumps "off" and "onto" the line. For instance, a jump from address 1F80 to address 2800 would look something like this:



When you issue the command `RESET <addr> RB`, the DEBUG features install vectors for the software breakpoint and insert some code—usually a NOP and RTI—in the reserved area. The instruction at `<addr>` is copied elsewhere, then overwritten by the software breakpoint opcode. The overlay area (usually right above the reserved area) is saved also, uploaded to the host PC's memory.

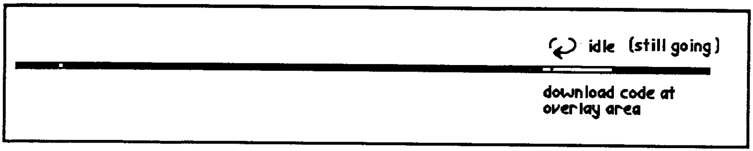


The command RB starts the analyzer, with the idle register "armed." When the software breakpoint is executed, the program will jump to the software-interrupt vector (SWI), which contains the address of the reserved area:

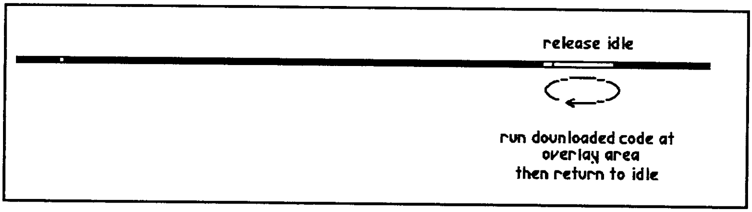


When the analyzer "sees" the address of the reserved area on the bus, it turns on the idle register. This two-byte register is placed on the data bus instead of emulation memory. It contains a "branch -2" instruction. While the idle register is turned on, the microprocessor is caught in the idle loop, and can do nothing but execute this branch instruction .

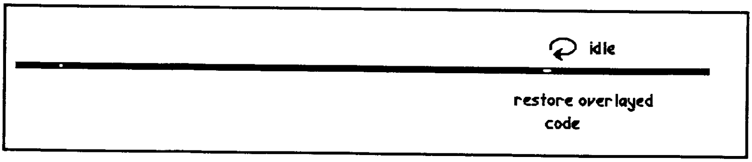
Now the UniLab prepares to gather data about the breakpoint. First, the UniLab downloads the DEBUG code into the overlay area:



Next, the UniLab releases the idle register, causing the microprocessor to run the downloaded code. A trace is captured and uploaded to the host. This provides all the data shown in the breakpoint display (register values, stack pointer, flags, program counter). Other DEBUG code segments can then be downloaded to change registers, and to read/write ports or target RAM:



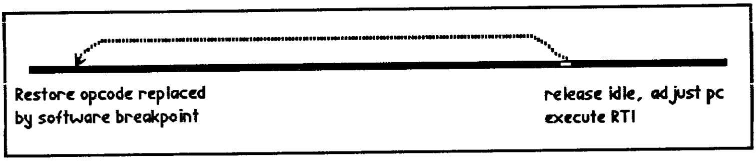
At the end of each such code segment, a branch instruction points back to the start of the reserved area. The idle register is still "armed," and puts the microprocessor into a loop again:



Finally, the original code from the overlay area is downloaded from the host and returned to its place, and the software-interrupt opcode is overwritten by the original instruction from that address. "Beneath" the idle register, waiting to be executed, is an RTI instruction, usually the second byte of the reserved area.

The processor is kept in the tight loop until another DEBUG function is executed (e.g., `STEP-OVER`, `GB`, `=register`, target RAM operation), or until the processor is released from the breakpoint (e.g., `RZ`, `G`, or `GW`).

When you release the processor from a breakpoint, the program counter is "backed up," so the RTI instruction will cause the original instruction at the breakpoint address to execute:



Hardware breakpoints work the same way, but an NMI (or IRQ) signal sends the microprocessor to the hardware vector, not to the software-interrupt vector.

EXAMINING INTERRUPT ROUTINES

Interrupt-driven code on microcontroller boards can be difficult to debug with traditional techniques. Single-stepping to examine program flow can give misleading information about programs that respond to timer counters or to externally generated interrupt service requests.

When single-stepping, the microprocessor is effectively slowed down to one instruction per step. Meanwhile, peripheral devices (including on-chip timer counters) continue to operate at full speed.

Further, this continuous servicing of interrupt requests can mask, or even cause, subtle bugs. For example, suppose a program exhibits a problem, but only if a device's interrupt occurs within a few instructions of the interrupt-enable instruction. The program and hardware design may prevent that code from being interrupted during real-time execution. During single-stepping (with interrupts enabled), that block will be interrupted immediately.

The UniLab provides several ways to analyze and debug interrupt-driven code, depending on what you want to examine. With the UniLab bus-state analyzer (to capture real-time traces) and the UniLab DEBUG features (to set breakpoints and single-step), you should be able to fix tough interrupt bugs quickly.

To watch interrupt routines in action:

1. Use a trigger to capture a real-time trace of the interrupt routines.
2. Or set a breakpoint in the interrupt routine and single-step through it.
3. With some processors, you can set a breakpoint in non-interrupt code and use `STEP-INTO` (F4) to single-step into the interrupt code.

To watch your code without seeing interrupt routines:

1. Use a filtered trigger (conditional recording) to capture a real-time trace.
2. Or set a breakpoint in non-interrupt code and use `STEP-OVER` (F3) and RB to single-step, while transparently servicing your interrupts.
3. Or set a breakpoint in non-interrupt code and disable interrupts before single-stepping with `STEP-INTO` (F4).

Real-Time Trace of Interrupts

In general, a real-time trace is the best way to examine the flow of a program, especially of interrupt routines. In its destined environment, your target will operate in real time. The best way to examine your program's behavior in real time is to capture traces with the analyzer.

When you see trouble in the program flow, you may want to set a breakpoint and examine the internal registers. But the trace should be used as the main tool for spotting such problems; the DEBUG features are used only after a program flow problem has been found.

Examples: Trigger Analyzer on Interrupt

To capture a trace of your interrupt service routine, specify the trigger with this command line:

```
<interrupt_vector_address> AS
```

AS invokes NORMT, so the trigger in this example would be near the top of the buffer. To see more of the program prior to the interrupt, type:

```
NORMM <interrupt_vector_address> ADR S
```

NORMM sets the trigger at the middle of the buffer, so the first half of the buffer will contain the cycles that preceded the interrupt, and the other half will contain post-interrupt cycles.

If resetting is enabled (with RESET), the UniLab will issue a reset pulse to the target system before starting the bus-state analyzer. If resetting is disabled (with RESET'), the bus analyzer will monitor the program in progress.

To trigger on the second execution of an interrupt-service routine, type:

```
NORMM <int_vec_addr> ADR AFTER <int_vec_addr> ADR
```

To capture an ISR after it executes a specific number of times, add n PEVENTS to the above command line. Now the analyzer can be started with S to capture a trace of the "nth + 1" execution of the interrupt routine.

To capture a trace of the interrupt-service routine after some other code executes, use the command line:

```
NORMM <int_vec_addr> ADR AFTER <other_addr> ADR
```

to delay the search for the interrupt vector address until after the other address is seen. As above, you can use PEVENTS to capture a trace of the interrupt routine after the nth execution of the other routine.

Breakpoints and Interrupts

After using the bus-state analyzer to spot a problem, you may want to set breakpoints and examine the internal registers of your processor. Keep in mind that your timer counters will continue to count down when your processor is stopped at a breakpoint, and that peripheral devices will continue to operate in real time while your processor single-steps.

Use RESET <addr> RB or the optional hardware-interrupt commands STEP-INTO (F4) or RI <trigger_spec> SI to set a breakpoint. Interrupt routines in the target system will not interfere with a breakpoint unless they are non-maskable. To examine non-maskable interrupts, stick with the analyzer techniques discussed above.

Once the processor is stopped at a breakpoint, an interrupt can occur any time, but it will not be serviced. The various DEBUG commands interact with pending interrupt requests in different ways, described below:

- In general, STEP-OVER (F3) transparently services interrupt requests, and re-establishes DEBUG control when the processor returns from the interrupt.

- <addr> RB behaves similarly. RB releases the processor from the current breakpoint and sets another software breakpoint set at the specified address. DEBUG will get a breakpoint when that address is executed, whether it is in the interrupt routine or in other code.

- RI <trigger_spec> SI releases your processor from the current breakpoint and gets a new hardware breakpoint after the specified trigger occurs.

- The hardware STEP-INTO (F4) exhibits different behaviors on different processors, depending on the structure and timing of the interrupts. For example, on the 8051 family of processors, STEP-INTO single-steps into an interrupt service routine if the request is asserted while the processor is at the breakpoint. But on 65/xx processors, STEP-INTO will "stall" at the same address when there is a pending interrupt request—at that point, you must service the interrupt (use STEP-OVER, RB, or RI ... SI) or disable interrupts.

Examples: Set Breakpoints on Interrupts

To get a breakpoint in your interrupt service routine, you have two choices. Type:

```
RESET <interrupt_service_routine_address> RB
```

to restart the target system with a software breakpoint set up at the specified address. Or use:

```
RI <int_routine_addr> ADR
```

to specify the trigger. Then use SI to start the bus-state analyzer, with the hardware interrupt "armed."

The first will get a breakpoint at the address you specify, the second will get a breakpoint (via a hardware interrupt) after the address appears on the bus.

Both RB and RI ... SI disable resetting after the breakpoint is reached.

To capture the second execution of your interrupt service routine, first set a breakpoint within the interrupt service routine and then use LP to get a breakpoint the next time that routine is executed. Or use:

```
RI <int_routine_addr> ADR 2 PEVENTS SI
```

to set a breakpoint after the second time the interrupt routine address appears on the bus.

Note that RI sets a special trigger spec that causes the address you specify to appear as the first qualifier. In general, you shouldn't use an additional AFTER in an RI ... SI trigger specification. Thus, to trigger on its second appearance, just use PEVENTS. To capture the "nth" execution of your interrupt service routine, type:

```
<n> PEVENTS
```

in the above command line. Or set a breakpoint in the interrupt service routine and use the LP command n - 1 times.

To capture a trace of the interrupt service routine after some other code executes, first set a breakpoint on the other code with RB or RI ... SI. Then set the subsequent breakpoint on the interrupt service routine.

Notes:

OBJECT- AND SYMBOL-FILE FORMATS

The UniLab software can directly load binary images and Intel hex files. All other file formats must be processed by Orion's LOADER utility, which converts object files into binary images, and converts symbolic information into the UniLab symbol file format.

The formats handled by LOADER are described in this appendix, along with the Intel hex format and the UniLab symbol-file format. See the glossary entry for LOADER for details about using the utility.

Define a .MAP File

You can use Microsoft or Orion format .MAP files, which are just ASCII files that consist of one series of records for each source file. You can have any number of records per .MAP file. (See MAPSYM for details.)

Sample Orion format .MAP file:

```
SOURCE SIMPLE1.C
2 0034
5 0040
6 0050
 <blank line>
SOURCE SIMPLE2.C
3 0055
5 0070
 <blank line>
 <blank line>
```

Explanation

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 of the source lines generated code.

Compatible Symbol Tables

2500AD
Specify file format `M` to the linker to output a Microtek-format file.

8051-family compilers
Most will output an OMF-51 file—look for the OMF-51 or Intel-compatible option.

8088- and 8086-family compilers
Most will output an OMF-86 file—look for the OMF-86 or Intel-compatible option.

8080-family compilers
Most will output an OMF-80 file—look for the OMF-80 or Intel-compatible option.

Archimedes—8051
Specify file format `-F AOMF8051` to the linker to output an OMF-51 file.

Archimedes—other processors
Specify the `-f` linker option with any Archimedes 8-bit C compiler to produce an OMF-51 output file, allowing the UniLab to provide high-level language support. If you do not want high-level lines in the symbol file—e.g., to save `LOADER` file conversion time—specify `-F MSD-I` to output a Manx file (ASCII file with a value followed by the name).

Avocet
Specify option `-PlainDump` to the linker to output an Avocet file (ASCII file with the name followed by a value).

Allen-Ashley
Produces an Avocet file (ASCII file with the name followed by a value).

Manx
Can produce an OMF86 output file, allowing the UniLab to provide full high-level language support. If you do not want high-level language lines in the symbol file—e.g., to save `LOADER` file conversion time—you can specify option `-T` to output a Manx symbol file.

Software Development Systems, Inc. (formerly Uniware)
The loader can process the output file of the SDSI cross-compiler. Specify option `-f` to the compiler to produce symbols in the output file. Set the downloader output format to "image" for a binary object file, and use the `-W` option with the downloader to prevent loading into target RAM.

OBJECT FORMATS

Intel Extended Hex format

Intel Extended Hex format represents an object image by using a fairly simple ASCII file. Intel literature also refers to this format as the Hexadecimal Object File Format.

Four types of records can occur in an Intel Hex file:

Data Record
Extended Address Record
End-of-File Record
Start Address Record

The UniLab command HEXLOAD will process the first two types, terminating when the end-of-file record occurs. The start address record is ignored.

The Extended Address Record is used to specify the "segment base address" for subsequent Data Records. If the target program stays within a single 64K physical segment, the Hex file does not need Extended Address Records.

The Data Record contains up to 255 data bytes, as well as the offset address for loading the data bytes. If there are no Extended Address Records, the offset address is the physical address (offset from the current UniLab EMSEG).

All Intel hex records consist of a single line of ASCII characters, starting with a colon (:) and terminating with a checksum, followed by a carriage return and line feed (ASCII codes 0Dh and 0Ah). Adding all the hex digits (grouped in pairs), including the checksum, gives a zero result in the low byte.

Use TYPE to view the file, and you will see the ASCII characters for the addresses, record lengths, data bytes, and checksums. Perform a binary dump of the file, and you will see the ASCII codes for the characters.

Data Record (type ØØ)

Record mark	Record length	Load address	Record type	Data	Check sum
":"			"ØØ"	(variable number of chars)	

Example data record

```
:ØEØØF8ØØ112233445566778899AABBCCDDEEØ1
```

Split into fields for readability:

```
: ØE ØØF8 ØØ 112233445566778899AABBCCDDEE Ø1
```

Explanation
This example will load ØEh bytes into memory, starting at offset ØØF8h.

Extended Address Record (type Ø2)

Record mark	Record length	Filler-- Zeroes	Record type	USBA	Check sum
":"	"Ø2"	"ØØØØ"	"Ø2"		

Example

```
:Ø2ØØØØØ2Ø9ØØF3
```

Split into fields for readability:

```
: Ø2 ØØØØ Ø2 Ø9ØØ F3
```

Explanation
This example will set the USBA (Upper Segment Base Address) to Ø9ØØ. This means that the address specification in subsequent data records will be an offset from the 20-bit absolute address Ø9ØØØh.

Start Address Record (type Ø3)

Record mark	Record length	Filler–Zeroes	Record type	CS	IP	Check sum
":"	"Ø4"	"ØØØØ"	"Ø3"			

Example

```
:Ø4ØØØØØ3FFFØØØ23E7
```

Split into fields for readability:

```
: Ø4 ØØØØ Ø3 FFFØ ØØ23 E7
```

This example will set the "start address" to FFFØ:ØØ23 (20-bit address FFF23). The UniLab's HEXLOAD command will ignore this record.

End-of-File Record (type Ø1)

Record mark	Record length	Filler–Zeroes	Record type	Check sum
":"	"ØØ"	"ØØØØ"	"Ø1"	"FF"

Example

```
:ØØØØØØØ1FF
```

Split into fields for readability:

```
: ØØ ØØØØ Ø1 FF
```

Explanation
The end of file record always looks the same. The UniLab's HEXLOAD command will terminate when it sees a record with a record-length value of zero.

Tekhex format

Standard Tektronix Hexadecimal format (Tekhex) represents an object image using a fairly simple ASCII file. (Extended Tekhex, not yet supported by Orion, includes symbolic information as well.)

Tektronix documentation refers to the different record types as "blocks." Three types of records can occur in a standard Tekhex file:

Data Record
Termination Record
Abort Record

LOADER utility will process the first type, terminating when either of the other two is encountered. The Termination Record should mark the formatted end-of-file. The Abort Record is used for "termination upon communication error" of a transmission from a host computer to a remote instrument.

Tekhex data records have two checksums. The first is for the load address and the byte count. The second checksum is for the data. The checksum equals the low byte of the sum of the hex digits (grouped in pairs).

Data Record

Record mark "/"	Load address	Byte count	First Check sum	Data (variable number of chars)	Second Check sum

Example

```
/12ØØØ719Ø1Ø2Ø3Ø4Ø5Ø6Ø71C
```

Split into fields for readability:

```
/ 12ØØ Ø7 19  Ø1Ø2Ø3Ø4Ø5Ø6Ø7 1C
```

Explanation

This example will load seven bytes into memory, starting at address 12ØØ.

Termination Record

Record mark "/"	Transfer address	Byte count "ØØ"	Check sum

Example

```
/12340046
```

Split into fields for readability:

```
/ 1234 ØØ 46
```

Explanation
The transfer address for this record is the starting address of the code. LOADER ignores the transfer address.

Abort Record

Record mark "/"	Second slash mark "/"	ASCII message alerting user to cause of termination.

Example

```
// HANDSHAKE FAILURE, TRANSMISSION ABORTED
```

Explanation
The LOADER terminates when it encounters an abort record.

Motorola Hexadecimal format (S-records)

Motorola Hexadecimal format represents an object image by using a fairly simple ASCII file. It is popularly called the Motorola S-record format, because the character "S" marks the start of a record, which never contains more than 80 characters. Adding all the hex digits (grouped in pairs), including the checksum, gives the result FFh in the low byte; this doesn't include the record mark. Motorola documentation refers to groups of records as "blocks."

There are ten S-record types:

S0	group header record
S1	data and 16-bit load address
S2	data and 24-bit load address
S3	data and 32-bit load address
S5	number of records in the group
S7	termination record for a group of S3 records
S8	termination record for a group of S2 records
S9	termination record for a group of S1 records

Orion's LOADER utility will process the S1, S2 and S3 records. The others will be ignored, any symbol information they contain discarded.

Generalized S-record

Record mark	Record length	Load address	Data	Check sum
"S"X		4, 6, or 8 chars	(variable number of chars)	

Example

```
S1Ø8Ø3ØØ1122334455F5
```

Split into fields for readability:

```
S1 Ø8 Ø3ØØ 1122334455 F5
```

Explanation

This S1 record is eight bytes long, not including the record mark or length. It loads five bytes, starting at address Ø3ØØ.

SYMBOL FORMATS

Manx symbol file

The Manx symbol file represents symbolic information in a simple ASCII format. This format is produced by the Manx Aztec C compiler. Many other compilers and assemblers produce similar or identical files.

The Manx format—Each line in the file defines a single symbol, and consists of a four-digit hex value, followed by a space (one or more blanks) and a variable-length name. Every line ends with a carriage return and line feed (ASCII codes ØDh and ØAh).

The file terminates with an end-of-file mark (ASCII code 1Ah) on a line by itself. The loader will not process any symbol information that is on the same line as the end-of-file mark.

Example Manx-format symbol file:

```
ØØ23 firstsym
ØØ67 another_sym
ØØ1Ø third_sym
ØØ45 last-one
```

Binary dump of example:

```
30 30 32 33 20 66 69 72 73 74 73 79 6D 0D 0A 30      0023 firstsym..0
30 36 37 20 61 6E 6F 74 68 65 72 5F 73 79 6D 0D      067 another_sym.
0A 30 30 31 30 20 74 68 69 72 64 5F 73 79 6D 0D     .0010 third_sym.
0A 30 30 34 35 20 6C 61 73 74 2D 6F 6E 65 0D 0A     .0045 last-one..
1A                                                   .
```

Avocet symbol file

The Avocet symbol file represents symbolic information in a simple ASCII format. This format is produced by the Avocet cross-assemblers. Many other compilers and assemblers produce similar or identical files.

The Avocet format—Each line in the file defines a single symbol, and consists of a variable length name, followed by a space (one or more blanks) and a four-digit hex value. Every line ends with a carriage return and line feed (ASCII codes ØDh and ØAh).

The file terminates with an end-of-file mark (ASCII code 1Ah) on a line by itself. The loader will not process any symbol information that is on the same line as the end-of-file mark.

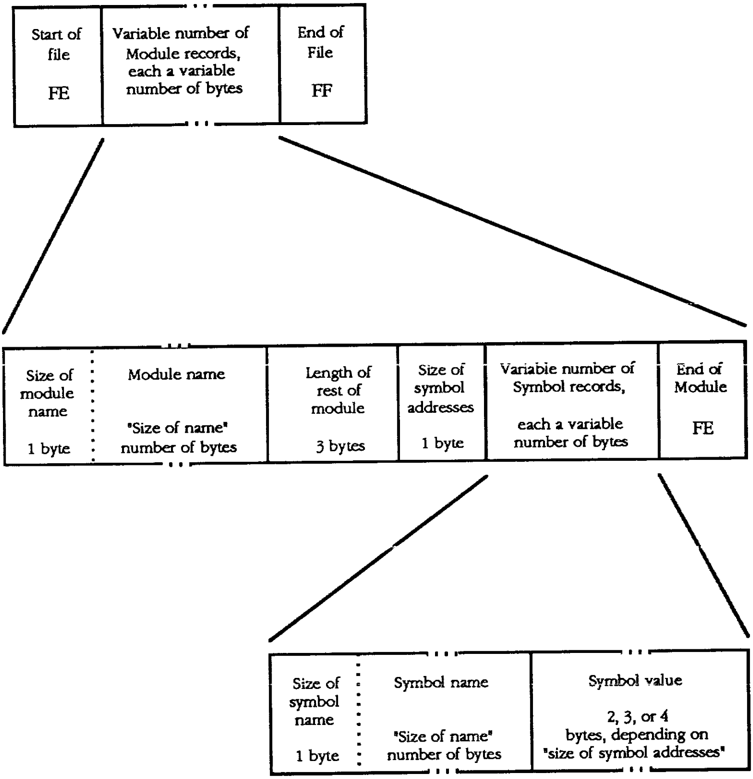
Example Avocet-format symbol file:

```
firstsym    1023
another_sym 2067
third_sym   3010
last-one    f045
```

Binary dump of example:

```
66 69 72 73 74 73 79 6D 20 20 20 20 31 30 32 33      firstsym  1023
0D 0A 61 6E 6F 74 68 65 72 5F 73 79 6D 20 20 32      ..another_sym 2
30 36 37 0D 0A 74 68 69 72 64 5F 73 79 6D 20 20      067..third_sym
20 20 20 33 30 31 30 20 0D 0A 6C 61 73 74 2D 6F      3010 ..last-o
6E 65 20 20 20 20 20 20 20 66 30 34 35 0D 0A 1A      ne    f045...
```

Microtek symbol file (produced by the 2500AD assembler)
The Microtek format file is not in ASCII format:



Start of file FE	Variable number of Module records, each a variable number of bytes	End of File FF

Size of module name 1 byte	Module name "Size of name" number of bytes	Length of rest of module 3 bytes	Size of symbol addresses 1 byte	Variable number of Symbol records, each a variable number of bytes	End of Module FE

Size of symbol name 1 byte	Symbol name "Size of name" number of bytes	Symbol value 2, 3, or 4 bytes, depending on "size of symbol addresses"

Legitimate values for "size of symbol addresses":

2 = 16-bit address, stored as low byte, high byte.
3 = 24-bit address, stored as high byte, middle byte, low byte.
4 = segmented address stored as low,high of segment, then offset.
5 = 32-bit address, stored in high-to-low order.

SYMFIX Formats

The LOADER option SYMFIX allows you to load a wide variety of fixed-record-length files. This option is used with files that consist only of a series of records, each of which has the same format and length. (SYMFIX cannot skip header information in files.)

The SYMFIX arguments describe fixed-file formats:

```
    <a> <b> <c> <d> <e> <f> SYMFIX
```

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

**Note:* Parameter a is "zero-indexed"; if the name field starts on the third byte, use 2. Parameter c is "one-indexed"; if the address field starts on the third byte, use 3. Parameter f is a count; if there are five bytes in the record, use 5.

Example spec for a 2500AD abbreviated symbol-table file:

```
C> LOADER 0 0 B 1 0 E SYMFIX MYFILE.SYM
```

Explanation

The above example will properly process files that follow this format:

no offset from start-of-record to start-of-name (a=0)
ten (decimal) bytes for the symbol name
two bytes for the binary, MSB-first symbol value (b=0 and d=0)
two pad bytes, for a total of 0Eh bytes (f=0E)
the symbol value, thus, starts at offset 0Bh in the record (c=0B)
the pad bytes are value 00 (e=00)

Binary dump of example:

```
4E 45 58 54 00 00 00 00 00 00 08 2B 00 00 53 54        NEXT.......+..ST
42 31 00 00 00 00 00 00 08 36 00 00 42 55 52 4E        B1 .....6..BURN.
00 00 00 00 00 00 08 53 00 00 42 4C 4F 4F 50 00        .......S BLOOP..
```

This is a dump of the 2500AD format. SYMFIX can handle many other fixed-file formats as well.

Software Development Systems, Inc. Files

The Software Development Systems, Inc. output file format is proprietary and will not be described here. The .OUT file contains object code, line-number information, and global symbols. They have a very flexible symbol and line handler to produce expanded symbol and line information. The loader will provide a symbol format readable by the UniLab.

MIXED SYMBOL AND OBJECT FORMATS

OMF-51 (Object Module Format for 8051 family)
An OMF-51 file represents symbolic information, debug information, and data records. The file is not in ASCII form. It is also referred to as AOMF-51 (i.e., Ashling, or Absolute, OMF-51). This rather complicated format has become the *de facto* standard for encoding 8051-family object modules. Most compiler vendors now support OMF-51, for the sake of compatibility with Intel tools.

The OMF-51 specification includes "relocatable" records and absolute records. Orion's `LOADER` supports only the absolute records. (Relocatable records have not yet had their absolute locations in memory determined.) Use a linker/locator package to translate a file with relocatable and "fixup" records into a file with only absolute records.

No documentation on OMF-51 is publicly available. A close relative of the format is described in a 40-page Intel publication:
MCS 80/85 Relocatable Object Module Formats
Order #121747-001

OMF-86 (Object Module Format for 8086 family)
An OMF-86 file represents symbolic information, debug information, and data records. The file is not in ASCII form. This rather complicated format has become the *de facto* standard for encoding 8086-family object modules. Most compiler vendors now support the OMF-86.

The OMF-86 specification includes "relocatable" records and absolute records. Orion's `LOADER` supports only the absolute records. (Relocatable records have not yet had their absolute locations in memory determined.) Use a linker/locator package to translate a file with relocatable and "fixup" records into a file with only absolute records.

OMF-86 is described in a 120-page Intel publication:
8086 Relocatable Object Module Formats
Order # 121748-ØØ1

For a good article on OMF-86:
PC Tech Journal, pp. 62 – 81, October 1985, Vol. 3 No. 10.
Focuses on the use of OMF in the MS-DOS environment, but serves as a good introduction to the format.

OMF-80 (Object Module Format for 8080 family)
No documentation on OMF-80 is publicly available.

UNILAB SYMBOL-FILE FORMAT

Here, the normal UniLab file facilities are used to reveal the symbol format. If you understand the basic UniLab commands, this will show how to investigate the symbol file's properties for yourself.

```
COMMAND> clrsym
COMMAND> 100 is one_hundred
COMMAND> 200 is two_hundred
COMMAND> 40 is forty
COMMAND> 25 is twenty-five

COMMAND> symlist
sym0001 0025 TWENTY-FIVE
sym0002 0040 FORTY
sym0003 0100 ONE_HUNDRED
sym0004 0200 TWO_HUNDRED

COMMAND> symsave xx.sym
COMMAND> 0 7ff 0 mfill
COMMAND> 0 7ff binload xx.sym end = 67

COMMAND> 0 6f mdump
 0  9E 92 01 00 40 00 08 00 00 00 00 00 00 00 00 00      ....@...........
10  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00      ................
20  9E 92 01 00 00 00 01 0B 4F 4E 45 5F 48 55 4E 44      ........ONE_HUND
30  52 45 44 01 00 00 00 02 0B 54 57 4F 5F 48 55 4E      RED.....TWO_HUN
40  44 52 45 44 01 00 00 40 00 05 46 4F 52 54 59 01      DRED...@..FORTY.
60  2F 00 24 00 02 00 13 00 00 00 00 00 00 00 00 00      /.$.............
```

The four parts of the header of a UniLab-format symbol file:

SYMBOL FILE ID# (this is also at 20 bytes into file):

```
0  9E 92 01 00 40 00 08 00 00 00 00 00 00 00 00 00          ....@...........
```

VERSION NUMBER (currently 1; check your system for current number—it may change.)

```
0  9E 92 01 00 40 00 08 00 00 00 00 00 00 00 00 00          ....@...........
```

POINTER START OF INDEX (minus 20-byte header):

```
0  9E 92 01 00 40 00 08 00 00 00 00 00 00 00 00 00          ....@...........
```

NUMBER OF SYMBOLS (times 2):

```
0  9E 92 01 00 40 00 08 00 00 00 00 00 00 00 00 00          ....@...........
```

The UniLab format for symbol records:

The first number is the symbol type—symbols defined by `IS` or translated by the `LOADER` are of type 01.

```
20    01 00 00 00 01 0B 4F 4E 45 5F 48 55 4E 44          ........ONE_HUND
30    52 45 44
```

The next 32 bits are the symbol value. The name follows in ASCII, preceded by a count byte (B characters in this example).

```
20    01 00 00 00 01 0B 4F 4E 45 5F 48 55 4E 44          ........ONE_HUND
30    52 45 44
```

At the end of the file is an index to all symbols. Here, the first symbol starts 002F bytes after the header, the second at 0024 bytes after, the third at 0002 bytes after, and the last at 0013 after.

```
60    2F 00 24 00 02 00 13 00 00 00 00 00 00 00 00 00    /.$.............
```

Note: The index has the symbols sorted by value. If you make or translate a symbol file for the UniLab with your own program, you ***must*** sort them by value for the symbol interface to function properly.

Notes:

HIGH LEVEL LANGUAGE SUPPORT

UniLab has several features that help you debug code written in high level languages such as C. You can use function names and global variable names[1]† and refer to statements (by line number) in your source code. This dramatically simplifies debugging. In addition, UniLab Source Tracking automatically displays the appropriate sections of your source file while you work. You will no longer have to search through hard-copy listings, cross reference maps, and link tables.

In general, the main requirement for UniLab high-level language support is that the compiler generates an extended symbol table that relates lines in your source text with addresses in the compiled code. The addresses represent the code that is generated by these high level instructions. (On-line help is provided for each processor to help you create the proper file format with your compiler and on how to load this high level information into the UniLab.)

The UniLab can correlate lines from your source text with the compiled code while you use the analyzer, inspect or modify memory, and perform DEBUG functions.

The UniLab treats these high level line "tags" as special types of symbols. When loaded into the UniLab they appear in the symbol table along with normal symbols. There are two different types of high level symbols:

1. Addresses which point to lines in a source module.
In the symbol table these appear as
```
sym00AC   0657   L0171_M5 of module TEST51.C
```
This tag shows that symbol #00AC is address 0657 in the compiled code and was generated by line 171 in source file number 5 (named TEST51.C).

2. Names of source modules.
In the symbol table these appear as
```
sym00BA   0005     module TEST51.C
```
This tags the source file named TEST51.C as file number 5. The number 5 is a cross-reference "ID number" used by

†Note: every UniLab system can support high level symbols, however, not every cross-compiler generates this type of information. Check information in your processor-specific Application Notes (located in each PPAK) and on the LOADER to see which compilers are supported for your processor.

UniLab to identify the different source files. You can have many source modules. When the UniLab shows a line in a disassembly or at a breakpoint, the source file is opened and the line is printed.

Lines from the source text will be shown just like symbols when disassembling memory, viewing a trace, or at breakpoints.

High level source lines with disassembly from memory:

```
MAIN          03C8 90000C    MOV DPTR,#C
              03CB 120323    LCAL ?ENTER_L17
L0050_M3     set_timers();  /* setup the timers */
              03CE 7900      MOV R1,#0
              03D0 1203B7    LCAL SET_TIMERS
L0051_M3     set_int0();     /* setup the int pin */
              03D3 7900      MOV R1,#0
              03D5 1203A8    LCAL SET_INTO
L0057_M3     time = 0x800;  /* set default delay   */
              03D8 7B00      MOV R3,#0
              03DA 7A08      MOV R2,#8
```

High level source lines in the trace display:

```
 cy# CONT                 ADR DATA
  82  7F                  0359 C0F0     PUSH B_
  86  7F                  035B 22       RET
L0037_M3   output(TCON,0x51); /* set t0,1    */
  8A  7F                  03BD 7B51     MOV R3,#51
  8C  7F                  03BF 8B88     MOV TCON,R3
L0038_M3    output(TMOD,0x10);  /* t1=8-bit,t0=13b*/
  90  7F                  03C1 7B10     MOV R3,#10
  92  7F                  03C3 8B89     MOV TMOD,R3
L0039_M3               }
  96  7F                  03C5 02035C   LJMP ?LEAVE_L17
  9A  7F ?LEAVE_L17       035C D006     POP 6
  9E  7F                  035E D007     POP 7
  A2  7F                  0360 D0E0     POP ACUM
  A6  7F                  0362 F581     MOV SP,A
  A8  7F                  0364 C007     PUSH 7
  AC  7F                  0366 C006     PUSH 6
  B0  7F                  0368 22       RET
L0051_M3   set_int0();                /* setup the int pin */
  B4  7F                  03D3 7900     MOV R1,#0
  B6  7F                  03D5 1203A8   LCAL SET_INTO
  BA  7F SET_INTO         03A8 900000   MOV DPTR,#RESET_
  BE  7F                  03AB 120323   LCAL ?ENTER_L17
```

High level line source line at a DEBUG breakpoint:

```
SP=2F A=2C DPTR=0002 IE=60 PSW=01 (cafbbv-P)
 R0=DD R1=00 R2=06 R3=C3 R4=06 R5=2A R6=00 R7=2B T0=0    T1=0
 B_=05 P0=DD P1=FF P2=05 P3=FF IP=E0  CON=00  TCON=00   TMOD=00
L0150_M4    set_timers();  /* setup the timers */
             05DD 7900      MOV R1,#0
```

You may also use the line numbers as symbols. For instance, you can use L0171_M5 instead of an address to set a trigger spec or to get a breakpoint. If you use the Debug PopUp to set a breakpoint, you can enter L0171_M5 when you are requested to enter the address.

Normally you will not have to invoke any command or special feature to enable high-level support. Simply load the symbol table after loading your program code into the UniLab. This feature is enabled by default. If for some reason you wish to turn it off, use the command SOURCE' (the ending apostrophe is part of the word). To re-enable, use the command SOURCE.

IMPORTANT: In order for the UniLab to open the proper file and show the proper lines, the source file must be in the current directory.

Source Tracking

The UniLab's Source Tracking feature allows you to see automatically more than a single line from the source file when you are disassembling, using the analyzer or DEBUG. Source tracking automatically correlates the compiled code with your source text.

In order to use Source Tracking, select SPLIT screen display mode (F2). Use the lower window to do your normal activities. When a source line appears in a trace, disassembly or at a breakpoint, the upper window will show an entire section from the source file. You will see the section of your source code which generated the compiled code in the upper screen window while you inspect and debug the actual code in the lower window. You won't have to sift through reams of print-outs of your source code while you are debugging and analyzing.

If you wish to use the upper window for other purposes, you can turn off the Source Tracking feature with the command TRACK' (don't forget the apostrophe). Re-enable with TRACK.

Single Stepping through Source Lines

As mentioned above, the UniLab can set breakpoints on line numbers. You can set a breakpoint on any line that is executed in your target code and use line numbers to begin execution at other places in your code.

In order to DEBUG high-level code quickly, the UniLab has a STEP-LINE method of breakpointing. Like the STEP-OVER function which sets a breakpoint on the next instruction in memory, STEP-LINE sets a breakpoint on the next address tagged by a high level line symbol. STEP-LINE is assigned to function key Alt-F4.

You must be careful using this feature. Its primary use is to get quickly to a point in the code, so you then can use the lower level single-step routines to analyze the compiled code in detail. If you use a structure such as

```
Line100 (expression)
Line101       if  (expression)
Line102       else  (expression)
Line103  (expression)
```

and you use Alt-F4 to single step, you will probably get to a point where DEBUG returns the message "Waiting for address xxxx (Line101_M3)". Like STEP-OVER (F3), STEP-LINE (Alt-F4) sets a breakpoint at the next higher expression. If the expression never executes, then you will not get a breakpoint. In the example above, if you are at a breakpoint at Line 100, then used STEP-LINE, a breakpoint would be set at Line101. At Line101 if you use STEP-LINE again, the next breakpoint would be set at Line 102. If the "if" part of the statement is taken, the "else" part will not be executed. If you want to step through this kind of code, use STEP-INTO (F4) to follow the low level execution of the processor until you get to the line tagged by the next high level symbol.

Here is an example of `STEP-LINE`. We'll begin at a breakpoint set on the symbol `MAIN`:

```
SP=2C A=00 DPTR=06C3 IE=60 PSW=00 (cafbbv-p)
 R0=08 R1=00 R2=06 R3=C3 R4=06 R5=C3 R6=00 R7=2A T0=0  T1=0
 B_=00 P0=D7 P1=FF P2=05 P3=FF IP=E0 SCON=00  TCON=00   TMOD=00
MAIN          05D7 900002   MOV DPTR,#2

Command> STEP-LINE (or press Alt-F4)
SP=2F A=2F DPTR=0000 IE=60 PSW=01 (cafbbv-P)
 R0=9D R1=00 R2=06 R3=10 R4=06 R5=2F R6=05 R7=E2 T0=0 T1=0
 B_=04 P0=E2 P1=FF P2=05 P3=FF IP=E0 SCON=00  TCON=71   TMOD=10
L0151_M4          set_int0();      /* setup the interrupt pin */
              05E2 12CD03   LCALL SET INT0
```

Line 151 from source module 4 was the next address tagged with a high level line number in the symbol table.

IN

Intel
3065 Bowers Ave.
Santa Clara, CA 95051
(800) 874-6835

Description
C cross compiler. Must also purchase the RLL (relocate, link, load) utility to create ROMable code.

Symbol support
After linking and locating the code, load the absolute object module with option "OMF86" of Orion's LOADER.

IT

Introl
647 West Virginia St.
Milwaukee, WI 53204
(414) 276-2937

Description
C cross-compiler, with assembler included.

Symbol support
Produces "AVOCET" or "MANX" symbol files, with the Introl symbol-name lister ISYM. Which to use is a matter of personal taste. Consult the Introl documentation if more details are needed.

AVOCET: First, make an ISYM command file "avo.sym" with the contents:

```
%n   %c(Ø4H)
```

Then, use the following command to produce a symbol file that can be loaded by option "AVOCET" of Orion LOADER:

```
isym -cavo.sym <obj. file>
[,<obj. file>]  >  <outfile>
```

MANX: First, make an ISYM command file "mnx.sym" with these contents:

```
%c(Ø4H) %n
```

Then, use the following command to produce a symbol file that can be loaded by option "MANX" of Orion's LOADER:

```
isym -cmnx.sym <obj. file>
[,<obj. file>]  >  <outfile>
```

SOURCES OF CROSS-ASSEMBLERS & C CROSS-COMPILERS

The UniLab software is designed to work with any assembler or compiler. To load the target program, however, the UniLab first needs the object code in binary or Intel hex format.

The UniLab LOADER utility converts Motorola S-records, TEKHEX files, OMF51, OMF80, and OMF86 into binary files. It also loads many common symbol file formats. See the appendix on object and symbol file formats.

As a service to our users, we have compiled this list of inexpensive cross-assemblers and compilers. The two-character abbreviations refer to sources listed on the following pages.

For Aztec C, see MANX Software Systems (MX).
For UniWare, see Software Development Systems (SD).

Cross-Assemblers

(Following pages explain vendor codes across top row.)

	25	AA	AV	CY	EN	MT	SD	UN
1802/5	•	•	•		•		•	
6301	•	•	•	•	•		•	
6305		•	•			•		
HD64180	•	•	•		•	•	•	
6502	•	•	•		•	•		
6800/2/8	•	•	•		•		•	
6801/3	•	•	•	•	•		•	
6805	•	•	•		•		•	
6809	•	•	•		•		•	
68000	•		•		•		•	•
68HC11	•	•	•	•	•		•	
NSC800	•	•	•		•			
8048-50/41	•	•	•	•	•			•
8051/31	•	•	•	•	•		•	•
8080	•	•	•	•	•			•
8085	•	•	•	•	•		•	
8086/8	•			•			•	•
8096	•	•	•	•	•		•	
Z8	•	•	•	•	•		•	
Z-80	•	•	•		•	•	•	
Z-8000	•				•			•
SUPER 8	•	•						
Symbol support?	Y	Y	Y	N	Y	N	Y	?

C Cross-Compilers

(Following pages explain vendor codes across top row.)

	25	AR	BC	FR	IM	IN	IT	MC	MT	MX	OK	SD	UN	WI
1802/5							none							
6301	●	●					●				●			
6305			●											
HD64180		●								●		●		
6502	●									●				
6800/2/8														●
6801/3	●	●					●				●			●
6805			●											
6809	●				●		●							●
68000	●				●		●		●	●		●	●	
68HC11		●			●		●							●
NSC800	●								●			●		
8048-50/41							none							
8051/31	●	●		●				●			●			
8080							none							
8085									●					
8086/8					●	●			●	●			●	
8096		●												
Z8								●						
Z-80	●	●			●					●		●		
Z-8000													●	
SUPER 8								●						
Symbol support?	Y	Y	?	Y	?	Y	Y	?	N	Y	N	Y	Y	?

Vendor List

Contact the vendor directly for recent information. This list is a service to our customers, and does not constitute a recommendation.

25 2500AD Software Inc.
P.O. Box 480
Buena Vista, CO 80017
(719) 395-8683
(800) 843-8144

Description
They include recursive macros, nested conditional assembly, listing control, and a linker. C cross-compilers available, too.

Symbol support
Specify file format M to 2500AD linker. Produces symbol files loadable by option "2500AD" of Orion's LOADER.

AA Allen Ashley
395 Sierra Madre Villa
Pasadena, CA 91107
(818) 793-5748

Description
Resident editing capability, assembly to memory. Macro/relocatable versions also available.

Symbol support
Produces symbol files loadable by option "AVOCET" of Orion LOADER.

AR Archimedes
1728 Union St.
San Francisco, CA 94123
(415) 567-4010

Description
C cross-compiler for 8051. Also, Z80/64180, 6301/3, 6801/3, 68HC11, 8096.

8051 symbol support: Specify file format -F AOMF8051 to the Archimedes linker. Produces symbol files loadable by option "OMF51" of Orion's LOADER.

Other processors: All of the processors supported by Archimedes can be linked to produce an OMF51 output file. This format is preferable, since it contains all symbols and line number information. You may also specify file format -F MSD-I to the linker to produce symbol files which can be loaded via the "MANX" option of Orion's LOADER.

AV

Avocet Systems Inc.
120 Union Street
P.O. Box 490
Rockport, ME 04856
(800) 448-8500

Description
CP/M-80 or MS-DOS versions.

Symbol support
Specify file format -PlainDump to the Avocet linker. Produces symbol files loadable by option "AVOCET" of Orion's LOADER.

BC

Byte Craft
421 King Street North
Waterloo, ON N2J 4E4
Canada
(519) 888-6911
FAX: (519) 746-6751

Description
6805 C cross-compiler.

Symbol support
Planned for future. Contact Orion for details.

CY

Cybernetic Micro
Systems
P.O. Box 3000
San Gregorio, CA 94074
(415) 726-3000

Description
Conditionals, macros. Written in 8088 assembler.

Symbol support
Not known.

EN

Enertec Inc.
19 Jenkins Ave.
Lansdale, PA 19446
(215) 362-0966

Description
Cross-assembler.

Symbol support
Produces ASCII-format symbol files readable by the loader. Enertec calls this option the "Orion" symbol file.

FR

Franklin Software, Inc.
888 Saratoga Avenue, #2
San Jose, CA 95129
(408) 296-8051

Description
C cross-compiler for 8051.

Symbol support
Produces OMF51 output files for high-level support by the UniLab.

IM

InterMetrics
733 Concord Ave.
Cambridge, MA 02138
(617) 661-1840

Description
C cross-compiler.

Symbol support
Not known.

IN

Intel
3065 Bowers Ave.
Santa Clara, CA 95051
(800) 874-6835

Description
C cross compiler. Must also purchase the RLL (relocate, link, load) utility to create ROMable code.

Symbol support
After linking and locating the code, load the absolute object module with option "OMF86" of Orion's LOADER.

IT

Introl
647 West Virginia St.
Milwaukee, WI 53204
(414) 276-2937

Description
C cross-compiler, with assembler included.

Symbol support
Produces "AVOCET" or "MANX" symbol files, with the Introl symbol-name lister ISYM. Which to use is a matter of personal taste. Consult the Introl documentation if more details are needed.

AVOCET: First, make an ISYM command file "avo.sym" with the contents:

```
%n  %c(Ø4H)
```

Then, use the following command to produce a symbol file that can be loaded by option "AVOCET" of Orion LOADER:

```
isym -cavo.sym <obj. file>
[,<obj. file>]  >  <outfile>
```

MANX: First, make an ISYM command file "mnx.sym" with these contents:

```
%c(Ø4H) %n
```

Then, use the following command to produce a symbol file that can be loaded by option "MANX" of Orion's LOADER:

```
isym -cmnx.sym <obj. file>
[,<obj. file>]  >  <outfile>
```

MC MicroComputer Control
(609) 466-1751

Description
C cross-compiler for 8051, Super 8, Z80.

Symbol support
Not known.

MS Microsoft
10700 Northup Way
Bellevue, WA 98004

Description
C compiler for 8086, 8088.

Symbol support
Can load .MAP file with UniLab command MAPSYM. But you must manipulate the output file (normally in .EXE format) to make it suitable for embedded controller applications, or purchase a linker from another company. The linker/locater from Systems&Software, for example, will output an absolute OMF-86 format file, which can be loaded by Orion's LOADER.

Link&Locate from:
Systems&Software
3303 Harbor Blvd., C11
Costa Mesa, CA 92626
(714) 241-8650

PC-Locate from:
Aldia Systems
P.O. Box 37634
Phoenix, AZ 85069
(602) 866-1786

Phar Lap Software
60 Aberdeen Avenue
Cambridge, MA 02138
(617) 661-1510
Their product generates ROM-able code from other manufacturers' C compilers and assemblers.

For some details of putting Microsoft code into ROM, and a LOCATE utility, see "Putting ROM Code in its Place," by Rick Naro, *Dr. Dobb's Journal*, December 1987.

MT

Microtec Research
Box 60337
Santa Clara, CA 94088
(408) 733-2919

Description
C cross compilers for 68000, 68008, 68010, 68020, 8085, Z80, 64180, 8088, 8086, 80188.

Symbol support
Produces OMF86 output for the 8088/8086 family.

MX

Manx Software Systems
One Industrial Way
Eatontown, NJ 07724
(800) 221-0440

Description
C cross-compilers for 8086, 68000, 8080, Z80, 6502.

Symbol support
Specify file format -T to linker to produce symbol files readable with the "MANX" option of Orion's LOADER. OMF86 output is available for the 8088/8086 family.

OK

Okapi Systems, Inc.
P.O. Box 3095
Everett, WA 98203
(206) 258-1163

Description
C cross-compilers for 8051, 6801/3, 6301/3.

Symbol support
No OMF51 output files produced at this time.

SD

Software Development
Systems
(formerly UniWare)
3110 Woodcreek Dr.
Downers Grove, IL 60515
(312) 971-8170
(800) 448-7733

Description
Range of 8- and 16-bit cross-assemblers. C cross-compiler for 68000 and Z80 family (NSC-800 and 64180).

Symbol support
Specify option `-f` to the SDS compiler to produce symbols in the output file. See previous discussion.

UN

Unidot Inc.
602 Park Point Dr. #225
Golden, CO 80401
(303) 526-9263

Description
C cross-compilers.

Symbol support
Not known.

WI

Wintek
1801 South St.
Lafayette, IN 47904
(317) 742-8428

Description
C cross-compilers.

Symbol support
Not known.

Notes:

UNILAB SPECIFICATIONS

Host Computer Interface

Parallel Orion bus, card fits in short slot.
IBM PC 5 1/4", MS-DOS format diskettes.

Emulator

- Download in five seconds for 64K binary file from AT hard disk.
- 150 ns. max. access time emulation ROM memory, 32K x 8-bit or 16K x 16-bit standard (128K bytes optional).
- Individual 2K segments selectable in any combination in a 128K range.
- 20-bit enable address decoding.
- Standalone 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 target-register and -memory display/change, breakpoints, single step, next step, and a real-time display).
- 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.
- Extensive filtering, 2730 bus-cycle trace buffer
- 48 data inputs (two groups of eight can be separately clocked).
- Four clock signal inputs, gated to form one bus clock: 297 ns. minimum cycle. Clock-edge filter prevents retrigger before 100 ns.
- 1 microsecond crystal-controlled timer.
- Address demultiplexing latches (also used by emulator).

Analyzer Trigger

- Four-step sequential trigger.
- RAM truth tables allow search for any eight-bit function/value at each of six input groups, for each of four trigger steps.
- Eight truth tables x four-step trigger = thirty-two (256-bit tables).
- 16-bit inside/outside range detection on address lines.
- Four-bit segment enable gives 20-bit address capability. 20-bit single address detection, or 16-bit range detection, in any four-bit segment.
- Pass Counter: wait up to 65,382 events or cycles before fourth step.
- Before/After at pass-count trigger enable.
- Delay Counter: wait up to 65,382 events or cycles, then stop the trace.
- Filter, records only cycles that match the trigger.
- Oscilloscope-sync output on trigger.
- Interrupt-target output on trigger, (if enabled).
- LED indicates waiting for trigger, standalone operation supported while waiting.

Software Features

PopUp or command driven, with single context for all four instruments:

- 48-Channel Bus-State Analyzer
- In-Circuit Emulator
- PROM Programmer
- Stimulus Generator
- Line-by-line assembler
- High-level language support
- Supports common cross-assembler symbols, provides a loader utility for Intel OMF files, Motorola S records, and Tektronix Hex files.
- Extensible macro capability.
- Cursor-key control of text and trace display
- Pop-up, mode-switch displays
- User-definable split-screen displays
- On-Line Help, with glossary
- 40 user-definable, soft keys
- Bonus on-screen features: calculator, ASCII table, IC pinout library, memos/messages, direct DOS access, EGA/ECD support (or use monochrome CRT).

Software Options

- Program Performance Analyzer
- target-specific Disassembler/DEBUG software

EPROM Programmer

- Smart programming algorithm for high speed.
- 28-pin Textool (zero-insertion-force socket) handles 24- and 28-pin devices.
- Programs single-supply EPROMs.
- See *User's Guide* chapter "EPROM Programmer" for a detailed list of supported PROM/EPROM devices.
- Optional programming module available for 27512, 2716B, 2732B.

Signal Inputs

- TTL logic levels (74ALS inputs).
- 0.1 ma. maximum loading, includes emulator and analyzer.

Signal Outputs

- TTL logic levels (74LS244 outputs).
- 100 ohms forward-terminating resistors on Emulator data lines.
- Reset output (RES-): open collector, 7406 through 47 ohms.
- Interrupt output (NMI-): open collector, 7406, low true.
- Nine Stimulus outputs (at EPROM socket): 8255 NMOS outputs.

Physical

Size: 2.1" high 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 briefcase.

Power

100 kHz switching supply
110V + 10% 50/60 Hz 15 watts (standard)
220V + 10% 50/60 Hz 15 watts (optional)

Host Computer Requirements

IBM-compatible PC, XT, AT, PS/2
MS-DOS 2.0 or above
320K RAM
dual floppy drives, or hard drive
one short slot for Orion parallel-bus adaptor card
monochrome and EGA compatible

Accessories Included
User's Guide
Reference Manual
16-pin IC clip
Input stimulus cable
Component-clip adaptor probes (2)
Programming modules PM16, PM32, PM64, PM56

Accessory Options
Personality Paks for more than 150 microprocessors include:

- Disassembler/DEBUG software (DDB-xxx).
- Optional programming module available for 27512, 2716B, 2732B.
- Connection hardware, consisting of *either:*
  a ROM emulator cable (8-bit, 24-pin version #C8-24 unless otherwise specified; see below) and an analyzer cable preconfigured for your target processor.
  or:
  an Emulation Module. Replaces the processor on your target board.

Some Personality Paks also include a MicroTarget, an expandable target board. Check with your Orion Sales Representative for current product information.

Emulator Cable options:

Description	Part Number
8-bit, 24-pin ROM emulator cable	C8-24 (standard)
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

Notes:

CUSTOM CABLES

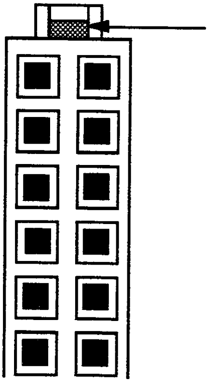
The Sockets

The two 50-pin connectors on the front of the UniLab carry extra signals, so operation of the instrument can be easily altered for different target processors. Because clock-logic requirements vary from one processor family to another, jumpers are used on the connector to make some of the connections.

Altering Standard Cables

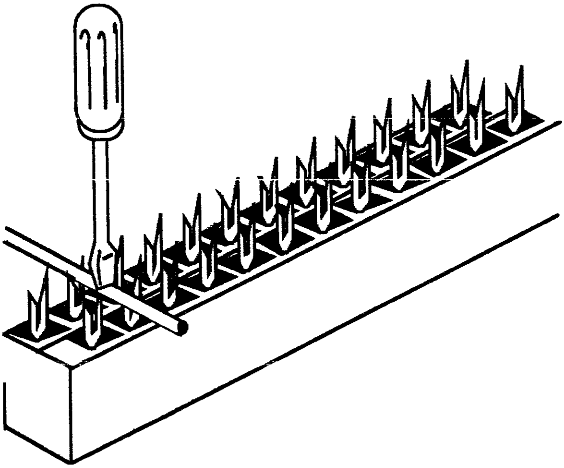
Standard ribbon cables are provided that will work for most systems. In some cases, these cables must be reconfigured for proper operation with a target system. The connections are made with the kind of insulation-displacement U-contacts used in "Scotchflex" and "Quick-Connect" prototyping systems, so they can be changed easily.

Pop the cable clamp off the connector by gently prying open the end with a small screwdriver at the points shown below. Carefully release each end of the connector before you try to remove the cover.



Pry end of screwdriver in this hole, at each end, while holding the connector body.

Wires can be disconnected by just pulling them out of the connecting pin's wedge with small needle-nose pliers. If you are doing changes like this often, you can get a special tool from a 3M distributor to push the wire into the connections. Otherwise, carefully push the wire down in its wedge by placing a screwdriver tip adjacent to the wedge connector (not in it) to force the wire down so that the insulation is cut and the wire makes contact. Note that these connections are identical to the ones used on "Quick-Connect," "Speedwire," and "Scotchflex" prototyping boards.



You can use #26-30 solid or stranded wire for making connections. Wire-wrap wire works nicely. If you are jumpering a probe wire to a second pin (as on pins 21 – 22 of cable 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 push it onto the connector.

If you are working with several families of processors and require different jumper options, you should probably buy additional analyzer cables to avoid changing jumpers every time you change processor families.

The Analyzer Cable

The analyzer is internally connected to all the signals on the ROM cable. To pick up any additional signals required for full monitoring of bus operations, connect patch wires from the analyzer cable to your processor pins. This is usually done with a 40-pin dip-clip. The wires can also be plugged into .025" wire-wrap pins.

The UniLab comes with an analyzer cable preconfigured for the processor of your choice. You can purchase additional cables or alter the original cable to support other processor families.

Problems with Decoded OE- Signals

Most analyzer-cable configurations (all but B, H, M, and N—see diagrams at the end of this appendix) assume 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, or 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, because it is used as a low-true master enable for the emulator's tri-state data bus outputs. Thus, this signal is needed in order to prevent the UniLab from getting onto the bus at the wrong time in systems that multiplex addresses over the data bus.

Most 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 onto the bus during I/O cycles.

With address decoding in the OE- signal, the UniLab will be unable to emulate memory for other ROM sockets.

Analyzer Connector Signals

Pin#	Signal	Remarks
1	M7	MISC-byte input to analyzer.
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	Emulation enable. 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.

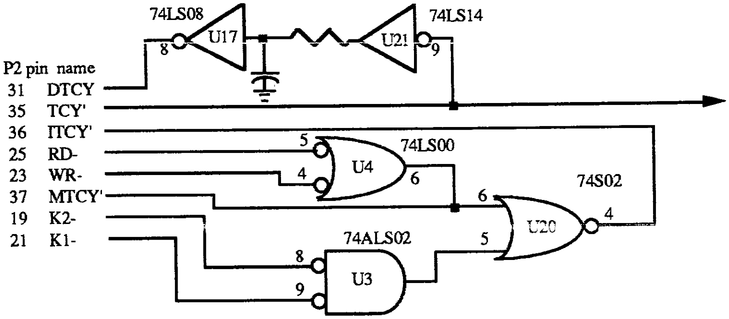
(Continued)

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 and 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 shown in the schematic and textual descriptions that follow. The list of the analyzer connector's signals, preceding, should assist further. While most of those signals are self-explanatory, the analyzer input register clocking deserves some explanation. The analyzer logic, and three of the six analyzer input bytes, are clocked by the + edge of TCY' (pin 35). The other three 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.

UniLab Clock 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 when RD- or WR- goes low, or when both K1- and K2- go low. Clocking, in this case, occurs at the end of the WR- or RD- pulse. DTCY is an inverted, delayed version of TCY'. By using this signal to clock the CONTROL input register (CCK'), the source of the clock (e.g., WR-) can be captured reliably as an analyzer input.

Avoid Memory Contention

When using a ROM cable, all ROMs being emulated must be removed from their sockets, in order to prevent bus contention. The ROM cable plugs into only one of the sockets—except in 16-bit systems, where there must be a second ROM plug in one of the most-significant-byte ROMs. In eight-bit systems, the most-significant data bits are brought out in a separate cable, and can be used as extra general-purpose analyzer inputs.

ROM Connector & UniLab Circuitry

The next page presents the ROM connector signals. It is followed by a schematic diagram of the UniLab's input circuitry and by diagrams of all Orion's standard cable jumpers. Along with the chart of ROM connector signals (preceding), these should provide all the information needed to customize your emulator cable.

The ROM Cable

There are four types of standard ROM cables (also see the appendix "UniLab Specifications"):

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 cable has an A11 pin, which can be left in 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 connected to the A11 signal at the processor.

Other MSB address signals are connected to the processor similarly. Pin numbers for making these connections to the major processors are shown in this appendix. Note that 24-pin cables will work fine in 28-pin ROM sockets, if they are just plugged in leaving pin #1 of the socket open. Extra address signals are picked up at the processor.

To minimize interconnections and signal loading, the analyzer data and address connections are also taken from the ROM cable. 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 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 systems). Or you can order a custom cable (C824HD, C1624HD), or a C8D or C16D cable that makes all connections at the processor.

Avoid Memory Contention

When using a ROM cable, all ROMs being emulated must be removed from their sockets, in order to prevent bus contention. The ROM cable plugs into only one of the sockets—except in 16-bit systems, where there must be a second ROM plug in one of the most-significant-byte ROMs. In eight-bit systems, the most-significant data bits are brought out in a separate cable, and can be used as extra general-purpose analyzer inputs.

ROM Connector & UniLab Circuitry

The next page presents the ROM connector signals. It is followed by a schematic diagram of the UniLab's input circuitry and by diagrams of all Orion's standard cable jumpers. Along with the chart of ROM connector signals (preceding), these should provide all the information needed to customize your emulator cable.

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 address 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 addresses' LSB and MSB paralleled for 8-bit.)
28	D10E	" "
29	D12E	" "

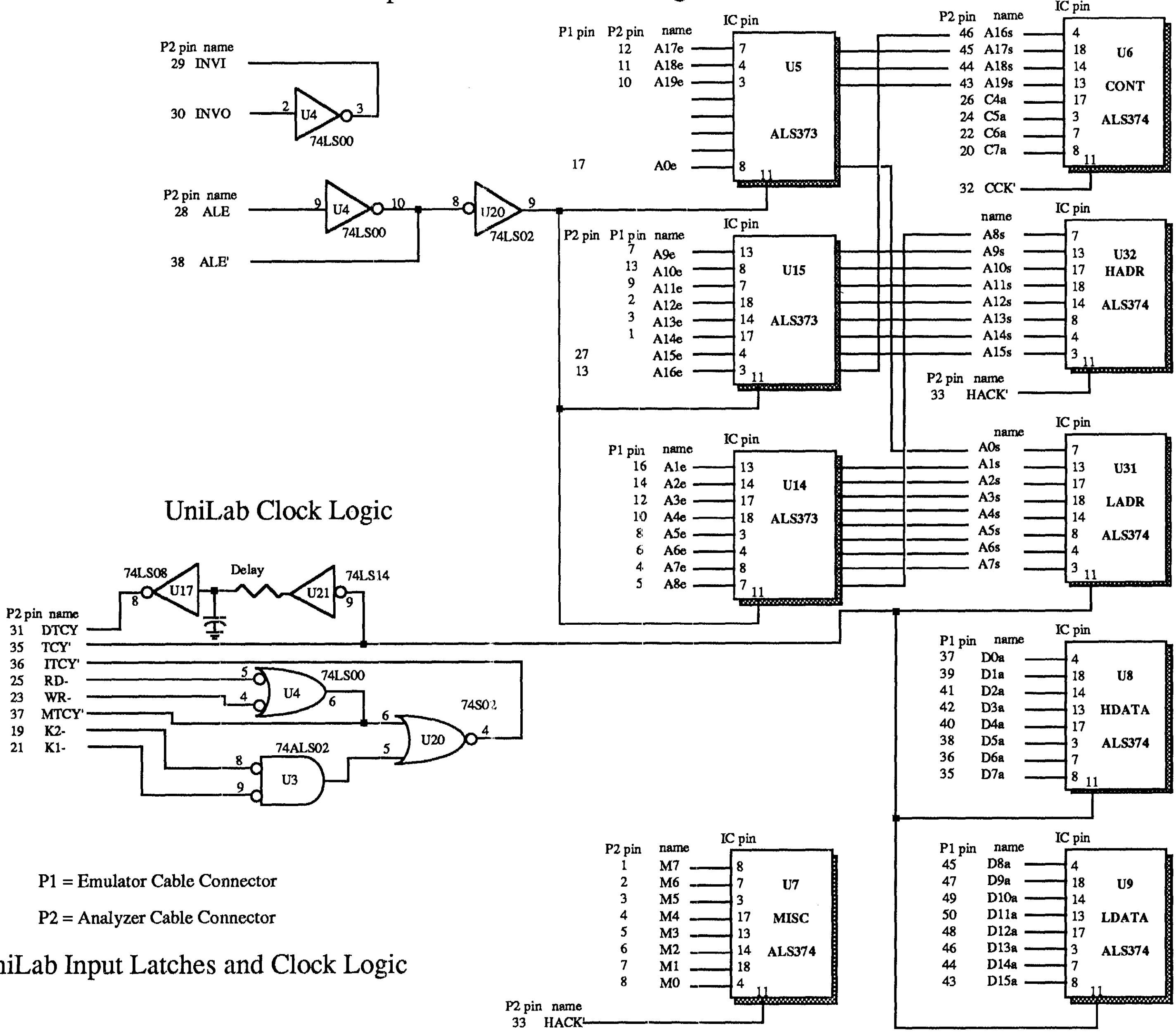
(Continued)

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 and MSB 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.

Notes:

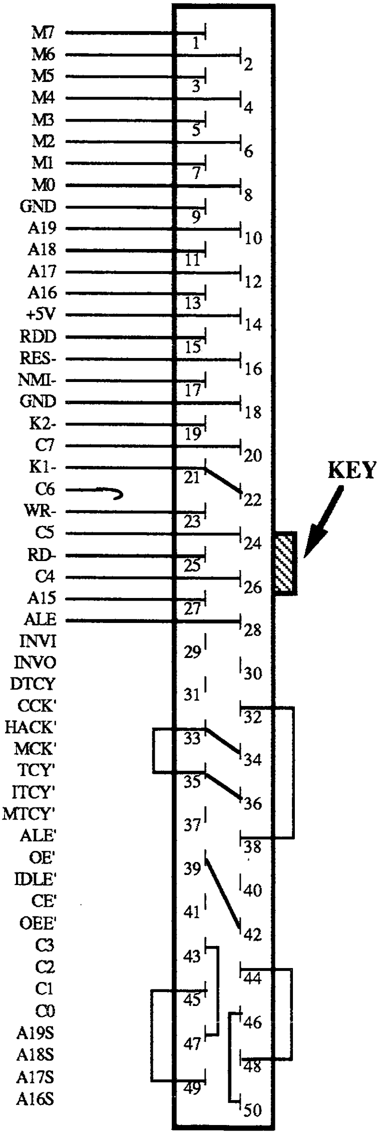
UniLab Input Latches and Clock Logic



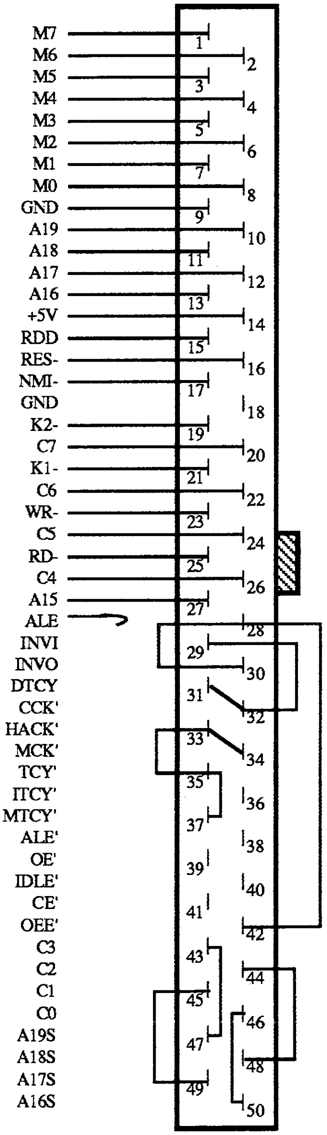
P1 = Emulator Cable Connector

P2 = Analyzer Cable Connector

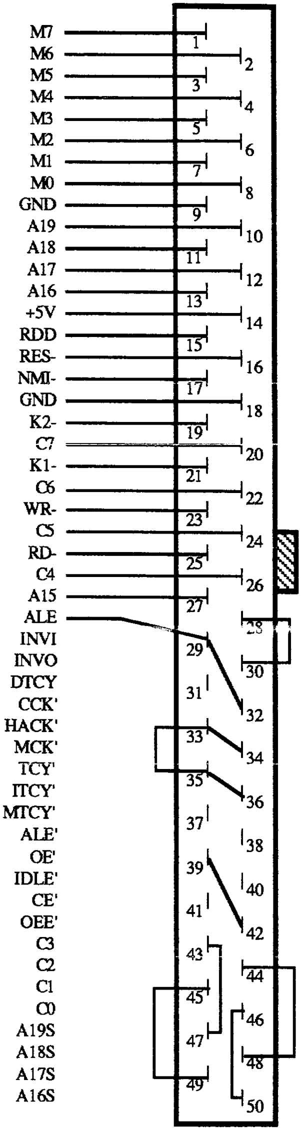
UniLab Input Latches and Clock Logic



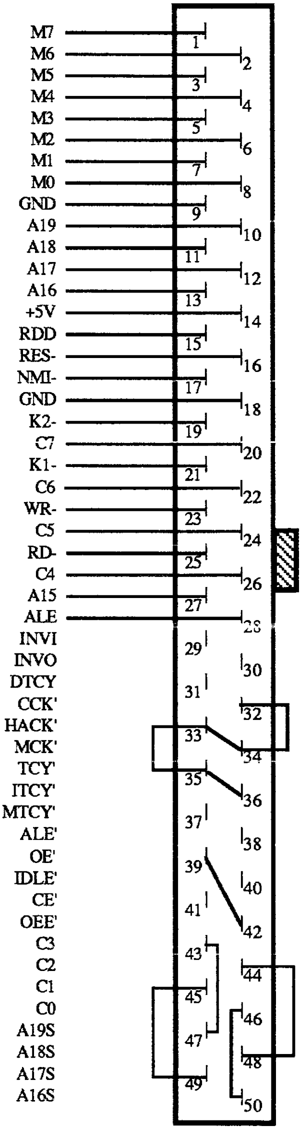
CABLE A
for 8085,
80186/188 (<12 Mhz),
8086 min, 8088 min



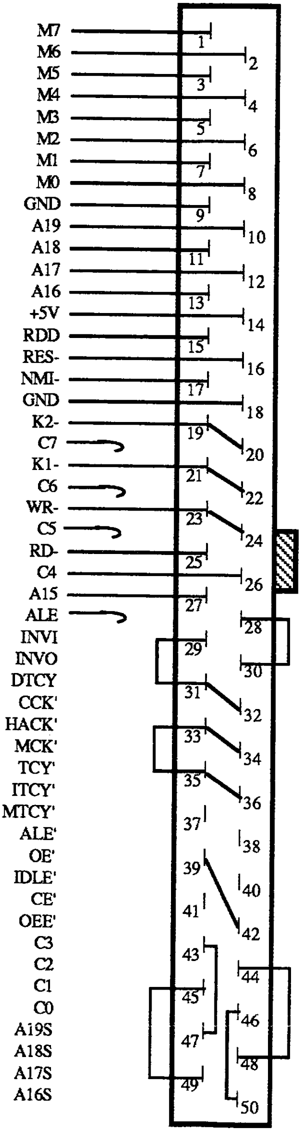
CABLE B
for 6303, 6502
6511Q, 6800,
6802, 6803,
6805 (romless),
6809, 6809E



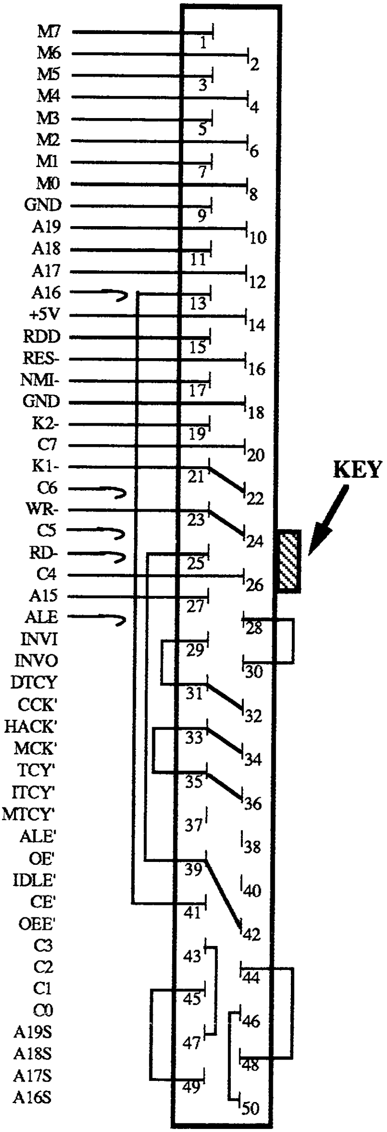
CABLE C
for Z8000 family



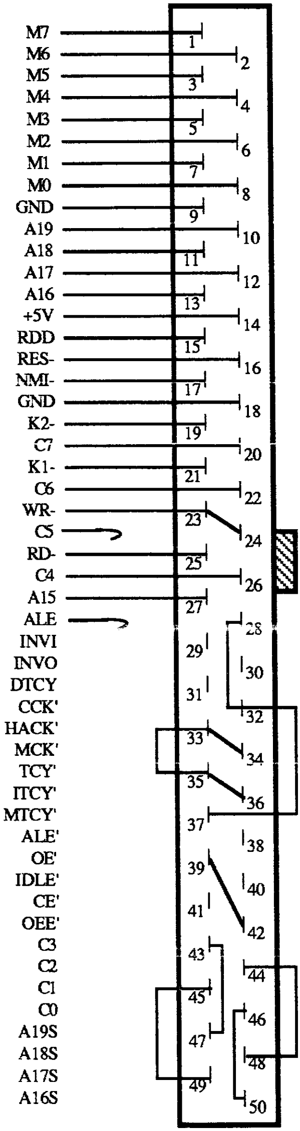
CABLE D
for
Z-8 expanded,
Super8 expanded



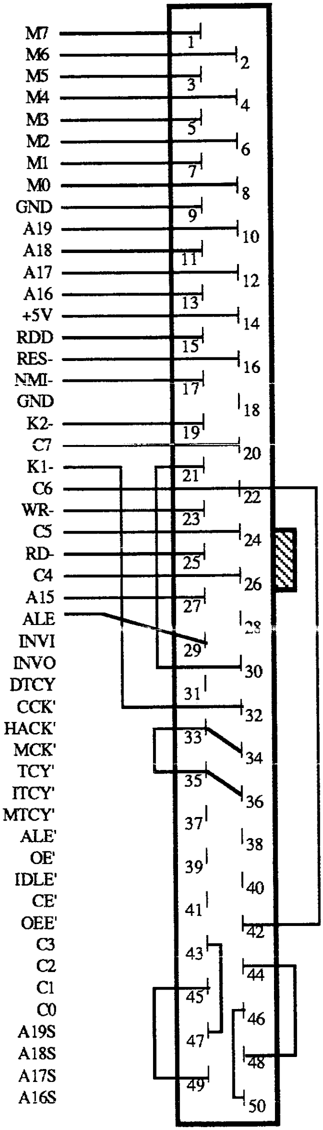
CABLE E
for 64180,
8048 (romless),
Z8 piggyback,
Super8 piggyback,
Z80



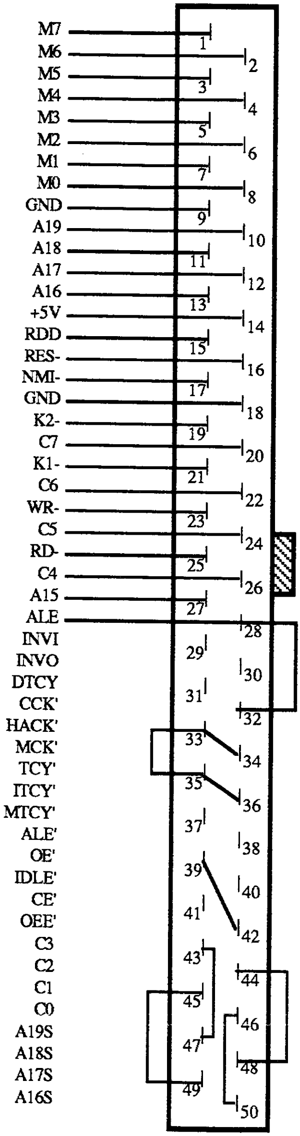
CABLE F
for
8048 piggyback
(ROM clocked)



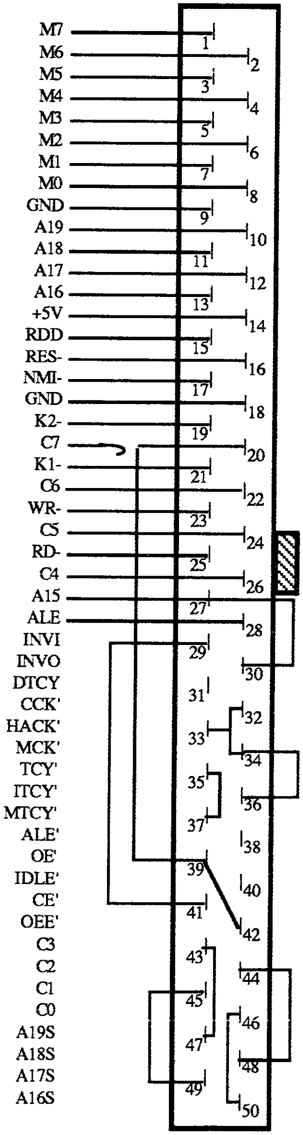
CABLE G
for 1802



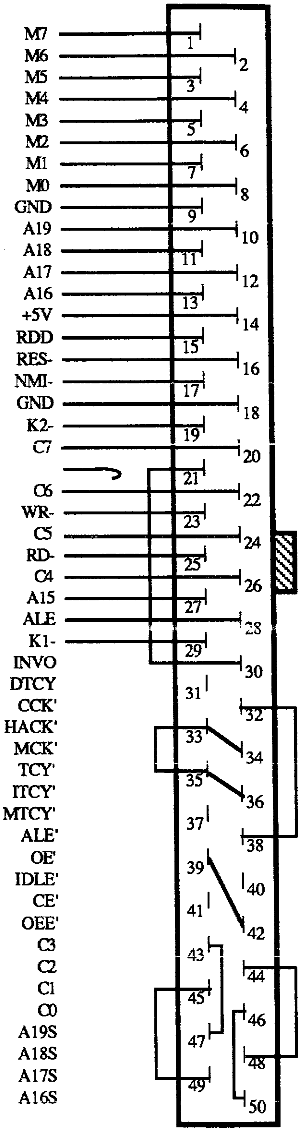
CABLE H
for 8080



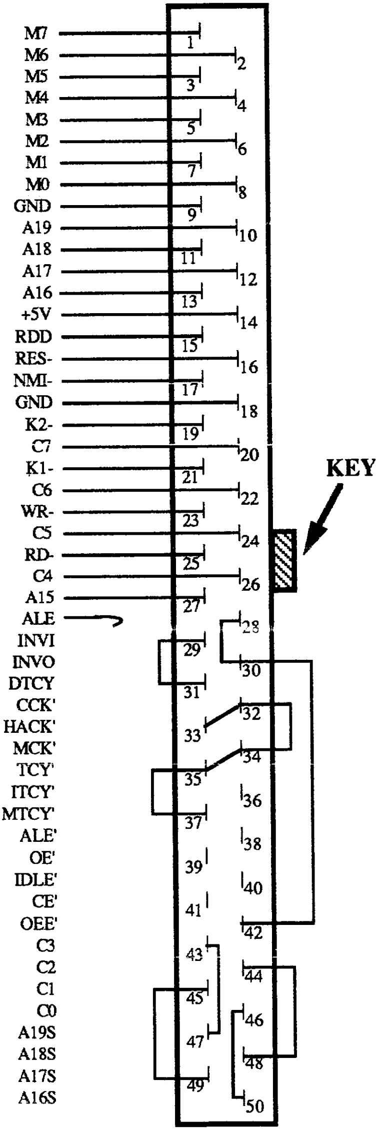
CABLE I
for 80286



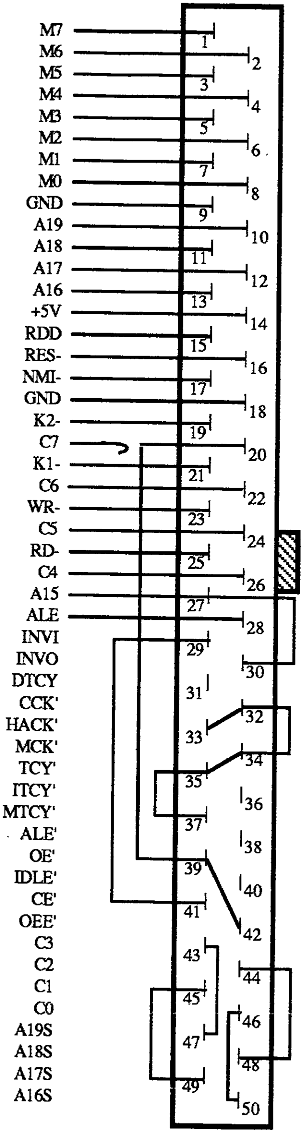
CABLE K
for 65/11
piggyback family



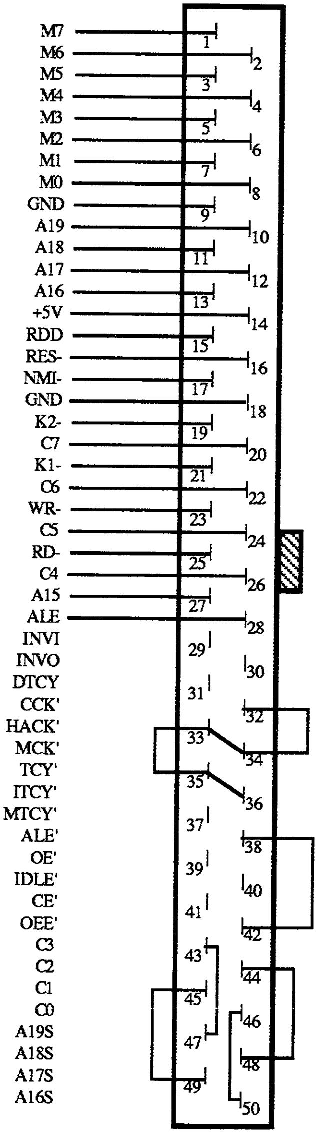
CABLE L
for 8086 max
8088 max



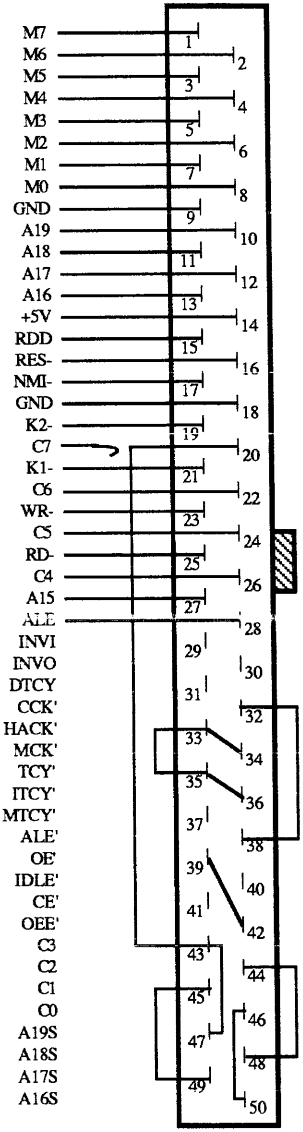
CABLE M
for 6805 piggyback
family
(with clock
divider circuit)



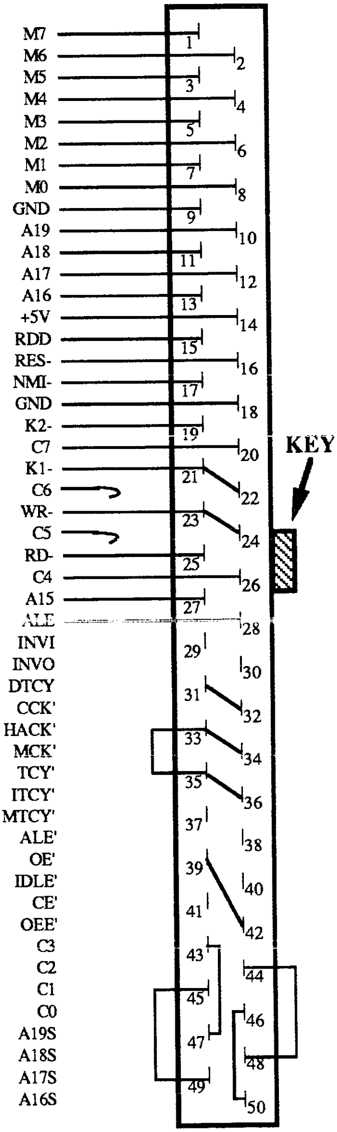
CABLE N
for 6801 piggyback
family, 6301
piggyback family



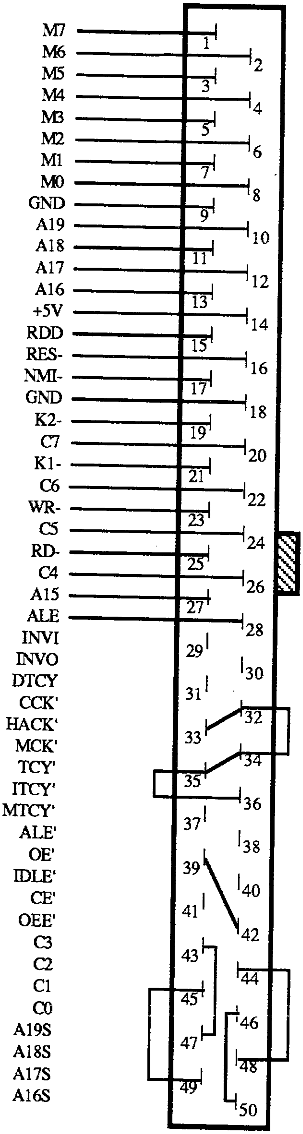
CABLE P
for 68000
68008



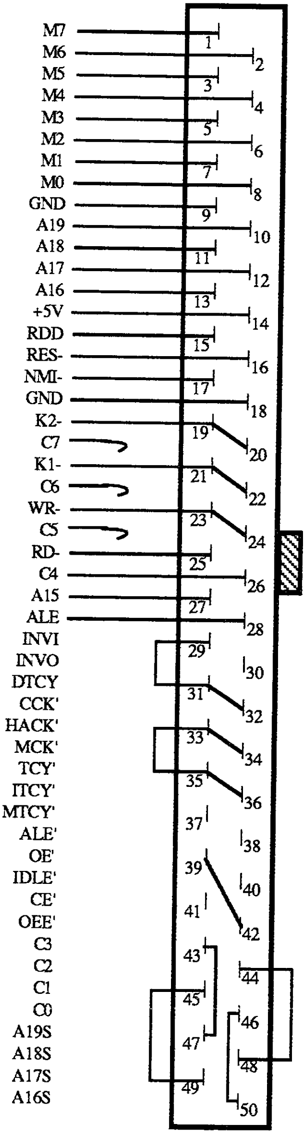
CABLE Q
for NSC-800



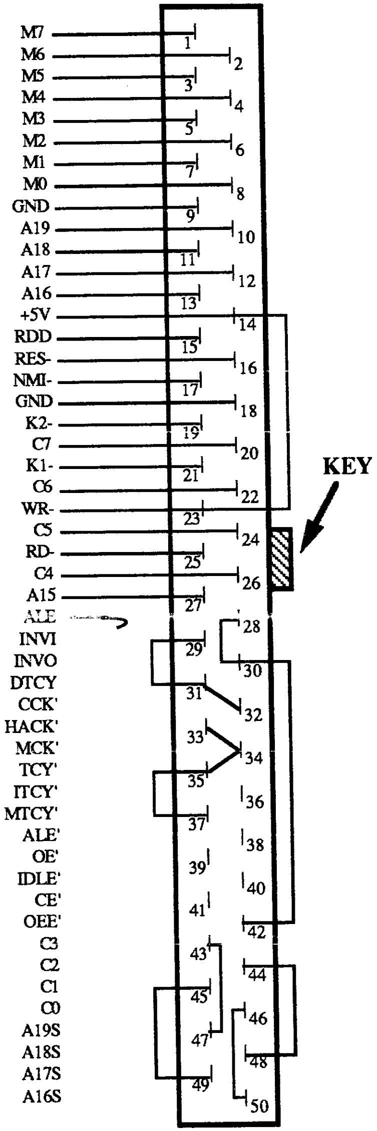
CABLE R
for 8096 family



CABLE S
for 80186/188
(>12 Mhz)



CABLE T
for 8031
8051 piggyback



CABLE U
for 68HC11

Cable Specifications

The following descriptions refer to the analyzer cables whose schematics appear in this appendix. They briefly explain how the analyzer and emulator clocking is derived, and are intended as a basis for customizing your own cable hook-up scheme.

Since +5 volts is available at the connector, you could 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 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 piggyback processor, with all signals except the RD' strobe directly connected. This sacrifices universality and some transparency, but it might be more convenient in some situations. The Orion Emulation Modules use this feature.

Cable A The CONTROL byte input register clock (pin 32) is isolated from the other input clocks and connected to ALE' (pin 38) on the 8088 and 8086 family (MIN mode). 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', 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- and 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 needs to be connected only if you have a 16-bit processor, as it is taken care of by the eight-bit ROM cable.

Cable B The UniLab address latch enable and CCK' pin are jumpered to the inverted DTCY signal, so that control signals and addresses will be latched after the rise of the E clock signal on Motorola processors. 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.

Cable C The ALE signal is inverted, using the uncommitted inverter on pins 29 and 30. The control signals are latched at the end of the address strobe.

Cable D A very simple configuration with all analyzer inputs clocked by ITCY.

Cable E C5, C6, and C7 are jumpered to WR-, K1-, and 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. WR', M1', and IORQ' are all used for clocking, and are captured by the analyzer to identify the cycle type on the Z80. The address latches are enabled by DTCY' to prevent trouble from the short address hold time on Z-80 A', B', and 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 only need to use `7 =EMSEG` when this signal is low.

Cable F 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, in order 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 use `F =EMSEG` to make the UniLab ignore this signal, then the `EMENABLE` statement; followed by `ALSO E =EMSEG`, then repeat the `EMENABLE` statement.

Cable G Used only for the 1802 family, 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.

Cable H Used only for the Intel 8080, the O/2TTL 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 O/2·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 – 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.

Cable I Used only for the 80286, the ALE signal from the 82288 is jumpered to the CONTROL clock input so that S0 and S1 will be captured.

Cable K Connects C7 to OE', which is R'/W; inverts CE and connects it to A15. A15 can be jumpered to A12 – 14 at the end of the cable for true address display.

Cable L Identical to cable A, except the uncommitted inverter is used to invert the DEN signal on an 8288 bus controller. This inverter output is jumpered to the K1- input. Connect the DEN wire to pin 16 of the 8288 bus controller. A0 needs to be connected only if you have a 16-bit processor, as it is taken care of by the eight-bit ROM cable.

Cable M This cable is used on Motorola piggyback development chips. Since no bus clock signal is provided by the processor, a circuit board is provided that derives a 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.

Cable N Identical to cable K, except the clock polarity is reversed.

Cable P Identical to cable D, except the OEE' input is grounded so that emulation will be enabled when either half of the data bus is read.

Cable Q Similar to cable A, except C7 is connected to A19 to help the disassembler to distinguish between memory and I/O cycles. On the NSC-800, A19 is then connected to M/IO. Also, K1- is not connected to C6 as on cable A.

Cable R Used only for the 8096 family, DTCY is used to properly clock in the status lines INST and WR-.

Cable S Similar to cable A, except CCK- is clocked in by TCY-. This allows the 80188/186 family to run at higher speeds, although the status lines S0, S1, and S2 must be pre-latched with an external D-type latch.

Cable T Identical to cable E, except ALE is connected directly to the processor's ALE.

Cable U Similar to cable E, except the emulator output enable (OEE-) is derived from DTCY-. Used only on 68HC11.

INDEX

Specific commands appear alphabetically in the glossary, and are listed by page number and are also grouped by general function in the introductory material.

UniLab

analyzer-emulator

Volume II Reference Guide

ORION INSTRUMENTS

ORION INSTRUMENTS, INC.

180 Independence Dr., Menlo Park, CA 94025
Telephone 415-327-8800
TLX 530942 FAX 415-327-9881

ORION
INSTRUMENTS