EZCA Primer

Nicholas T. Karonis
Argonne National Laboratory
Advanced Photon Source
Accelerator Systems Division
Controls Group

1 Introduction

This document provides a quick introduction to EZCA, a library that was designed to provide an easy to use interface to Channel Access (CA). As such, this document is *not* a user's manual, where a more detailed explanation of EZCA can be found. In short, this document is designed to get users to a state where they can be writing EZCA code as quickly as possible. It is not a document that answers all EZCA questions.

2 Getting Started

Below is an example of a program, *usr1.c*, that uses EZCA.

usr1.c

```
/* following needed when using EZCA */
#include <tsDefs.h>
#include <cadef.h>
#include <ezca.h>

main()
{
double d;
short s[2];

    s[0] = 10; s[1] = -21;
    ezcaPut("mywaveform", ezcaShort, 2, s);

    ezcaGet("mygenerator", ezcaDouble, 1, &d);
} /* end main() */
```

First note the three include files *necessary for all EZCA programs*. The first two, **tsDefs.h** and **cadef.h**, are EPICS include files. The last, **ezca.h**, is a new EPICS include file. The files must appear in the same order as they appear above.

Our first call is to **ezcaPut**. The first argument specifies the process variable we wish to write to, **mywaveform**. The next two arguments state that we are supplying the data as C short variables and that there are 2 of them. The last argument is a pointer to the data.

Our last call is to **ezcaGet**. We are retrieving a scalar from the process variable **mygenerator** in the form of a C double and are placing it into the variable **d**.

2.1 Data Types

EZCA allows the user to read/write information in a number of different formats. In the example above we wrote data supplied as **ezcaShort** and requested the data in our read as an **ezcaDouble**. Here are the allowable EZCA data types.

- **ezcaByte**
- **ezcaString**
- **ezcaShort**
- **ezcaLong**
- **ezcaFloat**
- **ezcaDouble**

2.2 Makefile

Below is the *makefile* used to make the example program *usr1.c*.

makefile

```
CC = acc

EPICSDIR = /home/opiinj2/epics
EPICS_ADD_ON_DIR = $(EPICSDIR)/add_on

EZCAINCDIR = -I$(EPICS_ADD_ON_DIR)/include
EPICSINCDIR = -I$(EPICSDIR)/R3.11.6/share/epicsH
INCLUDEDIRS = $(EZCAINCDIR) $(EPICSINCDIR)

EZCALIBDIR = $(EPICS_ADD_ON)/lib
EZCALIB = -lezca

CALIBDIR = $(EPICSDIR)/R3.11.6/Unix/sun4/bin
CALIBS = -lca -lCom -lUnix

LIBDIRS = -L$(CALIBDIR) -L$(EZCALIBDIR)
LIBS = $(EZCALIB) $(CALIBS)

CFLAGS = -c
DEFINES = -DUNIX -USUN4 -DOLDCA

usr1: usr1.o
        $(CC) -o usr1 usr1.o $(LIBDIRS) $(LIBS)

clean:
        /bin/rm -f *.o usr1

.c.o:$*.c
        $(CC) $(CFLAGS) $(INCLUDEDIRS) $(DEFINES) $*.c
```

3 Main Functions

EZCA is a library of composed of 25 or so functions. Some of the functions deal with error handling, grouping, monitors, and tuning EZCA. All of the functions are described in this document.

In this section we introduce the main work functions in EZCA, the functions that read/write data. There are 8 functions that read data all starting with **ezcaGet** and one function to write data, **ezcaPut**.

- **int ezcaGet(char *pvname, char ezcatype, int nelem, void *data_buff)**

This is the fundamental retrieval function in EZCA. The process variable is named in **pvname** and the request type is specified as an EZCA data type in **ezcatype**. The requested number of elements is specified in **nelem**. **data_buff** is a user-supplied buffer that must be large enough to store **nelem** data values of type **ezcatype**. **ezcaGet** places the value into that buffer.

- **int ezcaGetControlLimits(char *pvname, double *low, double *high)**
- **int ezcaGetGraphicLimits(char *pvname, double *low, double *high)**
- **int ezcaGetNelem(char *pvname, int *nelem)**
- **int ezcaGetPrecision(char *pvname, short *precision)**
- **int ezcaGetStatus(char *pvname, TS_STAMP *timestamp, short *status, short *severity)**
- **int ezcaGetUnits(char *pvname, char *units)**

These functions all retrieve information *about* EPICS database records rather than their values. The process variable is named in **pvname**. All other fields are user-supplied buffers that EZCA fills.

TS_STAMP is an EPICS type (found in *tsDefs.h*). It reflects the last time the record was processed. There is an EPICS library to manipulate this data type.

In **ezcaGetUnits** the character array **units** must be at least **EZCA_UNITS_SIZE** (defined in *ezca.h*) big.

- **int ezcaGetWithStatus(char *pvname, char ezcatype, int nelem, void *data_buff, TS_STAMP *timestamp, short *status, short *severity)**

This is nothing more than an **ezcaGet** and an **ezcaGetStatus** wrapped up into one function.

- **int ezcaPut(char *pvname, char ezcatype, int nelem, void *data_buff)**

This is the write function in EZCA. The process variable is named in **pvname**. The type and amount of supplied data are specified in **ezcatype** and **nelem**, respectively. The data is in the user-supplied buffer **data_buff**, which is immediately ready for re-use upon return of the function.

4 Error Handling

4.1 Return Codes

EZCA functions return status codes indicating the success/failure of the call. In EZCA a return of 0 (**EZCA_OK**) always means success. Anything else indicates a problem. Following are the return codes EZCA uses.

- **EZCA_OK**
- **EZCA_INVALIDARG**
- **EZCA_FAILEDMALLOC**
- **EZCA_CAFAILURE**
- **EZCA_UDFREQ**
- **EZCA_NOTCONNECTED**
- **EZCA_NOTIMELYRESPONSE**
- **EZCA_INGROUP**
- **EZCA_NOTINGROUP**

4.2 Automatic Error Reporting

Although the return codes are useful, the real information associated with anomalous return codes is found in the error messages. By default, EZCA prints error messages (to *stdout*) as soon as they are encountered. The user can toggle this automatic error reporting feature with the following functions.

- **void ezcaAutoErrorMessageOn()**
- **void ezcaAutoErrorMessageOff()**

The default state is on. The user may call these as often as he wishes and at any time throughout the program.

4.3 Requested Error Messages

For tighter control over error messages, EZCA provides two error handling mechanisms.

- **void ezcaPerror(char *prefix)**
- **int ezcaGetErrorString(char *prefix, char **buff)**

Both of these error reporting facilities report the status, including success, of the *last* EZCA call (except for at the end of groups discussed later). In both functions, **prefix** is a user-supplied character string (possibly **NULL**) that EZCA will use as a prefix to its error message.

ezcaPerror prints the optional prefix with the error message to *stdout*. **ezcaGetErrorString** allocates a buffer and fills it with the optional prefix and error message. It then returns the address of the buffer in the user-supplied pointer **buff**. It then becomes the user's responsibility to free the buffer using the following.

- **void ezcaFree(void *buff)**

Following is the program *usr2.c*, a modification of *usr1.c,* that uses these new error functions.

usr2.c

```
#include <stdio.h>

/* following needed when using EZCA */
#include <tsDefs.h>
#include <cadef.h>
#include <ezca.h>

main()
{

double d;
short s[2];
char *error_msg_buff;

    ezcaAutoErrorMessageOff();
```

```
    s[0] = 10; s[1] = -21;
    if (ezcaPut("mywaveform", ezcaShort, 2, s))
        ezcaPerror("Put Error:");

    if (ezcaGet("mygenerator", ezcaDouble, 1, &d))
    {
        ezcaGetErrorString(NULL, &error_msg_buff);
        printf("Get Error: %s\n", error_msg_buff);
        ezcaFree((void *) error_msg_buff);
    } /* endif */

} /* end main() */
```

The first thing to notice is the new include file *stdio.h*. It was brought in because we used the constant **NULL** in our call to **ezcaGetErrorString**.

We introduced a new variable, **error_msg_buff**, where we plan to have **ezcaGetErrorString** place the address of the buffer it allocates.

Our first call is to **ezcaAutoErrorMessageOff**. This turns off the automatic error message reporting that is the default in EZCA. We could have left it on. Doing so in this program would have resulted in errors being reported twice, once automatically and once with our explicit requests.

Our call to **ezcaPut** is the same. This time we test the return status. Recall a return status of 0 is always an indication of success. Anything else indicates a problem. Here we use **ezcaPerror** to report the problem with the write using the prefix **"Put Error:"**. If something went wrong with the write, the error message would be written to *stdout* with our specified prefix.

Our call to **ezcaGet** is also the same. Here we used **ezcaGetErrorString** without specifying the optional prefix to have the error message placed into an allocated buffer and the address of the buffer placed into our variable **error_msg_buff**. We print the error with our own prefix message specified in the print statement rather than the EZCA function and then free the allocated string using **ezcaFree**.

Assuming that both process variables cannot be found on any IOCs, executing *usr2* would look like this.

```
% usr2
Put Error: ezcaPut(): channel not currently connected
Get Error: ezcaGet(): channel not currently connected
%
```

5 Groups

When doing a large block of unconditional reads and/or writes, it is more efficient to do them in a group rather than individually. Groups are delineated using the following two EZCA functions.

- **int ezcaStartGroup()**
- **int ezcaEndGroup() or int ezcaEndGroupWithReport(int **rcs, int *nrcs)**

Making an EZCA call in the context of an EZCA group merely checks the validity of the arguments. The actual work is postponed until the end of the group is encountered.

Not all of the EZCA functions respect the context of a group. Only those functions mentioned in section 3 have their work postponed. All other EZCA functions are always performed immediately.

Consider the following program *usr3.c* which is another modification of *usr1.c*.

usr3.c

```
#include <stdio.h>
/* following needed when using EZCA */
#include <tsDefs.h>
#include <cadef.h>
#include <ezca.h>

main()
{

double d;
short s[2];
char *error_msg_buff;

    ezcaAutoErrorMessageOff();

    ezcaStartGroup();

    s[0] = 10; s[1] = -21;
    ezcaPut("mywaveform", ezcaShort, 2, s);

    ezcaGet("mygenerator", ezcaDouble, 1, &d);

    if (ezcaEndGroup())
        ezcaPerror(NULL);

} /* end main() */
```

Note that we have removed the error reporting functions found in *usr2.c* from **ezcaPut** and **ezcaGet**, although we were not required to do so. We have also placed both of these functions in a group by surrounding them with **ezcaStartGroup** and **ezcaEndGroup**.

By placing these functions into a group we have postponed the work until **ezcaEndGroup** is encountered. Each function simply checks the validity of the arguments and places the work onto a list to be processed later. **ezcaEndGroup** performs all the batched work. It returns the *first encountered* non-successful return code (based on their order of appearance) in the group. If all the batched work returned successfully, **ezcaEndGroup** returns **EZCA_OK** which is 0.

ezcaPerror behaves a little bit differently here. When called after **ezcaEndGroup** (and before a call to any other EZCA function) it prints a status line *for every function in the group*. This includes those functions with successful return codes.

Under the same assumption as before, that none of the process variables can be found, executing *usr3* would look like this.

```
% usr3
ezcaPut(): channel not currently connected
ezcaGet(): channel not currently connected
%
```

Assuming that all the arguments to all the functions in the group are valid, there is no difference in placing **ezcaStartGroup** and **ezcaEndGroup** around the EZCA functions as they appear in *usr1.c* or in *usr2.c*. In *usr2.c* the return code from each EZCA function would have indicated success (assuming all the arguments are valid).

If some of the arguments were invalid, then surrounding the EZCA calls with **ezcaStartGroup** and **ezcaEndGroup** in *usr1.c* and *usr2.c* would simply produce different output. In *usr2.c* the invalid argument error message would appear twice, once immediately after the function call and once as a result of the **ezcaPerror** at the end of the group.

Users interested in the return status of all the *all* the EZCA calls in a group should use **ezcaEndGroupWithReport** instead of **ezcaEndGroup**.

Like **ezcaEndGroup**, **ezcaEndGroupWithReport** performs all the batched work and returns the first encountered non-successful return code or **EZCA_OK**. Additionally, **ezcaEndGroupWithReport** allocates a vector of return codes, one element for each member of the group, and returns that vector as well as its length to the user. It then becomes the user's responsibility to free the acquired vector using **ezcaFree**.

Consider the following program *usr4.c*, a modification of *usr3.c* where we replaced **ezcaEndGroup** with **ezcaEndGroupWithReport**.

usr4.c

```
#include <stdio.h>
/* following needed when using EZCA */
#include <tsDefs.h>
#include <cadef.h>
#include <ezca.h>

main()
{

double d;
short s[2];
char *error_msg_buff;
int i, *rcs, nrcs;

    ezcaAutoErrorMessageOff();

    ezcaStartGroup();

    s[0] = 10; s[1] = -21;
    ezcaPut("mywaveform", ezcaShort, 2, s);

    ezcaGet("mygenerator", ezcaDouble, 1, &d);

    ezcaEndGroupWithReport(&rcs, &nrcs);

    for (i = 0; i < nrcs; i ++)
        if (rcs[i] != EZCA_OK)
            printf("Call %d had abnormal return status %d\n", i, rcs[i]);

    ezcaFree((void *) rcs);

} /* end main() */
```

5.1 Bad Grouping

Not all programs should use EZCA groups. The following is an example program that is a poor candidate for groups.

bad.c

```
#include <stdio.h>
/* following needed when using EZCA */
#include <tsDefs.h>
#include <cadef.h>
#include <ezca.h>

main()
{

double d;
short s[2];

    ezcaAutoErrorMessageOff();

    ezcaStartGroup();

    ezcaGet("mygenerator", ezcaDouble, 1, &d);
```

```
    if (d < 0)
    {
        s[0] = 10; s[1] = -21;
        ezcaPut("mywaveform", ezcaShort, 2, s);
    } /* endif */

    if (ezcaEndGroup())
        ezcaPerror(NULL);

} /* end main() */
```

Here we have changed the order of the read and write. We read first, and based on the value we read we *conditionally* write. This program is destined to fail. Recall that the **ezcaGet** read is postponed until **ezcaEndGroup** is executed. This means that the value found in the variable **d** is garbage when the test on it is performed.

6 Monitors

Another optimization (in addition to groups) available to users are monitors. If the user has a process variable whose value will not change very often but will be read frequently, then the user should establish an monitor on that process variable.

Monitors can be placed and removed at any time using the following.

- **int ezcaSetMonitor(char *pvname, char ezcatype)**
- **int ezcaClearMonitor(char *pvname, char ezcatype)**

There is no difference in the way a user reads the value, i.e.,all the **ezcaGet** family of functions are called exactly the same way. Calling **ezcaSetMonitor** simply instructs EZCA to immediately establish a CA monitor of the specified request type on the named process variable. Any time the value of the process variable changes (presumably infrequently) EZCA automatically and silently caches the new value. All subsequent reads of that process variable under that request type will not generate a CA read, but rather, will simply read the cached value. This arrangement continues until the monitor is removed using **ezcaClearMonitor**.

6.1 Monitor Check

The user can also poll the monitor to see if a new value has come in since the last time the value was read. This is done with the following function.

- **int ezcaNewMonitorValue(char *pvname, char ezcatype)**

This function returns a non-zero value if there is a new (unread) value in the monitor, otherwise it returns 0. This function is particularly useful when the read operation is expensive in time, e.g., reading large arrays.

6.2 Delay

Users must exercise caution when using monitors. Because of the way CA is implemented, it is possible to lose changes in process variables if there is a substantial amount of time between any two adjacent EZCA calls. Substantial here is a relative term. It depends on how frequently the values are likely to change.

To alleviate this problem, EZCA provides a function that should be called whenever using monitors and there is a substantial amount of time between any two adjacent EZCA calls. Between all such pair of calls, the user should call the following function.

- **int ezcaDelay(float sec)**

Where **sec** is always greater than 0, values around 0.01 should suffice.

7 Tuning EZCA

EZCA uses two tunable parameters to determine when to stop waiting for connections and confirmations of reads and writes. They are **timeout** and **retrycount**. EZCA uses them by waiting **timeout** seconds and then, if necessary, waiting **timeout** seconds a maximum of **retrycount** more times, resulting in a maximum total timeout time of **timeout+(timeout*retrycount) = timeout*(1+retrycount)**.

The default values for these parameters strikes a balance that hopefully serves most users efficiently. The hope is that those users that can connect quickly are served as well as those that require a little more time.

Users can not only discover the values of these parameters using

- **float ezcaGetTimeout()**
- **int ezcaGetRetryCount()**

but they can also set these parameters using

- **int ezcaSetTimeout(float sec)**
- **int ezcaSetRetryCount(int retry).**

Here all arguments must be greater than 0.

Empirically, we have observed that under normal circumstances EZCA can reliably process (read or write) 200 process variables per second. Users should adjust **timeout** and **retrycount** accordingly.

For example, with a **timeout** of 0.2 seconds and a **retrycount** of 9, EZCA will wait a maximum of 0.2*(1+9) = 2 seconds. This will allow the user to process groups of up to 400 gets and/or puts reliably. If a particular group has 600 operations to perform, the user must increase the maximum timeout to at least 3 seconds to more adequately assure reliable processing. This may be done by increasing **timeout** and/or **retrycount** accordingly.

8 Escape to Raw Channel Access

EZCA is designed to provide an easy to use interface to CA. In doing so, we were forced to sacrifice some of the functionality and efficiency found in CA. Users of EZCA that occasionally need to make a call directly to the CA library can do so in an EZCA program by calling the EZCA function that converts a process variable name to a CA chid.

- **int ezcaPvToChid(char *pvname, chid **cid)**

The **cid** should already be connected for you and ready to use with any CA function. Calls to CA functions may be mixed freely with EZCA function calls.

9 New Channel Access

EZCA is designed and implemented to use blocking writes. That is, to wait for acknowledgment that the value successfully got into the process variable and all processing that resulted from that write has successfully completed (**ca_array_put_callback**). This was only available in CA as of EPICS release 3.11.6.

Prior to release 3.11.6, writing in CA simply initiated the write and there was no way to automatically validate that the value got to the process variable or that all resulting processing was successfully completed (**ca_array_put**).

Users performing **ezcaPut** to process variables residing on IOCs booted with versions of EPICS *prior to* 3.11.6 will always get return codes indicating timeout errors for all such writes. The antiquated versions of EPICS never acknowledge the write, so EZCA is fooled into thinking that the write was not successful.

In order to eliminate these annoying error messages, we have provided a backward compatible version of EZCA, one that uses older **ca_array_put** instead of the newer **ca_array_put_callback**. To use this version the user simply adds one switch to the *compile* line of the makefile. No change is necessary to the user code. Below is a copy of the *makefile* presented in section 2.2. It already has the compile switch in place.

makefile

```
CC = acc

EPICSDIR = /home/opiinj2/epics
EPICS_ADD_ON_DIR = $(EPICSDIR)/add_on

EZCAINCDIR = -I$(EPICS_ADD_ON_DIR)/include
EPICSINCDIR = -I$(EPICSDIR)/R3.11.6/share/epicsH
INCLUDEDIRS = $(EZCAINCDIR) $(EPICSINCDIR)

EZCALIBDIR = $(EPICS_ADD_ON)/lib
EZCALIB = -lezca

CALIBDIR = $(EPICSDIR)/R3.11.6/Unix/sun4/bin
CALIBS = -lca -lCom -lUnix

LIBDIRS = -L$(CALIBDIR) -L$(EZCALIBDIR)
LIBS = $(EZCALIB) $(CALIBS)

CFLAGS = -c
DEFINES = -DUNIX -USUN4 -DOLDCA

usr1: usr1.o
        $(CC) -o usr1 usr1.o $(LIBDIRS) $(LIBS)

clean:
        /bin/rm -f *.o usr1

.c.o:$*.c
        $(CC) $(CFLAGS) $(INCLUDEDIRS) $(DEFINES) $*.c
```

This changes *all* writes to use the old CA write. It does not force the old version to acknowledge the write, it simply instructs EZCA to use the old CA write (**ca_array_put**) and not to wait for the acknowledgment.

To take advantage of the new CA write (**ca_array_put_callback**), simply remove **-DOLDCA** from the *makefile* and re-compile.