labCA — An EPICS Channel Access Interface for *scilab* and *matlab*

Till Straumann <straumanATslac.stanford.edu>, 2003

April 13, 2006

`Id: manual.tex,v 1.31 2006/04/17 01:11:32 strauman Exp`

1 Introduction

The *labCA* package provides an interface to the EPICS channel access client library which can be integrated with the *scilab* or *matlab* applications. Both, *scilab* and *matlab* feature an API for interfacing to user binaries written in a programming language such as C or Fortran. *labCA* properly wraps the essential channel access routines and makes them accessible from the *scilab*[1] command line.

labCA actually uses an extra layer, the *ezca* library which transparently manages and caches channel access connections. A modified version of *ezca* comes with *labCA*, adding thread-safety and hence EPICS 3.14 fitness.

A very convenient feature of *labCA* is the ability to execute *ezca* calls on groups of PVs, simply by passing the respective *labCA* routine a column vector of PV names.

labCA has been tested with EPICS 3.13, 3.14, *scilab*-2.7 .. *scilab*-4.0, *matlab*-6.5, *matlab*-7 on *linux*, *solaris* and *win32*. Note that while some of these combinations have been tested and been known working in the past, only the latest versions of the respective components have been tested and verified to build and run successfully with the current version of *labCA*. Modifications to the `Makefiles` might be necessary to build older versions. Note also that support for EPICS 3.13 is being deprecated. Future versions of *labCA* will have the old `Makefiles` removed.

[1]throughout this text, references to *scilab* usually mean *scilab* or *matlab*.

2 Supported EZCA Calls

labCA implements an interface to almost all public *ezca* routines[2]. Note that the arguments and return values do not exactly correspond to the respective *ezca* originals but have been adapted to make sense in the *scilab* environment.

2.1 Common Arguments and Return Values

2.1.1 PV Argument

All *labCA* calls take a *PV* argument identifying the EPICS process variables the user wants to connect to. PVs are plain ASCII strings. *labCA* is capable of handling multiple PVs in a single call; they are simply passed as a column vector:

```
pvs = [ 'PVa'; 'b'; 'anotherone' ]
```

Because *matlab* doesn't allow the rows of a string vector to be of different size, the *matlab* wrapper expects a *cell-* array of strings:

```
pvs = { 'PVa'; 'b'; 'anotherone' }
```

All channel access activities for the PVs passed to a single *labCA* call are batched together and completion of the batch is awaited before returning from the *labCA* call. Consider the following example:[3]

```
lcaPut( 'trigger', 1 ) \\
data=lcaGet( ['sensor1' ; 'sens2'] );
```

- It is guaranteed that writing the "trigger" completes (on the CA server) prior to reading the sensors.[4]
- Reading the two sensors is done in "parallel" — the exact order is unspecified. After the command sequence (successfully) completes, all the data are valid.

2.1.2 Timestamp Format

Channel access timestamps are "POSIX struct timespec" compliant, i.e. they provide the number of nanoseconds expired since 00:00:00 UTC, January 1, 1970. *labCA* translates the timestamps into *complex numbers* with the seconds (`tv_sec` member) and nanoseconds (`tv_nsec`) in the real and imaginary parts, respectively. This makes it easy to extract the seconds while still maintaining full accuracy.

[2] the *matlab* implementation may still lack some of the more esoteric commands

[3] In *matlab*, the square brackets ("[]") must be replaced by curly braces ("{}").

[4] If the remote sensors have finite processing time, the subsequent CA read may still get old data — depending on the device support etc.; this is beyond the scope of channel access, however.

2.2 lcaGet

2.2.1 Calling Syntax

```
[value, timestamp] = lcaGet(pvs, nmax, type)
```

2.2.2 Description

Read a number of m PVs, which may be scalars or arrays of different dimensions. The result is converted into a $m \times n$ matrix. The number of columns, n, is automatically assigned to fit the largest array among the m PVs. PVs with less than n elements have their excess elements in the result matrix filled with NaN.

If all PVs are of native type DBF_STRING or DBF_ENUM, the values are returned as character strings; otherwise, all values are converted into double precision numbers. Explicit type conversion into strings can be enforced by submitting the 'type' argument described below.

2.2.3 Parameters

pvs Column vector (in matlab: $m \times 1$ *cell-* matrix) of m strings.

nmax (*optional argument*) Maximum number of elements (per PV) to retrieve (i.e. limit the number of columns of `value` to *nmax*). If set to 0 (default), all elements are fetched and the number of columns, n, in the result matrix is set to the maximum number of elements among the PVs. The option is useful to limit the transfer time of large waveforms (unfortunately, CA does not return the valid elements ("NORD") of an array only — it always ships *all* elements).

type (*optional argument*) A string specifying the data type to be used for the channel access data transfer. Note that unless the PVs are of native "string" type or conversion to "string" is enforced explicitely (type = `char`), *labCA* always converts the data to "double" locally.

It can be desirable, however, to use a different data type for the transfer because by default CA transfers are limited to ≈ 16kB. Legal values for `type` are `byte`, `short`, `long`, `float`, `double`, `native` or `char` (strings). There should rarely be a need for using anything other than `native`, the default, which directs CA to use the same type for transfer as the data are stored on the server.

Occasionally, conversion to `char` can be useful: retrieve a number of PVs as strings, i.e. let the CA server convert them to strings (if the PVs are not native strings already) and transfer them.

If multiple PVs are requested, either none or all must be of native DBF_STRING or DBF_ENUM type unless explicit conversion to `char` is enforced by specifying this argument.

Note that while `native` might result in different types being used for different PVs, it is currently not possible to explicitely request different types for individual PVs (i.e. `type` can't be a vector).

value The $m \times n$ result matrix. n is automatically assigned to accomodate the PV with the most elements. If the `nmax` argument is given and is nonzero but less than the automatically determined n, then n is clipped to `nmax`. Excess elements of PVs with less than n elements are filled with NaN values.

The result is either a 'double' or a (*matlab*: *cell*-) 'string' matrix (if all PVs are of native string type or explicit conversion was requested by setting the 'type' argument to '`char`').

labCA checks the channel access severity of the retrieved PVs and fills the rows corresponding to *INVALID* PVs with NaN[5]. In addition, warning messages are printed to the console if a PV's alarm status exceeds a configurable threshold (see 2.14). The refusal to read PVs with *INVALID* severity can be tuned using the `lcaSetSeverityWarnLevel` call as well.

timestamp (*optional result*) A $m \times 1$ column vector of *complex* numbers holding the CA timestamps of the requested PVs. The timestamps count the number of seconds (real part) and fractional nanoseconds (imaginary part) elapsed since 00:00:00 UTC, Jan. 1, 1970.

2.2.4 Examples

```
// read a PV
    lcaGet( 'thepv' )
// read multiple PVs along with their EPICS timestamps
    [ vals, tstamps] = lcaGet( [ 'aPV' ; 'anotherPV' ] )
// read an 'ENUM/STRING'
    lcaGet( 'thepv.SCAN' )
// read an 'ENUM/STRING' as a number (server converts)
    lcaGet( 'thepv.SCAN', 0, 'float' )
// enforce reading all PVs as strings (server converts)
// NOTE: necessary if native num/nonnum types are mixed
    lcaGet( [ 'apv.SCAN'; 'numericalPV' ] , 0, 'char' )
// limit reading a waveform to its NORD elements
    nord = lcaGet( 'waveform.NORD' )
if nord > 0 then
      lcaGet( 'waveform', nord )
end
```

[5] Actually, all fields of an EPICS database record share a common severity, (which itself is a field/PV — the .SEVR field). However, the *INVALID* status actually only applies to the .VAL field of a record — other fields (e.g. .EGU) may still hold meaningful data. Consequently, *INVALID* PVs are returned as NaN only if they refer to a record's .VAL field.

2.3 lcaPut

2.3.1 Calling Syntax

```
lcaPut(pvs, value, type)
```

2.3.2 Description

Write to a number of PVs which may be scalars or arrays of different dimensions. It is possible to write the same value to a collection of PVs.

2.3.3 Parameters

pvs Column vector (in matlab: $m \times 1$ *cell-* matrix) of m strings.

value $m \times n$ matrix or $1 \times n$ row vector of values to be written to the PVs. If there is only a single row in `value` it is written to all m PVs. `value` may be a matrix of "double" precision numbers or a (*matlab*: *cell-*) matrix of strings (in which case the values are transferred as strings and converted by the CA server to the native type — this is particularly useful for DBF_ENUM / "menu" type PVs).

It is possible to write less than n elements — *labCA* scans all rows for NaN values and only transfers up to the last non-NaN element in each row.

type (*optional argument*) A string specifying the data type to be used for the channel access data transfer. Note that *labCA* always converts numerical data from "double" locally.

It can be desirable, to use a different data type for the transfer because by default CA transfers are limited to ≈ 16kB. Legal values for `type` are `byte`, `short`, `long`, `float`, `double`, `char` or `native`. There should rarely be a need for using anything other than `native`, the default, which directs CA to use the same type for transfer as the data are stored on the server. If `value` is a string matrix, `type` is automatically set to `char`.

Note that while `native` might result in different types being used for different PVs, it is currently not possible to explicitely request different types for individual PVs (i.e. `type` cannot be a vector).

2.3.4 Examples

```
// write a PV
    lcaPut( 'thepv', 1.234 )
// write as a string (server converts)
    lcaPut( 'thepv', '1.234' )
```

```
// write/transfer as a short integer (server converts)
    lcaPut( 'thepv', 12, 'short' )
// write multiple PVs (use { } on matlab)
    lcaPut( [ 'pvA'; 'pvB' ], [ 'a'; 'b' ] );
// write array PV
    lcaPut( 'thepv' , [ 1, 2, 3, 4 ] )
// write same value to a group of PVs (string
// concatenation differs on matlab)
    lcaPut( [ 'pvA'; 'pvB' ] + '.SCAN', '1 second' )
// write array and scalar PV (using NaN as a delimiter)
    tab = [ 1, 2, 3, 4 ;   5, %nan, 0, 0 ]
lcaPut( [ 'arrayPV'; 'scalarPV' ], tab )
```

2.4 lcaPutNoWait

2.4.1 Calling Syntax

```
lcaPutNoWait(pvs, value, type)
```

2.4.2 Description

`lcaPutNoWait` is a variant of `lcaPut` that does *not wait* for the channel access put request to complete on the server prior to returning control to the command line. This call can be useful to set PVs that are known to take a long or indefinite time to complete processing, e.g., arming a waveform record which is triggered by a hardware event in the future or starting a stepper motor.

2.4.3 Parameters

See `lcaPut`.

2.5 lcaGetNelem

2.5.1 Calling Syntax

```
numberOfElements = lcaGetNelem(pvs)
```

2.5.2 Description

Retrieve the element count of a number of PVs. Note that this is not necessarily the number of *valid* elements (e.g. the actual number of values read from a device into a waveform) but the maximum number of elements a PV can hold.

2.5.3 Parameters

pvs Column vector (in matlab: $m \times 1$ *cell*- matrix) of m strings.

numberOfElements $m \times 1$ column vector of the PV's number of elements ("array dimension").

2.6 lcaGetControlLimits

2.6.1 Calling Syntax

```
[lowLimit, hiLimit] = lcaGetControlLimits(pvs)
```

2.6.2 Description

Retrieve the control limits associated with a number of PVs.

2.6.3 Parameters

pvs Column vector (in matlab: $m \times 1$ *cell*- matrix) of m strings.

lowLimit $m \times 1$ column vector of the PV's low control limit.

hiLimit $m \times 1$ column vector of the PV's high control limit.

2.7 lcaGetGraphicLimits

2.7.1 Calling Syntax

```
[lowLimit, hiLimit] = lcaGetGraphicLimits(pvs)
```

2.7.2 Description

Retrieve the graphic limits associated with a number of PVs.

2.7.3 Parameters

pvs Column vector (in matlab: $m \times 1$ *cell*- matrix) of m strings.

lowLimit $m \times 1$ column vector of the PV's low graphic limit.

hiLimit $m \times 1$ column vector of the PV's high graphic limit.

2.8 lcaGetStatus

2.8.1 Calling Syntax

```
[severity, status, timestamp] = lcaGetStatus(pvs)
```

2.8.2 Description

Retrieve the alarm severity and status of a number of PVs along with their timestamp.

2.8.3 Parameters

pvs Column vector (in matlab: $m \times 1$ *cell*- matrix) of m strings.

severity $m \times 1$ column vector of the alarm severities.

status $m \times 1$ column vector of the alarm status.

timestamp $m \times 1$ *complex* column vector holding the PV timestamps (see 2.1.2 about the timestamp format).

2.9 lcaGetPrecision

2.9.1 Calling Syntax

```
precision = lcaGetPrecision(pvs)
```

2.9.2 Description

Retrieve the precision of a number of PVs.

2.9.3 Parameters

pvs Column vector (in matlab: $m \times 1$ *cell*- matrix) of m strings.

precisiom $m \times 1$ column vector of the PV's precision.

2.10 lcaGetUnits

2.10.1 Calling Syntax

```
units = lcaGetUnits(pvs)
```

2.10.2 Description

Retrieve the engineering units of a number of PVs.

2.10.3 Parameters

pvs Column vector (in matlab: $m \times 1$ *cell*- matrix) of m strings.

units $m \times 1$ column vector (on *matlab*: *cell*- matrix) of strings holding the PV EGUs.

2.11 lcaGetRetryCount, lcaSetRetryCount

2.11.1 Calling Syntax

```
currentRetryCount = lcaGetRetryCount()
lcaSetRetryCount(newRetryCount)
```

2.11.2 Description

Retrieve / set the *ezca* library retryCount parameter (consult the *ezca* documentation for more information). The retry count multiplied by the timeout parameter (see 2.12) determines the maximum time the interface waits for connections and data transfers, respectively.

2.12 lcaGetTimeout, lcaSetTimeout

2.12.1 Calling Syntax

```
currentTimeout = lcaGetTimeout()
lcaSetTimeout(newTimeout)
```

2.12.2 Description

Retrieve / set the *ezca* library timeout parameter (consult the *ezca* documentation for more information). Note that the timeout parameter is actually the *minimum* timeout the library pends for activity. The *maximal* time spent waiting for connections and/or data equals the timeout multiplied by the retry count (see 2.11).

2.13 lcaDebugOn, lcaDebugOff

2.13.1 Calling Syntax

```
lcaDebugOn()
lcaDebugOff()
```

2.13.2 Description

Switch the *ezca* library's debugging facility on and off, respectively.

2.14 lcaSetSeverityWarnLevel

2.14.1 Calling Syntax

```
lcaSetSeverityWarnLevel(newlevel)
```

2.14.2 Description

Set the warning threshold for `lcaGet()` operations. A warning message is printed when retrieving a PV with a severity bigger or equal to the warning level. Supported values are $0..3$ (No alarm, minor alarm, major alarm, invalid alarm). The initial/default value is 3.

If a value $>= 10$ is passed, the threshold for refusing to read the `.VAL` field of PVs with an *INVALID* severity can be changed. The rejection can be switched off completely by passing 14 ($= 10 + INVALID\_ALARM + 1$) or made more sensitive by passing a value of less than 13 ($= 10 + INVALID\_ALARM$), the default.

2.15 lcaClear

2.15.1 Calling Syntax

```
lcaClear(pvs)
```

2.15.2 Description

Clear / release (disconnect) channels. This is particularly useful with EPICS 3.14 to clean up invalid PVs (e.g., due to typos). Nonexisting PVs are continuously searched for by a CA background task which may result in cluttered IOC consoles and resource consumption. All monitors on the target channel(s) are cancelled/released as a consequence of this call.

2.15.3 Parameters

pvs Column vector (in matlab: $m \times 1$ *cell-* matrix) of m strings. Alternatively, `lcaClear` may be called with *no* rhs argument thus clearing *all* channels (and monitors).

2.15.4 Examples

```
\\ clear a number of channels
  lcaClear( ['aUseless_PV'; 'misTyppedPV' ] )
\\ purge all channels (dont use parenthesis in matlab)
  lcaClear()
```

2.16 lcaSetMonitor

2.16.1 Calling Syntax

```
lcaSetMonitor(pvs, nmax, type)
```

2.16.2 Description

Set a "monitor" on a set of PVs. Monitored PVs are automatically retrieved every time their value or status changes. *labCA* sets an internal flag when this happens. Monitors are especially useful under EPICS-3.14 which supports multiple threads. EPICS-3.14 transparently reads monitored PVs as needed. Older, single threaded versions of EPICS require periodic calls to *labCA* e.g., to `lcaDelay` (see 2.18), in order to allow *labCA* to handle monitors.

Use the `lcaNewMonitorValue` (see 2.17) call to check monitor status (local flag). If new data are available, they are retrieved using the ordinary `lcaGet` (see 2.2) call.

Note the difference between polling and monitoring a PV in combination with polling the local monitor status flag. In the first case, remote data are fetched on every polling cycle whereas in the second case, data are transferred only when they change. Also, in the monitored case, `lcaGet` reads from a local buffer rather than from the network.

There is currently no possibility to selectively remove a monitor. Use the `lcaClear` (see 2.15) call to disconnect a channel and as a side-effect, remove all monitors on that channel. Future access to a cleared channel simply reestablishes a connection (but no monitors).

2.16.3 Parameters

pvs Column vector (in matlab: $m \times 1$ *cell*- matrix) of m strings.

nmax (*optional argument*) Maximum number of elements (per PV) to monitor/retrieve. If set to 0 (default), all elements are fetched. See (2.2.3) for more information.

type (*optional argument*) A string specifying the data type to be used for the channel access data transfer. The native type is used by default. See (2.2.3) for more information.

The type specified for the subsequent `lcaGet` for retrieving the data should match the monitor's data type. Otherwise, `lcaGet` will fetch a new copy from the server instead of using the data that was already transferred as a result of the monitoring.

2.16.4 Examples

```
lcaSetMonitor('PV')
// monitor 'PV'. Reduce network traffic by just have the
// library retrieve the first 20 elements. Use DBR_SHORT
// for transfer.
lcaSetMonitor('PV', 20, 's')
```

2.17 lcaNewMonitorValue

2.17.1 Calling Syntax

```
[flags] = lcaNewMonitorValue(pvs, type)
```

2.17.2 Description

Check if monitored PVs need to be read, i.e, if fresh data are available (e.g., due to initial connection or changes in value and/or severity status). Reading the actual data must be done using `lcaGet` (see 2.2).

2.17.3 Parameters

pvs Column vector (in matlab: $m \times 1$ *cell*- matrix) of m strings.

type (*optional argument*) A string specifying the data type to be used for the channel access data transfer. The native type is used by default. See (2.2.3) for more information.

Note that monitors are specific to a particular data type and therefore `lcaNewMonitorValue` will only report the status for a monitor that had been established (by `lcaSetMonitor`) with a matching type. Using the "native" type, which is the default, for both calls satisfies this condition.

flags Column vector of flag values. A value of zero indicates that no new data are available – the monitored PV has not changed since it was last read (i.e., the data *not the flag*). A value of one indicates that new data are available for reading (`lcaGet`). A negative The flag value can be negative, indicating a problem e.g.,

-1: no monitor established (`lcaSetMonitor` (see 2.16) never called for this PV/data type),

-2: non-existing PV (no successful CA search so far),

-3: invalid type argument,

-4: invalid PV argument,

-10: currently no connection.

2.17.4 Examples

```
lcaNewMonitorValue('PV')
```

2.18 lcaDelay

2.18.1 Calling Syntax

```
lcaDelay(timeout)
```

2.18.2 Description

Delay execution of *scilab* or *matlab* for the specified time to handle channel access activity (monitors). *Using this call is not needed under EPICS-3.14* since monitors are transparently handled by separate threads. These "worker threads" receive data from CA on monitored channels "in the background" while *scilab*/*matlab* are processing arbitrary calculations. You only need to poll the library for the data being ready using the `lcaNewMonitorValue()` (see 2.17)) routine.

Under 3.13 however, the user must periodically call `lcaDelay()` or another *labCA* routine in order to yield the CPU to the CA library.

2.18.3 Parameters

timeout A timeout value in seconds.

3 Building and Using *labCA*

3.1 Build

labCA comes with a '`configure`' subdirectory and Makefiles conforming to the EPICS build system. Following a configuration step which involves editing two small files, '`make`' is executed to install the generated libraries and scripts.

Note that EPICS 3.13 is being deprecated. This is the last version of *labCA* supporting the old EPICS build system. In order to use it, you must manually replace all `Makefiles` by the respective `Makefile.R13`s and you have to edit the `CONFIG` and `RELEASE` files in the `config` rather than the `configure` directory.

Prior to invoking the *scilab* or *matlab* application, the system must be properly set up in order for the applications to locate the *labCA* and channel access libraries.

3.1.1 Prerequisites

labCA needs an EPICS BASE installation that was built *with shared libraries*. The main reason being that *matlab*'s *mex* files cannot have multiple entry points. Hence, when statically linking multiple *mex* files against *ezca, ca, Com* etc. multiple copies of those libraries would end up in the running *matlab* application with possible adverse effects. It should be possible to build and use the *scilab* interface with static libraries — minor tweaks to the Makefiles might be necessary.

labCA has been tested with *matlab*-6.5, *matlab*-7.0 and *scilab*-2.7 .. *scilab*-4.0 under a variety of EPICS releases ranging from 3.13.2 to 3.14.8.2 on *linux-x86*, *linux-ppc*, *solaris-sparc-gnu* and *win32*.

Note that the binary distribution of *labCA* usually ships with the necessary EPICS base libraries so there is no need to download anything besides *labCA*. The *linux* and *solaris* versions have these libraries built-in, the *win32*version comes with the appropriate DLLs.

3.1.2 Configuration

Two files, '`configure/CONFIG`' and '`configure/RELEASE`' need to be adapted to the user's installation:

`CONFIG`: A handful of configuration parameters must be defined in this file.

`MAKEFOR`: Setting the `MAKEFOR` variable determines the target application program the interface library is built for. Valid settings are `MAKEFOR=SCILAB` or `MAKEFOR=MATLAB`. Any setting other than `MATLAB` is treated like `SCILAB`.

`CONFIG_USE_CTRLC`: Set this to `YES` or `NO` to enable or disable, respectively, code for handling "Ctrl-C" keystroke sequences. When enabled, *labCA* operations (*except for* `lcaDelay`) may be aborted by hitting "Ctrl-C". Note that *labCA* polls for an "abort condition" with a granularity of the timeout parameter (see 2.12). Unfortunately, neither *matlab* nor *scilab* feature a documented API for handling Ctrl-C events and therefore Ctrl-C support – the implementation using undocumented features of *labCA* and *matlab*– must be considered "experimental" i.e., it might cause problems on certain operating system and/or *scilab*/*matlab* versions.

`INSTALL_LOCATION`: Set this variable to install in a location different from the *labCA* top directory.

`RELEASE`: In this file, paths to the EPICS base ('`EPICS_BASE`' variable) and *scilab* ('`SCILABDIR`' variable) or *matlab* ('`MATLABDIR`' variable) installations must be specified.

Under *win32*, an additional variable '`MATLIB_SUBDIR`' must be set directing the build process to select the correct `libmx.lib` and `libmex.lib` library variants. The setting of this variable is compiler dependent.

`MATLABDIR=<path>` `<path>` defining the *matlab* installation directory where the '`extern`' subdirectory can be found (e.g. `/opt/matlabR14beta2`).

`SCILABDIR=<path>` `<path>` defining the *scilab* installation directory where the '`routines`' subdirectory can be found (e.g. `/usr/lib/scilab-2.7`).

`MATLIB_SUBDIR=<pathelem>` `<pathelem>` chosing the subdirectory corresponding to the C-compiler that is used for the build. (e.g. `win32/microsoft/msvc60` for the microsoft visual c++ 6.0 compiler).

Any irrelevant variables (such as `MATLABDIR` if `MAKEFOR=SCILAB`) are ignored.

Note that the EPICS build system has problems with path names containing white space as they are commonly used on *win32*. Although I have tried to work around this, you still might encounter problems. I found that setting the environment variable `MATLAB` to point to the *matlab* directory helped (*cygwin*). It is best to avoid white space in path names, however.

3.1.3 Building *labCA*

After setting the '`EPICS_HOST_ARCH`' (for EPICS 3.14) and '`HOST_ARCH`' (EPICS 3.13) environment variables, *GNU* make is invoked in the *labCA* top directory. Note that the compiler toolchain must be found in the system `PATH`[6].

3.2 Using *labCA*

labCA consists of a set of shared libraries which in turn reference other shared libraries (most notably the channel access client libraries from EPICS BASE). It is of *crucial importance* that the operating system locates the correct versions at *run-time* (i.e. the same versions *labCA* was *linked* against). Otherwise, the run-time linker/loader could fail to load the required objects — leaving the user (expecially in *matlab*) at the prompt with obscure error messages.

Under *linux* or *solaris*, the `LD_LIBRARY_PATH` environment variable or the `ld.so.conf / ldconfig` facility are used to point to the executable shared libraries (located in `lib/<arch>`). If using a binary distribution, the `PATH` variable also should point to `bin/<arch>` so that the *CA repeater* executable is found. If you build from source using your own EPICS base installation then we assume that locating the *CA repeater* has already been taken care of.

Under *win32*, the `PATH` environment variable must point to the correct EPICS BASE and *labCA* DLLs (located in `bin/<arch>`).

Note that the paths to the correct EPICS BASE and *labCA* shared libraries must be set up *prior* to starting the *scilab* or *matlab* application. It is usually not possible to change the system search path from within an application.

Possible problems could occur because

- third party setup scripts modify `LD_LIBRARY_PATH / PATH`.
- your EPICS BASE was not built with shared libraries but shared libraries of a different release are found somewhere.
- a "innocent-looking" directory present in `LD_LIBRARY_PATH` before EPICS BASE actually contains shared libraries of an older release.
- Note that the system search path is *not* identical with the *matlab* path — changing the *matlab* path (from withing *matlab*) has no influence on the system search path.

3.2.1 Using *labCA* with *scilab*

Set up the shared library search path (as described above) and start *scilab*. Run the `labCA.sce` script which was generated by the build

[6] Under *win32*, the *msvc* compiler features a `.BAT` file for setting up the necessary environment.

process (<*labCA*-top>/bin/<arch>/*labCA*.sce) to load the *labCA* interface. The script also adds to the %helps variable making on-line help available.

The script can be installed at any convenient location — the lines setting up the %helps path need to be adapted in this case.

It is also possible to permanently link *labCA* to *scilab*. Consult the *scilab* documentation for more information.

3.2.2 Using *labCA* with *matlab*

Every entry point / lcaXXX routine is contained in a separate shared object (AKA "mex-") file in the <*labCA*-top>/bin/<arch>/labca[7] directory which must be added to the *matlab* search path. Note that this is in *addition* to setting the system library search path which must be performed prior to starting *matlab* (see previous section). All necessary objects and libraries are transparently loaded simply by invoking any of the entry point routines. Note that on-line help files are also installed to be located automatically.

[7] the extra subdirectory was added in order for the *matlab* help command to easily locate the Contents.m file when the user types help labca.