PROMICE™

Universal Firmware Development Tools

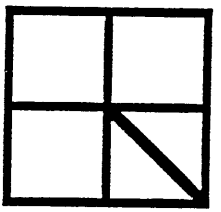
USER MANUAL

Grammar Engine Inc

PROMICE™

User Manual

Version 1.5



Grammar Engine, Inc.
3314 Morse Rd. Columbus, OH 43231
(614) 471-1113

PROMICE User Manual

Version 1.5

PRINTED IN THE UNITED STATES OF AMERICA

PROMICE User Manual

TABLE OF CONTENTS

ORIENTATION

You have purchased the best ROM emulator available on the market. The PROMICE is a very capable tool. It is configurable, expandable, upgradable and has potential that is way beyond that of a simple ROM emulator. Not only can the PROMICE emulate virtually any ROM, it can just as easily emulate RAM. It can load data over the serial port or the parallel port. The parallel port can also be used as a bi-directional link, thus saving the serial port on your Host. The PROMICE can also be set up to establish a direct communication link between the Host and the Target systems, thus you can not only emulate the ROMs but also talk to the downloaded code from the same Host over the same port.

To get your unit doing useful work in short order it is recommended that you familiarize yourself with a few of the basic concepts. If the unit fails to operate properly you may go to the very back of this manual and look for trouble shooting hints. But first lets us orient you to a few basic concepts.

PROMICE replaces the ROM/PROM/EPROM from your target system with a low-power, low-noise circuit that contain a micro controller, static RAMs and buffers etc. It is attached to your target ROM socket with a ribbon cable. Data intended for your ROM is down-loaded over the serial or parallel cable from your host computer. PROMICE powers itself either parasitically by drawing power over the ROM cable from your target system's ROM socket, or via the external power supply directly attached to the unit. You must make the proper power source selection on the back-panel of the unit by moving the shorting block to the appropriate jumpers.

LoadICE is an application program that is supplied with your PROMICE unit and is intended to be run on your host system. LoadICE will communicate with, configure and down-load one or more PROMICE units. You must help LoadICE program in order to properly operate your units. The few basic things that must be described to the LoadICE application are, number and size of ROMs that you are emulating. The word length configuration of the data being down loaded (8, 16 or 32 bits).

The actual data files to be down loaded. The baud rate of the serial port and whether you are also using the parallel port. All these specifications are best specified by putting them in the *loadice.ini* file on your host system. The data files can be specified in this file as well as on the command line.

LoadICE will automatically figure out the format if your files are in any of the popular HEX formats (HEX format files are ASCII files where the actual binary data is encoded as ASCII characters). Binary files must be explicitly specified to LoadICE. In order to ensure that the data gets loaded into the right place in the PROMICE unit(s) the information required to properly map the HEX records or binary files must be specified with each file. In some of the cases it can be very straight forward where as in some other cases a little bit of pre-planning can avoid hassles later on.

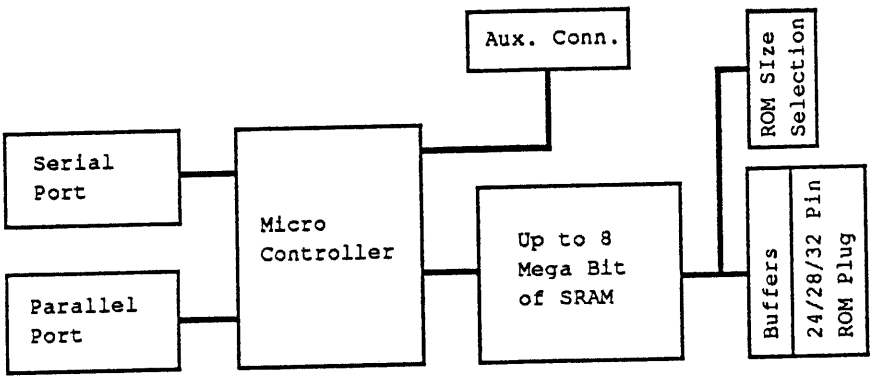
If you are anxious to get going and will read the documentation only when absolutely necessary, then type the *release.ini* file on your screen, look at it, copy it to *loadice.ini*, edit it to have sensible contents for your situation and run the *LoadICE* application. In most cases the going beyond this point is relatively easy. If you think that the stuff still doesn't make much sense, then we recommend that you read at least the relevant sections in the manual. Also try *loadice ?* to get some help printout.

PRODUCT DESCRIPTION

PROMICE is an In Circuit Emulator for a (Programmable) Read Only Memory. It is a self contained unit that can emulate any 24, 28 or 32 pin JEDEC standard ROM (2716 thru 27080). A given unit can be expanded to emulate up to two ROMs. Furthermore, the units can be daisy chained to emulate more than two ROMS. Current limit is set at 256 ROMs.

The full blown PROMICE system consists of up to three modules. A *master module* contains the micro controller and the host interface circuitry as well as one bank of emulation RAM. A *slave module* containing another bank of emulation RAM may be added to extend the capability to emulate two ROMs. And finally an *analysis interface* board may be added to allow PROMICE to be used for more intensive firmware development.

The following is a simplified block-diagram for the PROMICE master module.



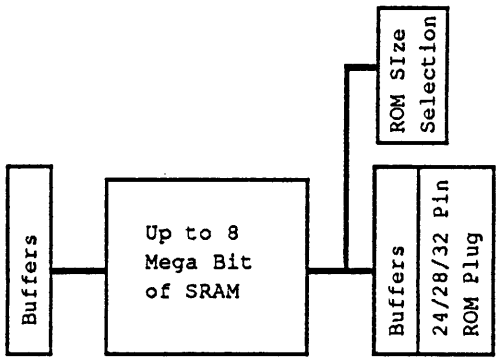
The product is based on an intelligent micro controller (Signetics 87C451) communicating with the host processor (PC etc.) over the RS-232C serial link or the parallel link. The emulator is connected to the ROM socket on the target via a ribbon cable. The auxiliary connector provides a programmable 'reset' and other signals for target control

Hex or binary data may be down-loaded to the unit. The unit can also self-test, diagnose, up-load and down-load data. The user interface allows complete control of the unit's functions either from the host or from a terminal. The parallel port may be used to down load data much faster than the serial port. Parallel ports can not be daisy chained, except by external switching.

The unit is designed with CMOS parts where ever possible resulting in very low power consumption. The unit can be powered parasitically by the target system over the ROM cable(s). However, external power supply may be used for powering the unit separately.

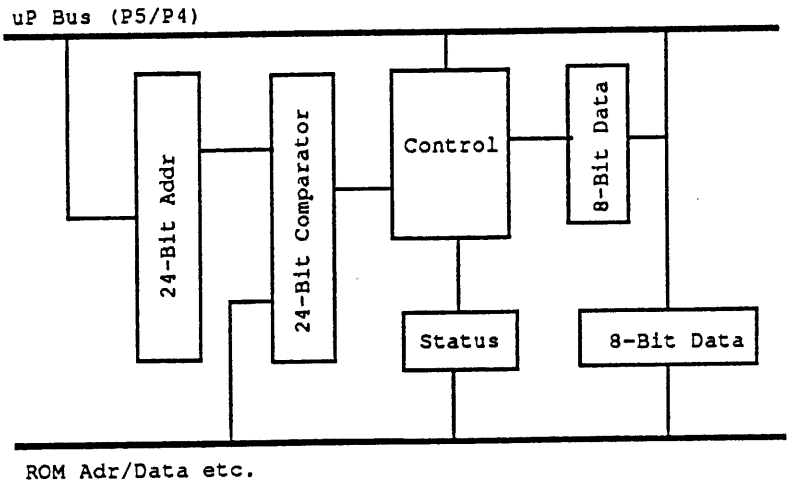
All user settable options are either specified and controlled via the host software or are selectable from the outside (back panel) of the unit. The unit is constructed from quality materials and is warranted for one year.

The slave module is mounted in the same box as the master. The micro controller controls both the master and the slave modules' loading, testing and emulation functions. Here is a simplified block-diagram of the slave module:



The slave module extends a given unit's capacity to emulate two ROMs. The maximum capacity of a unit equipped with a slave module is thus 2 mega bytes or 16 mega bits!

The analysis interface contains a proprietary circuit that allows for bidirectional communication between the host and the software (debugger) running in the target system without the need for any special modifications to the target. This circuit enables the debugger and the host software to coordinate their activities via the micro controller and allow easy firmware development. Here is a simplified block diagram of the analysis interface module:



A four byte area of the emulation ROM space is used as a control block by this circuit. It allows for 8-bit bidirectional communication with status bits indicating when data is available and when data has been read. The debugger interface to these functions is accomplished by a small driver software that operates the four bytes block.

Additionally the analysis interface allows for a hardware break point to be set that is qualified by some target supplied signals (write; ram select etc.). Upon detection of this trap a signal is generated to trigger the target.

Another important feature of the analysis interface is to allow changing of any byte in the emulated ROM space (master or slave module) on the fly. This feature can be used to set and remove 'break-points' or otherwise 'patch' code without interrupting the Target system.

Analysis interface allows the PROMICE to be used as a "Universal Firmware Development Tool". In order to adapt the PROMICE to a different target processor only a different debugger needs to be down-loaded.

The basic PROMICE unit is also capable of emulating a RAM with an externally supplied 'write' signal. It is also capable of operating the Target BusRequest/BusGrant (HOLD/HOLDA) protocol and thus allowing modification and examination of emulation space by the Host processor, without crashing the Target system. It can also generate interrupts to the Target system and accept interrupts from it. These capabilities are exploited to accomplish Host/Target communication without the use of the Analysis Interface. See Appendix-6 for more detail.

HOW TO USE THE PROMICE

HARDWARE

Attaching the unit to the target system:

The unit is attached to the target system via the ribbon cables provided. One end of the cable has a DIP plug for insertion into the target's ROM socket. The other end has a female header (34 pins (2x17)) to be inserted in to the male header on the back of the unit. Pin-1 orientation is marked on the back panel. Notice that it is always on the right when looking from the back of the unit. We make cables to emulate 24, 28 or 32 pin devices. You may also have a custom cable to handle PLCC or 40-pin DIP or any other ROM foot-print. Please follow any special instructions included with custom cables.

If your unit is a duplex unit then there is also a slave module present. It is almost identical to the master module and another ROM cable can be attached to the slave module from your target's second ROM socket. This will allow you to emulate two ROMs from one unit. They may be emulating a 16 bit target system or just be two ROMs in an 8 bit system or working with another duplex PROMICE unit in a 32 bit system.

Setting emulation ROM size:

There is a 10-position dip switch on the back panel to the left of the ROM cable header. The switches are marked 1 through 10 on the package itself. On the back panel there are numbers printed that go with individual switches. The numbers are paired as follows

Switch#	*ROMpart# 27xxx*	*Address line connected*
10	32	A11
9	64	A12
8	128	A13

7	256	A14
6	512	A15
5	010	A16
4	020	A17
3	040	A18
2	080	A19

In order to emulate a given size ROM turn on switches starting from #10 and going down till you have turned on the one for the particular size that you are emulating. The switches connect the address lines from the ROM cable through to the emulation RAM. If you fail to turn on the proper switches, the unit may fail to emulate properly, since your target system will end up accessing wrong space within the unit. It is therefor very important that the proper switches be turned on, and only the proper ones be turned on. However, it is possible that different modules in a given unit or configuration be emulating different size ROMs.

Setting power supply options:

On the back of the unit to the right of the ROM cable header, there is a row of male headers. The first four positions on this header select power source for the unit. Power may come from two different sources, the external power supply (9VDC) or from the target via the ROM cable (parasitic operation). Use the shorting-block (shunt) to jumper the pins appropriately to supply power to the unit. There are three choices for parasitic power, depending on the ROM cable size (24, 28 or 32 position). *In all cases there should be only one shunt on the back of a given module selecting the power source, otherwise you may be connecting the external and target power supplies!!.*

When using the external power supply make sure that the setting on the slave module (if present) is also set for external power position. Wrong setting on the slave can result in PROMICE powering your target system over the ROM cable. In that case you are likely to overload the power supply circuit in the PROMICE unit.

If the slave module is present:

If your unit includes a slave module then it is mounted on top of the master module. It has identical headers and switches etc on the back, as described so far. The master module is always the first ROM in the unit and the slave is the second ROM. Follow the same instructions for setting and hooking up the slave as for the master.

If you are emulating a 16 bit system then the master and the slave module will form odd and even byte pair or even and odd byte pair depending on the byte addressing used by your target system. If your data files are per ROM files (i.e. they contain 8 bit data) then you can arbitrarily specify which file goes into which ROM (PROMICE module). If your data files contain 16bit data then the first byte always goes into master module and the second byte into the slave module. Hook the appropriate module to the appropriate ROM socket on the target system.

Connecting RESET signal to the target (optional):

Next to the power options there are other header pins on the master module that provide various auxiliary functions. The one marked 'RST' provides a programmable reset signal that can be used to reset the target. This signal is internally driven by the micro controller through an active buffer. The default is to drive this signal on power up for 500 milli seconds. The signal is driven low unless the switch position #1 on the left most side of the backpanel is on (this switch is unmarked on the back panel). In that case the signal is high asserted (e.g. reset on 8051 family micro). This signal can also be asserted by giving commands to the PROMICE via LoadICE and its time period can be specified (8ms to 2.3 seconds).

This concludes the procedures required to hook the PROMICE to your target system.

Connecting the unit to the host:

PROMICE is connected to the host computer (PC etc.) via the RS-232C link. If you are also going to use the parallel link for faster download then additionally the parallel cable is also connected to the parallel port on the host system. However you may choose to use the parallel link by itself. In that case the parallel link is used in bi-directional mode. (This is supported on the microCode version 4 and later). Bi-directional parallel ports can not be daisy chained.

The RS-232 link is connected via a 6-conductor modular cable and a modular to DB-25 or DB-9 adapter. All cables and adapter are provided with each unit for easy hookup. The parallel cable is designed to hook directly to the printer port (DB-25 female) on the back of the PC or compatibles. On the Macintosh system a DIN-8 to DB-25 cable is provided additionally.

If you wish to make a cable of your own or have doubts about whether the cable is working properly, check the 'Technical specification' elsewhere in this manual for the pin outs. The PROMICE unit needs only three wires for proper communication over RS-232, namely 'receive data', 'transmit data' and 'signal ground'.

Connecting Multiple units to the host:

Multiple units can be daisy chained from a single RS-232 link. Up to 256 unique ROMs can be emulated on a single daisy chain. This limit is strictly enforced by the software only (IDs are 8-bits). For each unit after the first one, a special daisy chaining connector is used. These connectors are provided with all multiple unit orders. Any word size up to 2048 bits can be emulated. See appendix for how to use the daisy chain connector and how to determine ROM IDs for chained units.

If you have the parallel port option on the PROMICE, then connect the parallel port cable between the PC and the PROMICE. Two units can be daisy chained on the parallel port. We make a special cable that has two connectors on one end to connect to two PROMICE units. By using some extra signals on the parallel port we can talk to two units. The serial port must also be used in conjunction with the parallel port. For 32 bit emulation the parallel port only supports duplex units.

If you are emulating 16 or 32 bit systems then special attention should be paid to the byte numbering scheme forced by the arrangements of the units. If data contained in the data files is 16 or 32 bit then the first byte goes to ID0 and so on. If data files contain 8 bit data then any file may be loaded into any module. Remember that the units are essentially modeless and the 16 and 32 bit data is handled and assigned to modules as a result of implementation choices made within the LoadICE program.

Unit IDs and ROM numbers:

On start up the LoadICE software assigns IDs to each ROM it finds on the Host to Promice link. The first unit's master ROM will be ID 0 and if a slave is present, that will be ID 1. Next unit's master will take the next ID# (1 or 2) and so on. LoadICE lets you specify what data should go into which ROM. The units operate in a modeless manner, i.e. they are always in 8 bit mode with direct communication between LoadICE and the particular ROM. Various word lengths are emulated by LoadICE sifting the data out to per ROM images and then loading particular ROM with its image etc. The host to PROMICE protocol is specified in the appendix.

SOFTWARE

Installing the software:

The software is distributed on a floppy disk. Insert the disk in a drive and copy all the files to a specific directory. The disk contains an executable LoadICE image as well as all the sources. There are README and RELEASE files that should be printed out and read for the latest information about the software. If your host is not a PC or a MAC then you will need to recompile the software. The sources are compilable on PC, MAC, UNIX and VMS systems. If you have a different system it will have to be ported. Contact us first, since we may have a customer who has already ported the software to your type of system. You may also prefer to copy the LoadICE executable image to your directory where all the other commands live. Use the *release.ini* file as a template and copy it to *loadice.ini* name. Then edit and modify this new file to reflect your particular setup. If you wish to access some on-line help then make sure that *loadice2.hlp* and *loadice.hlp* files are also in your working directory.

To install software on non-DOS machines see the instructions in the README file distributed with the software.

Specifying working parameters:

You will want to specify what baud rate to use on the serial port or to use the parallel port, what files to load in which ROM , whether to verify data after it is loaded, whether your host can handle the full speed transmission from the PROMICE (applicable to serial I/O only). These things are best specified in the initialization file called *LoadICE.ini.* This file is used by the LoadICE if it is present. Almost everything can be specified in this file. The command line options over-ride or add to the stuff from the initialization file. In the dialog mode, you can interactively specify information. You will use a combination of these specifications, depending on your particular needs. For example, you may specify baud rate etc.

in the *ini* file and file names on the command line etc. The whole thing can be put in to a *make* or a *batch* file.

Initialization file:

The initialization file is named *loadice.ini.* It is read every time the *loadice* command executes. The file as well the parameters are optional. It contains the following parameters:

Parameter	comment (*not in file*)
`baud=rate`	1200,2400,4800,9600,19200,57600
`fill=character`	x00-xFF
`ffill=character`	x00-xFF force fill all ROM space
`output=device`	com1; /dev/tty2; tta2:
`rom=size`	27256, 27040, 64k, 131072 etc.
`word=size`	8, 16, 32
`noverify`	don't verify down-loaded data
`high`	full speed response from PROMICE
`parallel lptn`	down-load on parallel port
`pponly lptn`	parallel port is bi-directional
`reset nnn`	assert *reset* for *nnn* ms, after loading
`file=filename offset=ID:Address`	specifies hex file to be down loaded
`image=filename skip=ID:Address`	specifies binary file to be down loaded

FILE SPECIFICATIONS: The file specifications in the *ini* file specify whether the file is hex or binary (See File Formats in Appendix 7). For hex files the *offset* specifies the address in hex files and its is mapped to the given unit (*ID*) at the given *Address* . This allows loading of a hex file to any desired address in the ROM. For binary files the *skip* value specifies the number of bytes that must be skipped from the beginning of the file and the *ID:Address* specifies the location where the data in binary file must end up. In addition to all this up to 55 files specifications may be included in the *ini file.*

PARALLEL PORT: If your unit has the parallel port option, then the 'parallel' specification tells *Loadice* that it should

use the parallel link for down loading the data. We can daisy chain two units on the parallel port, thus allowing emulation and downloading of a 32 bit system over parallel link. If you are connecting only one unit to the Host then you may prefer to use the parallel port in the bi-directional mode. To invoke that option specify 'pponly' in the initialization file. 'pponly' units can not be daisy chained.

HIGH SPEED RESPONSE (RS-232C ONLY): In normal mode of operation the PROMICE units will send back any response etc. at a baud rate that does not exceed 9000 baud. This is done to avoid overflow of data in a system where no buffering is provided. For those systems where the data buffering is built in (such as UNIX systems or the Mac), or for higher performance PC systems the *high* option specifies that the response be sent at full speed of the selected baud rate. If *Loadice* seems to hang (specially during verification) then more than likely some characters were lost due to overflow. It is recommended that the keyword *high* or the command line specification of *-h* be removed. In some versions of the system *Loadice* might report a time-out error for the same reasons.

COMMENTS: Any line in the *ini* file may be started with an '*' and will be treated as a comment line. This is a quick way to disable commands in the *ini* file without deleting them.

Example *loadice.ini* file:

```
baud=19200
output=com1
rom=27512
word=16
file=file1.hex f0000=0:0
image=file2.bin 100=2:0
```

This *loadice.ini* file sets the baud rate to 19200 and configures the *loadice* program to use the *com1* port. The ROMsize is set for a 27512 ROM. The hex file *file1.hex,* which is linked at address F0000 (i.e. the address in the hex records starts at F0000) will be loaded into PROMICE unit 0 and 1 starting at address zero. The binary image file *file2.bin* will be loaded

into the units 2 and 3 after 100 (hex) bytes of header information from the front of the file is skipped over.

LoadICE *command:*

A typical invocation of the LoadICE software might look like this:

```
loadice filename
```

The file is an ascii file containing hex records. LoadICE automatically figures out record type of data in the file. For binary files and some peculiar hex formats the file type has to be specified with switches. The following is a formal description of the LoadICE command:

LoadICE establishes communication with one or more ROMs being emulated by PROMICE units connected or daisy chained from a single serial port. It establishes number and size for each emulated ROM. Then it processes binary or hex data files containing 8, 16 or 32 bit data and down loads the data to the particular ROMs. It can also verify and otherwise manipulate ROM data. It controls other function of the PROMICE units such as resetting target system, and interactive editing of ROM contents. It can also test the emulation RAM and obtain microcode version number. If the command is entered with no file specifications on the command line or in the initialization file then the interactive (dialog) mode is entered. Interactive mode is also entered if requested via switch specification on the command line.

Command line options:

The minimum command line is:

```
loadice
```

loadice ? will display command line option help file. Command line options over-ride or augment initialization file

parameters. All options and fields are delimited by one or more spaces. Following can be specified on the command line as arguments to *loadice*

-b *rate*
Specifies the baud rate for the serial port. Allowable values are 1200, 2400, 4800, 9600, 19200, 57600. However you must assure that your system is capable of supporting the rate you choose. On PC and MAC *loadice* can support any of these rates automatically.

-f *fillcharacter*
Specifies the fill character to be used for filling gaps when building ROM image from hex records. By default the memory contents are uninitialized.

-ff *fillcharacter*
Same as fill above except, entire ROM image is filled instead of only up to last good data.

-h
Set PROMICE to send response at full baud rate (instead of <9000 baud)

-i *skip filename*
Specified a binary file and a number of bytes to skip from the beginning of the file. The file can be loaded at any desired location in ROM space by specifying an offset used as the start address. See -*s* below.

-k *startaddress endaddress storeaddress*
Checksum the ROMimage from *start address* to the *end address* and store the result in *store address*. The checksum is a simple byte wide sum of all the addressed bytes and it is complemented before it is stored.

-l
Used in conjunction with -*d* options, when specified it forces down loading of data before entering the dialog mode.

-m *ID:StartAddress EndAddress*
Only loads the data in the specified range to the referred unit.

-o *device*
Specifies the serial port, can be COM1-4 on the PC, *modem* or *printer* on the MAC and appropriate device on other machines.

-p
Enable parallel port option. Same as *parallel* in *LoadICE.ini* file.

-q
Enable bi-directional parallel port option. Same as *pponly* in *LoadICE.ini* file.

-r *romsize*
Specifies the emulation ROM size. It can be any value from the matrix below

size in bytes	size in K bytes	generic part#
2048	2k	2716
4096	4k	2732
8192	8k	2764
16348	16k	27128
32768	32k	27256
65536	64k	27512
131072	128k	27010
262144	256k	27020
524288	512k	27040
1048576	1m	27080

-s *offset filename*
Specifies the offset that must be added to each hex record in the specified file. This allows mapping of a file anywhere desired. Negative offsets are allowed. Precede the number with '-' sign for negative offset. E.g. -x400000. The -s option

also applies to binary files. In that case the file name is preceded by both the *-s* and *-i* options.

-u *ID*
Specifies which unit a file gets loaded into, used like *-i* and *-s* options. Applies to next *filename* on the command line.

-v
If specified it inhibits verifying of the downloaded code. It is strictly for speed purposes. Once you are comfortable with your setup and it works reliably then the verification can be turned off for speed.

-w *width*
Specifies a general operating mode. The data in files is assumed to be organized in the width specified. Currently 8, 16 and 32 bit widths are supported.

-x
Inhibits validating checksum on hex records.

-z
Ignores *Address out of Range* errors. The errors are still reported but the processing continues. This case is most frequent when the HEX data contains initialized RAM data and hence will fall way out of the range of addresses for the ROM space.

-d
Enter *dialog* mode. This mode is entered after any specified files have been processed into images and before any PROMICE units are loaded. It is also entered automatically when no files are specified on the command line or in the initialization file. In this mode *loadice* will take interactive commands from the user and perform following functions:

c
Compare the PROMICE contents against the image built by *Loadice*. No data is down loaded, it is only checked. This allows verification of PROMICE contents against software built by the host.

e *id:address*
Examine and Deposit. Data is examined and any new value typed is deposited. An ^ character will backup the address one byte.

d *id:address*
Dump 16 bytes.

f *start-address end-address data*
fill image with data.

g *filename*
Get image file.

h *filename*
get hex file.

l *[filename][ID:start end]*
down load current image or given *filename*. OR load the given ROM (*ID*) from addresses *start* to *end*. This allows partial load of a ROM. (same as *-m* on command line)

m *start-address end-address dest-address*
move stuff around in *image*

r *time*
reset target system for *time* milliseconds.

s *filename*
Save the current image to the *filename*.

t *id*
Test PROMICE emulation RAM.

v *id*
Report PROMICE micro code version #.

x
exit. Also '.' works the same way.

?
dialog mode help display.

! *string*
Escapes the *string* as a command for the operating system (not supported all systems).

Command line example:

```
loadice -b 19200 -o com1 -r 2764 -n 1 -w 8 -ff
xff -i 0 -s x200 tut.bin -s -xf000 tut.int -s
-xf00 tut.mot -v
```

This command will perform the same operation as the example for *loadice.ini* file. (This command is actually typed all on one line, here the word processor broke in to three lines). See the TUTORIAL for more insight into the example on the disk.

MESSAGES:

When *loadice* is executed it displays copyright information, including its version#. If an initialization file is present, it will inform about processing it. It then tries to establish link with the PROMICE units on the serial chain. This is followed by processing of any files and loading of ROM images etc. If interactive mode is invoked that also happens at this time. As the processing proceeds appropriate messages are displayed. If errors are encountered then usually an error message is displayed and processing stops. Following error messages may be expected from *loadice*.

```
System error - (any system related error
message)
ERROR - loadice specific error message
Location -> string (indicating argument or data
in error)
```

System error messages are displayed when appropriate. *loadice* error messages may be any of the following:

`End-O-File`	unexpected end of file reached.
`Open failed`	failed to open a file specified.
`Address out of range`	An address in a file was out of range of the ROM in which it is to be loaded
`Checksum error`	A hex record failed checksum
`Bad argument`	*loadice* unable to interpret the command line argument
`I/O error`	check system error for the real problem.

Most of the message printed out of LoadICE are understandable in the context they appear.

Technical Specifications

IDENTIFICATION:

P1*nnn*- Simplex: Single (master) module for emulating 1 ROM.
P2*nnn*-Duplex: Two modules (master and slave in one box) for emulating 2 ROMs.
where *nnn* is one of the following indicating the maximum capacity of the ROM

256	Emulates 2716 - 27256	(32KBytes)
512	Emulates 2716 - 27512	(64KBytes)
010	Emulates 2716 - 27010 (128KBytes)	
020	Emulates 2716 - 27020 (256KBytes)	
040	Emulates 2716 - 27040 (512KBytes)	
080	Emulates 2716 - 27080	(1MBytes)

further affixes that may be added to indicate the following options:
PP: Parallel Port on the master module for faster loading.
AI: Analysis Interface for special firmware development features.

POWER:

Promice Master: +5VDC (+-5%) < 100mA (85mA typical)
Promice Slave : +5VDC (+-5%) < 50mA (45mA typical)
Promice Analysis Interface: +5VDC(+-5%) < 100mA (90mA typical)

Power Jack:
Pin and sleeve plug, with pin as +ve and sleeve as ground.

INTERFACES:

Serial Port:
RS232-C, connects to host or a terminal via 6 conductor modular cable. Pinout (pins numbered from left to right) is RD-3, TD-4, CTS-5, RTS-2, GND-1. Only TD, RD and GND are required for communication.

Parallel Port:
Centronics compatible parallel printer port configured for direct connection to DB25 connector on back of PC or Compatibles. Can operate bidirectionally by using 'error lines' for sending data back 4 bits at a time.

ROM Socket:
JEDEC 24/28/32 pin DIP socket w/150ns access. Non-JEDEC and non-DIP footprints handled via custom cables.

Indicators:
Red LED indicating power-on.

Switches:
Back-panel: 10-position dip switch for selection of emulation ROM sizes. Switches 2-10, when in 'on' position, connect through to address lines for emulation of 2732(sw10), 2764(sw9), 27128(sw8), 27256(sw7), 27512(sw6), 27010(sw5), 27020(sw4), 27040(sw3), 27080(sw2). Switch position 1 is used to set RESET signal polarity, when all switches (2-10) are in 'off' position a 2716 is being emulated.

Jumpers:
Back-panel: Master module has 2x10 header. Slave module has 2x5 header. The first four position on Master are identical to Slave. The four positions (from left, on both modules) determine power supply options. Position-1 for external power source, position-2 through 4 for parasitic operation from a 32, 28 and 24-pin ROM cables correspondingly. Jumper 5 is not used on the Slave, on master it provides the bus request and bus grant signal hookups for PiCOM protocol (see appendix 6). Jumper position 6 has signal for resetting the target system available on the lower pin and the 'nmi' signal generated by the AI systems hardware trap feature (see appendix 6). Jumper 7 has the HandShakeOut and HandShakeIn positions. They provide interrupts to and from the target system and used by PiCOM. Jumper position 8 allows for attaching 'write enable' signals for the Promice modules (for effective RAM emulation) The rest of the header pins are for analysis interface board and carry extended address lines and external chip select.

ENCLOSURE:
5.08" Wide, 1.5" High w/o rubber feet, 5.25" Deep, Impact-resistant, ABS-molded, Grade DFA/R Plastic.

Standard ROM Cable:
Standard (.6") DIP plug on 12" 24 and 28 Conductor Ribbon cable with a 34-position Female Header for mating with connector on back of unit.

Modular Adapter:
Modular to DB25 Male or Female and Modular to DB9 Male or Female are available. The pin out at the DB25 side of the adapter is RD-3, TD-2, RTS-4, CTS-5, GND-7, DTR-20 and the DB9 side of the adapter is RD-2, TD-3, RTS-7, CTS-8, DTR-4, GND-5.

ENVIRONMENTAL RESTRICTIONS:
Operating Temperature: 5 to 32 degree C (41 to 90 degrees F)
Storage Temperature: -40 to 70 degrees C (-40 to 158 degrees F)
Humidity: 90% maximum without condensation.

Appendix-1

PROMICE internal memory addressing scheme

PROMICE can address up to 1 megabyte of memory per module. Master and slave module are addresses by switching the external circuit to select either module. There a total of 20 address lines required to access the 1 megabyte of memory. When less than 1 meg of memory is present the higher address lines are pulled up high. The exact reason for this is that the micro controller in the PROMICE can set its i/o lines to off state and the internal pull up resisters will pull the signals up. The emulation size switches on the back of the unit let the user connect those line that are supplied by the target through. The unused address lines will remain pulled up high. Therefor internally PROMICE addresses memory by using this map:

address 0	last address	Emulated ROM size
0F F8 00	0F FF FF	2K
0F F0 00	0F FF FF	4K
0F E0 00	0F FF FF	8K
0F C0 00	0F FF FF	16K
0F 80 00	0F FF FF	32K
0F 00 00	0F FF FF	64K
0E 00 00	0F FF FF	128K
0C 00 00	0F FF FF	256K
08 00 00	0F FF FF	512K
00 00 00	0F FF FF	1M

When using a terminal or a terminal emulator to talk to the PROMICE unit the actual address specified to the various commands must have the appropriate bits turned on to address a given location. An example would be a unit equipped with 128k bytes of memory when emulating a 32k byte ROM will have to use address 0F8000 to access the proper space. This will compute to be the highest addressed one of the four chunks of 32k that comprise the 128k unit.

This also explains why the LoadICE software must be told exactly what size ROM you are emulating. It ensures that the data is loaded at the proper place in the internal space. Also the switches on the back on the unit must be set for proper emulation size, otherwise the target system will end up accessing the wrong emulation space.

APPENDIX - 2

Host/PROMICE interface

The host communicates with the PROMICE units over the serial interface. When the parallel interface is present, the same protocol is followed there as well, how ever any responses that are generated will be sent over the serial link. Therefore, in a unit with both interfaces, both must be connected for proper use. This is not to say that parallel only interface could not be used, it is just that you will not be able to know if all commands completed properly etc. etc. The following protocol is used by the host software in order to properly communicate with the PROMICE units (all numbers are in hex):

1. Establishing baud rate: '03' (control-c) is sent by the host software at the desired baud rate. The host will keep sending '03's until is receives '03' back from the PROMICE chain. '03' is used by the PROMICE units to determine the incoming baud rate. If the parallel only mode is being used then this step is used only to step the PROMICE through this part of its internal code. The PROMICE unit will actually set the parallel port to be bidirectional if it receives the '03' on the parallel port at this point in its internal processing.

2. Establishing ROM IDs: Once '03' is received by the host. The host will proceed to determine the number of ROMs present and assign then IDs. This is done by sending a packet '00 00'. The first unit receiving this packet will assign ID-0 to it self and if it has a slave unit then will assign ID-1 to the slave ROM. It will then send out '00 01' or '00 02'. The next unit in chain will process the command to further assign IDs. The host will ultimately receive '00 nn' where nn is between '01' and 'FF' (actually one of '01','02',...'FF','00') indicating the actual number of ROMs present on the link. Once the link is established the two systems communicate in an unbalanced mode, i.e. host is always the master of the link, PROMICE units respond to host sent commands.

3. Packet oriented communication: The host sends out commands to the PROMICE units. Each command starts with an ID followed by the command byte. That is followed by a count byte for the count of all the data that follows. A count of '00' means 256 bytes. This allows easy pass through of the commands by each unit that is not addressed by the command. Following commands may be issued by the host with corresponding response from PROMICE unit addressed:

command	coding	response
Load Pointer	ID,00,03,EX,HI,LO	ID,80,01,RC
Write	ID,01,CC,DATA	ID,81,01,RC
Read	ID,02,01,CC	ID,82,CC,DATA
Restart	ID,03,01,DD	ID,83,01,RC
Mode	ID,04,01,mm	ID,84,01,RS
Test RAM	ID,05,01,pc	ID,85,01,RC
response if Test failed		ID,85,03,EX,HI,LO
Reset Target	ID,06,01,pw	ID,86,01,RC
Modify Byte	ID,07,01,DD	ID,87,01,RC
Establish Link	ID,08,03,EX,HI,LO	ID,88,01,RC
Write Message	ID,09,CC,DATA	ID,89,01,RC
Read Message	ID,0A,01,CC	ID,8A,CC,DATA
Link Mode	ID,0B,01,mm	ID,8B,01,RC
AI BreakPoint	ID,0C,04,Addr,DD	ID,8C,01,RC
AI HdwTrap	ID,0D,03,Addr	ID,8D,01,RC
Pi Execute	ID,0E,nn,data	ID,8E,nn,data
Report Version#	ID,0F,01,DD	ID,8F,04,VERSION#

Where ID = unit id; EX,HI,LO/Addr = three bytes of address; CC=character count; mm=mode; pc=pass count; pw=pulse width; RC=response code; RS=ram size

In addition the commands may have a no-response bit set that will cause the PROMICE unit to suppress the response. This technique is useful in fast loading of data.

DATA TRANSFER: In order to transfer the data to and from PROMICE units, three things must be done. 1) put the unit in LOAD mode by using the Mode command. 2) Load the address to which input or output is to be done. 3) Do the actual Read or

Write commands. After I/O is done the Mode command should be used to put the unit back in to Emulation mode.

A brief description of each command follows: (if the command byte is OR'd with x20 then PROMICE will generate no response to the command. LoadICE uses this primarily on *write* commands to speed up data transfer.)

Command 00: *Load Pointer:* This command is used to send a 20 bit pointer to the PROMICE unit. This is the address at which read/write operation will be done. For each byte transferred the pointer is incremented internally. *The micro-controller internally only has a 16 bit pointer that it increments on each access. It is the responsibility of the external software to ensure that the pointer is reloaded with proper values every time it crosses the 64k boundary!!.*

Command 01: *Write Data:* This command transfers up to 256 bytes of data to the PROMICE. The count specified should be a value between 00 and FF, with 00 meaning 256. The data is stored in the RAM starting at the current pointer value.

Command 02: *Read Data:* Count bytes of data is sent from the PROMICE to the host. Since there is no flow control, the external software must ensure proper buffering or request small amounts of data at a time. Once again the count value of 00 implies 256 bytes.

Command 03: *Restart unit:* This code is normally used when the external software has done talking to the unit all together. A unit will go back to where it will seek for auto baud code.

Command 04: *Mode:* This command is used for changing operating mode of a given module. The response always contains the Ram Size encoded as four bits. In case of the master module the high nibble also contains the size of the slave module's memory. This way the external software can make a distinction between slave and master units. The size is encoded as follows: 0-no ram; 1-2k; 2-4k; 3-8k; 4-16k; 5-32k; 6-64k; 7-128k; 8-256k; 9-512k; A-1m. All sizes are in bytes. To change the operating mode of the unit the argument passed is bit encoded as follows: Bit-0/0-emulate, 1-load. Bit-7/1-send response at full baud rate, 0-send response at baud rate <9000.

Command 05: *Test RAM:* This command will cause the RAM on the given module to be tested. The test will run as many passes as specified by the pass-count. The response code will be zero unless an error occurs, in which case the response is the address at which the error was encountered.

Command 06: *Reset Target:* This command drives the reset pin on the male header on the back of the unit. The signal is driven for as many units of time as specified in the argument (pw). The basic unit of time is 8.9 milli seconds. The maximum value that can be specified is FF, or approximately 2.5 seconds. The signal is driven low unless switch 1 on the back panel is on, in which case the signal is driven high.

Command 07: *Modify Byte:* This command will cause the unit to be taken out of emulation mode, a byte at the current pointer modified and the unit is put back in emulation mode. There is a *request/grant* protocol that is followed to suspend the target system during the modification. The reset signal serves the dual purpose as *request* line and HSI signal on the back panel is used to sense the *grant* line. When the target *grants* the *request,* the unit is taken out of emulation and the byte changed. The unit is put back into emulation and the *request* is released.

Command 08-0E: *Host/Target Communication Protocol Commands:* See Appendix -6 for description of these commands.

Command 0F: *Report Version#:* This command is used to inquire the micro code version#. The unit sends back four bytes or ASCII data as the version#.

4. Flow control: The PROMICE can keep up with the data coming in at any of the supported baud rates. However, very often a host might not be able to accept data from the PROMICE at the full rate. By default the PROMICE unit will communicate back at a maximum of about 9000 baud. This is accomplished by using a timer. The mode command may be used to turn off this slow transmission. There is no other flow control scheme implemented by PROMICE.

On the parallel port there is a hardware handshake for each unit of data transfer. Therefor the flow is regulated by the Host systems ability to serve the parallel interface.

5. RS-232C Parameters: Set the transmission for 8 bits, no parity, and two stop bits. The two stop bits are necessary only if you are going to daisy chain multiple units.

The specific reason for the two stop bits originates from the fact that the UART in the micro controller (87C451) is buffered on the receive side but is unbuffered on the transmit side. Since some processing is required to determine when the transmitter is empty and loading the transmit buffer with a new character, no matter what the baud rate, the transmitter can not keep up with a receiver getting data at full speed. The insertion of an extra stop bit gives the micro enough time to do its processing for transmitting the data to units further down the chain. Even though the transmitter is transmitting with one stop bit , the successive units will not see the problem since the data flow rate is being limited by the transmitter speed of the first unit.

APPENDIX-3

Terminal mode of operation

If a PROMICE unit detects '0D' (carriage return) code at the auto baud time it decides that the user is communicating with it from a terminal. It further assumes that only one unit is present and no daisy chaining is supported. The unit will then take commands interactively from the user and can also directly load hex records (not implemented in current versions). Only Motorola S1, S2 and S3 records and Intel hex (standard and extended) records are supported. The commands all begin with a '.' Following are all the valid commands:

.a address	operate AI as a tty with UART mapped at the given address (control-shit-2 to exit).
.d address data	deposit the data at the given address.
.e address range	examine one or more bytes at given location. Promice will display the actual address being accesses (see Appendix-I). For slave unit the high nibble contain a 1.
.g	go into emulation mode.
.h	print help
.m nn	report or set mode, also reports size
.p address data	do AI patch function with given data to be patched at given address.
.q	quit emulation, go in to load mode
.r nn	reset the target for nn*8.9 milliseconds
.t	test emulation RAM
.x	switch from master to slave or vice-versa
q	*restart interface*

In the above commands all numbers are in hex. The examine and deposit commands will open the next location if a line-feed is typed instead of carriage-return. Test command can be interrupted by hitting a key. typing 'q' will restart the

interface allowing re-establishing the communication link at a different baud rate or via *loadice* program.

It may be notes that maximum transfer rate out of PROMICE is limited to ~9000 baud as a result of default flow control scheme implemented in the PROMICE.

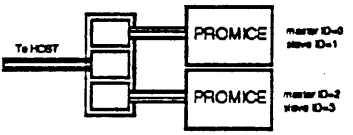
The AI board if present in the system can be operated from the terminal interface to emulate a transparent operation of the PROMICE unit. If you have a piece of code running on the Target system that can communicate to a serial line then it is trivial to interface it to operate via the AI system. The '.a' command is used to specify the mapped address of the AI and enable this transparent mode. This way for all practical purposes the PROMICE serial port can be used directly to communicate with the downloaded code.

There is no particular reason why the terminal interface can not be operated from the parallel port when it is used bidirectionally. However it requires that some Host based terminal emulator be able to handle the bidirectional parallel protocol. The detail of this protocol can be found in the LoadICE source code.

APPENDIX-4

USING PROMICE IN 32 BIT MODE

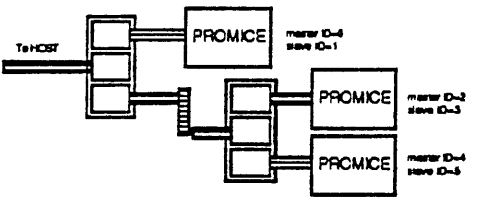
PROMICE units can be daisy chained by using the *daisy-chain module* . The daisy-chain module will allow you to hookup two Promice units, each with two emulation modules to emulate four ROMs in 32 bit mode. Refer to the following diagram for connection of units and their IDs:



The IDs of the four modules are fixed by their position on the daisy chain. When down-loading data that is in 32 bit format in the host, the first byte goes into ID=0 and second into ID=1 and third into ID=2 and fourth into ID=3.

If the data is in 8 bit format, i.e. file per ROM then it can be loaded in any particular ID by specifying where each file goes.

In general multiple units can be daisy chained by attaching successive daisy-chain modules to each other. After the first unit, one daisy chain module is required per additional unit. A maximum of 256 ROMs worth of units can be daisy chained.



The IDs assigned in the above picture assume the units to be all duplex models. If some of the model were simplex the IDs will still be assigned sequentially, i.e. there are no holes in the ID assignments.

DAISY CHAINING UNITS ON PARALLEL PORT:

Only two units can be daisy chained on the parallel port. This requires a cable made specially to support two units. The hookup is very simple, the cable has a DB25 male adapter on one end to connect to the PC parallel port and two female headers to connect to the two PROMICE units. Currently only the duplex units to operate as four ROMs are supported. However this restriction is strictly a result of *Loadice* software.

The serial port also needs to be daisy chained for this operation. the parallel port only operation is not possible on daisy chained units.

APPENDIX-5

Emulating RAM

The PROMICE may be used to emulate a RAM. However some consideration must be made when doing so. We do not emulate a RAM in the sense that you can not simply plug the unit into a RAM socket. The 'write' signal is provided to the PROMICE over a separate wire. Use a mini-clip wire and attach the jumper end to the pin marked 'WRT' on the right angle headers sticking out of the back panel. Make sure that you connect only to the marked pin. Attach the other end of the cable, which has a clip, to the R/W_ line of the target processor. It is assumed that write signal is low asserted.

If the target processor does a write cycle to the ROM, i.e. chip_select is low and write signal is low (output_enable is ignored), then you will end up modifying the addressed byte in the ROM. No other operation is affected.

Appendix-6

HOST/TARGET COMMUNICATIONS PROTOCOLS

PROMICE product offers two sophisticated methods of establishing communications between the software in the Host system and the software in the Target system. The physical link for these protocols is achieved via the PROMICE ROM emulation cable connected to the Target's ROM site and the RS232C link or the parallel link connected to the appropriate port on the Host.

The first method, called PICOM, utilizes the basic design of the PROMICE unit (Pi) and some capabilities of the Target system. In general the communication mechanism uses the ability of the Pi unit to emulate RAM from the Target side, and the use of Bus Request/Bus Grant (HOLD/HOLDA) handshake to read/write the emulation memory without crashing the Target system. Furthermore, Pi can interrupt the Target system when the emulation RAM is written into by the Pi with Host data. The Target can also interrupt the Pi by either directly driving an interrupt line to the Pi or by a special modification in the Pi that allows Target driven write cycles to the ROM space (RAM emulation) to cause interrupts within the Pi unit. A status byte is used in conjunction with above mechanism to qualify significant data movements, i.e. actual data transfer to/from Host vs. setting break points or patching code. This method also requires hooking up the Request and Grant lines from the back of the unit to the Target system. In addition if interrupts to the target are desired then an interrupt line needs to be hooked to the target as well. The interrupt from the target can also be hooked up to the Pi or the target initiated memory writes into the emulation space can cause interrupts.

The second method, called AICOM, is based on GEI's proprietary Analysis Interface option (AI). This option adds a board to the PROMICE system. AI implements a full duplex UART that operates through the ROM space. AI also allows changing of the down-loaded code for setting break points etc. as well as the specification of a hardware

trap. This method of communication is the least intrusive on the Target system's hardware. No other connection besides the emulation ROM cables are required between the Target and the PROMICE unit.

PICOM:

In order to establish communication between the Host and the Target system, following steps and commands are involved:

1. Establishing the Link:

The link is established by passing to the PI a pointer to the Data Transfer Area (DXA). This area is a contiguous piece of emulation space (within the Master ROM module) and consists of the following:

```
|status_byte|byte_count|data_characters.....
..|
```

The status byte contains the following bits:

```
|ENB|x|x|BUSY|ERR|IACK|HDA|TDA|
```

PI will monitor the Status byte on each interrupt from the Target for completion of data transfer.

Status bits are as follows :

0:TDA : Target Data Available: Set by the Target after data has been transferred into the DXA. Cleared by the PI when data has been transferred to the Host. PI also clear the BUSY bit.

1:HDA : Host Data Available: Set by the PI when data from the Host is stored into the DXA. Cleared by the Target when data has been read out of the DXA. Target also clear the BUSY bit.

2:IACK: Interrupt Acknowledgement: Set by the PI when it has seen the status byte updated by the Target. This is to cope with the circumstance when PI has missed the Target interrupt that updated the TDA ,HDA or BUSY bits. If the Target does not see the IACK bit after the status byte is written, it will interrupt the PI again (directly or by writing the status byte in the DXA).

3:ERR: Error: This bit is set by PI to inform the Target that a Host *write* command has failed. This would happen if somehow the PI was not able to accept Host data.

4:BUSY: This bit is set either by the Target or the Pi when either wishes to transfer data to the DXA and BUSY is not already set. At the end of the data transfer either TDA or HDA will be set by the respective system. BUSY remains on until the data has reached its eventual destination.

5:6:X: unused.

7:ENB: Enable the Interface: This bit is set or cleared by the Pi to turn the whole interface off or on. This bit is controlled by the *mode* command (see #4 below). *When the interface is turned off the interrupts from the target are disabled and the write command will result in the error 'link is not up'.*

The following command will setup the link, ';' separates bytes:

`ID;CLINK+MASK;03;EX;HI;LO`

Where ID is the unit id of the PROMICE unit, typically 0.

CLINK is the command code for this specific command (in low nibble).

MASK contains the modifier bits in the high nibble of the command byte. These bits are as follows:

Bit# 7 - 1 - set to indicate this protocol instead of the AI protocol.
Bit# 6 - INIT - set to initialize the interface - zero's the status byte (set 1st time).

Bit# 5 - NORSP* - set to indicate no response to host at command completion.
Bit# 4 - TINT - set to indicate, interrupt the target when done.

03: count of data to follow.

EX:HI:LO: three bytes containing 20 bit address of the DXA.

The following is the typical response to the *link* command:

`ID;CLINK+MASK;01;dd...`

---

* NORSP is a global bit used in all commands to speed up downloads and throughput.

where MASK contains the following bits:

Bit# 7 - DONE - command completed (always set).
Bit# 4 - ERR - if command encountered error.

If no ERR is set then dd=00 else:

dd=error code - one byte error code (standard Pi error codes listed elsewhere).

2. Data Transfer from the Host to the Target (WRITE):

This is done via the *write* command. The operation will write the byte count and the data into the DXA and then interrupt the target.

The following command will cause data to be transferred:

`ID;CWRITE+MASK;cc;dd...`

Where ID is the unit id of the PROMICE unit, typically 0.

CWRITE is the command code for the write command (in low nibble).

MASK contains the modifier bits in the high nibble of the command byte. These bits are as follows:

Bit# 7 - 1 - set to indicate this protocol instead of the AI protocol.
Bit# 6 - ASYNC - set to indicate asynchronous response required at completion of data transfer to the Target. i.e. when the Pi sees data transfer complete as a result of status update by the Target a response will be generated and sent to the Host. This bit in conjunction with NORSP can effectively synchronize write operations. Host sees a full completion before doing the next *write*.
Bit# 5 - NORSP- set to indicate no response to Host at command completion.
Bit# 4 - TINT - set to indicate, interrupt the Target when done transferring data to the DXA.

cc - is the byte count of the data that follows, the byte count is stored in the DXA.

dd.... - is the actual data, there must be cc number of bytes.

When the *write* command is issued, Pi will check to see if DXA is free. If not ,it will send back response indicating command completion with error. The response will look like this:

ID;CWRITE+MASK;02;error code;status byte

where MASK contains the following bits:

Bit# 7 - DONE - command completed (always set).
Bit# 4 - ERR - command encountered error.

error code - one byte error code (standard Pi error codes listed elsewhere).

status byte - the status byte from the DXA.

If no error is encountered in the *write*, then immediate execution of the command will result only in storing the data etc. in the DXA. If NORSP is clear then the following response will be sent to the Host:

ID;CWRITE+MASK;01,00

The MASK will only contain the DONE bit.

If NORSP is set and ASYNC is also set then the response will be generated only when Pi detects that the target has read the data. The response will be same as above.

If NORSP is clear and ASYNC is set, then two responses just like the above will be generated. One upon the immediate completion of write (this basically will indicate that the data was written into the DXA) and another when the Target has read the data.

3. Data Transfer from the Target to the Host (READ):

This is done via the *read* command. The operation will return any data in the DXA that may have been placed there by the Target.

The following command will cause data to be transferred:

`ID;CREAD+MASK;01;00`

Where ID is the unit id of the PROMICE unit, typically 0.

CREAD is the command code for the read command (low nibble).

MASK contains the modifier bits in the high nibble of the command byte. These bits are as follows:

Bit# 7 - 1 - set to indicate this protocol instead of the AI protocol.
Bit# 6 - ASYNC - set to indicate that if no data is available from the Target (right then) then to send data later whenever it is available. This will happen when on

interrupt from Target Pi finds that there is data in DXA for the Host. It will then proceed to send that data to Host at that time. This bit in conjunction with the NORSP bit effectively synchronizes the read operation.
Bit# 5 - NORSP- set to indicate no response to Host at command completion.
Bit# 4 - TINT - set to indicate, interrupt the Target when done sending data to the Host.

Normally the *read* operation will not succeed without knowing if Target has put the data in the DXA. However, by setting the ASYNC bit a *read* can be issued to be completed later. The following is the typical response to the *read* command:

```
ID;CREAD+MASK;cc;dd...
```

where MASK contains the following bits:

Bit# 7 - DONE - command completed (always set).
Bit# 4 - ERR - if command encountered error.

If ERR is set then cc=2 and the two bytes that follow are as below:

error code - one byte error code (standard Pi error codes listed elsewhere).
status byte - the status byte from the DXA.

If no ERR then:

cc - is byte count of data to follow.

dd... - is the data from the DXA.

4. Mode setting:

The polarity and control of various signals is specified by the mode command.

```
ID;CMODE+MASK;01;MODE
```

Where ID is the unit id of the PROMICE unit.

CMODE is the command code for the mode command, in the low nibble.

MASK is the high nibble of the command byte, it contains the following bits:

Bit# 7 - 1 - set to indicate this protocol instead of the AI protocol.
Bit# 6 - CHANGE - change the mode per mode byte (clear when only interrupting the Target).

Bit# 5 - NORSP- set to indicate no response to Host at command completion.
Bit# 4 - TINT - set to indicate, interrupt the Target when done, i.e. now.

MODE - This byte contains bit encoded values as follows:

Bit# 7 - PICOM - Turn the protocol on and off (controls ENB bit in status in DXA).
Bit#6 - ASYNC - global asynchronous mode bit, all *reads* are async. i.e. send data to Host whenever the Target has data ready for it.
Bit# 5 - REQH - set to indicate that BusRequest (HOLD) be high asserted.
Bit# 4 - ACKH- set to indicate that BusGrant (HOLDA) will be high asserted.
Bit# 3 - INTH - set to indicate, that interrupt to Target is to be high asserted.
Bit# 2 - GRINT - global sync read interrupt flag, goes with ASYNC above. Causes the Pi to interrupt the Target on async reads.

The following is the typical response to the *mode* command:

ID;CMODE+MASK;01;dd...

where MASK contains the following bits:

Bit# 7 - DONE - command completed (always set).
Bit# 4 - ERR - if command encountered error.

If no ERR is set then dd=00 else:

dd=error code - one byte error code (standard Pi error codes listed elsewhere).

5 Miscellaneous:

Generally it is expected that the above protocol should work fine as laid out. There is no provision in the protocol for contention over DXA by the Host and the Target at the same time. The BUSY bit should cope with most of the contention cases, except where the Target and Pi are reading the status byte very close to each other. In reality since the two entities are in communication, it is expected that they will be effectively in half-duplex mode, i.e. Host sends a command and the Target executes it and sends the results back. However there may be cases where an interrupt type of mechanism may be desired to get the

other ends attention. In such cases the Host may choose to reinitialize the link and /or interrupt the Target via *mode* command.

In addition the standard *modify_byte* command is available for the Host to change any data in the emulation memory by using the REQ/ACK mechanism. This operation is transparent to the PICOM protocol.

NOTE: The asynchronous operation relies upon the Targets systems ability to interrupt the PROMICE micro. If the target can not interrupt the micro then the I/O is expected to be accomplished by the CWRITE and CREAD commands only. In other words, there is no way for PROMICE to know when the Target is reading or writing the data to/from the DXA and only at the execution of the CWRITE and CREAD commands the micro looks for empty or full buffer. Therefor if a previous CWRITE did not complete, i.e. Target system did not read the data, then only a successive CWRITE can indicate so by returning an error. In the same way only a CREAD can determine if the target has put any data in to the DXA.

AICOM:

In order to establish communication between the Host and the Target system, following steps and commands are involved:

1. Establishing the Link:

The link is established by passing to the Pi a pointer to the AI Control Buffer (ACB). This area is a contiguous piece of emulation space (within the Master ROM module) and consists of the following: four bytes

```
|1|0|HostData|Status|
```

The status byte contains the following bits:

```
|x|x|x|x|0|0|HDA|TDA|
```

Pi will monitor the Status byte on each pass through its Main Scheduling Loop and take appropriate action as programmed.

Status bits are as follows :

0:TDA : Target Data Available: This bit is set when the Target has transferred one data byte to the AI interface. It is cleared automatically when the Pi reads the Target data.

1:HDA : Host Data Available: Automatically set when Pi writes a byte of data into the HostData location of the ACB. It is cleared automatically when the Target reads the same location (HostData location in the ACB.)

2-3:00: These bits are always zero. The target uses this information to successfully decide that the AI communications interface is active. *This is as opposed to the AI break or trap interface being active, in addition to the AI system being initialized (and therefor inactive).*

The following command will setup the link:'

`ID;CLINK+MASK;03;EX;HI;LO;` *(' separates the bytes)*

Where ID is the unit id of the PROMICE unit, typically 0.

CLINK is the command code for this specific command (in low nibble).

MASK contains the modifier bits in the high nibble of the command byte. These bits are as follows:

Bit# 7 - 0 - clear to indicate this protocol instead of the PI protocol.
Bit# 6 - INIT - set to initialize the interface.
Bit# 5 - NORSP - set to indicate no response to host at command completion.
Bit# 4 - TINT - set to indicate, interrupt the target when done.

03: count of data to follow.

EX:HI:LO: three bytes containing 24 bit address of the ACB.

The following is the typical response to the *link* command:

ID;CLINK+MASK;01;dd...

where MASK contains the following bits:

Bit# 7 - DONE - command completed (always set).
Bit# 4 - ERR - if command encountered error.

If no ERR is set then dd=00 else:

dd=error code - one byte error code (standard PI error codes listed elsewhere).

2. Data Transfer from the Host to the Target (WRITE):

This is done via the *write* command. The operation will write the byte count and the data into the internal buffer (IBUF) within Promice. Later in the Main Scheduler Loop this data will be sent over the AI interface to the Target system.

The following command will cause data to be transferred:

`ID;CWRITE+MASK;cc;dd...`

Where ID is the unit id of the PROMICE unit, typically 0.

CWRITE is the command code for the write command (in low nibble).

MASK contains the modifier bits in the high nibble of the command byte. These bits are as follows:

Bit# 7 - 0- clear to indicate this protocol instead of the PI protocol.
Bit# 6 - ASYNC - set to indicate asynchronous response required at completion of data transfer to the Target. i.e. when the Pi has completed the data transfer a response will be generated and sent to the Host. This bit in conjunction with NORSP can effectively synchronize write operations. Host sees a full completion before doing the next *write*.
Bit# 5 - NORSP- set to indicate no response to Host at command completion.
Bit# 4 - TINT - set to indicate, interrupt the Target when done. I.e. data is ready for the target in the IBUF. Pi will keep looking for opportunities to send this data during its Main Scheduling Loop.

cc - is the byte count of the data that follows, the byte count is stored in the IBUF.

dd.... - is the actual data, there must be *cc* number of bytes.

When the *write* command is issued, Pi will check to see if IBUF is free. If not ,it will send back response indicating command completion with error. The response will look like this:

```
ID;CWRITE+MASK;02;error code;status byte
```

where MASK contains the following bits:

Bit# 7 - DONE - command completed (always set).
Bit# 4 - ERR - command encountered error.

error code - one byte error code (standard Pi error codes listed elsewhere).

status byte - the status byte from the ACB.

If no error is encountered in the *write*, then immediate execution of the command will result only in storing the data etc. in the IBUF. If NORSP is clear then the following response will be sent to the Host:

```
ID;CWRITE+MASK;01,00
```

The MASK will only contain the DONE bit.

If NORSP is set and ASYNC is also set then the response will be generated only when Pi has sent all the data

successfully to the Target over the AI interface. The response will be same as above.

If NORSP is clear and ASYNC is set, then two responses just like the above will be generated. One upon the immediate completion of write (this basically will indicate that the data was written into the IBUF) and another when all of the data has been transferred to the Target.

Note that a time-out error may occur during the actual data transfer, in that case the ASYNC bit will cause an error response to be generated and sent to the Host.

3. Data Transfer from the Target to the Host (READ):

This is done via the *read* command. The operation will return any data in the IBUF that may have been received from the Target over the AI interface.

The following command will cause data to be transferred:

```
ID;CREAD+MASK;01;00
```

Where ID is the unit id of the PROMICE unit, typically 0.

CREAD is the command code for the read command (low nibble).

MASK contains the modifier bits in the high nibble of the command byte. These bits are as follows:

Bit# 7 - 0 - clear to indicate this protocol instead of the PI protocol.
Bit# 6 - ASYNC - set to indicate that if no data is available from the Target then to send data later whenever it is available. This will happen when in its Main Scheduler Loop PI finds that there is data coming in from the Target system. It will accept the data from the AI interface and put it in the IBUF. If ASYNC bit is set then it will proceed to send this data to the Host . This bit in conjunction with the NORSP bit effectively synchronizes the read operation.
Bit# 5 - NORSP- set to indicate no response to Host at command completion.
Bit# 4 - TINT - set to indicate, interrupt the Target when done sending data to the Host. I.e. there is room in IBUF to accept more data.

Normally the *read* operation will not succeed without knowing if PI has put some Target data in the IBUF.

However, by setting the ASYNC bit a *read* can be issued to be completed later. The following is the typical response to the *read* command:

`ID;CREAD+MASK;cc;dd...`

where MASK contains the following bits:

Bit# 7 - DONE - command completed (always set).
Bit# 4 - ERR - if command encountered error.

If ERR is set then cc=2 and the two bytes that follow are as below:

error code - one byte error code (standard PI error codes listed elsewhere).
status byte - the status byte from the ACB

If no ERR then:

cc - is byte count of data to follow.

dd... - is the data from the IBUF.

4. Mode setting:

The mode command does some minor tasks including interrupting the target if so desired.

`ID;CMODE+MASK;01;MODE`

Where ID is the unit id of the PROMICE unit.

CMODE is the command code for the mode command, in the low nibble.

MASK is the high nibble of the command byte, it contains the following bits:

Bit# 7 - 0 - clear to indicate this protocol instead of the PI protocol.
Bit# 6 - CHANGE - change the mode per mode byte (clear when only interrupting the Target).
Bit# 5 - NORSP- set to indicate no response to Host at command completion.
Bit# 4 - TINT - set to indicate, interrupt the Target when done, i.e. now.

MODE - This byte contains bit encoded values as follows:

Bit# 7 - AICOM - Turn the protocol on and off (disables or enables AI circuit).
Bit#6 - ASYNC - global asynchronous mode bit, all *reads* are async. i.e. send data to Host whenever the Target has data ready for it.

Bit# 5 - unused.
Bit# 4 - unused.
Bit# 3 - INTH - set to indicate, that interrupt to Target is to be high asserted.

The following is the typical response to the *mode* command:

ID;CMODE+MASK;01;dd...

where MASK contains the following bits:

Bit# 7 - DONE - command completed (always set).
Bit# 4 - ERR - if command encountered error.

If no ERR is set then dd=00 else:

dd=error code - one byte error code (standard Pi error codes listed elsewhere).

5. Break Points

In order to set a break point the Host must specify the address and the new data value for any emulated ROM space within the Master or the Slave module of the PROMICE unit. Furthermore, the Host must pass this information to the Target system before requesting the break point from the Pi.

The Target system will commence reading the location specified by the break point address from the Host. When it detects that the contents of the addressed location have changed, it will go on to reconnect with the Pi unit over the AI interface.

The following command will cause the Pi to set up break point at the given address, the actually setting is caused by the Target reading the location:

`ID;CBREAK+MASK;05;EX;HI;LO;DD;TT`

Where ID is the unit id of the PROMICE unit.

CBREAK is the command code for the break-point command, in the low nibble.

MASK is the high nibble of the command byte, it contains the following bits:

Bit# 7 - 0 - clear to indicate this protocol instead of the PI protocol.
Bit# 6 - ASYNC - set to request response when the break point is set. The Target may fail to set the break point in

which case the Pi will time out waiting for the Target. It will at that point try to restore the communications link. The failure to set the break point will be reported back to the Host if ASYNC bit is set. This bit in conjunction to the NORSP bit will synchronize this command.
Bit# 5 - NORSP- set to indicate no response to Host at command completion.
Bit# 4 - TINT - set to indicate, interrupt the Target when communication link is restored.

EX;HI;LO are the 24 bit address of the location where break point is to be set.

DD- is the new data to be stored at the break point location.

TT - is the time-out value in units of 2.358 seconds each, for Pi to wait for the break point to set. A value of 00 will not time the event.

The following is the typical response to the *break-point* command:

ID;CBREAK+MASK;cc;dd...

where MASK contains the following bits:

Bit# 7 - DONE - command completed (always set).
Bit# 4 - ERR - if command encountered error.

If no ERR is set then dd=00 else:

dd=error code - one byte error code (standard Pi error codes listed elsewhere).

6. Hardware Trap

This feature of the AI system allows the Host to specify a location within a 24 bit address space for a trap condition. The lower 20 bits of this space are provided by the Master module's emulation address bus. The target supplies the top 4 lines via the header on the back of the PROMICE unit. In addition the Target also supplies a signal to be used as chip_select for the trap condition. The purpose here to catch an arbitrary event in a 24 bit address space that can be qualified by a 20 bit address derived from the ROM cable interface and additional 4 (programmable as to be 0s, 1s or don't cares) and an external select line. This trap must be setup by the Pi; armed by the Target system and then the Target must

execute the offending code to cause the trap to occur. The PI will generate a low asserted signal on the back of the unit as soon as the trap condition is detected. If this signal is used as in interrupt or to trigger some other device then the offending event occurred just as the signal is asserted.

The following command will cause the PI to set a trap to the given location:

```
ID;CTRAP+MASK;04;EX;HI;LO;MM;TT
```

Where ID is the unit id of the PROMICE unit.

CTRAP is the command code for the trap command, in the low nibble.

MASK is the high nibble of the command byte, it contains the following bits:

Bit# 7 - 0 - clear to indicate this protocol instead of the PI protocol.
Bit# 6 - ASYNC - set to request response when the trap has occurred The Target may fail to arm or trigger the trap in which case the PI will time out waiting for it. It will at that point try to restore the communications link. The failure to trigger the trap will be reported back to the Host if the ASYNC bit is set. This bit in conjunction to the NORSP bit will synchronize this command.
Bit# 5 - NORSP- set to indicate no response to Host at command completion.
Bit# 4 - TINT - set to indicate, interrupt the Target when communication link is restored.

EX;HI;LO are the 24 bit address of the location where break point is to be set.

MM is a one byte mask that specifies the treatment of the top 4 lines of the 24 bit address space. They are specified as follows:

```
|a23|a22|a21|a20|A23|A22|A21|A20|
```

If the 'a' bit is on then the corresponding address is expected to be a zero for the trap condition and if the 'A' bit is on then it is expected to be a one. If both 'a' and 'A' are zero then the bit is don't care.

TT - is the time-out value in units of 2.358 seconds each, for PI to wait for the trap to occur. A value of 00 will not time the event.

TRAP OPERATION: Before executing the above command the Host must pass the trap information to the Target system. At this point the Target will proceed to generate the trap condition over and over. The Pi in mean time will setup the trap circuit. The Target will know when the circuit is armed, when it is able to trigger the trap. At this point the Target will execute the offending code and wait for the trap to occur. The trap must be cleared by the Target by once more causing the trap intentionally.

These activities of the Target are monitored by the Pi and when the last trap has occurred it will restore the AI communications link. The target must execute its link establishment procedure to reestablish link with Pi.

It is possible for the Target to fail to arm or trigger the trap. In that case the Pi can time out and report the error back to the Host if ASYNC bit is set.

The following is the typical response to the *trap* command:

ID;CTRAP+MASK;cc;dd...

where MASK contains the following bits:

Bit# 7 - DONE - command completed (always set).
Bit# 4 - ERR - if command encountered error.

If no ERR is set then dd=00 else:

dd=error code - one byte error code (standard Pi error codes listed elsewhere).

DEFINITION OF ALL THE BITS AND BYTES USED BY PiCOM AND AiCOM

Here are all the bits and variables and other things used in PiCOM
They are directly taken from the micro-code listings:

Command codes for COM commands

CLINK	0x08	establish link
CWRITE	0x09	write data
CREAD	0x0a	read data
CMODE	0x0b	mode command
CBREAK	0x0c	set break point (AI only)
CTRAP	0x0d	set hardware trap (AI only)

Command modifier masks for above commands:

NRSP	0x20	no response to command execution (all commands)
MBRD	0x40	do a read byte (modify byte command only)
PICOM	0x80	command is for PiCOM (as to AiCOM)
CINIT	0x40	initialize the link (CLINK only)
ASYNC	0x40	do asynchronous i/o (CWRITE & CREAD)
CHANGE	0x40	change mode bits (CMODE only)
INTT	0x10	interrupt the target at command completion

Mode byte as passed by the COMDE command:

COMCOM	0x80	turn on/off the comm link
ASYNC	0x40	global asynchronous read
REQH	0x20	HOLD is high asserted
ACKH	0x10	HOLDA is high asserted
INTH	0x08	Interrupt to target is high asserted
GRINT	0x04	Global async read interrupt (for ASYNC above)

PiCOM status bytes bits:

PITDA	0x01	target data available
PIHDA	0x02	host data available
PIACK	0x04	interrupt ack; Pi has seen status change

PIERR	0x08	host write failed (overrun)
PIBUSY	0x10	interface busy (has TDA or HDA)
PIENB	0x80	interface is enabled

Error Codes returned in the first byte when command is in error (second byte will contain the status byte from data transfer area)

NONE	-1	no resource available (i.e. No AI board or Link is down)
BUSY	-2	interface is busy
TERR	-3	timed out waiting for ACK
CERR	-4	host write failed (overrun)
NODAT	-5	no data to read from target
NYET	-6	not implemented yet (AI only)

HOOKING UP PROMICE AUXILIARY SIGNALS TO TARGET SYSTEM:

In order to hookup the PROMICE unit for HOST/TARGET communication hookup the various mini-clips as follows:

BackPanel	*Target signal*	*Description*
RST	Reset	system reset signal
WRT	Write	system write signal
HSO	Interrupt	interrupt to the Target
HSI	Interrupt	interrupt from the Target
REQ	Request	System Bus Request (HOLD)
ACK	Grant	System Bus Grant (HOLDA)
XCS	Chip_select	Chip Select for Hardware Trap
A20	Address	Extra address or other inputs -
A21	Address	- for
A22	Address	- Hardware
A23	Address	- Trap use

Appendix-7

FILE FORMATS

LoadICE supports HEX and BINARY file formats. The HEX files are essentially ASCII files containing binary data that has been encoded as ASCII. This is done by taking each 8-bit bytes and encoding it as two characters, one per hex digit. So if the data byte is hex 'B9' (binary '10111001'), it will be encodes as two bytes containing the character 'B' (hex '42') and the next byte containing the character '9' (hex '39'). The most popular HEX formats are INTEL HEX and MOTOROLA HEX (also called S-RECORDS). Besides containing the ASCII encoded binary data there is other information contained in each HEX records. There is the record type identifier (':' for INTEL and 'S' for MOTOROLA etc. followed by a number somewhere in the record). Then there is the address information that specifies where the data in this particular record is to be loaded. Typically there is also a checksum in the record.

BINARY file on the other hand is not readable by human beings. It contains binary data and is normally directly produced by the compiler/linker etc. Most native development systems will produce a binary file as their intermediate and final output. Whereas most cross-development systems will produce a HEX file as their final output. Usually the BINARY file is an executable image intended to run on the native system. There is no particular address information in a BINARY file like the kind that is contained in each HEX record. However, if a BINARY file is to be down-loaded in to the PROMICE then the user must specify the address where it is to be loaded (typically at address 0 unless loading multiple binary files) furthermore, a BINARY file may contain information that is used by the native system (loader) that specifies things like execution start address, stack size, uninitialized data area size etc. This information is usually at the beginning of the file, hence the user must specify the number of bytes that the LoadICE must skip from the beginning of the BINARY file.

LoadICE supports the following HEX file formats and they are recognized by the software automatically:

Motorola S-record format S1, S2, S3.
Intel standard format, 64k byte limit.
Intel extended format, 1m byte limit.
Tektronix Hex format (Standard TEKHEX).
Tektronix Extended Hex format (Extended TEKHEX).
RCA COSMAC format.
MOSTEK format (as used by most 6502 assemblers).
Motorola DSP56001 format

Here are some example HEX records:

INTEL HEX RECORDS:

```
:20000000020034D219C2...6C2AAC0E074805249D0E032000081
:20002000000000C299C2...0F608B880FB1209D6120A79
:200040000B75880575A8...5C207C202C200D20A7A06120941
```

MOTOROLA HEX RECORDS:

```
S00600004D414427
S2204000000000040000...400508004005160040052447
S22040001C0040053200...40056A00400578004005861C
```

Appendix-8

Modifier Boards

In order to adapt non-standard ROMs such as PLCC or 16-bit devices, Grammar Engine currently supplies custom cables and adapters. There are typically hand made and are subject to breakage besides the high manufacturing costs. There are also some ROMs that we currently do not support as a result of not having the functionality within the PROMICE unit. An example of such a device is ROMs that either have address latches in them or paged ROMs line 27513 etc. where the page address is expected to be latched from the data bus.

Then there are other emulation related problems such as detecting target system power shutdown so that we can turn off the emulation on the PROMICE. This normally causes problems for very low power CMOS systems. The target shutting itself down will appear to be accessing the ROM and hence the PROMICE will drive the data lines out. This causes the target system to draw power through input protection diode and in some cases actually remain powered up even though the power has been shut down. Parasitic operation of the PROMICE would normally solve this problem but if the target is running from a very limited supply or batteries then it is not desirable to power the PROMICE from the target.

In any case, to find a solution to all these problems, we have invented the ROM Function Modifier Boards. On the back of the PROMICE unit, the ROM cable is attached to a 17x2 male header. We are in process of designing small boards that are approximately as long as the header and about an inch or so wide. These boards will contain the various specialized functions and will be inserted between the female header on the ROM cable and the male header on the PROMICE unit. In some cases these adapter boards may be inserted between the cable itself, i.e. one short ROM cable and one short cables with female headers at both ends.

TUTORIAL

Copy all the files onto your working directory.
At DOS (system) prompt type `loadice -d` which will start the LoadICE application and then enter the dialog mode.
You should see:

```
LoadICE V 1.5a
Copyright (C) 1990 Grammar Engine, Inc.
Initializing defaults from file: loadice.ini
Force fill with ffff
filename = TUT.BIN  skip = 0  offset = 200
filename = TUT.INT  offset = - f000
filename = TUT.MOT  offset = - f00
Establishing connection with Promice
Module #0 Model 512 Emulating 8192 bytes
Operating Mode 8 bits
Building ROM image
LoadICE: t 0      TEST THE PROMICE MEMORY
Unit number 0 passed the memory test
LoadICE: v 0      FIND OUT THE MICRO CODE VERSION#
Promice Version 4.0h
LoadICE: l        LOAD THE IMAGE INTO PROMICE
Loading @ baudrate = 19200
Loaded
Loading complete - 8192 data bytes transfered
LoadICE: d 0:0    DUMP FIRST 16 BYTES OF TUT.INT
00:00000 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
LoadICE: d 0:100  DUMP FIRST 16 BYTES OF TUT.MOT
00:00100 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
LoadICE: d 0:200  DUMP FIRST 16 BYTES OF TUT.BIN
00:00200 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
LoadICE: e 0:0    EXAMINE/EDIT THE FIRST BYTE OF CODE
00:00000 00   [00] FF
00:00001 01   [01] F0
00:00002 02   [02] 0D
00:00003 03   [03] ^
00:00002 0d   [02] X
LoadICE: d 0:0    DUMP 16 BYTES TO VIEW CHANGES
00:00000 ff f0 0d 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
LoadICE: C        COMPARE PROMICE WITH IMAGE
(0:00000 I=00 P=ff)(0:00001 I=01 P=f0)(0:00002 I=02 P=0d)
Verified
LoadICE: x        EXIT LOADICE
```

The initial setup parameters are set using file loadice.ini The file supplied on the disk looks like this:

ROM=2764	SET THE ROM SIZE TO 8KX8
NUMBER=1	TELL LOADICE YOU ARE USING 1 PROMICE
WORD=8	SET 8 BIT LOAD MODE
OUTPUT=COM1	SET SERIAL PORT TO COM1
BAUD=19200	SET BAUD RATE TO 19200
IMAGE=TUT.BIN 0=0:200	LOAD BINARY IMAGE FILE TUT.BIN, MAPPING THE FIRST BYTE INTO PROMICE ADDRESS 200
FILE=TUT.INT F000=0:0	LOAD INTEL HEX FILE TUT.INT, MAPPING ADDRESS F000 INTO PROMICE ADDRESS 0
FILE=TUT.MOT 1000=0:100	LOAD MOTOROLA HEX FILE TUT.MOT, MAPPING ADDRESS 1000 INTO PROMICE ADDRESS 100
NOVERIFY	SET THE NO VERIFY OPTION TO SPEED UP LOADING

TROUBLE SHOOTING

Before getting frustrated check a few basic things:

1. POWER SOURCE: Make sure that you know how your unit is being powered. The shorting block on the back of the unit should be on one of the four positions marked for power. THERE SHOULD ONLY BE ONE JUMPER FOR POWER PER MODULE OR YOU WILL BE CONNECTING YOUR EXTERNAL POWER TO YOUR TARGET SYSTEM POWER ETC. If you are powering the unit parasitically, make sure that the jumper is set on the right pins for your size of ROM cable (24, 28 or 32). If you have a duplex unit then make sure that the slave unit power selection is also set properly.

In some cases the unit may appear to have power to it (red LED is on!) but actually is improperly powered. One such case is when the unit is set for external power operation, it is connected to the target system and the target system is powered on. In this case the CMOS address buffers in the unit are drawing power through the protection diodes on their input. The micro in the unit may actually be operating but the non-volatile memory controller in the unit will not let the memory be accesses. Usually the LoadICE application will detect this case, it can also be detected by the low level of illumination of the power LED. The other case is when the external power supply is inadequate to power the unit. This can happen if the address and or data buffer in the unit are replaced with some other type that draw far more power than the external supply can provide. You may choose to use a heftier supply but it is preferred that the unit be operated parasitically in such cases as the internal regulator may overheat.

2. ROM SIZE: Make sure that you specify the proper ROM size to LoadICE program. This is very crucial to proper emulation since the wrong specification can result in the data getting loaded in the wrong place in the unit. The size specification is usually in the *loadice.ini* file or on the command line. Check the message printed by LoadICE when it is run to see what size ROM its thinks it is emulating.

Also check the switches on the back of the unit. You should have all the switches up to and including the size of the ROM you are emulating in the 'on' position. If you don't have some 'on' or have some extra ones 'on' then the target won't be accessing the right space within the unit.

3. FILE SPECIFICATION: It is best to specify the files in the *ini* file or on the command line. Make sure that you have specified proper 'offset' for files that do not have data starting at address zero in the file, or if you have multiple files then to make sure that they get mapped properly.

If you are using binary files, then you must make sure that you tell LoadICE so. When you specify files in the 'dialog' mode (interactive at *Loadice*: prompt) make sure that you are making a distinction between HEX and binary files.

4. RS-232 HOOKUP: Make sure that you are using the cables and adapter provided by us. If you are unable to establish communication with the unit, most likely the problem is with improper linkage to your host computer. A quick way to check the unit's sanity is to hook a terminal or run a terminal emulator (even Kermit will do) on your PC and hit the <CR> key a few times. If the hookup is proper the unit will respond with its prompt message. While you are in that mode do the '.h' (help) command and few other things like '.t' (test ram) commands etc.

n. SUPPORT CALLS: If things are still out of control call us for prompt and courteous help at : - (614) 471-1113.

