MATLAB OBJECTS FOR EPICS CHANNEL ACCESS

J. Chrin, Paul Scherrer Institut, 5232 Villigen PSI, Switzerland

Abstract

With the substantial dependence on MATLAB for application development at the SwissFEL Injector Test Facility (SITF), the requirement for a robust and more extensive EPICS (Experimental Physics and Industrial Controls System) Channel Access (CA) interface became increasingly imperative. To this effect, a new MATLAB Executable (MEX) file has been developed around an in-house C++ CA interface library (CAFE), which serves to expose comprehensive control system functionality to within the MATLAB framework. Immediate benefits include support for all MATLAB data types, a richer set of synchronous and asynchronous methods, a further physics oriented abstraction layer that uses CA synchronous groups, and compilation on 64-bit architectures. An account of the MOCHA (MATLAB Objects for CHannel Access) interface is presented.

MOTIVATION

MATLAB [1] is an established fourth-generation programming language and numerical computing environment that provides matrix operations, algorithm procedures, data plotting tools and a Graphical User Interface (GUI) framework, as well as support for Object Oriented Programming (OOP). Its use for the control and operation of particle accelerators has surged in recent times (Fig. 1), initiating special focus within the accelerator community [2]. It is also the principal choice of physics application developers at the SwissFEL Injector Test Facility (SITF) [3]. While a critique of MATLAB is not intended here, it is acknowledged that the conciseness of the code adds to its ability to program rapidly and effectively. This is particularly enticing during the SITF commissioning phases [4] where novel applications may be quickly developed and put to the test. The consequential increase in the usage of MATLAB, coupled with the need to perform a number of essential measurements, however, inevitably led to a reappraisal of the MATLAB interface to the underlying Experimental Physics and Industrial Controls System (EPICS) [5] and its communication protocol, Channel Access (CA) [6]. The need for a stable and more extensive interface to that provided by the initially adopted MATLAB Channel Access (MCA) package [7, 8], became apparent. The following lists a number of necessary prerequisites.

- Support for all MATLAB data types.
- Incorporation of CA synchronous groups into MATLAB objects for specific operations such as machine snapshots and the acquisition of orbit data.
- Improved CA reconnection management to ensure stability and robustness in every eventuality.
- Compilation on 64-bit Linux architectures.

In the meantime and per contra, a new C++ CA interface library, CAFE [9], has recently been developed in-house that provided the required underlying functionality. Since CAFE further sought to act as a CA host library to scripting and domain-specific languages, the development of bindings to MATLAB presented itself as a natural extension. To achieve this necessitated gaining knowledge and experience in the use of the MATLAB C Application Programming Interface (API). A dynamically loadable MATLAB Executable (MEX) file could then be constructed that would allow CAFE routines to be called from within the MATLAB framework, in the same manner as MATLAB built-in functions.

The following recapitulates CAFE's functionality before describing how the new MOCHA (MATLAB Objects for CHannel Access) MEX-file is created. The MOCHA syntax is then presented and MATLAB's Object Oriented Programming (OOP) capability is briefly exemplified with a specialized MOCHA aware MATLAB class.

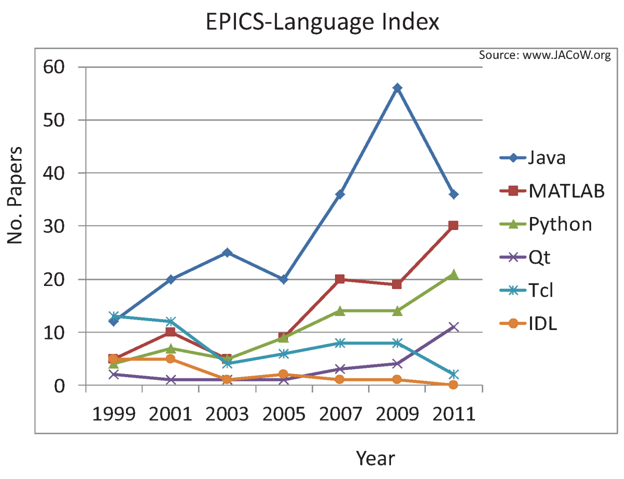

Figure 1: Approximate number of ICALEPCS papers citing EPICS together with a given language. The figure is indicative of trends only.

THE CAFE IMPLEMENTATION

CAFE is a C++ library that offers a multifaceted interface to the native C-based CA API. It provides remote access to EPICS Process Variables (PVs), which encapsulate the EPICS channel controls data, residing in the Input Output Contoller (IOC) or other devices hosting a CA server.

ISBN 978-3-95450-139-7

Functionality includes synchronous and asynchronous interactions for both individual and collections of PVs. At its core is a Boost multi-index container [10] which takes ownership of the data object (CAFEConduit) elements, which store all the current details of the associated channel. Callback functions, which have been implemented for all operations involving CA connection handlers, event handlers and access right handlers, report their data to the CAFEConduit object. The container also provides multiple, distinct interfaces that allow for fast retrieval and modification of the element's data. Stability and robustness are thus accomplished by ensuring that connectivity to PVs remains in a well defined state and results of all operations are captured and reported with integrity.

Figure 2 shows how the CAFE method first queries the CAFEConduit object within the container to assess the channel's current state. Only if the specified preconditions are met (e.g. channel is connected) is the input data verified and the message sent over the network (as for handle 1 in Fig. 2). If otherwise, the request is declined and an error message returned to the client (as for handle 4 in Fig. 2).

With the prescribed controls functionality in place within the CAFE library, the task of the MOCHA MEX-File is to provide the necessary CAFE bindings to MATLAB (Fig. 3).

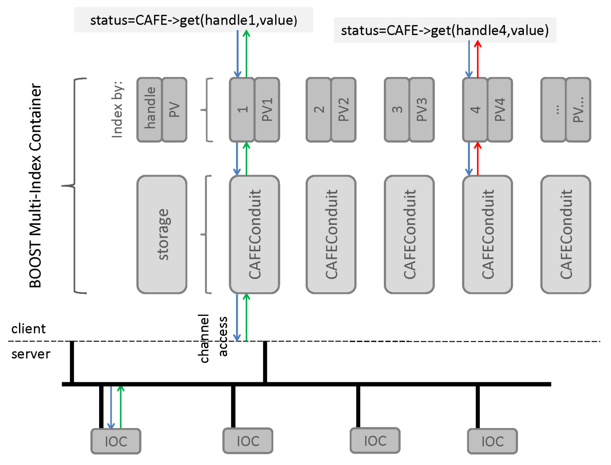
status=CAFE->get(handle1,value)
status=CAFE->get(handle4,value)
BOOST Multi-Index Container
Index by:
handle
PV
1
PV1
2
PV2
3
PV3
4
PV4
...
PV...
storage
CAFEConduit
client
server
channel access
IOC

Figure 2: A schematic diagram of CAFE's multi-index container composed of two indices, the handle (or object reference) and the process variable name. CAFE methods first query the CAFEConduit object within the container, shown here to be indexed by handle, to verify the channel's current state before deciding to send a message to the CA server hosted by the IOC.

THE MOCHA MEX FILE

The groundwork for MATLAB CA connectivity had, in many respects, already been laid by the MCA package. MOCHA, however, unlike MCA, implicitly shelters underlying CA routines from the MEX-file. Since these are delegated to the CAFE library, the resulting MEX-file is greatly simplified. A similar approach is adopted by the labCA package [11] where multiple MEX-files provide interfaces to the long established C-based EZCA (E-Z Channel Access) interface [12]. It is thus within MOCHA that the MATLAB-specific techniques in interpreting input data and packaging return data are to be mastered. The software strategy for constructing the MOCHA MEX-file is described.

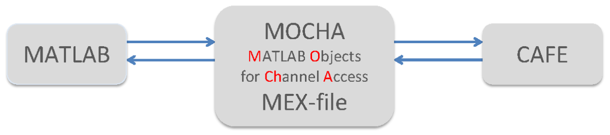

Figure 3: The MOCHA MEX-file provides the gateway from MATLAB to the CAFE library. Functions provided in the MEX-file are thus exposed to the MATLAB workspace.

Data Flow

The MEX-file consists of a gateway routine, named `mexFunction`, where variables are passed into the function through input and output arguments. The `mexFunction` syntax is:

```
void mexFunction (int nlhs, mxArray *plhs[], int
nrhs, const mxArray *prhs[])
```

where `nhls, nrhs` refer to the number of output and input parameters, respectively, which in the resulting MEX-file syntax corresponds to the number of parameters on the left-hand and right-hand side of the MEX-file function (as will be evident from Listing 1); `plhs, prhs` refer to an array of pointers to the input and output parameters (of length `nlhs` and `nrhs`) which themselves are encapsulated in MATLAB's underyling fundamental data type, the `mxArray`. All MATLAB's variables, namely scalars, vectors, matrices, strings, cell arrays, and structures are stored as `mxArrays`.

Source code is written using the MATLAB C API which comprises two libraries. The MX Matrix library provides the `mx*` functions to read/write input/output arguments from/to MATLAB, while the MEX library provides the `mex*` functions to perform operations in the MATLAB environment (and access global variables).

The MOCHA MEX-file is constructed such that the first input argument identifies the CAFE method to be invoked, while the second input argument is a handle to the previously obtained CAFEConduit object for the relevant EPICS PV. Any input data arguments then follow. The principle steps of the data flow within the MEX cycle are as follows.

- The input MATLAB data types are validated using the `mxIs*` functions and their data is extracted with the `mxGet*` functions.
- The first input argument is matched to an index in a switch statement that identifies the corresponding CAFE method.

ISBN 978-3-95450-139-7

- The MATLAB data types of any remaining input arguments are validated and their data is extracted.
- mxCreate* functions are used to create the MATLAB array for output arguments (most methods return at least a status).
- The corresponding CAFE function is invoked, with input and output data pointers passed as parameters.

Establishing communication between CAFE methods and the MATLAB workspace is thus largely reduced to mapping MATLAB data types to their equivalent CAFE/C++ data types and vice-versa. Once accomplished, and after implementing just a few CAFE methods, a definite pattern emerges that renders the making of a new MATLAB CA MEX-file refreshingly uncomplicated. Listing 1 shows the MOCHA syntax, for selected `set`, `get` and `monitor` methods. Note that a single `set` method can be applied for all MATLAB data types. Similarly the `getStruct` method will return a structure with value(s) given in the native data type (along with the alarm status, alarm severity and timestamp).

Listing 1: The MOCHA Syntax

```
% retrieve reference handles
handle = mocha('open','pvName');
groupHandle = mocha('openGroup','gName');
% returns value(s) in native data type
dataStruct = mocha('getStruct',handle);
[val,status] = mocha('get',handle); %returns double
[val,status] = mocha('getString',handle);
% synchronous group
[values,statae,isStatusOK] = mocha('getGroup',
    groupHandle);
% control display data
ctrlStruct = mocha('getCtrlStruct', handle);
status = mocha('set',handle,data); % all MATLAB types
% initiate a monitor and invoke 'action' on update
status = mocha('monitor',handle,'action');
% returns latest cached value from monitor (or get)
[val,status]=mocha('getCache',handle);
status = mocha('close',handle); % close channel
mocha('close'); % terminate all CA connections
```

CAFE Exceptions

CAFE is equipped with its own custom exceptions for reporting CA and CAFE-specific errors. An unexpected predicament arose, however, with the realization that CAFE's structured exceptions were not being caught from within the MEX-file, unless relucantly classified as an unknown exception. This behaviour could be attributed to the MATLAB version linker script, `mexFunction.map`. MATLAB's proposed solution was to link against a static version of the MEX library or to create a wrapper library that does not need to conform to the `mexFunction.map`. Since the latter approach is more inclusive, the previous contents of the `mexFunction` were moved to a separate library that could be easily called directly from the MOCHA MEX-file. The MOCHA `mexFunction` thus calls a single function embodied in a separate library.

Compilation on 64-bit Architectures

MEX-files constructed on 32-bit platforms can be similarly built on 64-bit architectures, without modification to source files, by enabling a 32-bit compatibility layer through the `-compatibleArrayDims` flag in the `mex` build command. To benefit from MATLAB's 64-bit indexing functionality, however, requires an update to the MEX source code, particularly with regard to the usage of the MX Matrix Library. To properly handle large arrays, variables used as array sizes or indices were converted from the 32-bit `int` type to the `mwSize` and `mwIndex` MATLAB preprocessor macros, which grant cross-platform flexibility. Similarly, 64-bit MX Matrix functions were identified and their signatures re-examined to ensure correct types are assigned to the input/output arguments. The MEX-file is finally compiled using the large array handling API, which is explicitly enabled by the inclusion of the `-largeArrrayDims` flag in the `mex` build. In a future MATLAB release, this flag will become the default option. The implementation of these changes to the MEX-file also renders the code platform (i.e. 32-bit, 64-bit) independent, thereby dispensing with the need to maintain a separate code base for building 32-bit MEX-files.

The 64-bit Scientific Linux 6 architecture is now standard in the SITF control room [13].

MATLAB OBJECTS

MATLAB releases since 2008 provide support for OOP. Features include classes, pass-by-value and pass-by-reference semantics, access control for methods and properties (public, private, protected), multiple inheritance, exceptions, operator overloading, and more. Of particular interest here is the use of class definition files which define object properties (i.e. data), methods and events. Pass-by-reference semantics is obtained by sub-classing the `handle` class which provides `Listeners` and `Events` that may be respectively configured to monitor object properties and trigger an associated event (or action). These possibilities can be put into good effect to produce MOCHA aware classes for certain specific operations, such as the acquisition of orbit data (Listing 2) and machine snapshots, though data verification here was undertaken within the MEX-file to enhance performance. Typically these objects make use of CAFE's abstract layer that addresses related data sets as a single logical unit. An XML configuration mechanism is used for the initialization of such group/collection objects and their data may be retrieved or set with a single method invocation. Meta data related to group/collection members, such as node positions, are extracted, upon object initialization, from CAFE's interface to an XML-based

Listing 2: A MATLAB Class Definition File for Orbit Data

```
classdef injectorOrbit < handle
  properties (SetAccess = private)
    gName='gDBPM'; gHandle=0; bpmNames=''; s=0;
    x=0; y=0; I=0; isStatusOK=0;
    ...
  end % properties
  events
    % listener defined in cafeErrorHandler class
    cafeErrorReport
  end % events
  methods
    function orbit = injectorOrbit();
      orbit.gHandle = mocha('openGroup',orbit.gName);
      [orbit.bpmNames,orbit.s] =
        mocha('getStaticData','cDBPM','master.xml');
      cafeErrorHandler.add(orbit)
      return
    end % function orbit
    function get(orbit)
      [orbit.x,orbit.y,orbit.I,...,orbit.isStatusOK]
           = mocha('getOrbit',orbit.gHandle);
      if orbit.isStatusOK !~ 1
        notify(orbit,'cafeErrorReport');
      end
      return
    end % function get
  end % methods
end % classdef
```

SITF accelerator database, also referred to as the master XML accelerator file [14]. MOCHA aware classes have also been provided to retrieve/set controls data or to start monitors which cache updated values and, following the convention in MCA, may optionally execute a MATLAB accessor script to e.g. update a widget value.

Interestingly, although the object oriented capabilities appear to be sufficiently comprehensive to allow established design patterns [15] to be effectively applied, they are yet to be fully exploited here. This may possibly be due to the observation that the core functionalities in MATLAB remain in the procedural style, inclining developers to program in a similar ‘function calls’ capacity.

DISCUSSION

MOCHA is a straightforward MEX-file that provides CA functionality to MATLAB through the CAFE interface. MOCHA in itself should not be viewed in isolation, but rather also as a demonstration of CAFE’s suitability to act as a host CA interface for other C/C++ based declarative and fourth-generation languages. In this respect, MOCHA is a natural extension to its CAFE roots driven by MATLAB user requirements at the SITF. The resulting boundary between the CA components, which are designated to the CAFE library, and the specifics of the MATLAB API, ensures a code base that is inherently convenient to maintain. Further developments include a refactoring of the CAFE C++ code with the intent of making the internal structure more comprehensible and easier to interpret. The corresponding MOCHA MEX-file may then finally be consolidated to produce a general purpose MATLAB CA interface. In the meantime, the experience gained to date in programming CA clients and constructing MEX-files has allowed us to provide a stable, robust and extensive MATLAB interface to EPICS that serves the needs of application developers at the SITF.

REFERENCES

[1] MATLAB®, `http://www.mathworks.com`

[2] J.W. Corbett, “MATLAB Workshop Report”, PCaPAC 2010, Saskatoon, Saskatchewan, Canada, Paper Code ID: THRA01, *presentation only*, and references therein.

[3] “SwissFEL Injector Conceptual Design Report”, Ed. M. Pedrozzi, PSI Report Nr. 10-05, July 2010.

[4] T. Schietinger, “Update on the Commissioning Effort at the SwissFEL Injector Test Facility”, LINAC 2012, Tel-Aviv, Israel, pp. 504–506.

[5] EPICS, `http://www.aps.anl.gov/epics/`

[6] J.O. Hill, R. Lange, “EPICS R3.14 Channel Access Reference Manual”, `http://www.aps.anl.gov/epics/docs/ca.php`

[7] T. Terebilo, “Channel Access Client Toolbox for MATLAB”, ICALEPCS 2001, San Jose, California, USA, pp. 543–544.

[8] MATLAB Channel Access (MCA), `http://sourceforge.net/apps/trac/epics/wiki/MatlabChannelAccess`

[9] J. Chrin, M. Sloan, “CAFE, A Modern C++ Interface to the EPICS Channel Access Library”, ICALEPCS 2011, Grenoble, France, pp. 840–843.

[10] Boost Multi-index Containers Library, `http://www.boost.org/libs/multi_index/`

[11] T. Straumann, “labCA - An EPICS Channel Access Interface for Scilab and Matlab”, Internal Document 2003, SLAC National Accelerator Laboratory, updated June 2010, `http://www.slac.stanford.edu/~strauman/labca/`

[12] N.T. Karonis, “EZCA Primer”, Internal Document, Argonne National Laboratory, Jan. 1995, `http://www.aps.anl.gov/epics/extensions/ezca/`

[13] P. Chevtsov *et al.*, “Status and Perspective of the SwissFEL Injector Test Facility Control System”, ICALEPCS 2013, San Francisco, California, USA, Paper Code ID: MOPPC112, *These Proceedings*.

[14] J. Chrin, L. Hubert, R. Krempaska, G. Prekas, T. Pelaia, “XML Constructs for Developing Dynamic Applications or Towards a Universal Representation of Particle Accelerators in XML”, IPAC 2011, San Sebastián, Spain, pp. 2295–2297.

[15] E. Gamma, R. Helm, R. Johnson, J. Vlissides, “Design Patterns, Elements of Reusable Object-Oriented Software”, Addison-Wesley (1994), ISBN 0-201-63361-2.

ISBN 978-3-95450-139-7