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* 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.

As of *labCA* version 3 further improvements to *ezca* have been made that exploit features of the multi-threaded CA library (EPICS 3.14 only) in order to speed up response time. Earlier versions always handed control to *ezca* in multiples of the *labCA* timeout, i.e., even if data from a channel were available quicker the library would wait until the next timeout quantum expired. Since version 3 a *labCA* call returns immediately after the underlying request completes.

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\(^1\) 3.14, *scilab*-2.7 .. *scilab*-4.1, *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

\(^1\)throughout this text, references to *scilab* usually mean *scilab* or *matlab*.

\(^2\)Support for 3.13 has been dropped as of *labCA* version 3
version of \textit{labCA}. Modifications to the \texttt{makefiles} might be necessary to build older versions.
2  Supported EZCA Calls

labCA implements an interface to almost all public ezca routines\footnote{the matlab implementation may still lack some of the more esoteric commands}. 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\footnote{throughout this text, references to scilab usually mean scilab or matlab.} 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:

\[
pvs = \begin{bmatrix} 'PVa'; 'b'; 'anotherone' \end{bmatrix}
\]

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:

\[
lcaPut('trigger', 1) \\n\text{data}=lcaGet(['sensor1'; 'sens2']);
\]

- It is guaranteed that writing the “trigger” completes (on the CA server) prior to reading the sensors.\footnote{In matlab, the square brackets (\['\]) must be replaced by curly braces (\{"\}).}
- 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 member) in the real and imaginary parts, respectively. This makes it easy to extract the seconds while still maintaining full accuracy.

\footnote{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 Error Handling

All errors encountered during execution of labCA calls result in the call being aborted, a message being printed to the console and an error status being recorded which can be retrieved using scilab’s lasterror command. The recommended method for handling errors is scilab's try -- catch -- end construct:

```scilab
try
val = lcaGet(pvvector)
catch
err   = lasterror()
// additional information is provided
// by the 'lcaLastError' routine
stats = lcaLastError()
// handle error here
end
```

Many labCA calls can handle multiple PVs at once and underlying CA operations may fail for a subset of PVs only. However, lasterror only supports reporting a single error status. Therefore the lcaLastError (see 2.11) routine was implemented: if a labCA call fails for a subset of PVs then a subsequent call to lcaLastError returns a column vector of status values for the individual PVs. The error codes are shown in Table 1.

---

As of version 3; earlier versions didn’t consistently “throw” all errors so that they could be caught by the try-catch-end mechanism but would merely print messages when encountering some minor errors. Also, versions earlier than 3 would not report error IDs/messages to lasterror nor implement the lcaLastError (see 2.11) routine.
<table>
<thead>
<tr>
<th>#</th>
<th>Matlab Error ID</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>labca:unexpectedOK</td>
<td>No error</td>
</tr>
<tr>
<td>1</td>
<td>labca:invalidArg</td>
<td>Invalid argument</td>
</tr>
<tr>
<td>2</td>
<td>labca:noMemory</td>
<td>Not enough memory</td>
</tr>
<tr>
<td>3</td>
<td>labca:channelAccessFail</td>
<td>Underlying CA operation failed</td>
</tr>
<tr>
<td>4</td>
<td>labca:udfCaReq</td>
<td>Item(s) requested undefined for its/their native data type</td>
</tr>
<tr>
<td>5</td>
<td>labca:notConnected</td>
<td>Channel currently disconnected</td>
</tr>
<tr>
<td>6</td>
<td>labca:timedOut</td>
<td>No response in time</td>
</tr>
<tr>
<td>7</td>
<td>labca:inGroup</td>
<td>Currently in a EZCA group</td>
</tr>
<tr>
<td>8</td>
<td>labca:notInGroup</td>
<td>Currently not in a EZCA group</td>
</tr>
<tr>
<td>9</td>
<td>labca:usrAbort</td>
<td>EZCA call aborted by user (Ctrl-C)</td>
</tr>
<tr>
<td>20</td>
<td>labca:noMonitor</td>
<td>No monitor for PV/type found</td>
</tr>
<tr>
<td>21</td>
<td>labca:noChannel</td>
<td>No channel for PV name found</td>
</tr>
</tbody>
</table>

Table 1: labCA error codes. Numerical codes (scilab lasterror and lcaLastError()) and corresponding matlab error “ID”s (as returned by matlab lasterror).

2.3 lcaGet

2.3.1 Calling Syntax

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

2.3.2 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.

2.3.3 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 explicitly *(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 $\approx 16\text{kB}$. 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 explicitly 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 accommodate the PV with the most elements. If the $n_{\text{max}}$ argument is given and is nonzero but less than the automatically determined $n$, then $n$ is clipped to $n_{\text{max}}$. 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. In addition, warning messages are printed to the console if a PV’s alarm status exceeds a configurable threshold *(see 2.22)*. 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.

---

8 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.
2.3.4 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

2.4 lcaPut

2.4.1 Calling Syntax

lcaPut(pvs, value, type)

2.4.2 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.

2.4.3 Parameters

pvs Column vector (in matlab: m x 1 cell- matrix) of m strings.
value m x n matrix or 1 x 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 $\approx 16$kB. 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 explicitly request different types for individual PVs (i.e. `type` cannot be a vector).

### 2.4.4 Examples

```matlab
// 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 )
```
2.5 **lcaPutNoWait**

2.5.1 **Calling Syntax**

\[
\text{lcaPutNoWait}(\text{pvs}, \text{value}, \text{type})
\]

2.5.2 **Description**

\text{lcaPutNoWait} is a variant of \text{lcaPut} that does \textit{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.

2.5.3 **Parameters**

See \text{lcaPut}.

---

2.6 **lcaGetNelem**

2.6.1 **Calling Syntax**

\[
\text{numberOfElements} = \text{lcaGetNelem}(\text{pvs})
\]

2.6.2 **Description**

Retrieves the element count of a number of PVs. Note that this is not necessarily the number of \textit{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.

2.6.3 **Parameters**

\textbf{pvs} Column vector (in matlab: \(m \times 1\) cell- matrix) of \(m\) strings.

\textbf{numberOfElements} \(m \times 1\) column vector of the PV's number of elements ("array dimension").
2.7 lcaSetMonitor

2.7.1 Calling Syntax

\texttt{lcaSetMonitor(pvs, nmax, type)}

2.7.2 Description

Set a “monitor” on a set of PVs. Monitored PVs are automatically re-
trieved every time their value or status changes. 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 \texttt{labCA} e.g., to \texttt{lcaDelay} (see
2.23), in order to allow \texttt{labCA} to handle monitors.

Use the \texttt{lcaNewMonitorValue} (see 2.9) call to check monitor status
(local flag) or \texttt{lcaNewMonitorWait} (see 2.8) to wait for new data to
become available (since last \texttt{lcaGet} or \texttt{lcaSetMonitor}). If new data
are available, they are retrieved using the ordinary \texttt{lcaGet} (see 2.3)
call.

Note the difference between polling and monitoring a PV in combi-
nation with polling the local monitor status flag (\texttt{lcaNewMonitorValue}
see 2.9). 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, \texttt{lcaGet} reads from a local buffer
rather than from the network. It is most convenient however to wait
for monitored data to arrive using \texttt{lcaNewMonitorWait} (see 2.8) rather
than polling.

There is currently no possibility to selectively remove a monitor. Use
the \texttt{lcaClear} (see 2.10) 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).

2.7.3 Parameters

\textbf{pvs} Column vector (in matlab: \texttt{m x 1 cell} - matrix) of \texttt{m} strings.

\textbf{nmax} \textit{(optional argument)} Maximum number of elements (per PV) to
monitor/retrieve. If set to 0 (default), all elements are fetched. See 2.3.3 for more information.

Note that a subsequent \texttt{lcaGet} (see 2.3) must specify a \texttt{nmax} argument equal or less than the number given to \texttt{lcaSetMonitor}
— otherwise the \texttt{lcaGet} operation results in fetching a new set
of data from the server because the \texttt{lcaGet} request cannot be
satisfied using the copy locally cached by the monitor-thread.
**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 [2.3.3](#) 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.

### 2.7.4 Examples

```cpp
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')
```

### 2.8 `lcaNewMonitorWait`

#### 2.8.1 Calling Syntax

`lcaNewMonitorValue(pvs, type)`

#### 2.8.2 Description

Similar to `lcaNewMonitorValue` (see 2.9) but instead of returning the status of monitored PVs this routine blocks until all PVs have fresh data 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 2.3).

#### 2.8.3 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 [2.3.3](#) for more information.

Note that monitors are specific to a particular data type and therefore `lcaNewMonitorWait` will only report the status for a monitor that had been established (by `lcaSetMonitor` (see 2.7)) with a matching type. Using the “native” type, which is the default, for both calls satisfies this condition.
2.8.4 Examples

```matlab
try lcaNewMonitorWait(pvs)
vals = lcaGet(pvs)
catch
errs = lcaLastError()
handleErrors(errs)
end
```

2.9 lcaNewMonitorValue

2.9.1 Calling Syntax

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

2.9.2 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 2.3).

2.9.3 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 (2.3.3) 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` (see 2.7)) 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 (the data, that is, *not the flag*). A value of one indicates that new data are available for reading (`lcaGet`).

*NOTE: As of labCA version 3 the flags no longer report error conditions. Errors are now reported in the standard way (see 2.2), i.e., by aborting the labCA call. Errors can be caught by the standard scilab try-catch-end construct. The `lcaLastError` (see 2.11) routine can be used to obtain status information for individual channels if `lcaNewMonitorValue` fails on a vector of PVs. See also `lcaNewMonitorWait` (see 2.8).*
2.9.4 Examples

```matlab
try and(lcaNewMonitorValue(pvvec))
vals = lcaGet(pvvec)
catch
    errs = lcaLastError()
handleErrs(errs)
end
```

2.10 lcaClear

2.10.1 Calling Syntax

```matlab
lcaClear(pvs)
```

2.10.2 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.

2.10.3 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).

2.10.4 Examples

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

2.11.1 Calling Syntax

[err_status] = lcaLastError()

2.11.2 Description

This routine is a simple extension to scilab’s lasterror which only allows a single error to be reported. If labCA encounters an error of general nature then lasterror is sufficient and lcaLastError() reports redundant/identical information. However, if a labCA operation only fails on a subset of a vector of PVs then lcaLastError() returns an error code for each individual PV (as a \( m \times 1 \) vector) so that failing channels can be identified.

The error reported by lasterror corresponds to the first error found in the err_status vector.

Note that (matching lasterror’s semantics) the recorded errors are not cleared by a successful labCA operation. Hence, the status returned by lcaLastError() is only defined after an error occurred and the routine is intended to be used from the catch section of a try -- catch -- end construct.

2.11.3 Parameters

err_status \( m \times 1 \) column vector of (see 2.2) for each PV of the last failing labCA call or a scalar. Note that this routine can return a scalar even if the last operation involved multiple PVs if the error was of general nature (e.g., “invalid argument”). In this case the scalar is identical to the error reported by scilab’s lasterror.

2.11.4 Examples

try
  // lcaXXX command goes here
catch
  errors = lcaLastError()
  // errors holds status vector or single status code
  // depending on command, error cause and number of PVs.
end
2.12 lcaGetControlLimits

2.12.1 Calling Syntax

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

2.12.2 Description

Retrieve the control limits associated with a number of PVs.

2.12.3 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.

2.13 lcaGetGraphicLimits

2.13.1 Calling Syntax

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

2.13.2 Description

Retrieve the graphic limits associated with a number of PVs.

2.13.3 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.
2.14 lcaGetWarnLimits

2.14.1 Calling Syntax

\[\text{[lowLimit, hiLimit]} = \text{lcaGetWarnLimits}(\text{pvs})\]

2.14.2 Description

Retrieve the warning limits associated with a number of PVs.

2.14.3 Parameters

\textit{pvs} Column vector (in matlab: \(m \times 1\) \textit{cell-} matrix) of \(m\) strings.
\textit{lowLimit} \(m \times 1\) column vector of the PV’s low warning limit.
\textit{hiLimit} \(m \times 1\) column vector of the PV’s high warning limit.

2.15 lcaGetAlarmLimits

2.15.1 Calling Syntax

\[\text{[lowLimit, hiLimit]} = \text{lcaGetAlarmLimits}(\text{pvs})\]

2.15.2 Description

Retrieve the alarm limits associated with a number of PVs.

2.15.3 Parameters

\textit{pvs} Column vector (in matlab: \(m \times 1\) \textit{cell-} matrix) of \(m\) strings.
\textit{lowLimit} \(m \times 1\) column vector of the PV’s low alarm limit.
\textit{hiLimit} \(m \times 1\) column vector of the PV’s high alarm limit.
2.16 lcaGetStatus

2.16.1 Calling Syntax

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

2.16.2 Description

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

2.16.3 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).

2.17 lcaGetPrecision

2.17.1 Calling Syntax

precision = lcaGetPrecision(pvs)

2.17.2 Description

Retrieve the precision of a number of PVs.

2.17.3 Parameters

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

precision $m \times 1$ column vector of the PV’s precision.
2.18 lcaGetUnits

2.18.1 Calling Syntax

\[ \text{units} = \text{lcaGetUnits(pvs)} \]

2.18.2 Description

Retrieve the engineering units of a number of PVs.

2.18.3 Parameters

\textbf{pvs} Column vector (in \textit{matlab}: \( m \times 1 \) \textit{cell}-matrix) of \( m \) strings.

\textbf{units} \( m \times 1 \) column vector (on \textit{matlab}: \textit{cell}-matrix) of strings holding the PV EGUs.

2.19 lcaGetRetryCount, lcaSetRetryCount

2.19.1 Calling Syntax

\[ \text{currentRetryCount} = \text{lcaGetRetryCount()} \]
\[ \text{lcaSetRetryCount(newRetryCount)} \]

2.19.2 Description

Retrieve / set the \textit{ezca} library retryCount parameter (consult the \textit{ezca} documentation for more information). The retry count multiplied by the timeout parameter (see 2.20) determines the maximum time the interface waits for connections and data transfers, respectively.
2.20 lcaGetTimeout, lcaSetTimeout

2.20.1 Calling Syntax

currentTimeout = lcaGetTimeout()
lcaSetTimeout(newTimeout)

2.20.2 Description
Retrieve / set the ezca library timeout parameter (consult the ezca documentation for more information). Note that the semantics of the timeout parameter has changed with labCA version 3. The library no longer pends for CA activity in multiples of this timeout value but returns control to scilab as soon as the underlying CA request completes.

However, labCA checks for “Ctrl-C” key events every time (and only when) the timeout expires. Hence, it is convenient to choose a value < 1s.

The maximal time spent waiting for connections and/or data equals the timeout multiplied by the retry count (see 2.19).

2.21 lcaDebugOn, lcaDebugOff

2.21.1 Calling Syntax

lcaDebugOn()
lcaDebugOff()

2.21.2 Description
Switch the ezca library's debugging facility on and off, respectively.
2.22 lcaSetSeverityWarnLevel

2.22.1 Calling Syntax

lcaSetSeverityWarnLevel(newlevel)

2.22.2 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 $\geq 10$ 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.

2.23 lcaDelay

2.23.1 Calling Syntax

lcaDelay(timeout)

2.23.2 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 either poll the library for the data being ready using the lcaNewMonitorValue() (see 2.9) routine or block for data becoming available using lcaNewMonitorWait (see 2.8).

2.23.3 Parameters

**timeout** A timeout value in seconds.
3 Building and Using labCA

3.1 Build

NOTE: If the binaries distributed with labCA work for you then there is no need to build anything. If you want/need to build your own version then read on otherwise proceed to Subsection 3.2.

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.

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 enabled. 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-4.1 under a variety of EPICS releases up to 3.14.9 on linux-x86, linux-ppc, solaris-sparc-gnu, linux-x86_64, solaris-sparc, solaris-sparc64 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.

9 To be precise: only labCA needs to be a shared library (but must then have EPICS BASE linked in); we recommend an EPICS BASE installation with shared libraries enabled because modifications to the EPICS Makefiles are required to build a shared labCA library that is linked against static versions of EPICS BASE.

10 Note that not all possible combinations have been tested with the latest labCA release but rather the latest versions of the respective components on the platforms that are in the distribution.
### 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 `icaDelay`) may be aborted by hitting “Ctrl-C”. Note that `labCA` polls for an “abort condition” with a granularity of the timeout parameter (see [2.20]). 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 `scilab` 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. 
  
  **NOTE**: This method has been deprecated. Use `INSTALL_LOCATION_APP` in the `RELEASE` file instead.

**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.

- `INSTALL_LOCATION_APP=<path> <path>` defining the install location of the `labCA` package. If unset, the `labCA` top directory will be used.

- `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`).
MATLAB_SUBDIR=<pathelem> <pathelem> choosing 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). The libmex.lib and libmx.lib files for the applicable compiler are found there.

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' environment variable, GNU make is invoked from the labCA top directory. Note that the compiler toolchain must be found in the system PATH.

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

11 Under win32, the msvc compiler features a .BAT file for setting up the necessary environment.
• 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 within `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 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` 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.

---

12 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`. 24