CaPython

Geoff Savage
24 August 2000

A Python interface to the EPICS channel access protocol.

Table of Contents

1 Prerequisites

This document assumes that the reader is familiar with Python and the EPICS channel access protocol (CA). CA information can be found in:

- "EPICS R3.12 Channel Access Reference Manual"
- "Channel Access Client Library Tutorial, R3.13"
- Channel access header files: cadef.h, db_access.h, and caerr.h.

To learn more about Python see the web site, www.python.org.

2 Introduction

CaPython is a Python extension module that provides a Python interface to the EPICS channel access (CA) communication protocol. It is a wrapper around the CA C library. The intent of the Python interface to CA was to match the functionality in the C interface as closely as possible. As such the following statements from the "Channel Access Reference Manual" still apply:

- "All requests which require interactions with a CA server are accumulated and not forwarded to the IOC until one of ca_flush_io(), ca_pend_io(), ca_pend_event(), or ca_sg_pend() are called allowing several operations to be efficiently sent over the network in one message."
- "Any process variable values written into your program's variables by ca_get() should not be referenced by your program until ECA_NORMAL has been received from ca_pend_io()."

The final goal of this project is to produce an object oriented interface to CA in Python. This is accomplished in the CaChannel class. The CaChannel class serves as the example of how to use CaPython.

This document covers:

- CaPython interface - how to use CaPython from Python.
- SWIG pointer library – technique for accessing C variables from Python.
- Internal design – covers the software design and building issues

The interface to the scan group CA functions is included. However I have never tested them nor is the documentation included. It should be evident from the documentation here how to use them.

3 CaPython Interface

This section covers how to use CaPython from a Python script. Covered under this topic:

- Data types – CA specific data types and argument mapping between the C and Python code.
- Functions – all the CaPython functions available to the user.
- Callbacks – how to write CA callback functions in Python
- Functions not implemented – explains which CA functions are not included and why.

3.1 Data Types

Matching the Python and C interfaces requires a mapping between data types in function argument lists and return values. The SWIG pointer library is used to allocate C variables and transfer the value to Python variables. For instructions on using the pointer library in your code see the section on the SWIG pointer library.

3.1.1 Channel Access Types

Using the special channel access types (chid and evid) requires helper functions. The CA types are really typedefs of pointers to structures:

```
typedef    struct channel_in_use    *chid;
typedef    struct pending_event     *evid;
```

3.1.1.1 new_chid

Synopsis

chid = ca.new_chid()

Description

Malloc space for a channel id structure.

Arguments

None

Comments

Initialized when a call to ca.search() completes successfully. Used in other caPython calls to identify the channel.. No routine is needed to delete a chid structure; A structure is deleted after a call to ca_clear_channel() under control of channel access.

Returns

chid (string): SWIG pointer to a channel id structure

3.1.1.2 new_evid

Synopsis

evid = ca.new_evid()

Description

Malloc space for an event structure.

Arguments

None

Comments

Initialized in ca_add_event(). Identifies an event channel in other channel access calls.

Returns

evid (string): SWIG pointer to an event id structure

3.1.1.3 free_evid

Synopsis

ca.free_evid(evid)

Description

Free space of an event structure.

Arguments

evid (string): SWIG pointer to an event id structure

Comments

Use after the event has been cleared by ca_clear_event() or the channel has been cleared by ca_clear_channel().

Returns

None

3.1.2 Status Returns

The status values returned from the Python function calls are Python integers whose values match those returned by the channel access C routines. The error codes are accessed by adding the module name (ca) to the front of the error code. For example: ca.ECA_NORMAL. Refer to the channel access reference manual for the list of status codes returned from each function.

3.1.3 Function Arguments

This is the mapping used by the Python API to convert types between C and Python. The left-hand column shows the C variable types and the right-hand column displays the corresponding types that the API expects. A type mismatch results in a Python type exception.

C Type	Python Type
char *	String
char	Integer
short	Integer
int	Integer
long	Integer
float	Float
double	Float

3.1.4 CA I/O Values

This is the mapping between types requested in channel access calls and the Python data types expected by the function. Specifying a value in the left column forces CA to return a Python value of the type found in the right hand column.

Database Request Type	Python Type
ca.DBR_STRING	String
ca.DBR_CHAR	Integer
ca.DBR_ENUM	Integer
ca.DBR_SHORT	Integer
ca.DBR_INT	Integer
ca.DBR_LONG	Integer
ca.DBR_FLOAT	Float
ca.DBR_DOUBLE	Float

Array of DBR type	List of Python type

When reading or writing multiple elements as one action the return type is a list containing items of the type requested.

3.2 Python Functions

The Python functions are listed in the order that the corresponding C functions are found in the channel access reference manual. Descriptions are provided in the channel access reference manual.

3.2.1 task_initialize

Synopsis

status = ca.task_initialize()

Arguments

None

Comments

Called from the module's initialization function (initca()) when the ca module is imported into Python. The user does not need to explicitly call this function. It is executed when the module is imported.

3.2.2 task_exit

Synopsis

ca.task_exit()

Arguments

None

Comments

Called automatically when python exits. Users do not need to explicitly call this function.

3.2.3 search_and_connect

Synopsis

status = ca.search_and_connect(pvName, chid, 0, args)
status = ca.search(pvName, chid)

Arguments

pvName (string): name of process variable to connect to
chid (string): SWIG pointer to channel id structure
args (tuple): callback function followed by any user arguments

Comments

See the section on callbacks for a description of how caPython implements callbacks.

3.2.4 clear_channel

Synopsis

status = ca.clear_channel(chid)

Arguments

chid (string): SWIG pointer to a channel id structure

Comments

None

3.2.5 put

Synopsis

status = ca.array_put(dbrType, count, chid, value)
status = ca.put(dbrType, chid, value)
status = ca.bput(chid, value)
status = ca.rput(chid, value)

Arguments

dbrType (int): database request type (ca.DBR_XXXX)
count (int): number of values to transfer
chid (string): SWIG pointer to a channel id structure
value (string): SWIG pointer to the C data type matching dbrType

Comments

None.

3.2.6 array_put_callback

Synopsis

status = ca.array_put_callback(dbrType, count, chid, value, 0, args)
status = ca.put_callback(dbrType, chid, value, 0, args)

Arguments

dbrType (int): database request type (ca.DBR_XXXX)
count (int): number of values to transfer
chid (string): SWIG pointer to a channel id structure
value (string): SWIG pointer to the C data type matching dbrType
args (tuple): callback function followed by any user arguments

Comments

Ca.put_callback() uses a count of one. See the section on callbacks for a description of how caPython implements callbacks.

3.2.7 get

Synopsis

status = ca.array_get(dbrType, count, chid, value)
status = ca.get(dbrType, chid, value)
status = ca.bget(chid, value)
status = ca.rget(chid, value)

Arguments

dbrType (int): database request type (ca.DBR_XXXX)
count (int): number of values to transfer
chid (string): SWIG pointer to a channel id structure
value (string): SWIG pointer to the C data type matching dbrType

Comments

None

3.2.8 array_get_callback

Synopsis

status = ca.array_get_callback(dbrType, count, chid, 0, args)
status = ca.get_callback(dbrType, chid, 0, args)

Arguments

dbrType (int): database request type (ca.DBR_XXXX)
count (int): number of values to transfer
chid (string): SWIG pointer to a channel id structure
args (tuple): callback function followed by any user arguments

Comments

Ca.get_callback() uses a count of one. See the section on callbacks for a description of how caPython implements callbacks.

3.2.9 add_event

Synopsis

status = ca.add_masked_array_event(dbrType, count, chid, 0, args, evid, mask)
status = ca.add_array_event(dbrType, count, chid, 0, args, evid)
status = ca.add_event(dbrType, chid, 0, args, evid)

Arguments

dbrType (int): database request type (ca.DBR_XXXX)
count (int): number of values to transfer
chid (string): SWIG pointer to a channel id structure
args (tuple): callback function followed by any user arguments
evid (string): SWIG pointer to an event id structure
mask (int): trigger mask

Comments

Ca.add_event() uses a count of one. See the section on callbacks for a description of how caPython implements event callbacks. The mask is a logical or of ca.DBE_VALUE, ca.DBE_LOG, or ca.DBE_ALARM. The three fields which are currently set to floating point zeros in the C function are always set to zero as a convenience to the user.

3.2.10 clear_event

Synopsis

status = ca.clear_event(evid)

Arguments

evid (string): SWIG pointer to an event id structure

Comments

Once the event id structure is no longer needed use ca.del_evid() to free the pointer.

3.2.11 pend_io

Synopsis

status = ca.pend_io(timeout)

Arguments

timeout (float): maximum time before returning

Comments

None

3.2.12 test_io

Synopsis

status = ca.test_io()

Arguments

None

Comments

None

3.2.13 pend_event

Synopsis

status = ca.pend_event(timeout)

Arguments

timeout (float): maximum time before returning

Comments

None

3.2.14 poll

Synopsis

status = ca.poll()

Arguments

None

Comments

None

3.2.15 flush_io

Synopsis

status = ca.flush_io()

Arguments

None

Comments

None

3.2.16 signal

Synopsis

status = ca.signal(status, message)
SEVCHK(status, message)

Arguments

status (int): status return from a channel access function
message (string): SWIG pointer to a null terminated array of char

Comments

None

3.2.17 CA Macros

These are now implemented as functions that return the result of the macro.

3.2.17.1 field_type

Synopsis

fieldType (int) = ca.field_type(chid)

Arguments

chid (string): SWIG pointer to a channel id structure

Comments

None

3.2.17.2 element_count

Synopsis

nativeCount (int) = ca.element_count(chid)

Arguments

chid (string): SWIG pointer to a channel id structure

Comments

None

3.2.17.3 name

Synopsis

pvName (string)= ca.name(chid)

Arguments

chid (string): SWIG pointer to a channel id structure

Comments

None

3.2.17.4 state

Synopsis

state (int)= ca.state(chid)

Arguments

chid (string): SWIG pointer to a channel id structure

Comments

None

3.2.17.5 message

Synopsis

statusMessage (string)= ca.message(status)

Arguments

status (int): return value from a channel access call, one of ca.ECA_XXX

Comments

Converts the status code to an informational text string.

3.2.17.6 host_name

Synopsis

hostName (string) = ca.host_name()

Arguments

chid (string): SWIG pointer to a channel id structure

Comments

None

3.2.17.7 read_access

Synopsis

access (int) = ca.read_access(chid)

Arguments

chid (string): SWIG pointer to a channel id structure

Comments

None

3.2.17.8 write_access

Synopsis

access = ca.write_access(chid)

Arguments

chid (string): SWIG pointer to a channel id structure

Comments

None

3.2.18 Other Macros

The macros implemented are found in the header files, not the documentation. These are implemented as functions that return the result of the macro. Some of these functions wrap arrays instead of macros. The implementation is identical for wrapping macros and arrays.

3.2.18.1 dbr_size

Synopsis

size (int) = ca.dbr_size(dbrType)

Arguments

dbrType (int): database request type (ca.DBR_XXXX)

Comments

None

3.2.18.2 dbr_size_n

Synopsis

size (int) = ca.dbr_size_n(dbrType, count)

Arguments

dbrType (int): database request type (ca.DBR_XXXX)
count (int): number of elements

Comments

None

3.2.18.3 dbr_value_size

Synopsis

size (int) = ca.dbr_value_size(dbrType)

Arguments

dbrType (int): database request type (ca.DBR_XXXX)

Comments

None

3.2.18.4 dbr_text

Synopsis

dbrText (string)= ca.dbr_text(dbrType)

Arguments

dbrType (int): database request type (ca.DBR_XXXX)

Comments

None

3.2.18.5 dbf_text

Synopsis

dbfText (string)= ca.dbf_text(dbfType)

Arguments

dbfType (int): native database field type (ca.DBF_XXXX)

Comments

None

3.2.18.6 valid_db_request

Synopsis

bool (int) = ca.valid_db_request(dbrType)

Arguments

dbrType (int): database request type (ca.DBR_XXXX)

Comments

Return true (non-zero) if the dbrType is one of ca.DBR_XXXX.

3.2.18.7 invalid_db_request

Synopsis

bool (int) = ca.invalid_db_request(dbrType)

Arguments

dbrType (int): database request type (ca.DBR_XXXX)

Comments

Return true (non-zero) if the dbrType is not one of ca.DBR_XXXX.

3.2.18.8 dbf_type_to_DBR

Synopsis

dbrType (int)= ca.dbf_type_to_DBR(dbfType)

Arguments

dbfType (int): native database field type (ca.DBF_XXXX)

Comments

Since the ca.DBF_XXXX and ca.DBR_XXXX types correspond to each other this transformation is not currently needed.

3.2.18.9 dbf_type_to_DBR_STS

Synopsis

dbrType (int)= ca.dbf_type_to_DBR_STS(dbfType)

Arguments

dbfType (int): native database field type (ca.DBF_XXXX)

Comments

DBF_XXXX and DBR_XXXX are mapped in a one to one correspondence to each other. The argument can be either.

3.2.18.10 dbf_type_to_DBR_TIME

Synopsis

dbrType (int)= ca.dbf_type_to_DBR_TIME(dbfType)

Arguments

dbfType (int): native database field type (ca.DBF_XXXX)

Comments

DBF_XXXX and DBR_XXXX are mapped in a one to one correspondence to each other. The argument can be either.

3.2.18.11 dbf_type_to_DBR_GR

Synopsis

dbrType (int)= ca.dbf_type_to_DBR_GR(dbfType)

Arguments

dbfType (int): native database field type (ca.DBF_XXXX)

Comments

DBF_XXXX and DBR_XXXX are mapped in a one to one correspondence to each other. The argument can be either.

3.2.18.12 dbf_type_to_DBR_CTRL

Synopsis

dbrType (int)= ca.dbf_type_to_DBR_CTRL(dbfType)

Arguments

dbfType (int): native database field type (ca.DBF_XXXX)

Comments

DBF_XXXX and DBR_XXXX are mapped in a one to one correspondence to each other. The argument can be either.

3.2.18.13 dbr_type_is_XXXX

Synopsis

bool (int)= ca.dbf_type_is_valid(dbrType)
bool (int)= ca.dbf_type_is_plain(dbrType)
bool (int)= ca.dbf_type_is_STS(dbrType)
bool (int)= ca.dbf_type_is_TIME(dbrType)
bool (int)= ca.dbf_type_is_GR(dbrType)
bool (int)= ca.dbf_type_is_CTRL(dbrType)

Arguments

dbrType (int): native database field type (ca.DBR_XXXX)

Comments

None.

3.2.18.14 alarmSeverityString

Synopsis

severityString (string)= ca.alarmSeverityString(alarmSeverity)

Argument

alarmSeverity (int): severity of the current PValarm

Comments

None.

3.2.18.15 alarmStatusString

Synopsis

statusString (string)= ca.alarmStatusString(alarmStatus)

Argument

alarmStatus (int): status of the current PValarm

Comments

None.

3.2.19 ca_modify_user_name

Synopsis

status = ca.modify_user_name(pUserName)

Arguments

pUserName (string): SWIG pointer to an array of char containing the new user name

Comments

None

3.2.20 ca_modify_host_name

Synopsis

status = ca.modify_host_name(pHostName)

Arguments

pHostName (string): SWIG pointer to an array of char containing the new host name

Comments

None

3.3 Callbacks

CaPython callbacks can be either Python functions or methods. The only restrictions on actions inside the callback functions are that CA calls cannot be made from a callback.

3.3.1 Key Points

- Each Python callback function is required to have two arguments. The first argument is a dictionary containing the results of the action. The second argument is a tuple containing any user arguments specified when the caPython function was invoked. If no arguments were specified then the tuple is empty.
- Functions that result in a callback being invoked require a tuple for the user data argument. The first element is the Python callback function and is required. The second element is an optional tuple containing any data that the user wants to access in the Python callback function. This user data tuple appears as the second argument in a callback function.
- The user must maintain a reference to the tuple passed to ca.add_event(). Reference counting is handled automatically for the other three callbacks.

3.3.2 Connection callback

Callback for ca.search_and_connect()

Example:

```
def connection_cb(epics_args, user_args):
    chid = epics_args[0]
    connection_state = epics_args[1]
```

Values:

chid (string): SWIG pointer to this channels id structure
connection_state (int): the current state of the connection (ca.CA_OP_CONN_UP or ca.CA_OP_CONN_DOWN)

Comments:

The argument is short so a tuple was used instead of a dictionary. Use the name macro and chid to determine which channels connection state is being reported (name = ca.name(chid)).

3.3.3 Put callback

Example:

```
def put_cb(epics_args, user_args):
    chid = epics_args['chid']
    dbrType = epics_args['type']
    count = epics_args['count']
    status = epics_args['status']
```

Python Argument:

chid (string): SWIG pointer to this channels id structure
dbrType (int): database request type (ca.DBR_XXXX)
count (int): number of values to transfered
status (int): CA status return code (ca.ECA_XXXX)

Comments:

There is no process variable data returned on a write, only CA data.

3.3.4 Get callback

3.3.4.1 Plain get

Example:

```
def get_cb(epics_args, user_args):
    chid = epics_args['chid']
    dbrType = epics_args['type']
    count = epics_args['count']
    status = epics_args['status']
    val = epics_args['pv_value']
```

Python arguments:

chid (string): SWIG pointer to this channels id structure
dbrType (int): database request type (ca.DBR_XXXX)
count (int): number of values to transfered
status (int): CA status return code (ca.ECA_XXXX)
val (Python type corresponding to dbrType): value read

Comments:

If the count is greater than one then the data is returned in a tuple of length count. An array of strings does not exist in EPICS.

3.3.4.2 Status get

Callback for ca.array_get_callback(). Use the macro dbf_type_to_DBR_STATUS to convert the plain request type to a status request type.

Example:

```
def get_status_cb(epics_args, user_args):
    chid = epics_args['chid']
    dbrType = epics_args['type']
    count = epics_args['count']
    status = epics_args['status']
    val = epics_args['pv_value']
    pv_status = epics_args['pv_status']
    severity = epics_args['pv_severity']
```

Python arguments:

pv_status (int): current alarm status of channel
severity (int): current alarm severity of channel
The other arguments are discussed in the plain get section.

3.3.4.3 Time get

Callback for ca.array_get_callback(). Use the macro dbf_type_to_DBR_TIME to convert the plain request type to a time and status request type.

Example:

```
def get_status_cb(epics_args, user_args):
    chid = epics_args['chid']
    dbrType = epics_args['type']
    count = epics_args['count']
    status = epics_args['status']
    val = epics_args['pv_value']
    pv_status = epics_args['pv_status']
    severity = epics_args['pv_severity']
    seconds = epics_args['pv_seconds']
    nseconds = epics_args['pv_nseconds']
```

Python arguments:

seconds (int): time the value was last updated in seconds since the epoch
nseconds (int): time the value was last updated in nanoseconds
The other arguments are discussed in the status get section.

3.3.4.4 Graphic get

Callback for ca.array_get_callback(). Use the macro dbf_type_to_DBR_GR to convert the plain request type to a graphic and status request type.

Example:

```
def get_status_cb(epics_args, user_args):
    chid = epics_args['chid']
    dbrType = epics_args['type']
    count = epics_args['count']
    status = epics_args['status']
    val = epics_args['pv_value']
    pv_status = epics_args['pv_status']
    severity = epics_args['pv_severity']
    units = epics_args['pv_units']
    prec = epics_args['pv_precision']
    upper_display_limit = epics_args['pv_updislim']
    lower_display_limit = epics_args['pv_lodislim']
    upper_alarm_limit = epics_args['pv_upalarmlim']
    lower_alarm_limit = epics_args['pv_loalarmlim']
    upper_warning_limit = epics_args['pv_upwarnlim']
    lower_warning_limit = epics_args['pv_lowarnlim']
```

Python arguments:

units (string): engineering units
prec (int): precision for displaying the value
upper_display_limit (float): varies by record, usually HOPR
lower_display_limit (float): varies by record, usually LOPR
upper_alarm_limit (float): HIHI field
lower_alarm_limit (float): LOLO field
upper_warning_limit (float): HI field
lower_warning_limit (float): LO field
The other arguments are discussed in the status get section.

Comments

When dbrType is ca.DBR_GR_STRING the only values returned are value, pv_status, and pv_severity. When dbrType is ca.DBR_GR_ENUM the values returned are value, pv_status, pv_severity, pv_nostrings, and pv_statestrings. This is the number of strings that describe the possible states in the record.

3.3.4.5 Control get

Callback for ca.array_get_callback(). Use the macro dbf_type_to_DBR_CTL to convert the plain request type to a control and graphic request type.

Example:

```
def get_status_cb(epics_args, user_args):
    chid = epics_args['chid']
    dbrType = epics_args['type']
    count = epics_args['count']
    status = epics_args['status']
    val = epics_args['pv_value']
    pv_status = epics_args['pv_status']
    severity = epics_args['pv_severity']
    units = epics_args['pv_units']
    prec = epics_args['pv_precision']
```

```
upper_display_limit = epics_args['pv_updislim']
lower_display_limit = epics_args['pv_lodislim']
upper_alarm_limit = epics_args['pv_upalarmlim']
lower_alarm_limit = epics_args['pv_loalarmlim']
upper_warning_limit = epics_args['pv_upwarnlim']
lower_warning_limit = epics_args['pv_lowarnlim']
upper_control_limit = epics_args['pv_upctllim']
lower_control_limit = epics_args['pv_upctllim']
```

Python arguments:

upper_control_limit (float): record specific, DRVH for analog records
lower_control_limit (float): record specific, DRVL for analog records
The other arguments are discussed in the graphic get section.

Comments

See the comments in the graphic get section.

3.3.5 Event callback

Event callbacks are the same as get callbacks.

3.4 Functions Not Implemented

3.4.1 ca_change_connection_event

Synopsis

status = ca_change_connection_event(chid, userFunc)

Arguments

chid: channel index
userFunc: address of a user function to be run when the connection state changes

Comments

The caPython callback scheme requires an argument in the callback routine for user data. Ca_change_connection_event() doesn't have this argument. A function could be implemented in C for connect/disconnect notification. The logic would be hardwired in the callback function and would require reinvoking the Python interpreter.

3.4.2 ca_add_exception_event

Synopsis

status = ca_add_exception_event(userFunc, userArg)

Arguments

userFunc: address of a user function to be run when exceptions occur
userArg: pointer passed to userFunc

Comments

This function does meet the criteria for the caPython callback scheme. It appears that the necessary information to fully implement an exception handler is not available. "More information needs to be provided about

which arguments are valid with each exception. More information needs to be provided correlating exceptions with the routines which cause them."

3.4.3 ca_replace_printf_handler

Synopsis

status = ca_replace_printf_handler(userFunc, printArgs)

Arguments

userFunc: address of function called by CA to print formatted text
printArgs: variable argument list similar in format to the arguments expected by ANSI C vprintf()

Comments

This function does meet the criteria for the caPython callback scheme. I haven't attempted to provide an implementation.

3.4.4 ca_replace_access_rights_event

Synopsis

status = ca_replace_access_rights_event(chid, userFunc)

Arguments

chid: channel index
userFunc: address of a user function to be run when access rights change

Comments

The caPython callback scheme requires an argument in the callback routine for user data. Ca_replace_access_rights_event() doesn't have this argument. A function could be implemented in C for connect/disconnect notification. The logic would be hardwired in the callback function and would require reinvoking the Python interpreter. The same functionality can be implemented using the read/write access macros. Before issuing a put/get check the write/read access of the channel.

3.4.5 ca_puser macro

Synopsis

dataArea = ca_replace_access_rights_event(chid)

Arguments

chid: channel index

Comments

Not needed. Still used in C support routines.

3.4.6 ca_test_event

Synopsis

ca_test_event(struct event_handler_args handler_args)

Arguments

handler_args: arguments passed to all CA event handlers

Comments

User callbacks are implemented in Python not C. A Python class built on top of caPython should consider including this functionality in a method.

3.5 File Descriptor Manager

CaPython provides a simple interface to the functionality of the file descriptor manager (fdmgr) provided with EPICS. The model used follows the example in the "Channel Access Client Library Tutorial" in section 4, "Monitoring a PV". The caPython interface needs to accomplish two items:

1. Allow execution of CA background activity every 15 seconds.
2. Execute callbacks as needed by calling ca_pend_event().

caPython meets these two goals in a limited implementation. The limits are a severely restricted interface that does things my way not yours.

3.5.1 fdmgr_start

Synopsis:

ca.fdmgr_start()

Description:

Initialize a file descriptor manager (fdmgr) session. Register a function to call when a file descriptor is created or removed and register a function to call after one second. The registered functions are C routines not accessible through the interface.

3.5.2 fdmgr_pend

Synopsis:

ca.fdmgr_pend()

Description:

Wait for file descriptor activity or the timeout, 20 seconds, to occur. An applications waiting for events to occur should call ca.fdmgr_pend() frequently.

4 SWIG Pointer Library

The SWIG pointer library provides a collection of useful methods for manipulating C pointers. Conversion between C and Python variables is handled using the Python API. All SWIG pointers are represented as strings in Python. The string consists of the concatenation of the string representations of the memory address and C type that the pointer identifies.

4.1 pointer.i

This documentation is copied directly from SWIG.

4.1.1 ptrcreate

Synopsis:

ptr = ca.ptrcreate(type, [val], [nitems])

Comments:

Creates a new object and returns a pointer to it. This function can be used to create various kinds of objects for use in C functions. type specifies the basic C datatype to create and value is an optional parameter that can be used to set the initial value of the object. nitems is an optional parameter that can be used to create an array. This function results in a memory allocation using malloc(). Examples :

```
a = ptrcreate("double")        # Create a new double, return pointer
a = ptrcreate("int",7)         # Create an integer, set value to 7
a = ptrcreate("int",0,1000)    # Create an integer array with initial
                                 # values all set to zero
```

This function only recognizes a few common C datatypes as listed below:

int, short, long, float, double, char, char *, void

All other datatypes will result in an error. However, other datatypes can be created by using the ptrcast function. For example:

```
a = ptrcast(ptrcreate("int",0,100),"unsigned int *")
```

4.1.2 ptrfree

Synopsis:

ca.ptrfree(ptr)

Comments:

Destroys the memory pointed to by ptr. This function calls free() and should only be used with objects created by ptrcreate(). Since this function calls free, it may work with other objects, but this is generally discouraged unless you absolutely know what you're doing.

4.1.3 ptrvalue

Synopsis

val = ca.ptrvalue(ptr, [index], [type])

Comments:

Returns the value that a pointer is pointing to (ie. dereferencing). The type is automatically inferred by the pointer type--thus, an integer pointer will return an integer, a double will return a double, and so on. The index and type fields are optional parameters. When an index is specified, this function returns the value of ptr[index]. This allows array access. When a type is specified, it overrides the given pointer type. Examples:

```
ptrvalue(a)              #  Returns the value *a
ptrvalue(a,10)           #  Returns the value a[10]
ptrvalue(a,10,"double")  #  Returns a[10] assuming a is a double *
```

4.1.4 ptrset

Synopsis:

ca.ptrset(ptr, val, [index], [type])

Comments:

Sets the value pointed to by a pointer. The type is automatically inferred from the pointer type so this function will work for integers, floats, doubles, etc... The index and type fields are optional. When an index is given, it provides array access. When type is specified, it overrides the given pointer type. Examples :

```
ptrset(a,3)             # Sets the value *a = 3
ptrset(a,3,10)          # Sets a[10] = 3
ptrset(a,3,10,"int")    # Sets a[10] = 3 assuming a is a int *
```

4.1.5 ptrcast

Synopsis:

```
newPtr = ca.ptrcast(ptr, type)
```

Comments:

Casts a pointer to a new datatype given by the string type. Type may be either the SWIG generated representation of a data type or the C representation. For example:

```
ptrcast(ptr,"double_p");   # Python representation
ptrcast(ptr,"double *");   # C representation
```

A new pointer value is returned. ptr may also be an integer value in which case the value will be used to set the pointer value. For example :

```
a = ptrcast(0,"Vector_p");
```

Will create a NULL pointer of type "Vector_p". The casting operation is sensitive to formatting. As a result, "double *" is different than "double*". As a result of thumb, there should always be exactly one space between the C datatype and any pointer specifiers (*).

4.1.6 ptradd

Synopsis:

```
ca.ptradd(ptr, val, [index], [type])
```

Comments:

Adds a value to the current pointer value. For the C datatypes of int, short, long, float, double, and char, the offset value is the number of objects and works in exactly the same manner as in C. For example, the following code steps through the elements of an array.

```
a = ptrcreate("double",0,100);          # Create an array double a[100]
b = a;
for i in range(0,100):
   ptrset(b,0.0025*i);          # set *b = 0.0025*I
   b = ptradd(b,1);             # b++ (go to next double)
```

In this case, adding one to b goes to the next double.

For all other datatypes (including all complex datatypes), the offset corresponds to bytes. This function does not perform any bounds checking and negative offsets are perfectly legal.

4.1.7 ptrmap

Synopsis:

ca.ptrmap(ptr, val, [index], [type])

Comments:

This is a rarely used function that performs essentially the same operation as a C typedef. To manage datatypes at run-time, SWIG modules manage an internal symbol table of type mappings. This table keeps track of which types are equivalent to each other. The ptrmap() function provides a mechanism for scripts to add symbols to this table. For example:

ptrmap("double_p","Real_p");

would make the types "doublePtr" and "RealPtr" equivalent to each other. Pointers of either type could now be used interchangably.

Normally this function is not needed, but it can be used to circumvent SWIG's normal type-checking behavior or to work around weird type-handling problems.

4.1.8 captrcreate

Almost identical to ptrcreate. The difference is that when a char type is requested in ca.ptrcreate data is stored as a NULL terminated C string. In ptrcreate the data is stored in an array of char with no NULL terminator.

5 CaPython Internals

The internals section discusses the design and extra code added to the CaPython interface to implement the EPICS channel access protocol (CA). Much of the interface code was generated by SWIG (Simplified Wrapper and Interface Generator, www.swig.org). Code was added to handle the CA macros, CA data types, and callbacks.

5.1 Source Files

The source code for caPython comes in three files.

- ca.i
- ca_helper_functions.i
- ca_internal_functions.c
- captr.i

File with the ".i" extension are SWIG input files. The files are SWIG'ed, compiled, then linked into a shared object module, camodule.so.

5.1.1 ca.i

The main SWIG interface file. Contains:

- Channel access library routines
- Constant macros
  - Data base types (DBF_XXXX and DBR_XXXX)
  - Channel access error codes (ECA_XXXXXXX)
  - Connection states
  - Event masks
- Additions to the modules initialization routine

5.1.2 ca_helper_functions.i

Routines added to make the interface more functional. Included in ca.i. Contains:

- Create and delete channel access specific data types (chid and evid)
- Channel access macros (information on a connection)
- Data base type manipulation (dbf_type_to_XXX and dbr_type_is_XXX)
- Numerical to text conversion (DBR types and Alarms)

5.1.3 ca_internal_functions.c

Functions used to improve the interface that the user does not access. Included in ca.i. Contains:

- Callbacks
- Data conversion

5.1.4 captr.i

I had to modify the ptrcreate library function. To do this I created a new pointer library based on the SWIG pointer library (I just copied swig/swig_lib/python/ptrlang.i to captr.i.) I prepended "ca" to all the ptr functions in the library. I added the line %include captr.i to the SWIG input file, ca.i.

When handling C char types the library treated an array as a collection of bytes. I need to use the array as a string. Here is the modification I made in captr.i.

```
} else if (strcmp(type,"char") == 0) {
    char *ip,*ivalue;
    ivalue = (char *) PyString_AsString(_PYVALUE);
    ip = (char *) ptr;
    strcpy(ip, ivalue);
    /* strncpy(ip,ivalue,numelements-1); */
}
```

The change alters how values are copied from the local data space to the storage data space. The initial value passed in by the user is PYVALUE. This is translated to a null terminated string (PyString_AsString). The user has also specified the number of elements to be allocated in numelements. The original copy method:

```
strncpy(ip,ivalue,numelements-1); ,
```

moves the user values to the storage area. The null terminator is omitted. With out the null terminator the string was handled correctly most of the time but not always.

Sometimes extra characters were appended to the users string. Using a strcpy instead of a strncpy insures that the null terminator is added to the end of the string.

5.2 Callbacks

CaPython is designed to have callback functions written in Python. This requires that the Python interpreter be reinvoked from a C callback function. Using SWIG we specify which C function will be called for the different callbacks: connection, get, put, and event. From within the C function the interpreter is reinvoked and the Python callback function is called.

5.2.1 Argument Passing

To make this concrete let's look at how argument passing is handled for ca_search_and_connect(). The C synopsis is:

```
int ca_search_and_connect(
        char *CHANNEL_NAME,
        chid *PCHID,
        void (*USERFUNC)(struct connection_handler_args ARGS),
        void *PUSER);
```

When ca_search_and_connect() completes it calls USERFUNC. USERFUNC must have one argument of type struct connection_handler_args. Using the ARGS variable within the callback provides access to the data pointed to by PUSER. In the caPython wrapper code the USERFUNC argument is always connectCallback. When using ca_search_and_connect(), connectCallback is the C callback function that is always called. ConnectCallback has one argument of type struct connection_handler_args. Since the function pointer is occupied, the user data is used to transmit the Python callback function and any user data. The callback functions assume that the callback function and user data, when present, are members of a tuple. The first element is the callback function and the second element, when present, is a tuple containing as much user data as necessary. ConnectCallback must:

1. Prepare the channel access data for return to the Python callback function.
2. Retrieve the address of the user data (two element tuple).
3. Extract the Python function to call from the tuple (the first element).
4. Extract any user data specified.
5. Combine the user data in a tuple with the channel access data.
6. Call the Python function with the data tuple.

The data tuple always has two elements. The first element of the data tuple is data from channel access. The channel access data varies by callback. The second element of the data tuple is a tuple containing the user data. If no user data is specified then the tuple is empty.

5.2.2 Reference Counting

Python keeps a count of how many references there are too an object. When the reference count hits zero the memory used by the object is freed. The Python interpreter automatically tracks reference counts while in the C interface routines reference counting is the responsibility of the programmer. CaPython uses two methods for maintaining a reference to the argument tuple used in callbacks.

1. Increment the reference count to the tuple just before the channel access function that uses callbacks is called. Then decrement the reference count after the Python callback function is finished executing.
2. In cases where the tuple must be used again before a callback channel access function is called again, the user is responsible for keeping a reference to the tuple. This is the case for event callbacks. The alternative is to maintain a list of all registered events and the accompanying tuple in caPython.

5.3 File Descriptor Manager

CaPython provides a simple interface to the functionality of the file descriptor manager (fdmgr) provided with EPICS. The model used follows the example in the "Channel Access Client Library Tutorial" in section 4, "Monitoring a PV".

The caPython interface needs to accomplish two items:

3. Allow execution of CA background activity every 15 seconds.
4. Execute callbacks as needed by calling ca_pend_event().

caPython meets these two goals in a limited implementation. The limits are a severely restricted interface that does things my way not yours.

5.3.1 Preliminaries

Include the header file and global variables in the SWIG source file, ca.i.

```
#include "fdmgr.h"
fdctx *pfdctx;
static struct timeval twenty_seconds  = {20, 0};
static struct timeval fifteen_seconds  = {15, 0};
static struct timeval one_second       = { 0, 1000000};
```

The global variables are available in any added C functions.

5.3.2 caPython Interface

5.3.2.1 fdmgr_start

Synopsis:

ca.fdmgr_start()

Description:

Initialize a file descriptor manager (fdmgr) session. Register a function to call, fd_register, when a file descriptor is created or removed. Register a function to call after one second; caPollFunc.

Arguments:

None

Source:

```
void fdmgr_start() {
   /* Initialize an fdmgr session. */
   pfdctx = fdmgr_init();

   if(!pfdctx)
```

```
        printf("fdmgr_start: fdmgr_init() failed\n");

    /* Call fd_register each time a file descriptor is created. */
    SEVCHK(ca_add_fd_registration(fd_register, pfdctx),
                "fdmgr_start: ca_add_fd_registration failed");

    /* Call caPollFunc after one second.  Don't pass any user values. */
    fdmgr_add_timeout(pfdctx, &one_second, caPollFunc, (void *)NULL);

}
```

5.3.2.2 fdmgr_pend

Synopsis:

```
ca.fdmgr_pend()
```

Description:

Wait for file descriptor activity or the timeout, 20 seconds, to occur. An application waiting for events to occur should call ca.fdmgr_pend() frequently.

Arguments

None

Source:

```
void fdmgr_pend() {
    fdmgr_pend_event(pfdctx, &twenty_seconds);
}
```

5.3.3 C Interface

Functions in the C interface are not called by users directly. They are used in the caPython interface functions.

5.3.3.1 fd_register

Synopsis:

```
void fd_register(void *pfd, int fd, int opened);
```

Description:

Function called each time the channel access client library places a file descriptor into service or removes one from service.

Arguments

pfd – second parameter in ca_add_fd_registration()
fd – file descriptor specified by system
opened – non-zero = file descriptor placed in service
zero = file descriptor removed from service

Source:

```
void fd_register(void *pfd, int fd, int opened) {
    if(opened)
        fdmgr_add_callback(pfdctx, fd, fdi_read, caFDCallback, NULL);
    else
        fdmgr_clear_callback(pfdctx, fd, fdi_read);
```

```
}
```

Comments:

Each time a file descriptor becomes active a callback is waiting to execute. At this time caFDCallback() is called passing NULL as the only argument.

5.3.3.2 caFDCallback

Synopsis:

```
void caFDCallback(void *pParam);
```

Description:

Function called each time a registered file descriptor is waiting to execute.

Arguments

pParam – not used

Source:

```
void caFDCallback(void *pParam) {
   ca_poll();
}
```

Comments:

Calling ca_poll() allows execution of channel access callbacks (event handlers).

5.3.3.3 caPollFunc

Synopsis:

```
void caPollFunc(void *pParam);
```

Description:

Function called every 15 seconds to allow execution of channel access background activity.

Arguments

pParam – not used

Source:

```
void caPollFunc(void *pParam) {
   ca_poll();
   fdmgr_add_timeout(pfdctx, &fifteen_seconds, caPollFunc, (void *)NULL);
}
```

Comments:

Calling ca_poll() allows execution of channel access callbacks (event handlers). Then we schedule caPollFunc() to be called again in 15 seconds.