

# MRCI

---

Model Reference Current Injection  
Reference Manual  
Edition 1.9.7

Robert Butera  
Ivan Raikov

---

Copyright © 1999-2005 Robert J. Butera, Jr., Ivan Raikov, and the Georgia Institute of Technology.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the Free Software Foundation.

# 1 Introduction

This document describes the third public release of MRCI, which is an acronym for Model Reference Current Injection, also known as the dynamic clamp [3] or artificial current injection.

## 1.1 Purpose of MRCI

MRCI stands for Model Reference Current Injection. The basic idea behind it is to simulate the ionic currents of a neuron in real-time, and aid the researcher in performing the following experiments:

- artificially insert ion channels into a neuron
- connect in vitro neurons with simulated synapses
- connect simulated neurons to in vitro neurons

Why use our software? It offers the following features:

- Fast. Works reliably up to 30 kHz, even with several state variables.
- Real-time logging of internal state variables.
- Arbitrary model specifications – if you can code it, you can do it.
- Real-time parameter changes (no need to shut the software off).
- Open Source. Modify it. Let us know (see the copyright on the next page).

## 1.2 Changes since the previous release

### 1.2.1 Changes between this release and release 1.9.1

1. Fixed a bug where the initial timestep (1 ms) was set incorrectly in seconds instead of milliseconds. Thanks to Marin Manuel for the bug report.
2. The outputs of the DAQ board are now initialized to zero when a model is loaded or unloaded.
3. The `setvar` command was extended to support assigning the value of a variable to another variable.

### 1.2.2 Changes between release 1.9.1 and release 1.9.0

1. Minor changes to the configure script to better support building Debian packages.
2. `mrci_extract` changed to output all variables when no options are supplied.

### 1.2.3 Changes between release 1.9.0 and release 1.8.0

1. Added support for COMEDI and RT-Linux Pro. The configuration scripts now supports option `-with-comedi`, which directs the MRCI build process to use the COMEDI drivers if available; and option `-with-rtlpro-shm`, which directs the build process to use the shared memory interface used with RT-Linux Pro. The RT-Linux Pro code is still experimental and is not considered stable.
2. Added support for the Debian package build system. There is now a binary distribution of MRCI in the form of `.deb` Debian packages and `.rpm` packages for RPM-based Linux distributions, such as RedHat or Mandrake.

### 1.2.4 Changes between release 1.8.0 and release 1.6.0

1. Added functionality to support remote mode—that is, a mode in which MRCI can be controlled from a remote computer via the network. A separate client application is now included in the MRCI distribution.

### 1.2.5 Changes between release 1.6.0 and release 1.5.0

1. Added commands `cd` and `pwd` which can change and print the current working directory. All MRCI commands that read or write files will use the current working directory.
2. Added support for shell commands: the user can now execute an arbitrary shell command from the MRCI command line. Shell commands must be prefixed by an exclamation mark.

### 1.2.6 Changes between release 1.4.0 and release 1.5.0

1. Added support for RTAI. Now the user can choose between using RT-Linux and RTAI at compile time.
2. Added two extra features to the event interface: 1) ability to compare two system quantities in event specification; 2) each event now has a refractory period associated with it, which is a time interval after event occurrence when new events are not recorded.

### 1.2.7 Changes between release 1.3.5 and release 1.4.0

1. Added functionality to detect and record events occurring during simulation. Commands `add event`, `enable event`, `disable event`, `del event`, `reset event`, `dump event` can be used to create and manipulate event specifiers, and write the event log to file.

### 1.2.8 Changes between release 1.3.0 and release 1.3.5

1. Added the `system` command which allows the user to switch between MRCI system modules.
2. Added a set of commands to save/load the command history and to show the current command history and delete commands from it.

### 1.2.9 Changes between release 1.2.0 and release 1.3.0

1. `Ctrl-C` now interrupts the current command, instead of terminating the program.

### 1.2.10 Changes between release 1.1.0 and release 1.2.0

1. Added "rampoff" command to turn off the currently active ramp.
2. Timing statistics now include information about the length of the computational cycle length; the core module measures the cycle length, in addition to the time between the beginning of each cycle.

## 1.3 Acknowledgments

We'd like to acknowledge the National Science Foundation for funding this project (DBI-9987074) and the Laboratory for Neural Control, National Institutes of Neurological Disorders and Stroke, National Institutes of Health for supporting initial development of this

software and ongoing testing and collaboration. In particular Drs. Jeffrey Smith, Chris Wilson, and Christopher DeNegro.

MRCI includes a driver for the National Instruments E-series data acquisition boards, with some modifications; this driver was originally developed by National Instruments Corporation and was modified by David Nguyen and Ivan Raikov.

The MRCI system description translator is based upon the Gnans translator, written by Bengt Martensson <[bengt@mathematik.uni-Bremen.de](mailto:bengt@mathematik.uni-Bremen.de)> of the Institute for Dynamical Systems. The MRCI translator was developed by Ivan Raikov.

MRCI includes the MCA2 real-time Linux math library, created by Stephen L. Moshier, Tobias Luksch <[luksch@fzi.de](mailto:luksch@fzi.de)>.

## 2 Installation and Running

### 2.1 Prerequisites

MRCI can be installed in one of two ways. A set of binary packages are available for installation on systems running Debian or Mandrake Linux. These packages can be found in the subsection of the download page entitled *Linux and RTAI packages*. After downloading and installing the packages for the distribution being used, proceed with the installation instructions in the next section.

The binary installation only provides a default configuration that uses the COMEDI data acquisition drivers and RTAI (Real-Time Application Interface) Linux. If the user desires to use a different driver, or RT-Linux, then MRCI has to be built from source code. In order to build MRCI, one needs a working installation of Real-Time Linux, also known as RT-Linux, or RTAI, which is an alternative to RT-Linux. RT-Linux ver. 3.2 can be found at <http://www2.fsmlabs.com/3.2-free.html>. RTAI can be downloaded from <http://www.rtai.org/>. Detailed installation instructions about both RT-Linux and RTAI can be found inside their respective distribution packages. Once RTAI or RT-Linux are installed, proceed with the instructions in section [Section 2.3 \[Unpacking\]](#), page 5.

### 2.2 Binary Installation

Under Debian Linux:

Add the following to `/etc/apt/sources.list`:

```
deb http://www.neuro.gatech.edu/mrci/debian testing main
```

Then install MRCI as follows:

```
apt-get update
apt-get install mrci-modules-2.4.27-adeos-3-arch
```

where *arch* is one of the following architectures:

<i>586mmx</i>	Pentium-compatible with MMX support
<i>p3</i>	Pentium III
<i>p3-smp</i>	Pentium III with SMP support
<i>p4</i>	Pentium 4
<i>p4-smp</i>	Pentium 4 with SMP and hyperthreading support
<i>k8</i>	AMD K8
<i>k8-smp</i>	AMD K8 with SMP support

After installing the kernel image is installed, reboot the system and boot the ‘`-adeos`’ kernel. Run MRCI as described in section [Section 2.5 \[Running\]](#), page 8.

## 2.3 Unpacking

Decompress and extract the files from the MRCI distribution archive:

```
tar xzf mrci-1.9.7.tar.gz
```

This will extract the ‘mrci’ directory with the following subdirectories:

‘app/’	MRCI application
‘common/’	A library of commonly used C++ classes
‘debian/’	Support files for the Debian package build system
‘doc/’	Documentation
‘elisp/’	Emacs mode for editing MRCI files
‘examples/’	Example system description files
‘math/’	Real-time Linux math library
‘module/’	MRCI kernel module
‘nical/’	NI calibration module
‘nidaq/’	NI driver module
‘remote/’	Support scripts for MRCI remote mode; source code for the MRCI remote client
‘scripts/’	MRCI scripts used to start and stop the system
‘translator/’	MRCI translator

## 2.4 Building and Installation

First, the configure script must be launched from within the location where the distribution archive was uncompressed:

```
./configure
```

Running `configure` takes awhile. While running, it prints some messages telling which features it is checking for. When `configure` is done, MRCI can be compiled by issuing the following command:

```
make
```

After the build process has completed successfully, MRCI can be installed with the following command, which requires root privileges:

```
su -
make install
```

By default, the MRCI program and accompanying scripts are installed in directory `/usr/local/bin`, the example and data files in `/usr/local/share/mrci`, and the kernel module is installed in `/lib/modules/osrelease/misc` where `osrelease` is the version of the currently running kernel (as reported by `uname -r`).

### 2.4.1 Configuring MRCI for use with RTAI

Options `--with-rtai`, `--with-rtai-scripts`, `--with-rtai-headers` specify the directory or directories where the RTAI headers and scripts are located.

The installer can work under one of two assumptions: a) RTAI 3.x is being used and the header files and scripts for RTAI reside in a single directory (`/usr/lib/realtime` by default)—in that case the argument to the `--with-rtai` option points to the top-level RTAI directory; or b) RTAI 2.x is being used and the headers and scripts have been installed in a custom directory or directories; in this case, the argument to option `--with-rtai-scripts` points to the directory that contains the RTAI scripts, and the argument to option `--with-rtai-headers` points to the directory that contains the RTAI header files.

For example, if the RTAI scripts have been installed in directory `/usr/local/bin` and the RTAI header files in directory `/usr/local/include`, the MRCI configuration script needs to be invoked with the following options in order for it to be able to use RTAI:

```
./configure --with-rtai-scripts=/usr/local/bin
            --with-rtai-headers=/usr/local/include
```

When using RTAI 3.x, the following will be sufficient for successful MRCI installation:

```
./configure --with-rtai=/usr/lib/realtime
```

or

```
./configure --with-rtai
```

which causes the installer script to search for RTAI in the default locations, `/usr/lib/realtime`, `/usr/local/lib/realtime`, `/usr/realtime`.

### 2.4.2 Configuring MRCI for use with RT-Linux and RT-Linux Pro

Option `--with-rtlinux` specifies the directory where the RT-Linux installation script has placed the RT-Linux header files, libraries, and executables (`/usr/rtlinux` by default).

Option ‘`--with-rtlpro-shm`’ directs the build process to compile MRCI with support for the shared memory interface used with RT-Linux Pro. This option is required when configuring MRCI for use with RT-Linux Pro. This code is still experimental and is not considered stable.

### 2.4.3 Configuring MRCI for use with COMEDI

Option ‘`--with-comedi`’ specifies that MRCI should be compiled with Comedi support. This is the recommended driver for use with MRCI, and configuration default as of version 1.9.0. If the option is given a value (e.g. ‘`--with-comedi=dir`’), this value is used as the directory where COMEDI is installed.

### 2.4.4 Configuring MRCI for use with the bundled NI driver

Option ‘`--with-nidaq`’ specifies that MRCI should be compiled with the NI driver that is bundled with the MRCI distribution. This driver will eventually be phased out; Comedi is the recommended driver for use with MRCI.

If MRCI is to be used with National Instruments ISA data acquisition boards, the following options need to be used in conjunction with ‘`--with-nidaq`’:

```
./configure --with-nidaq --with-type0=NI_AT_MIO_16E_1 --with-base0=0x100 \
--with-irq0=5
```

where ‘`--with-base`’ and ‘`--with-irq`’ specify a hexadecimal base address and interrupt request line, respectively. The integer in each option keyword specifies the board index (0—3).

The ‘`--typeN`’ option indicates the model of the board. A list of all supported models can be found in the file ‘`NI_BOARD_LIST`’, which is located in the MRCI distribution package, and is placed by the installation script in directory ‘`/usr/local/share/mrci`’ (provided the default installation directory hasn’t been changed).

MRCI supports several different gain ranges for data acquisition. By default, the acquisition range used is +/- 0.1V. To specify a different gain range, the option ‘`--with-gainN`’ can be used, where N is the board index (0—3) and the value of the argument is an integer between 0 and 2, where:

0	+/- 0.1 V
1	+/- 1V
2	+/- 10V

For example, if we want to specify a gain range of +/- 10V for board 0:

```
./configure ... --with-gain0=2
```

### 2.4.5 Other configuration options

Option ‘`--with-waveform-buffers=N`’ can be used to change the default number of waveform buffers supported by MRCI (four, if this option is not specified). For detailed explanation on waveform buffers, see [Section 3.29 \[load\]](#), page 19, [Section 3.31 \[play\]](#), page 20.

For further help on the parameters of the configure script, use the ‘`--help`’ option:

```
./configure --help
```

## 2.5 Running

MRCI is started by invoking the `run_mrci` script. Among other things, the script loads the kernel modules of RT-Linux and MRCI, so it needs to be run with root privileges:

```
su -  
run_mrci system.mrci
```

When the MRCI application starts, it attempts to locate a file called `.mrcirc` in the home directory of the user running MRCI (usually root). If this file is found, all MRCI commands in it are executed prior to presenting the user with the MRCI command prompt. The location of this file can be specified with the `-i` option, thusly:

```
run_mrci -i /some/location/init-mrci system.mrci
```

The `-w` argument specifies the number of waveform buffers to be created by MRCI. For detailed explanation on waveform buffers, see [Section 3.29 \[load\]](#), page 19, [Section 3.31 \[play\]](#), page 20.

While running, the MRCI application maintains a history of all commands entered by the user. This list is by default saved at program exit and loaded at program startup. The file used for saving the command history is named `.mrci_history` and is located in the user's home directory. Only the last 4096 commands are saved in this file.

The above behavior can be changed with the following command-line options: `--history-file`, which specifies the location of a file to be used for saving and loading the command history, and `--history-limit` which can be used to alter the command history limit. For example:

```
run_mrci --command-history my-history --command-limit 100 system.mrci
```

This will make MRCI save the last 100 commands entered by the user in a file named `my-history`.

## 2.6 Uninstalling

Issuing the command

```
make uninstall
```

from the MRCI source directory will remove the MRCI executables, scripts and kernel modules from their respective installation locations. Root privileges are required in order to execute this command.

## 2.7 Running in Remote Mode

Beginning with version 1.7.0, MRCI supports the so-called *remote mode*, where the MRCI application is controlled by a remote computer over a network connection. A remote MRCI client is available, and its source code can be found in subdirectory `remote` in the MRCI distribution archive. The client is also available as a binary distribution for MS Windows, which can be downloaded from the MRCI web site.

MRCI's remote mode relies on the internet services daemon, `inetd/xinetd`, so either one must be installed if remote operation of MRCI is desired. It is presumed that the `inetd` configuration file is located in `/etc/inetd.conf`, and the `xinetd` service configuration files are placed in directory `/etc/xinetd.d`.

Because MRCI runs with root user privileges, it is necessary to secure the remote connection; Stunnel (<http://www.stunnel.org>) and OpenSSL (<http://www.openssl.org>) are used to encrypt the connection, and so they are required to be installed on the machine running MRCI.

MRCI can be set up to run in remote mode by using the ‘`--with-remote`’ command-line option when invoking the ‘`configure`’ script:

```
./configure --with-remote ...
```

The installation script will setup `inetd/xinetd` to listen on port 7091 (which is defined as the port for MRCI service) and upon a connection initiated by a client, to launch MRCI via the Stunnel wrapper. Also note that the `xinetd` MRCI service is set up to listen on the `localhost` interface; you will need to go into the directory where the `xinetd` service configuration files are stored (usually ‘`/etc/xinetd.d`’), open the file called ‘`mrcis`’, and change `localhost` in the following line to the name of the network interface where you want the MRCI service to listen:

```
bind                = localhost # change to desired interface
```

Because the connection between the machine running MRCI and the client controlling it is encrypted, *encryption certificates* need to be generated. When a MRCI client connects to a MRCI server, the server presents a certificate, essentially an electronic piece of proof that machine is who it claims to be. This certificate is signed by a *certificate authority* (or CA for short). A client will accept this certificate only if the certificate presented matches the private key being used by the remote end.

During installation, a certificate authority and a server-side certificate will be generated. The installation program will ask you the following questions. While it is not strictly necessary to provide correct and complete answers, it is good practice to do so. The most important question is the password one: remember the password that you use when creating the certificate authority, because you will need it afterwards: when you are creating a server-side certificate, when Stunnel is being set up, and when you are creating a client certificate (discussed below).

*Question*    Example Answers

*PEM pass phrase*  
\*\*\*\*\*

*Country name*  
PL, UK, US, CA

*State or Province name*  
Illinois, Ontario

*Locality*    Chicago, Toronto

*Organization Name*  
Wossamotta U, Acme Anvils

*Organizational Unit Name*  
Dept. of Math

*Common Name*  
MRCI certificate authority, MRCI server certificate

The installation script of MRCI generates a certificate authority, a server-side certificate, and places them in the right directories, so that MRCI is automatically set up on the server side. In order to connect to the MRCI machine with the MRCI client one needs to generate a *client certificate*, and copy that certificate to the client machine. The script `mrci_mkcert` can be used for the purposes of generating a client certificate, as seen in the following example.

Please note that the first password question pertains to the password to be used with the client certificate, i.e. this is the password you will be using when connecting with the MRCI remote client; the second password question is asking about the password of the certificate authority, i.e. this is the password you entered when the installation script prompted you to create one.

```
$ mrci_mkcert

Using configuration from /usr/local/share/mrci/remote/openssl.cnf
Generating a 1024 bit RSA private key
.....++++++
.....++++++
writing new private key to
'/usr/local/share/mrci/remote/MrciCA/clients/newreq.pem'

Enter PEM pass phrase: *****
Verifying password - Enter PEM pass phrase: *****
-----
You are about to be asked to enter information that will be
incorporated into your certificate request.
What you are about to enter is what is called a Distinguished Name or
a DN. There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [US]:
State or Province Name (full name) [Georgia]:
Locality Name (eg, city) [Atlanta]:
Organization Name (eg, company): Georgia Institute of Technology
Organizational Unit Name (eg, section): Laboratory for Neuroengineering
Common Name (eg, YOUR name): MRCI client certificate
Email Address []:

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []: (can be left blank)
Using configuration from /usr/local/share/mrci/remote/openssl.cnf
Enter PEM pass phrase: *****
Check that the request matches the signature
Signature ok
The Subjects Distinguished Name is as follows
```

```
countryName          :PRINTABLE:'US'  
stateOrProvinceName :PRINTABLE:'Georgia'  
localityName        :PRINTABLE:'Atlanta'  
organizationName     :PRINTABLE:'Georgia Institute of Technology'  
organizationalUnitName:PRINTABLE:'Laboratory for Neuroengineering'  
commonName          :PRINTABLE:'MRCI client certificate'  
Certificate is to be certified until Nov 28 16:19:27 2004 GMT (365 days)  
Sign the certificate? [y/n]:y
```

```
1 out of 1 certificate requests certified, commit? [y/n]y  
Write out database with 1 new entries  
Data Base Updated
```

The newly generated client certificate and key files are located at:

```
/usr/local/share/mrci/remote/MrciCA/clients/03.cert  
/usr/local/share/mrci/remote/MrciCA/clients/03.key
```

Please transfer these files to the computer where the MRCI remote client will be running.

## 3 Commands

### 3.1 Summary of Commands

<code>add event</code>	adds an event to the event list
<code>caplen</code>	sets length of capture/analysis window
<code>cap</code>	specifies variables for capturing or analysis
<code>capture</code>	logs system dynamics for a certain period of time and writes to a file
<code>cd</code>	changes MRCI's working directory
<code>clear history</code>	clears the command history
<code>del event</code>	deletes an event from the event list
<code>del history</code>	deletes a command from the command history
<code>dsamp</code>	specifies downsample ratio for capturing
<code>dump event</code>	writes the event log to a file
<code>enable/disable event</code>	enable/disable the logging of a particular event, or enables/disables event logging for all events
<code>exit</code>	exits MRCI
<code>getvar</code>	examines the values of all variables
<code>help</code>	prints a command summary
<code>incvar</code>	adds an increment to one or more variables
<code>list history</code>	lists the contents of the command history
<code>load</code>	loads data samples from a file into a memory buffer
<code>nocap</code>	removes variables from the capturing list
<code>on/off</code>	turns the MRCI computational engine on/off
<code>play</code>	continuously updates the value of a variable with data samples from a memory buffer
<code>pwd</code>	prints the current working directory
<code>rampf</code>	performs simultaneous ramping and variable capturing
<code>rampoff</code>	turns off the currently active ramp, if there is one
<code>ramp</code>	ramps a variable from an initial to final value over a specified time length

<code>read history</code>	reads the list of commands into the command history
<code>readvar</code>	reads values of variables from a file
<code>refract</code>	sets the refractory period of an event
<code>reset event</code>	clears the event log
<code>save history</code>	saves the history of commands to a file
<code>scalevar</code>	adds a scaling to one or more variables
<code>script</code>	runs a script of commands
<code>setrate</code>	specifies computation/acquisition rate
<code>table recalc</code>	recalculates lookup tables
<code>setvar</code>	sets the values of one or more variables
<code>sleep</code>	pauses for a specified number of seconds
<code>stats</code>	like capture, but performs stats (mean/sd/max/min) instead
<code>status</code>	current status of MRCI
<code>system</code>	loads a system module
<code>timing</code>	performs a timing analysis
<code>ver</code>	prints the version of MRCI
<code>writevar</code>	writes the current values of all variables to a file

## 3.2 Command-Line Editing

The following table describes the most basic keystrokes that you need in order to do editing of the input line.

<code>(Ctrl)-a</code>	Move to the start of the line.
<code>(Ctrl)-e</code>	Move to the end of the line.
<code>(Alt)-f</code>	Move forward a word, where a word is composed of letters and digits.
<code>(Alt)-b</code>	Move backward a word.
<code>(Ctrl)-l</code>	Clear the screen, reprinting the current line at the top.
<code>(Ctrl)-k</code>	Kill the text from the current cursor position to the end of the line.
<code>(Ctrl)-w</code>	Kill from the cursor to the previous whitespace.
<code>(Ctrl)-y</code>	Paste the most recently killed text back into the buffer at the cursor.
<code>(Up)</code>	Go to the previous command in history.
<code>(Down)</code>	Go to the next command in history.

### 3.3 Shell Commands

MRCI gives the user ability to execute arbitrary shell commands from the MRCI command prompt. All shell commands must be preceded by an exclamation mark; if a shell command is to span multiple lines, a backslash as the last character on the line indicates that the user wants to continue the command on the next line.

For example, the following command will show a listing of all files in the current working directory (see [Section 3.9 \[cd\]](#), page 15 for more information on changing and querying the current working directory):

```
Command (help for commands): !ls
PRC.mrci          leak-test.var          morris-lecar_varsetup.h
burst-test.var    ml.var                  ramp.mrci
calibrate.var     morris-lecar.mrci      stepscript.scr
example.mrci      morris-lecar.o          works.var
function.mrci     morris-lecar_constants.h
hh.mrci           morris-lecar_module.c
```

This example shows the syntax if a command were to be split in two lines:

```
Command (help for commands): !find . -name ramp.mrci -exec \
> touch '{} ' ;'
```

Note that while the user is typing a multi-line command, MRCI's command prompt changes to '>'.

### 3.4 on

Syntax: `on`

This command turns the MRCI computation engine on. Commands involving the acquisition of data or continuous update of variables will not work if MRCI is off.

### 3.5 off

Syntax: `off`

This command turns the MRCI computation engine off. The analog outputs are “frozen” at the last output values when MRCI is turned off.

### 3.6 status

Syntax: `status`

The status command returns the following information:

- Whether or not MRCI is active.
- The computation cycle rate (see [Section 3.20 \[setrate\]](#), page 17).
- The downsample ratio (see [Section 3.18 \[capture\]](#), page 16 and [Section 3.23 \[dsamp\]](#), page 17).
- Output variables selected for capture or statistics calculations (see [Section 3.18 \[capture\]](#), page 16 and [Section 3.19 \[stats\]](#), page 17).
- Current length of the capture/analysis episode and info on how big your resulting output file may be.

- Waveform buffers: the names of the files where samples were read from, the sampling rate of their content, the number of samples contained in the buffer (see [Section 3.29 \[load\]](#), page 19 and [Section 3.31 \[play\]](#), page 20).
- Event slots: the maximum number of events that can be logged, the number of events logged so far, whether event logging is enabled or disabled, and a list of all event specifiers (see [Section 3.40 \[add event\]](#), page 21).

### 3.7 setvar

Syntax: `setvar variable value [variable value] ... setvar variable variable ...`

Sets variables to the desired values. If multiple variable/value pairs are specified, they are all updated before the next computational cycle. The second form of the command sets the value of the first variable to the value of the second variable.

### 3.8 getvar

Syntax: `getvar`

Displays a list of current variable values.

### 3.9 cd

Syntax: `cd "directory"`

Changes the current working directory of MRCI. The directory name needs to be enclosed in double quotation marks. All MRCI commands that read or write files are affected by this: if a file name is relative (i.e. it does not start with a /), then MRCI will attempt to read from or write to this file in the current working directory. In addition, all shell commands are executed in the current working directory.

Command `pwd` will print the full path to the current working directory.

### 3.10 pwd

Syntax: `pwd`

Prints the full path to the current working directory. See [Section 3.9 \[cd\]](#), page 15 for information on how to change the current working directory.

### 3.11 writevar

Syntax: `writevar ["filename"]`

Writes the variables to a file. Specifying a filename is optional; otherwise the filename will be `'data-YYYY-MMDD-HHMMSS.var'` (with the appropriate year/month/day/hours/minutes/seconds inserted). The file name needs to be enclosed in double quotation marks.

It is perfectly OK to edit this file with a text editor. The parser simply expects one variable/value pair per line. Order of variables is not important (you need not set them all), and the file may have comments, indicated by a `'#'` on the first line.

### 3.12 readvar

Syntax: `readvar "filename"`

Reads the specified variable file and sets the variables accordingly. The file name needs to be enclosed in double quotation marks.

### 3.13 incvar

Syntax: `incvar value varname [varname] [varname] ...`

Adds a value to all of the variables in the list.

### 3.14 scalevar

Syntax: `scalevar value varname [varname] [varname] ...`

Multiplies every variable in the list by value.

### 3.15 cap

Syntax: `cap varname [varname] [varname] ...`

Adds the specified variables to the capture list. These variables are logged by the capture command or analyzed by the stats command.

### 3.16 nocap

Syntax: `nocap varname [varname] [varname] ...`

Removes the specified variables from the capture list.

### 3.17 caplen

Syntax: `caplen N`

Specifies the length of the capture episode (variables to be logged or analyzed) in SECONDS.

### 3.18 capture

Syntax: `capture ["filename-prefix"]`

Starts a data capture of the variables in the capture list (see `cap/nocap`) for the number of seconds specified by `caplen`. Data is dumped to a file named `'data-YYYY-MMDD-HHMMSS.dump'`. If a filename prefix is specified, the filename will be `'prefix-YYYY-MMDD-HHMMSS.dump'`. At the end of the acquisition episode, the data is written to disk.

Caution: the write to disk is a low-priority task, and may take a long time (minutes) with high MRCI rates or complex models. Also note that data capturing is disabled during event logging, because the same memory buffer is used by the event logging and data capturing facilities. The user can get around this obstacle by disabling event logging (see [Section 3.44 \[disable event\], page 23](#)), and then clearing the event log buffer (see [Section 3.46 \[reset event\], page 23](#)) or writing the event log to file (see [Section 3.45 \[dump event\], page 23](#)). After data has been captured, event logging get be enabled again.

During a data capture and while writing to the disk, the user may not interact with MRCI. Control is returned to the user after all data is written to disk. If `⌘-C` is pressed during the capturing episode, then all capturing data collected up to this point is discarded, the capturing process is stopped, and control is returned to the user.

The data file is a binary file, with information on the headers about the speed of the computational cycle, the value of `dsamp`, and the names of the variables logged. To extract data from this file, use the `mrci_extract` utility, also provided in the MRCI distribution. It has the following forms of usage:

`mrci_extract filename -i` —provides information contained in the header of the data file.

`mrci_extract filename [-t] [varname] [varname] ...`—outputs the specified variables from the data file, one line per timepoint. This output is to standard output. The `-t` option outputs the time in milliseconds. Time is assumed to start at zero.

### 3.19 stats

Syntax: `stats`

Similar to the capture command. However, the captured data is not written to disk. Instead, statistics (mean/stddev/max/min) are calculated from the time series of each acquired variable.

### 3.20 setrate

Syntax: `setrate F`

Sets the rate of MRCI in kHz.

### 3.21 table recalc

Syntax: `table recalc`

Recalculates all lookup tables in the currently loaded model. This is useful for updating lookup tables after a parameter is modified.

### 3.22 reset system

Syntax: `reset system`

Sets the model states to their initial values.

### 3.23 dsamp

Syntax: `dsamp N`

Sets the downsample ratio, where `N` is an integer. If greater than one, then only every `N` cycles is logged during a capture command. This setting does not affect the stats command.

The `dsamp` command is useful to reduce the size of generated data files when a lower degree of temporal resolution is acceptable for your logged data.

### 3.24 timing

Syntax: `timing [count ["filename"]]`

Performs a timing analysis. The optional count parameter specifies the number of CYCLES to perform the analysis over – the default is as many as possible (the stock code will perform 8 million). The optional filename (enclosed in double quotation marks and only valid if you specify a count) will dump all those time intervals to a file. You probably never want to do this.

When this command is executed, each computational cycle is time-stamped and the interval since the last computation (accurate to the precision of your microprocessor's clock) is written to a memory buffer. Statistics are acquired on two characteristics of the computational cycle: 1) the length of each cycle (mean/stddev/max/min) and 2) the time difference between the beginning of each successive cycle (mean/stddev/max/min).

You should ALWAYS run this command prior to actually using a model. Be sure that 1) the calculated mean interval length (reciprocal of the average cycle duration) is nearly identical to the specified rate and 2) the CV (coefficient of variation; stdev/mean) is less than 0.05.

This command should be run after `setrate` has set the system to the desired rate.

Hint: the default is to capture as many cycles as possible—this may be a lot! Our system as configured can log 8,000,000 intervals—if the rate is 10 kHz, that is 800 seconds (13 minutes) of timing, plus several minutes for subsequent analysis and disk writing, if necessary.

### 3.25 ramp

Syntax: `ramp mode varname start end duration(ms)`

Ramps the value of the variable `varname` from `start` to `end` for `duration` milliseconds. If `mode` is non-zero, the ramp will be for twice the duration (minus one cycle) and ramp bidirectionally, returning to the start value.

### 3.26 rampf

Syntax: `rampf mode varname start end duration(ms)`

Same as the `ramp` command, only data is logged to a file (as if `ramp` and `capture` were run simultaneously). The filename is similar to that in the `capture` command, only the filename prefix is `'ramp'`.

During a data capture and while writing to the disk, the user may not interact with MRCI. Control is returned to the user after all data is written to disk. If `Ctrl-C` is pressed during the capturing episode, then all capturing data collected up to this point is discarded, the capturing process is stopped, and control is returned to the user.

### 3.27 rampoff

Syntax: `rampoff`

Turns off the currently active ramp, if there is one.

### 3.28 script

Syntax: `script "filename"`

Runs a script. This is a file with a list of MRCI commands; the file can also have comments on lines beginning with '#'. The file name needs to be enclosed in double quotation marks.

This command is useful for designing protocols.

### 3.29 load

Syntax: `load "filename" buffer`

Loads a sampled waveform from a file into the buffer specified by an index number. This is the file format expected:

```
# 4 kHz synaptic output
4000
2.8138889e-04
2.8263889e-04
2.7597222e-04
2.5458333e-04
```

- Lines beginning with '#' (or '#' preceded by white space) are ignored.
- The integer in the first non-commented line is the sampling rate in Hz.
- Each sample should be the text representation of a floating-point number, which may be preceded by white space.
- The samples, as well as the sampling rate, should be each on its own line.

The buffer that is used to load the samples is indicated by a positive integer or zero; this index number should be less than the maximum number of waveform buffers specified. Suppose, for example, that MRCI is run with options which designate the maximum number of waveform buffers to be three. Then, we can load waveform data in buffers 0, 1, or 2, but 3 is an invalid buffer index. If we limit the maximum number of buffers to 2, then we can only load data in buffers 0 and 1. Each buffer has a maximum size of 8 MB (8,388,608 bytes), which is enough to hold 1,048,576 samples.

The file name needs to be enclosed in double quotation marks.

### 3.30 system

Syntax: `system "filename"`

This command unloads the currently loaded system module, if any, and compiles and loads the one specified by the user. If compiling the new system module fails, then the old one is not unloaded. The filename of the new system must be enclosed in double quotation marks.

The MRCI computational engine is turned off prior to compilation, and it is not turned back on automatically; this is done in order to enable the user to select a computation rate suitable for the new system.

Caution should be exercised when switching from one system to another; as the computation rate originally selected may turn out to be too high for the new system, which would cause MRCI to lock up when the computational engine is turned on.

### 3.31 play

Syntax: `play varname buffer`

Continuously updates the value of a variable with data samples from a waveform buffer. The given buffer needs to contain samples loaded with the `load` command. For more information, See [Section 3.29 \[load\]](#), page 19.

The buffer that is used to obtain the samples from is indicated by a positive integer or zero; this index number should be less than the maximum number of waveform buffers specified. Suppose, for example, that MRCI is run with options which designate the maximum number of waveform buffers to be three. Then, we can read waveform data from buffers 0, 1, or 2, but 3 is an invalid buffer index. If we limit the maximum number of buffers to 2, then we can only read data from buffers 0 and 1. Each buffer has a maximum size of 8 MB (8,388,608 bytes), which is enough to hold 1,048,576 samples.

If MRCI is running at a rate different than the sampling rate of the buffer data, then the variable is updated with interpolated data points, according to an algorithm which reads the two closest samples from the buffer, and produces their difference, multiplied by the factor of the two sampling rates.

The variable specified is updated until the set of samples in the buffer is exhausted.

### 3.32 sleep

Syntax: `sleep N`

Sleep (pause user interaction with MRCI) for N seconds. Does not affect the running of the computational engine.

This command is only really useful to provide required pauses in scripts. The user should not consider the specified number of seconds to be exact, since timing is done in user-space (unlike the MRCI engine). It is typically accurate within tens of milliseconds.

### 3.33 help

Syntax: `help`

Prints a summary help screen.

### 3.34 exit

Syntax: `exit`

Exits the MRCI application.

### 3.35 clear history

Syntax: `clear history`

Clears the command history.

### 3.36 save history

Syntax: `save history "filename"`

Saves the command history commands into a file. The named file will be overwritten with the current history of commands. The file name needs to be enclosed in double quotation marks.

### 3.37 read history

Syntax: `read history "filename"`

Reads the list of commands from the named file and replaces the current command history with that list. The file name needs to be enclosed in double quotation marks.

### 3.38 list history

Syntax: `list history`

Lists the contents of the command history. This includes commands loaded upon program startup (see [Section 2.5 \[Running\]](#), page 8) and commands since entered by the user. Each command is prefixed by its index number; this index can be used with the `del history` command (see [Section 3.39 \[del history\]](#), page 21).

### 3.39 del history

Syntax: `del history N`

Deletes the command at index N from the command history. The index of a command can be determined from the output of the `list history` command (see [Section 3.38 \[list history\]](#), page 21).

### 3.40 add event

Syntax: `add event var relation parameter`

Adds an event specifier to the list of events. An event is defined as the relationship between the value of a quantity and a certain threshold value—for example, the event when the value of quantity *Vin* becomes greater than 0.1, or the value of quantity *Vout* becomes less than or equal to quantity *Vthresh*.

MRCI can track up to 20 events; the conditions which describe each event are presented to the user as numbered *event slots*. Once an event is added, the user will need to use the corresponding event slot number in order to refer to that event. See [Section 3.6 \[status\]](#), page 14 for more information.

*var* is the name of a system quantity; *relation* is a relation operator (one of =, <, >, >=, <=). *Parameter* is the numerical constant or the name of the quantity that we are comparing against. The example events mentioned above could be specified as follows:

```
add event Vin > 0.1
add event Vout <= Vthresh
```

If all 20 event slots are full, an error message is printed, and the event is not added. New events are enabled by default (see [Section 3.43 \[enable event\]](#), page 22 and [Section 3.44 \[disable event\]](#), page 23 for more information on how to enable/disable events).

Note that while events are logged, MRCI cannot perform variable capture ([Section 3.18 \[capture\]](#), page 16) or timing tests ([Section 3.24 \[timing\]](#), page 18). This is because the same memory buffer is used for both event logging and data capture. If MRCI has logged events,

and the user wishes to capture data, they'll first need to disable event logging (Section 3.44 [disable event], page 23) and then either reset all logged events (Section 3.46 [reset event], page 23) or write them to file (Section 3.45 [dump event], page 23). After the data capturing or timing test is completed, event logging can be enabled again.

### 3.41 **refract**

Syntax: **refract event N**

Sets the *refractory period* of the specified event in milliseconds. A refractory period is an interval after an event has occurred, when another event of the same type will not be recorded. *event* is the event slot number, and *N* is a positive integer that represents the refractory period in milliseconds.

The exact length of the refractory period will be determined by MRCI, and may slightly differ from the one specified by the user. The actual length will be an integer multiple of the computational cycle length that the system is running at, i.e.  $\text{floor}(\text{Ref}/\text{Per}) * \text{Per}$ , where *Ref* is the requested refractory period and *Per* is the computational period. For example, if the user specifies a refractory period of 3 ms for a particular event, and the current computational rate is 15 kHz (0.067 ms cycle length), the actual refractory period will be set to 2.948 ms, which corresponds to the nearest integer multiple of the computational cycle length. If the user changes the computational rate, all refractory periods will be recalculated accordingly.

### 3.42 **del event**

Syntax: **del event N**

Deletes the event specifier in event slot *N*. A list of all events specifiers and their slot numbers can be obtained by issuing the **status** command (Section 3.6 [status], page 14).

If the event has been logged by MRCI, it cannot be deleted; the logged data must first be written to a file, or the event log needs to be reset with the **reset event** command. Whether occurrences of the event have been logged is indicated by the **status** command.

**del event all** deletes all defined events, provided that none of them has any logged data.

### 3.43 **enable event**

Syntax: **enable event N**

Enables the event at event slot *N*. Whether the event is enabled or disabled is indicated by the **status** command (Section 3.6 [status], page 14). When an event is enabled, it is actively logged by MRCI.

**enable event all** enables logging of events globally; note that even if individual events are enabled, event logging can still be disabled globally, and this is the default state of the system upon start-up. This means that after the user has added the events that they desire to record, they still need to enable global event logging by issuing the command **enable event all**. The global state of event logging is indicated by the **status** command.

### 3.44 disable event

Syntax: `disable event N`

Disables the event at event slot N. Whether the event is enabled or disabled is indicated by the `status` command (Section 3.6 [status], page 14). When an event is disabled, it is not logged by MRCI, although previously logged data from it can be written to file or reset.

`disable event all` disables logging of events globally; note that even if individual events are enabled, event logging can still be disabled globally, and this is the default state of the system upon start-up. The global state of event logging is indicated by the `status` command.

### 3.45 dump event

Syntax: `dump event ["filename"]`

Writes the event log to a file and clears the event log buffer. Specifying a filename is optional; otherwise the filename will be `'data-YYYY-MMDD-HHMMSS.event-log'` (with the appropriate year/month/day/hours/minutes/seconds inserted). The file name needs to be enclosed in double quotation marks.

The data file is a binary file, with information on the headers about the number of logged events, the number of event slots, and event descriptions. To extract event data from this file, use the `mrci_extract` utility, provided in the MRCI distribution. It has the following forms of usage:

`mrci_extract filename -i`—provides information contained in the header of the data file.

`mrci_extract filename`—prints the event log, where the first column is the relative time in microseconds, and the second column is the event slot number.

### 3.46 reset event

Syntax: `reset event`

Clears the event log. All events recorded by MRCI are deleted. The event specifiers are not deleted, and it is therefore possible that new events are recorded immediately after resetting, provided that event logging has not been disabled beforehand.

### 3.47 ver

Syntax: `ver`

Prints the version of the currently running MRCI application.

## 4 System Checkout and Tutorial

This section contains a system checkout and tutorial. It is designed with a wide-range of users and setups in mind. The initial checkout and first tutorial only requires an oscilloscope. Subsequent tutorial steps require a function generator and/or an electrophysiology amplifier with an RC cell model.

This tutorial utilizes the model (.mrci) and variable (.var) files located in the ‘examples’ directory of MRCI (‘/usr/local/share/mrci/examples’ by default).

### 4.1 Compiling the Model and Running MRCI

MRCI requires that the user be logged in as *root*. In order to load the example model, enter the following commands:

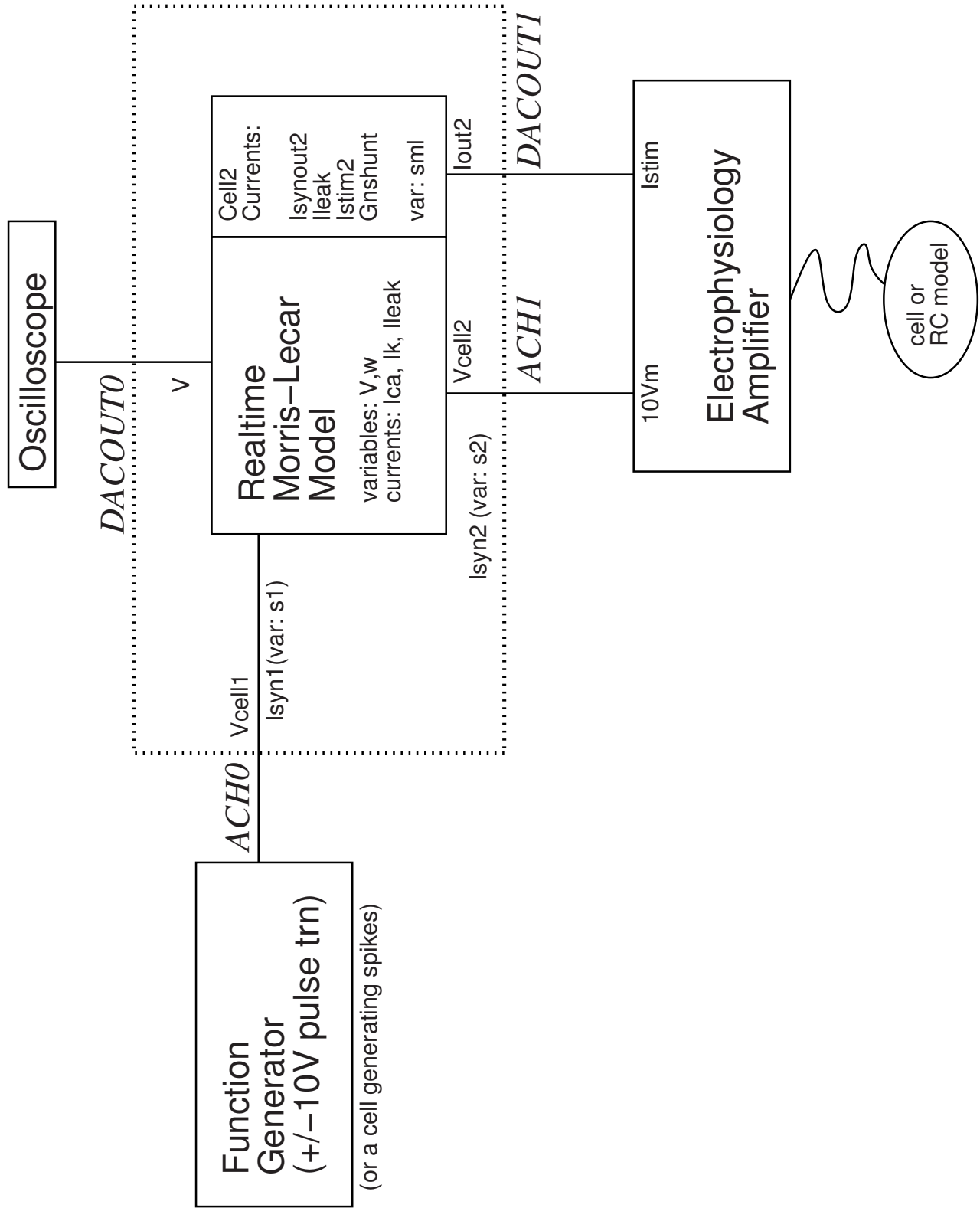
```
cd /usr/local/share/mrci/examples
run_mrci morris-lecar.mrci
```

### 4.2 Overview of Model

This model resides in file ‘*morris-lecar.mrci*’, found in the ‘examples’ MRCI directory, and by default installed in ‘/usr/local/share/mrci/examples’.

The experimental setup for our model system is illustrated in [Figure 1], page 25. It consists of a Morris-Lecar excitable cell model running in real-time, as well as some additional currents that can be added to the cell or RC model. The dotted box represents the MRCI system and the National Instruments board. The italicized labels are the corresponding inputs and outputs to/from the board.

*ACH0* is the input from an external cell—for these tests we will use a function generator (a real neuron generating spikes would also work!) outputting a spike train. A signal that is +/- 1V or larger should be sufficient. *ACH1* is the input from another external cell. For the tutorials that use this measurement, we use an electrophysiology amplifier connected to an RC model cell. *DACOUT0* outputs a scaled version of the membrane potential of the Morris-Lecar model running in real-time. *DACOUT1* outputs a voltage proportional to the current injected into the cell measured from *ACH1* (or RC model).



### 4.3 A Real-Time Morris-Lecar Model

For this tutorial you only need the oscilloscope monitoring *DACOUT0*. Start the MRCI system. At the MRCI prompt, type the following:

```
setrate 10
readvar "ml.var"
on
```

This sequence of commands sets the computation and update rate to 10 kHz, loads a variable file that sets some model parameters and initial conditions, and then activates the MRCI computations. Viewed on an oscilloscope, you should see a periodically firing action potential train. The outputs has been scaled so that 10V corresponds to 100mV in the cell model.

Type *getvar*. You will see a list of all of the model variables. Note that MRCI makes no distinction between variables and parameters (i.e., everything is a variable).

Type *status*. This shows you the general status of the system: whether MRCI is activated, what the computation rate is, and what variables (if any) are being logged.

Type *cap v*. This means that any subsequent executions of the *capture* or *stats* commands will include the variable *v* in any capture or data analysis.

Type *caplen 5*. This command specifies the number of *seconds* that data should be captured with the *stats* or *capture* commands.

Type *stats*. The system will pause for 5 seconds, logging *v* for every cycle, then computing some statistics on the collected time series. The mean, standard deviation, and extrema for *v* will be calculated and displayed.

Type *timing 50000*. The system will pause for the specified duration (50,000 cycles at 10 kHz, or 5 seconds) and collect timing information. Each cycle, a time-stamp is recorded, and the duration of each cycle period is computed and statistics are analyzed. Generally, the system is performing at a safe rate if the CV (standard deviation/mean) of the timing statistics is less than 0.1. The calculated cycle period and corresponding computation rate will be displayed—this should be similar to what you specified in the *setrate* command.

Set your timebase on your oscilloscope to show 5-10 seconds of activity. Type *setvar iapp 1.2*. This will depolarize the real-time cell, and it should spike at a faster rate. Reset it back to 0.7.

Type *ramp 0 iapp 0.7 1.2 5000*. This command will, over a duration of 5 seconds (5000 milliseconds), ramp the stimulus current from 0.7 to 1.2. You should see the cell increase in firing rate over that 5 second duration. The first parameter of the *ramp* command is whether the ramp is uni- or bi-directional. Repeat the command using *ramp 1 iapp 0.7 1.2 5000*. You will see a similar response, except that *istim* will increase from 0.7 to 1.2 over 5 seconds then decrease back to 0.7 over the next 5 seconds.

Type *capture*. The variable *v* (specified by *capvar* earlier) will be logged to disk, and the name of the file will be displayed.

Exit MRCI. Type *mrci\_extract FILE -i*, where *FILE* is the name of the file the data was captured to. You will see displayed information about your data file: the computation rate, the rate that data was saved at, and what variables were logged. Now type at your shell command prompt: *mrci\_extract FILE -t v*, This will output a time-series with time and voltage output once per row. Typing *mrci\_extract FILE -t v > foo.txt* would output

that data to a text file named ‘foo.txt’—this could easily be imported by any graphical and/or analysis program.

## 4.4 Synaptic Input to the Morris-Lecar Model

This tutorial extends the previous tutorial and verifies that external input works. First, connect an external voltage source (such as a power supply, or even a battery across a variable resistor) to input ACH0. Type the following:

```
setrate 10
readvar "ml.var"
on
getvar
```

Look at the screen—*VIN1* should correspond to the voltage that you are inputting from the power supply. Note that the gain setting specified when installing MRCI will affect the maximum voltage that can be measured. For example, if the gain is set to 10 (+/- 1V), anything larger than 1V will be measured as one volt. Vary the voltage on the power supply, and repeatedly type *getvar* to see that the new value is measured. See the comments in the header of the ‘morris-lecar.mrci’ file for more info on gain settings.

Connect an external function generation to input *ACH0*. Set it up to output a pulse 3 ms long every 1 second (or something similar). The pulse should have a baseline of -1V and be 2V in amplitude. The exit values do not matter too much, as long as the pulse crosses 0V. The synaptic model is described in the appendix, but basically is activated by positive crossings of 0 mV.

Monitor both the input signal and the output of the real-time Morris-Lecar model on an oscilloscope. At present, you should see the Morris-Lecar model not showing any response to the external input.

Type *setvar iapp 0.3*. The Morris-Lecar model should now stop firing action potentials and be at rest at about -46 mV (-4.6V if viewed on an oscilloscope with a  $10 \times V_m$  output).

Type *setvar gsynin1 0.3*. Your Morris-Lecar model should now show an EPSP (excitatory post-synaptic potential) every 1 second, triggered by the function generator.

Multiply this input by 2 by typing *scalevar 2 gsynin1*. Each synaptic input should now trigger an action potential.

## 4.5 Connecting to an Electrophysiology Amplifier

Now we’re going to connect the system to an electrophysiology amplifier. Input *ACH1* should be connected to the membrane potential output of an electrophysiology amplifier. The input gain of the installed MRCI system should be appropriately matched. If connected to an electrophysiology amplifier with a  $10 \times V_m$  output, as in [Figure 1], page 25, a 100 mV cellular signal corresponds to an input of 1V. MRCI should be run with ‘--gain=1’ (+/- 1V input range). See also the notes in the header of the ‘morris-lecar.mrci’ file for more input on gain settings. The variable *vin2gain* converts this input voltage (in volts) to the units used within our model—millivolts of  $V_m$  of the measured cell. For this example *vin2gain* = 100 to convert a 1V input signal to its 100 mV representation.

Connect your electrophysiology amplifier to an RC model cell.

Start the MRCI system (if not still running from the previous tutorial), using the same sequence of commands as in the tutorial above.

Type `getvar`. Look at the value of `vin2mv`. This should be *identical* to the voltage (in mV) across your RC cell model. Vary the DC bias to your RC cell model using the appropriate controls on your electrophysiology amplifier, and repeatedly type `getvar` to make sure similar numbers are being displayed on the MRCI system.

Hint: typing `(Ctrl)-P` and `(Ctrl)-N` will allow you to scroll up and down through your past commands without typing them!

## 4.6 Altering the Properties of an RC Cell Model

This tutorial follows up on the previous tutorial. Connect an RC “model-cell” (typically an RC circuit that comes with your electrophysiology amplifier) to your amplifier.

First, make sure the output scalings are correct. Restart MRCI, and type the following sequence of commands:

```
setrate 10
readvar "ml.var"
on
getvar
```

This loads a variable/parameter set with all conductances and currents set to zero. We assume that the input voltage from the electrophysiology amplifier is being read correctly (see previous tutorial) as the variable `vin2mv`. If not, adjust the input gains settings while running MRCI and/or the variable `vin2mv`.

## 4.7 Constant Current

In this tutorial we will make sure the output works correctly. The variable in the model called `vout2gain` is the gain to translate the model output current `iout2` into a voltage that is output to the electrophysiology amplifier. Our electrophysiology amplifier has a gain of 1 nA/V, so this value is set to one—adjust yours accordingly.

Monitor the voltage on your model cell. Type `setvar iapp2 1`. The voltage should increase as if 1 nA of current was injected into it. For example, with a 50 M $\Omega$  model cell, 1 nA would depolarize the cell 50 mV.

Does it work? If the expected value is off by a factor of 10, check your input/output gain settings (both the gain variables within MRCI, as well as the gain settings for the input ADCs used when starting MRCI). If off by some other factor, check whether your model cell also has modeled electrode resistances and/or a bridge circuit is (or is not) being used.

## 4.8 Modified RC Parameters

Now we’re going to modify the parameters of the RC. There is a model variable called `gnshunt`—this is a NEGATIVE conductance. The purpose of this is to effectively cancel out the resistance of the RC cell, typically in parallel with specifying our own leakage current in its place.

Reload the ‘`ml.var`’ variable set just used above.

Let  $X$  be the value of the conductance of the resistor in your RC cell in  $\mu S$  (assuming that you output current in nA). For a 50 M $\Omega$  resistor, this corresponds to 0.02  $\mu S$ . Zero out the DC bias on your model cell.

Type `on` if the MRCI system is not currently actively running.

Type the following: `setvar gshunt X gl2 0.1 e12 -70`. where  $X$  is the value determined above. The voltage on your electrophysiology amplifier should now read -70. In this example, we have negated the resistance of the RC cell using `gshunt` and added a passive leak current with a conductance of 0.5  $\mu S$  and a reversal potential of -70 mV.

Now the virtual cell has an effective resisting resistance of 10 M $\Omega$ . Using your electrophysiology amplifier (or typing `setvar iapp2 1` inject 1 nA of current into the model cell—it should depolarize by 10 mV.

Note that when using this shunt model, you may see oscillations on your oscilloscope. This is due to feedback instability, and can be corrected by increasing your capacitance compensation on your amplifier and/or altering your computation rate.

## 4.9 Synaptic Output from the Morris-Lecar Model

In this example we'll combine all of the above models, and use the Morris-Lecar model to generate action potentials and provide synaptic input to the model cell.

Perform the following (where  $X$  is again the conductance value from above):

```
readvar "ml.var"
setrate 10
setvar gshunt X gl2 0.1 e12 -70
on
```

Monitor both `DACOUT0` and  $V_m$  from your electrophysiology amplifier on an oscilloscope. So far you should see no response from the RC cell. There is an excitatory synapse between the real-time model and the RC cell, but its conductance is zero.

Type `setvar gsynout2 0.5`. You should see EPSPs about 10mV in amplitude triggered by each action potential. Type `setvar gsynout2 1.0`. They should now double in amplitude.

Type `ramp 1 gsynout2 0.0 1.0 10000`. This ramps the parameter `gsynout2` from 0 to 1 over 10 seconds (10000 ms), and back down for another 10 sec. The first parameter is a 1 for a bidirectional ramp, and 0 for a unidirectional ramp.

Keep this simulation running for the next tutorial.

## 4.10 A Scripting Example

Set `gsynout2` back to zero (`setvar gsynout2 0`).

Type `script "stepscrip.t.scr"` and observe. Every five seconds, you will see the amplitude of the EPSPs increase. This is an example of a *script*—a text file consisting of MRCI commands that are executed in sequential order. Below is a listing of the script:

```
setvar gsynout2 0.4
sleep 5
incvar 0.4 gsynout2
sleep 5
```

```
incvar 0.4 gsynout2
sleep 5
incvar 0.4 gsynout2
```

When running the script, you will see a special prompt indicating that a script is being executed. *You cannot interrupt a script, nor execute MRCI commands interactively while the script is running.* The `sleep` command is specifically for use within scripts, and it means to pause for *approximately* the specified number of *seconds*.

#### 4.11 A Stimulus/Pulse Generator (Inline C Code Example)

## 5 The System Description Language

MRCI simulates a dynamical system described in a special, equation oriented language. The description consists of declarations of states, functions, parameters, etc., and equations describing the behavior of the system. It is translated into the programming language C, which is subsequently compiled into machine-executable code.

This chapter describes the *System Description Language* of MRCI.

### 5.1 General

A *system description* describes the dynamical system to be studied. It consists of *declarations* and *equations*, which have a certain mathematical meaning, and are not to be confused with assignment statements in programming languages. (Strictly speaking there is the difference that the left hand side and the right hand sides are not exchangeable: those who write their differential equation  $f(x) = dx/dt$  may have problems. . .) These equations do not have to be entered in any particular order; they are automatically sorted by the translator so that functions are computed before the state equations that use them.

A formal grammar of the system description language is given in [Chapter 10 \[Specification of the System Description Language\]](#), page 44.

### 5.2 Concepts

The MRCI system description language is always case sensitive. The name ‘A’ is different from ‘a’. The language has a set of reserved words, which are always written in capital letters.

A *user-visible quantity* is something the user can access, change (provided that it makes sense), and which may have a documentation associated with it. MRCI allows all user-visible quantities to be manipulated by the user, and in that sense it does not distinguish between quantities with different mathematical meanings (such as constants, states, functions, etc.)

The names given to user-visible quantities must obey the same rules that apply to identifiers in the C programming language: an identifier consists of a sequence of (ASCII) alpha-numerical characters, the first of which is a character (the underscore ‘\_’ is counted as a character).

Once an identifier is used to declare a state, parameter etc. *it may not be used for any other purpose*, except for local variables and formal parameters in raw C-code.

Comments may be written anywhere. The syntax is as in the C and C++ programming languages: either enclosed between ‘/\*’ and ‘\*/’, or from ‘//’ and to end-of-line.

The system file is passed through the m4 macro processor before the translator sees it. You can thus define macros, use include files, etc. However, note that there are some dangers involved in using macros: It can be error prone, since the macro processor does not know anything about the language (use parentheses around arguments and the resulting expression!). The syntax will often look bizarre, and it is simple to write very unclear code. Error messages will refer to the code *after* macro expansion, which is confusing.

### 5.3 Structure of a MRCI System

A MRCI system file consists of a declaration section followed by a time block. The declaration section consists of a number of declarations. Every declaration is ended by a semicolon. The first declaration has to be a declaration of the system, such as

```
SYSTEM system_name;
```

where *system\_name* may contain any letters, numbers, or punctuation.

After the system declaration, there follow a number of declarations of states, parameters, and functions. There has to be exactly one time declaration.

### 5.4 Declarations

*States* are components of the state vector in the sense of a dynamical system. They are user visible. They are declared with the command `STATE`. The syntax is

```
STATE name = initial condition;
```

or

```
STATE name = initial condition "description";
```

where *name* is the name of the state as the user sees it, *initial condition* is the default initial condition (a real constant), and the optional *description* is the system authors message to the user. (Don't take the word "description" too literally: Feel free to put anything sensible here. For example, it can be pretty handy to have the exact look of the equations available to the user.)

A typical declaration can be

```
STATE omega = 3.1 "Angular velocity in rad/s";
```

where *omega* is the name of the state (the entity to be declared), '3.1' is the default initial condition (which the user can change). The documentation string is there for convenience. It is not read by any program, just by humans.

The above declaration will create a state variable which is integrated using the equation set in the time block, described later in this chapter. The default method for integration is Euler's method. MRCI also supports a method we call *multiply-add-update*, where the state variable being integrated is multiplied and added with the values returned by two functions dependent on *dt*. The method of integration can be specified with the `METHOD` attribute of the state definition, as follows:

```
STATE name = initial condition METHOD method_name;
```

or

```
STATE name = initial condition METHOD method_name "description";
```

where *method\_name* can be either "euler" or "mau", indicating Euler or multiply-add-update, respectively. Thus our example can be changed to

```
STATE omega = 3.1 METHOD "mau" "Angular velocity in rad/s";
```

*Parameters* are constants during an integration. Syntax is

```
PARAMETER = default_value "description"
```

where "*description*" is optional. (A description is always optional, we shall therefore not mention it for the rest of the chapter.)

*Functions* are quantities depending (statically) on other quantities in the systems. (The name is possibly badly chosen, they correspond neither to the mathematical nor the computer science meaning of the word.) The declaration syntax is

```
FUNCTION name "description";
```

*Function lookup tables* are a method of approximating a function by a table with a finite number of points  $(X, Y)$ . At initialization time, MRCI will compute all values for the function, using the user-supplied range of input values. At run-time, the input value given as an argument to the lookup function is the value of another quantity in the system (e.g. another function). This quantity is called a *lookup function dependency*.

For example, we could have a lookup function called 'F1', and another function, called 'F2', which is the dependency; at run-time, 'F2' would be computed first, then the resulting value would be looked up in the 'F1' table, and the result would be the value of 'F1'. The declaration syntax of 'F1' would be:

```
TABLE FUNCTION F1(v) = (1 + tanh(v)),
                    LOW = -10.1, HIGH = 10.1, STEP = 0.1,
                    DEPENDENCY = F2;
```

The various syntactic components of this statement have the following meanings:

- 'TABLE FUNCTION F(v)'—declaration of a function called 'F1', which has one argument, 'v'. Note that the function argument is only to be used inside the function expression; it is *NOT* (or doesn't have to be) the name of the dependency.
- '(1 + tanh(v))'—the actual function expression. See [Section 5.5 \[Time Blocks\], page 34](#), for details on arithmetic expressions in MRCI. Note the use of our function argument.
- 'LOW=-10.1,HIGH=10.1,STEP=0.1'—the lower boundary of input values, the upper boundary of input values, and the stepping for values between the two boundaries. MRCI will compute function values starting at the lower boundary and reaching to the upper boundary using the given step.
- 'DEPENDENCY=F2'—the name of the dependency. This can be a function, state, parameter, etc. At run-time, the value of this quantity will be computed first, then it will be given as an input to the lookup function.

*Integer states* are simply integers. They are declared as

```
INTEGER STATE name ;
```

*External states* are states whose value is either obtained through the data acquisition board (*external input*), or whose value is being output to the data acquisition board (*external output*). They are declared as

```
EXTERNAL INPUT Vin;
EXTERNAL OUTPUT Vout;
```

The system then can use the input state in equations and expressions; the output state may not be used in expressions, and it must be assigned a value.

The translator checks the equations to sort them, and to check that all symbols make sense. If using raw C code, it may thus be necessary to tell the translator that a certain name should be considered as resolved. The declaration is

```
C-IDENTIFIER name ;
```

This declares *name* as a quantity the translator should consider as well known. It does not correspond to a user visible variable.

C preprocessor commands (defined as lines starting with a ‘#’ character, extending to the end of the line) are passed unprocessed to the compiler.

For all systems, there is only one “time”, to be declared with the declaration `TIME`. This is mandatory in all systems. Syntax is

```
TIME name ;
```

No documentation is allowed.

After the standard declaration, but before the time blocks, there may be a section of raw C code. Example:

```
C-DECLARATION
{ /* Arbitrary C code */ }
```

It should be enclosed within a pair of matching braces.

## 5.5 Time Blocks

A time block describes the equations which are in effect during the named time. Dynamic equations are all in the `AT TIME t`-block (assuming that the system time is called `t`). The equations in a time block are, if possible, sorted in order so they can be sequentially executed. If the equations contain a circular dependence, the sorting will fail. The MRCI translator can not solve algebraic loops (It can be claimed that in this case, the user has not written complete and consistent equations for the system). Other sanity tests (like that every derivative is assigned to exactly once) are also performed.

Function expressions are specified in the following manner:

```
f1 = sin((1 + a) / 5)
```

The above statement will specify that at each iteration, the quantity *f1* will have the value computed with the given expression. *f1* then can be used in the expressions of other functions, differential equations, etc:

```
f2 = sin(f1 * 12)
```

On the right hand side, almost any scalar C expression is allowed: The exponential operator, denoted either ‘\*\*’ or ‘~’, has been added to the C syntax. The equation, which may run over several lines, is terminated with a semicolon. Further, the sequencing operator (the comma) is not allowed, since an expression sequence can hardly be an “equation”. See [Section 5.6 \[Expressions and Operators in the System Description Language\], page 36](#), for a complete description of the possible arithmetic expressions.

Standard mathematical functions are available with their usual names (log, cos, atan, etc. . .). See [Section 5.7 \[Mathematical Functions in the System Description Language\], page 38](#), for a list of all MRCI-supported mathematical functions.

Differential equations are specified in the following form:

```
d(state) = right-hand side;
```

Here  $d(\ )$  denotes the differential operator, and  $d(\mathbf{x})$  should here be interpreted as  $dx/dt$ . On the right-hand side, the same rules apply as for function expressions.

In cases where the desired method of integration requires more than one equation (such as the multiply-add-update method), the equations are written as a comma-separated list enclosed in brackets. Thus:

```
d(state) = [ exp1, exp2 ];
```

For difference equations, the dynamic equation takes the form

```
q(x) = right-hand side;
```

Here we think of  $q$  as the forward shift operator:  $q(\mathbf{x}(t)) = \mathbf{x}(t + 1)$ .

A time block may also contain a block of arbitrary C code. It should occur first in the time block. It will be executed before the equations. There is presently no possibility to put raw C code *after* the equations are executed. (However, one way to circumvent this would be to use function calls, e.g. of the type ' $d(\mathbf{z}) = \mathbf{f}(\mathbf{z}) + \text{function}();$ ' where `function` always returns 0.)

Declarations of states etc. are also allowed in the time block, as long as a quantity is declared before it is referenced. This feature is necessary for certain machine generated system descriptions (e.g. using macros). It is not the recommended practice to take advantage of it in manually written system descriptions.

One *time* is predefined: `START`. `AT TIME START` contains equations and/or C-code to be executed before the main integration. For example, functions can be set to their initial values in this section. If an error is detected the C statement '`return -1;`' should be executed. This will stop the simulation.

Dynamic equations (differential equations and difference equations) are only allowed in the `AT TIME t` block. Algebraic equations may occur only in the `AT TIME t` and the `AT TIME START` block.

## 5.6 Expressions and Operators in the System Description Language

An *expression* is any sequence of operators and operands in the C programming language that produces a value. The simplest expressions are parameter, function, and state names, which yield values directly. Other expressions combine operators and subexpressions to produce values.

An expression within parentheses has the same value as the expression without parentheses would have. Any expression can be delimited by parentheses to change the precedence of its operators.

All declared quantities can be used in conjunction with C operators to create more complex expressions. The following table presents the set of C operators.

Operator	Example	Description/Meaning
+ [unary]	+a	Value of <i>a</i>
- [unary]	-a	Negative of <i>a</i>
~	~a	One's complement of <i>a</i>
++ [prefix]	++a	The value of <i>a</i> after increment by one
++ [postfix]	a++	The value of <i>a</i> before increment by one
-- [prefix]	--a	The value of <i>a</i> after decrement by one
-- [postfix]	a--	The value of <i>a</i> before decrement by one
+ [binary]	a + b	<i>a</i> plus <i>b</i>
- [binary]	a - b	<i>a</i> minus <i>b</i>
* [binary]	a * b	<i>a</i> times <i>b</i>
/	a / b	<i>a</i> divided by <i>b</i>
%	a % b	Remainder of <i>a/b</i>
>>	a >> b	<i>a</i> , right-shifted <i>b</i> bits
<<	a << b	<i>a</i> , left-shifted <i>b</i> bits
<	a < b	1 if <i>a &lt; b</i> ; 0 otherwise
>	a > b	1 if <i>a &gt; b</i> ; 0 otherwise
<=	a <= b	1 if <i>a &lt;= b</i> ; 0 otherwise
>=	a >= b	1 if <i>a &gt;= b</i> ; 0 otherwise
==	a == b	1 if <i>a</i> equal to <i>b</i> ; 0 otherwise
!=	a != b	1 if <i>a</i> not equal to <i>b</i> ; 0 otherwise
& [binary]	a & b	Bitwise AND of <i>a</i> and <i>b</i>
	a   b	Bitwise OR of <i>a</i> and <i>b</i>
^	a ^ b	Bitwise XOR (exclusive OR) of <i>a</i> and <i>b</i>
&&	a && b	Logical AND of <i>a</i> and <i>b</i> (yields 0 or 1)
	a    b	Logical OR of <i>a</i> and <i>b</i> (yields 0 or 1)
!	!a	Logical NOT of <i>a</i> (yields 0 or 1)
?:	a ? e1 : e2	Expression <i>e1</i> if <i>a</i> is nonzero; Expression <i>e2</i> if <i>a</i> is zero
=	a = b	<i>a</i> , after <i>b</i> is assigned to it

<code>+=</code>	<code>a += b</code>	<i>a plus b (assigned to a)</i>
<code>--=</code>	<code>a -= b</code>	<i>a minus b (assigned to a)</i>
<code>*=</code>	<code>a *= b</code>	<i>a times b (assigned to a)</i>
<code>/=</code>	<code>a /= b</code>	<i>a divided by b (assigned to a)</i>
<code>%=</code>	<code>a %= b</code>	<i>Remainder of a/b (assigned to a)</i>
<code>&gt;&gt;=</code>	<code>a &gt;&gt;= b</code>	<i>a, right-shifted b bits (assigned to a)</i>
<code>&lt;&lt;=</code>	<code>a &lt;&lt;= b</code>	<i>a, left-shifted b bits (assigned to a)</i>
<code>&amp;=</code>	<code>a &amp;= b</code>	<i>a and b (assigned to a)</i>
<code> =</code>	<code>a  = b</code>	<i>a OR b (assigned to a)</i>
<code>^=</code>	<code>a ^= b</code>	<i>a XOR b (assigned to a)</i>

The C operators fall into the following categories:

- Unary operators, which take a single operand.
- Postfix operators, which follow a single operand.
- Unary prefix operators, which precede a single operand.
- Binary operators, which take two operands and perform a variety of arithmetic and logical operations.
- The conditional operator (a ternary operator), which takes three operands and resolves to the value of either the second or third expression, depending on the result of the evaluation of the first expression.

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator:

```
x = 8 + 4 * 2;      /* x is assigned 16, not 24 */
```

The previous statement is equivalent to the following:

```
x = 8 + ( 4 * 2 );
```

Using parenthesis in an expression alters the default precedence. For example:

```
x = (8 + 4) * 2; /* (8 + 4) is evaluated first */
```

In an unparenthesized expression, operators of higher precedence are evaluated before those of lower precedence. Consider the following expression:

```
A + B * C
```

The identifiers B and C are multiplied first because the multiplication operator (\*) has higher precedence than the addition operator (+).

A useful construction is the ternary ‘?’ operator. A good example of its use may be

```
step = t < t0 ? 0 : 1;
```

This expression states that `step` has the value 0 if `t < t0`, else the value 1.

## 5.7 Mathematical Functions in the System Description Language

Function	Description/Meaning
asin	Arc sine of x
atan	Arc tangent of x
atan2	Arc tangent of two variables
acos	Arc cosine of x
abs	Absolute value of an integer x
ceil	Smallest integral value not less than x
cos	Cosine of x
cosh	Hyperbolic cosine of x
cube	x cubed
exp	e raised to the power of x
floor	Largest integral value not greater than x
fabs	Absolute value of a floating-point number x
log	Natural logarithm of x
log10	Base-10 logarithm of x
pow	x to the yth power
sin	Sine of x
sinh	Hyperbolic sine of x
sqrt	Square root of x
sqr	x squared
tanh	Hyperbolic tangent of x
tan	Tangent of x

## 6 Emacs Editing Mode

### 6.1 Description

MRCI has an Emacs major mode for composing MRCI system description files and compiling them to object code using the MRCI translator and the MRCI system make file (`'Makefile.mrcicompile'`).

This mode offers several features to aid composing of MRCI system description files, including syntax colorization using font-lock, a syntax table appropriate to MRCI, and key bindings.

Once the system description file is composed, there is a function for compiling the script. The interaction with the MRCI translator and the C compiler is within a *comint* buffer.

The following key strokes are recognized:

- `(Ctrl)-c m f`  
inserts a MRCI function declaration
- `(Ctrl)-c m t`  
inserts a MRCI table function declaration
- `(Ctrl)-c m p`  
inserts a MRCI parameter declaration
- `(Ctrl)-c m s`  
inserts a MRCI state declaration
- `(Ctrl)-c m c`  
compiles the MRCI system description in the current buffer

Entering MRCI mode runs the hook *c-mode-common-hook*, then *mrci-mode-hook*. The customizable variable *mrci-compile-command* contains the command to compile MRCI system description files. Each buffer in MRCI mode has its own local copy of *mrci-compile-command*.

### 6.2 Installation

By default, the MRCI configure script places the file `'mrci-mode.el'` in `'/usr/local/share/emacs/site-lisp'`. This can be changed by the user by using the `'--prefix'` option. See the configure script help information for further details. You'll need to make sure that the directory where this file is placed is listed in the *load-path* variable. See the *load-path* documentation for details.

To autoload *mrci-mode* on any file with `'mrci'` extension, put this in your `'emacs'` file

```
(require 'mrci-mode)
```

This module has been tested with GNU Emacs 21.2 and 20.7.

## 7 Common Questions

1. Why is your driver so limited?

It is the only way we can achieve 30 kHz rates. There are technical issues with the National Instruments boards that make it difficult to do single-cycle high-speed acquisition with multiple input channels.

We are working on a COMEDI version (COMEDI is a Linux device driver package that supports a wide-range of data acquisition boards). This version will support multiple input channels, but will probably only run at 5 kHz, maybe 10 kHz. COMEDI also has "issues" autocalibrating the boards that we use.

2. When will we support the Digidata 1200A/B?

It depends. Axon Instruments no longer makes it, but a lot of people have it. They want me to pay \$2700 for one, which is outrageous since there are comparable boards on the market today that cost \$600! If someone wants to donate one (and a photocopy of the Appendix with the register-level programming information) and we have the time and manpower, we'll gladly write a driver for it. I've read the manual in the past and believe that this driver will be relatively easy to write.

3. What about Instrutech Boards?

A Linux driver exists for the ITC-18 see Instrutech's web site for a link to it. It should not be too hard to write a daq wrapper.c file for it.

## 8 Troubleshooting

Using an incorrect filename in a read command, such as `readvar`, may lock up MRCI and/or crash the computer. We've tried to find all of these situations and add error checking, but some bugs may still be lurking. If for some unfortunate reason MRCI crashes, you may have to run the `mrci_rmrt1` script before you are able to run MRCI again. If that doesn't help, reboot.

The purpose of the chapter is to provide a background on the translation, compilation and running process. It should help the user diagnose their mistakes, and to help diagnose bugs in MRCI or in its environment. It should be pointed out that MRCI is a program that works more intimately with the operating system environment than most other programs. If something does not work on your system, it may be that either MRCI *or* the configuration of your system is to blame.

### 8.1 Building MRCI Systems

The exact steps involved in the creation of the kernel module '`.o`' file are as follows: The task is carried out by the Unix program `make`, using a *make file* automatically generated during the installation of MRCI. The steps are as follows:

When trying to create the '`.o`' file, MRCI first checks if there is one already in the same directory, and if it is more recent than the system description file. If so, that one is used instead of creating it anew, which saves some time. Since the '`.o`' file is fairly small (typically < 50kB) it is probably a good idea not to delete it.<sup>1</sup>

1. First, the system file is processed by the m4 macro processor and subsequently by the MRCI translator, which is a separate program named `translator`. A C file will be generated. If the system file does not obey the syntax of the system description language, the translation will fail, and the translator will produce an error message which hopefully is understandable.
2. The generated C file is compiled. Unless UNSAFE status is returned from the translator, if this step fails it is to be considered as a bug in MRCI or in its environment. However, warnings about implicitly defined functions are to be taken seriously, and will indicate a real error. They indicate that the system called some function the compiler did not know about, say that you wrote '`sine(x)`', instead of '`sin(x)`'.
3. A Linux kernel module, the '`.o`' file, is generated using the `ld` program. (If this step fails, it is probably to be considered as installation error.)

---

<sup>1</sup> However, it is not guaranteed to be compatible between different MRCI versions, or different versions of Linux/RT-Linux. You should therefore delete all '`.o`' files when the MRCI version has changed, or when they were generated under another operating system release.

## 8.2 Fixing syntax errors

Just as when writing programs in any programming language, or text for any text formatter, you will make mistakes that need to be found and fixed. Unfortunately, you're likely to encounter bugs, in which case you are kindly asked to report reproducible failures.

If the translation/compilation/linking failed, you will get error messages. First, you have to see if the error came from the translator: if the error messages are prepended by 'mrci\_translator:' they are. If an input file does not conform to the syntax (for example: misspelled keywords, unbalanced parentheses, or forgotten semicolons), you will get error messages. These are called *syntactic errors*. If you get a syntactic error, the translator will tell you where it found the error. (The translator does not attempt error recovery, but just gives up after the first error). If you desire, you can run the translator outside of MRCI, typically with a command such as

```
/usr/local/libexec/mrci_translator system.mrci
```

You will also get error messages for doing some other things which are not allowed, such as leaving some variable unassigned, or assigning them twice. In this case, pointing to a particular line would in general be meaningless.

If the translation succeeds, the translator informs the user how sure it is concerning the correctness of the program: If no raw C code and no unknown function calls were encountered, it considers the program as *safe*, all subsequent steps of the creation and loading should succeed. If calls to functions not known to the translator are found, *semi-safe* state is returned, the compilation following should succeed, but errors may occur during the loading phase. By *unsafe* status, either the compilation or the loading may fail. It can be noted that the translator knows the normal mathematical functions such as 'sin', and checks its usage, instead of considering it semi-safe. It will detect if you call 'sin' with two arguments or 'atan2' with one.

## 9 Using Your Own Drivers

Starting with version 1.0.0 of MRCI, we have a file called ‘`daq_wrapper.c`’. This file contains a standardized application interface for MRCI calling acquisition device drivers.

If you want to use your own device drivers (ex., Comedi), simply edit the required functions with the calls to your driver.

### 9.1 `daqw_init_board`

```
void daqw_init_board (short, short);
```

Called by MRCI at module initialization time. Used to initialize the data acquisition board. The two arguments are number of input channels and number of out channels, respectively.

### 9.2 `daqw_read_input`

```
void daqw_read_input (short *);
```

Called by MRCI at the beginning of each computational cycle. Used to read samples from the data acquisition board. The supplied parameter is an array of `short` variables, which size depends on whether the system is one-channel or two-channel mode. The mode is determined by a global variable called `daq_mode`. If `daq_mode` is 0, that means one-channel mode. If `daq_mode` is 1, that means two-channel mode.

### 9.3 `daqw_start_input`

```
void daqw_start_input (void);
```

This function may contain code that needs to be invoked prior to the actual reading of samples from the data acquisition board.

### 9.4 `daqw_write_output`

```
void daqw_write_output (short *);
```

Called by MRCI at the end of each computational cycle. Used to write values from the external output states to the data acquisition board. The size of the supplied array depends on the mode, identically with `daqw_read_input`.

### 9.5 `daqw_conv_input_to_phys`

```
double daqw_conv_input_to_phys (short);
```

This function is used by MRCI is used to convert to volts a raw value read from the data acquisition board.

### 9.6 `daqw_conv_phys_to_output`

```
short daqw_conv_phys_to_output (double);
```

This function is used by MRCI to convert a voltage value, passed as an argument into raw value to be written to the data acquisition board.

## 10 Specification of the System Description Language

### 10.1 Terminal Symbols in the System Description Language

Token Name	Description
FLOAT	A floating point number, either in a floating decimal or scientific (e) notation, or the string 'Inf', to indicate infinity.
INT	A decimal integer.
ID	An identifier consisting of letters, digits and underscore, where the first character can only be a letter.
STRING	A string of characters enclosed by '"'.
C_BLOCK	A block of C code, enclosed by '{' and '}'.
OR	The C operator denoting logical OR ('  ').
AND	The C operator denoting logical AND ('&&').
EQ	The C operator denoting equality ('==').
NE	The C operator denoting non-equality ('!=').
GE	The C operator denoting greater-than or equal ('>=').
LE	The C operator denoting less-than or equal ('<=').
LL	The C operator denoting left bit shift ('<<').
GG	The C operator denoting right bit shift ('>>').
PLUSPLUS	The C operator denoting increment by one ('++').
MINUSMINUS	The C operator denoting decrement by one ('--').

#### Reserved Words:

TIME	FUNCTION
SYSTEM	C_IDENTIFIER
DEPENDENCY	ATTIME
LOW	
HIGH	
STEP	
TABLE	
EXTERNAL	
INPUT	
OUTPUT	
METHOD	
STATE	
INTEGERSTATE	
INTEGER	
DOUBLE	
PARAMETER	

## 10.2 Grammar of the System Description Language

```

system          ::= pass_thrus system_decl decls time_block time_blocks
decls           ::=
decls           ::= decls decl
decl            ::= component_decl
decl            ::= time_decl
decl            ::= function_def
decl            ::= pass_thru
time_blocks     ::=
time_blocks     ::= time_blocks time_block
time_block      ::= time_header C_block equations
time_header     ::= ATTIME ID ":"
C_block         ::=
C_block         ::= C_BLOCK
formal_param    ::= ID
formal_param_list ::= formal_param "," formal_param "," formal_param
formal_param_list ::= formal_param "," formal_param_list
function_def    ::= unary_function_def
function_def    ::= binary_function_def
function_def    ::= general_function_def
unary_function_def ::= ID "(" formal_param ")" "=" exp ";"
binary_function_def ::= ID "(" formal_param "," formal_param ")" "=" exp ";"
general_function_def ::= ID "(" formal_param_list ")" "=" exp ";"
equations       ::=
equations       ::= equations equation
equation        ::= diff_eqn
equation        ::= difference_eqn
equation        ::= alg_eqn
equation        ::= implicit_eqn
equation        ::= component_decl
equation        ::= pass_thru
diff_eqn        ::= "d" "(" ID ")" "=" arglist ";"
difference_eqn  ::= "q" "(" ID ")" "=" exp ";"
alg_eqn         ::= ID "=" exp ";"
implicit_eqn    ::= INT "=" exp ";"
multiplier      ::=
multiplier      ::= exp "*"
exp             ::= FLOAT
exp             ::= INT
exp             ::= ID
exp             ::= "q" "(" ID ")"
exp             ::= int_cast exp
exp             ::= double_cast exp
exp             ::= fname "(" ")"
exp             ::= fname "(" exp ")"
exp             ::= fname "(" exp "," exp ")"

```

```

exp                ::= fname "(" exp "," exp "," arglist ")"
exp                ::= ID bracket_args
exp                ::= exp "+" exp
exp                ::= exp "-" exp
exp                ::= exp "*" exp
exp                ::= exp "/" exp
exp                ::= exp "%" exp
exp                ::= PLUSPLUS ID
exp                ::= MINUSMINUS ID
exp                ::= ID PLUSPLUS
exp                ::= ID MINUSMINUS
exp                ::= "-" exp
exp                ::= "+" exp
exp                ::= "!" exp
exp                ::= "~" exp
exp                ::= exp "^" exp
exp                ::= "(" exp ")"
exp                ::= exp LL exp
exp                ::= exp GG exp
exp                ::= exp EQ exp
exp                ::= exp NE exp
exp                ::= exp LE exp
exp                ::= exp GE exp
exp                ::= exp "<" exp
exp                ::= exp ">" exp
exp                ::= exp "&" exp
exp                ::= exp "|" exp
exp                ::= exp AND exp
exp                ::= exp OR exp
exp                ::= exp "?" exp ":" exp
fname              ::= ID
fname              ::= fname "." ID
arglist           ::= exp
arglist           ::= arglist "," exp
int_cast          ::= "(" INTEGER ")"
double_cast       ::= "(" DOUBLE ")"
bracket_args      ::= "[" exp "]"
bracket_args      ::= bracket_args "[" exp "]"
time_decl         ::= TIME ID ";"
system_decl       ::= continuous_system
latin1id          ::= ID
latin1id          ::= ID8
continuous_system ::= SYSTEM latin1id ";"
component_decl    ::= state_decl
component_decl    ::= int_state_decl
component_decl    ::= par_decl
component_decl    ::= func_decl

```

```

component_decl ::= table_decl
component_decl ::= id_decl
component_decl ::= external_decl
state_decl ::= scalar_states_decl
scalar_states_decl ::= STATE scalar_states ";"
scalar_states ::= scalar_state_decl
scalar_states ::= scalar_state_decl "," scalar_states
scalar_state_decl ::= ID "=" real string method
scalar_state_decl ::= ID "=" real string
method ::= METHOD string
int_state_decl ::= INTEGERSTATE int_states ";"
int_states ::= int_state
int_states ::= int_state "," int_states
int_state ::= ID "=" INT string
par_decl ::= scalar_par_decl
scalar_par_decl ::= PARAMETER scalar_pars ";"
scalar_pars ::= scalar_par
scalar_pars ::= scalar_par "," scalar_pars
scalar_par ::= ID "=" real string
func_decl ::= FUNCTION functs_decl ";"
functs_decl ::= func
functs_decl ::= func "," functs_decl
func ::= ID string
table_decl ::= TABLE FUNCTION ID "(" formal_param ")" "=" exp "," LOW
               "=" real "," HIGH "=" real "," STEP "=" real ","
               DEPENDENCY "=" ID string ";"
external_decl ::= EXTERNAL externals_decl ";"
externals_decl ::= external
externals_decl ::= external "," externals_decl
external ::= OUTPUT ID string
external ::= INPUT ID string
id_decl ::= C_IDENTIFIER id_decls ";"
id_decls ::= id
id_decls ::= id "," id_decls
id ::= ID
string ::=
string ::= STRING
real ::= FLOAT
real ::= "-" FLOAT
real ::= INT
real ::= "-" INT
pass_thrus ::=
pass_thrus ::= pass_thrus pass_thru
pass_thru ::= ";"
pass_thru ::= COMMENT

```

## 11 References

[1] R. J. Butera, J. Rinzel, and J. C. Smith. Models of the neuronal kernel for respiratory rhythm generation in the pre-Botzinger complex of mammals: I. Bursting pacemaker neurons. *J. Neurophysiol.*, 82:382397, 1999.

[2] R. J. Butera, C. G. Wilson, C. A. DelNegro, and J. C. Smith. A methodology for achieving high-speed rates for artificial conductance injection in electrically excitable biological cells. *IEEE Trans. Biomed. Eng.*, 2001. Submitted.

[3] A. A. Sharp, M. B. O'Neil, L. F. Abbott, and E. Marder. Dynamic clamp: computer-generated conductances in real neurons. *J. Neurophysiol.*, 69:992995, 1993.

## Short Contents

1	Introduction . . . . .	1
2	Installation and Running . . . . .	4
3	Commands . . . . .	12
4	System Checkout and Tutorial . . . . .	24
5	The System Description Language . . . . .	31
6	Emacs Editing Mode . . . . .	39
7	Common Questions . . . . .	40
8	Troubleshooting . . . . .	41
9	Using Your Own Drivers . . . . .	43
10	Specification of the System Description Language . . . . .	44
11	References . . . . .	48

# Table of Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Purpose of MRCI	1
1.2	Changes since the previous release	1
1.2.1	Changes between this release and release 1.9.1	1
1.2.2	Changes between release 1.9.1 and release 1.9.0	1
1.2.3	Changes between release 1.9.0 and release 1.8.0	1
1.2.4	Changes between release 1.8.0 and release 1.6.0	2
1.2.5	Changes between release 1.6.0 and release 1.5.0	2
1.2.6	Changes between release 1.4.0 and release 1.5.0	2
1.2.7	Changes between release 1.3.5 and release 1.4.0	2
1.2.8	Changes between release 1.3.0 and release 1.3.5	2
1.2.9	Changes between release 1.2.0 and release 1.3.0	2
1.2.10	Changes between release 1.1.0 and release 1.2.0	2
1.3	Acknowledgments	2
<b>2</b>	<b>Installation and Running</b>	<b>4</b>
2.1	Prerequisites	4
2.2	Binary Installation	4
2.3	Unpacking	5
2.4	Building and Installation	6
2.4.1	Configuring MRCI for use with RTAI	6
2.4.2	Configuring MRCI for use with RT-Linux and RT-Linux Pro	6
2.4.3	Configuring MRCI for use with COMEDI	7
2.4.4	Configuring MRCI for use with the bundled NI driver	7
2.4.5	Other configuration options	7
2.5	Running	8
2.6	Uninstalling	8
2.7	Running in Remote Mode	8
<b>3</b>	<b>Commands</b>	<b>12</b>
3.1	Summary of Commands	12
3.2	Command-Line Editing	13
3.3	Shell Commands	14
3.4	on	14
3.5	off	14
3.6	status	14
3.7	setvar	15
3.8	getvar	15
3.9	cd	15
3.10	pwd	15
3.11	writevar	15

3.12	readvar	16
3.13	incvar	16
3.14	scalevar	16
3.15	cap	16
3.16	nocap	16
3.17	caplen	16
3.18	capture	16
3.19	stats	17
3.20	setrate	17
3.21	table recalc	17
3.22	reset system	17
3.23	dsamp	17
3.24	timing	18
3.25	ramp	18
3.26	rampf	18
3.27	rampoff	18
3.28	script	19
3.29	load	19
3.30	system	19
3.31	play	20
3.32	sleep	20
3.33	help	20
3.34	exit	20
3.35	clear history	20
3.36	save history	20
3.37	read history	21
3.38	list history	21
3.39	del history	21
3.40	add event	21
3.41	refract	22
3.42	del event	22
3.43	enable event	22
3.44	disable event	23
3.45	dump event	23
3.46	reset event	23
3.47	ver	23
<b>4</b>	<b>System Checkout and Tutorial</b>	<b>24</b>
4.1	Compiling the Model and Running MRCI	24
4.2	Overview of Model	24
4.3	A Real-Time Morris-Lecar Model	25
4.4	Synaptic Input to the Morris-Lecar Model	27
4.5	Connecting to an Electrophysiology Amplifier	27
4.6	Altering the Properties of an RC Cell Model	28
4.7	Constant Current	28
4.8	Modified RC Parameters	28
4.9	Synaptic Output from the Morris-Lecar Model	29
4.10	A Scripting Example	29

4.11	A Stimulus/Pulse Generator (Inline C Code Example) . . . . .	30
<b>5</b>	<b>The System Description Language . . . . .</b>	<b>31</b>
5.1	General . . . . .	31
5.2	Concepts . . . . .	31
5.3	Structure of a MRCI System . . . . .	32
5.4	Declarations . . . . .	32
5.5	Time Blocks . . . . .	34
5.6	Expressions and Operators in the System Description Language . . . . .	36
5.7	Mathematical Functions in the System Description Language . .	38
<b>6</b>	<b>Emacs Editing Mode . . . . .</b>	<b>39</b>
6.1	Description . . . . .	39
6.2	Installation . . . . .	39
<b>7</b>	<b>Common Questions . . . . .</b>	<b>40</b>
<b>8</b>	<b>Troubleshooting . . . . .</b>	<b>41</b>
8.1	Building MRCI Systems . . . . .	41
8.2	Fixing syntax errors . . . . .	42
<b>9</b>	<b>Using Your Own Drivers . . . . .</b>	<b>43</b>
9.1	daqw_init_board . . . . .	43
9.2	daqw_read_input . . . . .	43
9.3	daqw_start_input . . . . .	43
9.4	daqw_write_output . . . . .	43
9.5	daqw_conv_input_to_phys . . . . .	43
9.6	daqw_conv_phys_to_output . . . . .	43
<b>10</b>	<b>Specification of the System Description Language . . . . .</b>	<b>44</b>
10.1	Terminal Symbols in the System Description Language . . . . .	44
10.2	Grammar of the System Description Language . . . . .	45
<b>11</b>	<b>References . . . . .</b>	<b>48</b>