DRAFT

MIDDLE LAYER SOFTWARE FOR ACCELERATOR CONTROL

Gregory J. Portmann, Jeff Corbett, and Andrei Terebilo
November, 2003[1]

TABLE OF CONTENTS

APPENDICES

REFERENCE

[1] Most recent document in Q:\Groups\Accel\Controls\matlab\acceleratorcontrol\docs

1. INTRODUCTION

What makes Matlab so appealing for accelerator physics is the combination of a matrix oriented programming language, an active workspace for system variables, powerful graphics capability, built-in math libraries, and platform independence. At the ALS, Matlab is used for storage ring control including energy ramp, configuration save/restore, global orbit correction, local photon beam steering, insertion device compensation, beam-based alignment, tune correction, response matrix measurement, and script-based physics studies [1-4]. Simple Channel Access has been used to connect these programs to the EPICS control system.

At SSRL, parallel developments in Matlab led to the Accelerator Toolbox (AT) for machine simulations [1], Matlab Channel Access Toolbox (MCA) for EPICS connections [2], and LOCO for accelerator calibration, [3, 8]. In a collaborative effort between ALS and SSRL, many of the control functions developed at the ALS were ported to SSRL, re-structured to incorporate MCA and made *machine independent*. As a result, the methodology and structure of the control routines and functions is easily ported to other machines. The resulting "Middle Layer" software simplifies application program development and buffers the user from the details of MCA and cumbersome control system channel names.

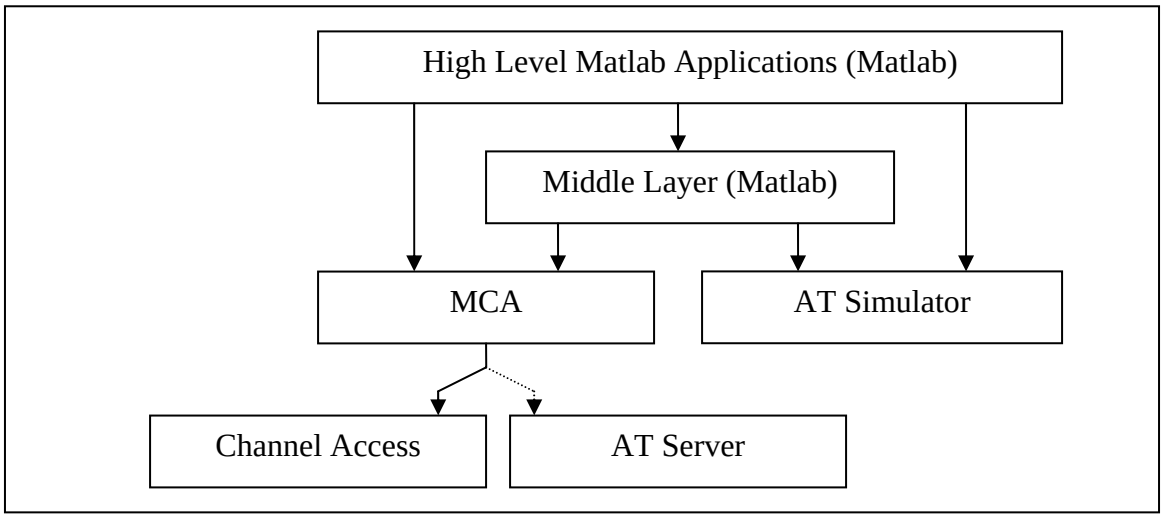


Fig. 1. Middle Layer Software Flow Diagram

As shown in Fig. 1 the Middle Layer software provides a set of functions that accesses either the machine hardware via the MCA toolbox or the AT simulator, [1, 2]. It can also connect to a remote AT simulator serving Channel Access. The ability to switch between online and simulate modes is helpful for analysis and debugging. The AT Serve mimics both the accelerator and the control system and requires no knowledge of the AT toolbox. The AT Simulator manipulates the local AT variables on your computer (THERING). One of the fundamental purposes of the Middle Layer is to change or interpret the hardware channel naming scheme used by the control system. Channel names are often quite obtuse so it is best not burden too many people with

deciphering what names goes with what piece of hardware. The Middle Layer organizes channel names into groups (families), subgroups (fields), and devices (elements). The Middle Layer tries to mimics naming schemes commonly used in particle tracking codes. Hence, the same language or terminology of tracking codes can be used to communicate with the online accelerator.

At the heart of the Middle Layer is a data structure containing the necessary information to setup the mapping from Family/Device to the control system hardware. The Matlab structure has been named the Accelerator Object (AO). The AO contains attributes for each Family (element indices, channel names, etc), handles for device control (MCA), hardware-to-physics conversion factors, etc. The complete set of Accelerator Objects is contained in a text file for easy editing. The AO resides in the memory location for application data in the Matlab command window. A parallel structure, called Accelerator Data (AD), contains directory locations, file names, and basic accelerator parameters. Accelerator Data structure also resides in the application data location of the command window. Running the Matlab command *aoinit* will setup these structures. The details of how to setup the Middle Layer is in the Appendix.

2. MIDDLE LAYER NOMENCLATURE

A standard set of naming conventions has been established for variables and functions.

Families/device

Family	= Group descriptor	(text string)
Field	= Subgroup descriptor	(text string)
DeviceList	= [Sector Element-in-Sector]	(two column matrix)
ElementList	= Element-in-family	(one column vector)
ChannelName	= Control System name	(text string)
CommonName	= Commonly used name	(text string) (not required)

Functions

The function prefix attempts to provide some indication for what the function does.

1. anal… – analyzes a data set
2. calc... – makes a calculation or conversion from existing data
3. get… – retrieve information from EPICS or a database (no setpoint changes)
4. meas… – perform a measure and return a result (usually setpoints are changed)
5. mon... – monitor a group of channels
6. ramp... – ramp a group of channels at a specified rate
7. set... – absolute setpoint change
8. step... – incremental setpoint change

3. MIDDLE LAYER FAMILIES

From a control system point of view each device usually has a unique channel name. However, accelerator physicists usually think in terms of a family (corrector, quadrupole, etc), how many elements are in a given family, and element attributes (length, strength, etc). For instance, all the beam position monitors (BPM) can be one family with different elements. Table 1 shows some typical family names.

Family Name	Function
BEND	Bend magnets
QF, QD	Quadrupoles
SF, SD	Sextupoles
SQSF, SQSD	Skew quadrupoles
HCM, VCM	Correctors
BPMx and BPMy	Beam position monitors

Table 1: Typical Families used for the ALS and SPEAR

Similar to most accelerator simulation codes, the Middle Layer software uses the same convention but associates both an *element index* and a *device index pair* with each individual piece of hardware. The "Element List" method specifies a Family member by the sequential order in the accelerator. Referring to Table 2, the third horizontal corrector is referred to in the Family/Element convention as (HCM, 3). Equivalently, the "Device List" method specifies a family member by the Sector and device number within the sector. For instance, a 12-fold symmetry storage ring is conveniently divided into 12 Sectors. If the ninth horizontal corrector is the first such magnet in Sector two, it can be referred to as (HCM, [2 1]). Hence, two ways are used to specify a desired piece of hardware – Family/ElementList and Family/DeviceList. Both have their merits.

Family-Element Method	Family-Device Method	Channel Name
HCM, 1	HCM, [1,1]	Unassigned
HCM, 2	HCM, [1,2]	SR01C___HCM2___AM01
HCM, 3	HCM, [1,3]	SR01C___HCSD1__AM00
HCM, 4	HCM, [1,4]	SR01C___HCSF1__AM02
HCM, 5	HCM, [1,5]	SR01C___HCSF2__AM03
HCM, 6	HCM, [1,6]	SR01C___HCSD2__AM01
HCM, 7	HCM, [1,7]	SR01C___HCM3___AM02
HCM, 8	HCM, [1,8]	SR01C___HCM4___AM03
HCM, 9	HCM, [2,1]	SR02C___HCM1___AM00
HCM, 10	HCM, [2,2]	SR02C___HCM2___AM01
- - -	- - -	- - -
HCM, 94	HCM, [12,6]	SR12C___HCSD2__AM01
HCM, 95	HCM, [12,7]	SR12C___HCM3___AM02

Table 2. Family/ElementList, Family/DeviceList, and Channel Names for the horizontal corrector magnets at the ALS.

Often there is an advantage to the Family/DeviceList method over the Element index because takes some thought to calculate an Element number but the device location in the repetitive sector structure of a storage ring or transport line is immediately apparent. More importantly, it is usually safer to hardcode the DeviceList method in an application program. For instance, a magnet referred as HCM [5 4] should never change even if additional correctors are added to the accelerator. However, adding a new HCM will change the element numbering in the ring unless it's the last magnet or a "placeholder" has created in advanced for new magnets (hence why

(HCM, 1) is included in Table 2 even though it does not yet exist). Functions *dev2elem* and *elem2dev* convert between the Element and Device conventions. All Middle Layer functions use these two methods interchangeably. It is also possible to reference devices by a Common Name method (possible the actual hardware name). A Common Name can replace a device list. Details of how to set this up are in the Appendix.

As an example, the ALS has 94 horizontal corrector magnets distributed in 12 sectors. Table 2 shows how these two methods work. In general, the hardware channel names are much more difficult to keep track of then the Family/DeviceList.

For example, the function *getam* returns the analog monitor value; *getam('HCM',4)* returns the value of the process variable assigned to the 4th horizontal corrector magnet in the ring. The same value can be accessed with *getam('HCM',[1 4])*. All functions allow for vectorized inputs. For instance, *getam('HCM',[1 3;1 5;7 8])* returns the 3rd and 5th HCM in Sector 1 and the 8th HCM in Sector 7 and *getam('HCM')* returns all HCM elements in the family.

Since it is easy to create families one might want to add special or temporary families for an experiment or task. For instance, in a ramping application an Accelerator Object with every magnet involved in the ramp can be created (or one could use a cell array of magnets which is sent to *getam*, *getsp*, or *setsp*). See the methods to create Accelerator Objects in the Appendix for more details.

4. BASIC MIDDLE LAYER FUNCTIONS

Although the Middle Layer function toolbox is well established, the complete toolbox continually expands. Wherever possible, Middle Layer functions are written in a machine independent way. However, hardware and control methods in different accelerators sometimes limits the degree to which machine independent code can be written. This section lists the basic functions which need to work in order for the Middle Layer to be useful.

Database Access Functions – These functions are used to communicate directly with the online hardware, Channel Access Server, or Accelerator Toolbox. The two main functions in this class are *getpv* (get EPICS process variable) and *setpv* (set EPICS process variable). Both functions accept a variety of input formats including multiple Families and timing information. For more information on these two functions refer to the Appendix or type *help getpv*. The suffixes for the database access functions are: pv – process variable, am – analog monitor (or any monitor), and sp – setpoint.

1. getpv – get a group of EPICS process variables (pv)
2. setpv – set a group of EPICS process variables (pv)
3. steppv – step a group of EPICS process variables (pv)
4. getam – get a group of monitors (am)
5. getsp – get a group of setpoints (sp)
6. setsp – set a group of setpoints (sp)
7. stepsp – step a group of setpoints (sp)
8. ramppv – ramp a set of EPICS process variables (pv)

9. switch2sim – changes family in online mode to simulate mode
10. switch2online – changes family in simulate mode to online mode
11. switch2physics – get/set family in physics units
12. switch2hw – get/set family in hardware units

Conversion Functions – These functions convert between naming conventions.

1. channel2dev – convert channel names to device list
2. channel2family – convert family to channel names
3. channel2common – convert common names to channel names
4. channel2handle – convert channel names to MCA handles
5. common2dev – convert device list for set of common names
6. common2channel – convert common names to channel names
7. common2family – convert common names to family names
8. common2handle – convert common names to MCA handles
9. dev2elem – convert element list to device list
10. elem2dev – convert device list to element list
11. family2dev or getlist – convert family to device list
12. family2channel – convert family to channel names
13. family2common – convert family name to common names
14. family2handle – convert family name to MCA handles
15. family2status – get the status information about a device (1-in operation, 0-removed from service)

Data Retrieval Functions – These functions retrieve data from various sources. Use *getfamilydata* to get family and control system parameters. Use *getphysdata* to get physics data. And use *getdata* to retrieve data from a file. Most of the other functions listed below are just aliases of these functions.

1. getdata – get data structure from a file
2. getfamilylist – returns the list of families
3. getfamilytype – returns a list of family types
4. getfamilydata – get specified data field for a family
5. getgolden – get the set of golden values for a family
6. getoffset– get the offset value for a family
7. getphysdata – get calibration data
8. getrespmat – get response matrix data from a file
    - getbpmresp
    - gettuneresp
    - getchroresp
    - getdispresp
9. getspos – get s-position in the ring for specified set of elements
10. getsigma – gets the standard deviation of the monitor (pre-measured)
11. gettol – get the allowed tolerance between the setpoint and monitor
12. isfamily – check for valid family name
13. minsp/maxsp – get minimum/ maximum setpoint for set of elements
14. setphysdata – set calibration data
15. setfamilydata – set data field for a family

Save/Restore Functions

1. getmachineconfig – get/save the lattice magnets and orbit (to a file or variable)
2. setmachineconfig – sets all lattice magnets (from a file or variable)

5. SHORTCUT FUNCTIONS

Shortcut functions are alias functions used to reduce number of parameters required in the function call. Two examples listed above include *getam* and *getsp*. These functions call *getpv* without an explicit request for monitor or setpoint. *setsp* and *stepsp* work in a similar mode.

Other shortcut functions include:

1. getbpm – general BPM function
2. getdcct – get electron beam current
3. getrf/setrf – get/set RF frequency
4. gettune – get storage ring tune
5. getx – get horizontal beam position
6. gety – get vertical beam position

Note: some of these shortcut functions many belong in the "special" functions category which is discussed in the next section. For instance, if TUNE is a family then *gettune* is just an alias to *getam*('TUNE') . However, making TUNE a family may not make sense for some accelerators, hence, a separate function name has been designated. Using shortcut functions makes it easy to write high level functions in a machine-independent way.

6. SPECIAL FUNCTIONS

Some devices do not fit neatly into the Accelerator Object method so individual functions are required to access the data. For instance, a family may not be one process variable per device or the data does not come from EPICs at all. The Accelerator Object file can usually be organized to still use the family method (see Appendix: Creating Families) or one can bypass the Accelerator Object entirely. For instance, the storage ring tune can be obtained from a special function and still be made into a family. Exactly how the Accelerator Object is setup and how the function calls are made will depend on what is appropriate for the specific machine or experiment. Special functions that do not refer to the Accelerator Object structure are likely to be machine-dependent; hence it is best to put them in a directory separate from the machine-independent Middle Layer functions. Examples include:

1. adcquantization – return the LSB of the ADC for a channel
2. dacquantization – return the LSB of the DAC for a channel
3. getid/setid – get/set the insertion device gap vertical position and velocity
4. getepu/setepu – get/set the EPU channels for horizontal motion
5. getlifetime – get beam lifetime (if lifetime channel exists, use measlifetime if not)
6. getrfcavitytemperature/setrfcavitytemperature
7. getrfpower / setrfpower
8. getscrap/setscrap – get/set the scraper position and runflag
9. getbpmv – get the raw BPM button voltages

10. power supply functions: on/off, ready, and reset (it is best to make these control a different field in the power supply family)
11. temperature monitors
12. vacuum pressure functions

7. MACHINE PHYSICS FUNCTIONS

The purpose of the basic Middle Layer functions is to provide support for accessing the accelerator hardware and model (simulator). The next step is to use this library to generate basic accelerator physics support. This section should continually expand with the life of the accelerator and as more accelerator facilities adopt the Middle Layer.

General Machine Physics Functions

1. amps2mm / mm2amps – converts a change in a corrector magnet from amps to max orbit change (based on response matrix)
2. bpm2orbit – converts the BPM reading on either side of the insertion device straight to position and angle at the insertion device center.
3. bend2gev – converts bend magnet current to electron beam energy (option to include the additional energy shift due to correctors)
4. buildlocoinput – assembles a LOCO input file
5. bumpinj – creates an injection bump
6. getenergy – returns the operating (desired) beam energy
7. getmcf – return the momentum compaction factor
8. gev2bend – converts electron beam energy to bend magnet current
9. hw2physcis – convert between hardware and physics units
10. monitor – monitors channels
11. measlifetime – computes the lifetime using beam current measurements (lease squares fit)
12. measdisp – measure the dispersion function
13. measchro – measure the storage ring chromaticity (uses SF & SD)
14. measbpmsigma – measures the standard deviation of the BPMs
15. monbpm – monitor, plot, and compute basic statistics like standard deviations on the BPMs
16. monmags – monitor, plot, and compute basic statistics like standard deviations on the storage ring magnets
17. physcis2hw – convert between physics and hardware units
18. raw2real – converts control system data (raw) to calibrated data (real)
19. real2raw – converts calibrated data (real) to control system data (raw)
20. setchro – sets the storage ring chromaticity
21. stepchro – steps the storage ring chromaticity
22. settune – sets the storage ring tune (uses quadrupoles and tune measurement)
23. steptune – steps the storage ring tune (uses quadrupoles)
24. turnoff – slowly ramps an entire magnet family off (for instance, sextupoles)

Response Matrix Functions

1. getrespmat – get a response matrix from a file
2. getbpmresp – get a BPM response matrix from a file
3. gettuneresp – get a tune response from a file

4. getchroresp – get a chromaticity response from a file
5. getrespmat – general response matrix retrieval
6. measrespmat – measure a response matrix (general function)
7. measbpmresp – measure a response matrix for the BPM family
8. measdispresp – measure the dispersion response matrix
9. measchroresp – measure the chromaticity response matrix
10. measTuneResp – measure a response matrix for the quadrupole family
11. plotbpmresp – plot the response matrix and analyzes symmetry

Insertion Device Compensation Functions

1. ffgettbl – gets a new insertion device feed forward table
2. fftest – tests the current feed forward table
3. ffanal – analyzes an existing feed forward table

System Checking

1. getrate – measures the data rate for a channel (channel must be noisy, ie, changes every update)
2. checkbpms – checks if the BPMs are functioning (based on response matrix)
3. checkmags – checks the magnets (setpoint, tolerance, on/off, etc)
4. checkorbit – checks the orbit (based on golden orbit)
5. magstep – checks the step response of a corrector magn
6. checkmachine – look for errors in the storage ring
    a. Power supply problems
    b. Orbit errors
    c. Temperatures
    d. Vacuum
    e. …

Simulator Functions

The Middle Layer can run independent of the AT simulator. However, it is can be very useful to use the AT model with the Middle Layer. *Switch2sim*/*switch2online* and the mode flag are usually used to access the AT model from the Middle Layer (or just use AT commands directly). However, it is helpful to have commands that always use the AT model. When using these commands one must make sure the simulator has the proper setpoints. This can be accomplished using the mode flag or in one step with *machine2sim* (as described below).

1. modelbeta – beta function of the model
2. modelchro – chromaticity function
3. modeldisp – dispersion function
4. modelmcf – returns the momentum compaction factor of the model
5. modeltune – returns the model tune
6. modeltwiss – returns model twiss functions
7. machine2sim – copy all the machine setpoints to the simulator
8. sim2machine – copy all the simulator setpoints to the machine

Miscellaneous Functions

1. addlabel – adds a label to an arbitrary location on a figure window
2. appendtimestamp – appends a d nd time string to the input

3. gettime – time in seconds (Note: starting time is different on PC vs Unix)
4. popplot – pops the current axes into a new figure window
5. sleep – delay in seconds
6. xaxis – just changes the horizontal axis
7. xaxiss – change all the horizontal axis in a figure
8. yaxis – just changes the vertical axis
9. yaxiss – change all the vertical axis in a figure
10. zaxis – just changes the z-axis in a 3d plot

8. DATA MANAGEMENT

Managing the all the data required for the setup and control of an accelerator becomes a fulltime job. Online databases are helpful but it takes cooperation and coordination of all the member of the physics, controls, and instrumentation groups to really do it well. And centralized method of data handling is usually not available on day one of operations and chaos and confusion often sets in. An attempt to mitigate (or deal with) the problem will be presented here. This is by no mean a complete or particularly good solution.

Machine data that is almost static

1. Physics to hardware conversion (however, there is an energy scaling that needs to be applied to this data).
2. Maximum/Minimum setpoints
3. Position of hardware in the ring
4. Magnet hysteresis data (*magnetcoefficients*)
5. …

Machine data that needs to be periodically updated

1. Offset orbit (based on magnet centers)
2. Model calibration data (LOCO output)
3. Golden parameters:
   - Orbit
   - Tune
   - Chromacity
   - Desired setpoints for applications like bump magnets, feedback systems, RF, etc
   - …
4. Magnet lattice save/restore files
5. Response matrices (measured and model)
   - Orbit (corrector to beam position)
   - Tune (quadrupoles to tune)
   - Chromaticity (sextupole to chromacity)
   - Dispersion (corrector to dispersion)
6. Standard deviations of monitors channels (like BPMs and magnets)
7. Insertion device feedforward tables
8. …

Machine data and parameter saves

Although most accelerators have online archiving of all database channels at periodic rates, it is necessary to have separate archiving in Matlab for a number of reasons. For one, it is often more convenient to save data directly then it is to remember the time and retrieve that data from an archived database (assuming the granularity of the archived data is even acceptable). And, accelerator parameters like dispersion and chromaticity are not database channels; it requires an experiment to determine them. Typical physics data which is often archived include:

1. Orbit
2. Tune
3. Dispersion
4. Chromaticity
5. Response matrices
6. Beta function
7. …

See Append: Data Storage, for information on where the data is saved.

Data Structures

It is convenient to save data with a consistent format. When using *getpv, getam, getsp, getx, gety, getrf, getdisp, getchro, etc* with the 'Struct' option, the following structure is returned.

Data:	Data (vector)
FamilyName:	Family name (string)
Field:	Field to set of get (string)
DeviceList:	Device list (2-column matrix)
Mode:	'Online' or 'Simulator'
Status:	1-device ok, 0-device bad (vector)
t:	time when the measurement started (vector)
tout:	time when the measurement completed (vector)
TimeStamp:	Matlab clock when the data was measured
GeV:	Energy [GeV]
Units:	'Physics' or 'Hardware'
UnitsString:	Actual units (string)
DataDescriptor:	Description (like, 'Horizontal Orbit', 'Vertical Dispersion)
CreatedBy:	Name of the function that created the data (string)

When possible, it is best to use this data structure as much as possible to minimize the learning curve when sharing data. Many functions have a 'Archive' option which will automatically save a data structure to a subdirectory of <DataRoot> (use DataRoot=getfamilydata('Directory', 'DataRoot') to view the location of 'DataRoot').

Response matrix data as return *measrespmat, measbpmresp, etc,* have a slightly different structure. See *help measrespmat* or the next section for more details. Chromaticity and dispersion data structure are essentially response matrix structures with a few extra fields required to define that particular measurement (see *help measdisp* and *meachro* for details).

9. RESPONSE MATRIX MEASUREMENT/SAVE/RESTORE

The function *measrespmat* is the most general function for measuring a response matrix between an actuator family and a set of monitor families.

```
>> help measrespmat

For family name, device list inputs:
S = measrespmat(MonitorFamily, MonitorDeviceList, ActuatorFamily, ActuatorDeviceList,
                ActuatorDelta, ModulationMethod, WaitFlag, ExtraDelay)

For data structure inputs:
S = measrespmat(MonitorStruct, ActuatorStruct, ActuatorDelta, ModulationMethod,
                WaitFlag, ExtraDelay)

Inputs:
  MonitorFamily       - AcceleratorObjects family name for monitors
  MonitorDeviceList   - AcceleratorObjects device list for monitors (element or device)
  or MonitorStruct can replace MonitorFamily and MonitorDeviceList

  ActuatorFamily      - AcceleratorObjects family name for actuators
  ActuatorDeviceList  - AcceleratorObjects device list for actuators
                        (element or device) or ActuatorStruct can replace
                        ActuatorFamily and ActuatorDeviceList

  ActuatorDelta    - AcceleratorObjects family name for monitors
  ModulationMethod - Method for changing the ActuatorFamily
                     bipolar' changes the ActuatorFamily by +/- ActuatorDelta
                              on each step {default}
                     unipolar' changes the ActuatorFamily from 0 to ActuatorDelta
                               on each step
  WaitFlag         - (see setpv for WaitFlag definitions) {default: -1}
  ExtraDelay       - extra time delay [seconds] after a setpoint change to wait before
                     reading the MonitorFamily {default: 0}

Output:
  S = the response matrix.

  If 'struct'  is an input, the output will be a response matrix structure
                           {default for data structure inputs}
  If 'numeric' is an input, the output will be a numeric matrix
                           {default for non-data structure inputs}

Notes:
1. If MonitorFamily and  MonitorDeviceList are cell arrrays, then S is a cell array
   of response matrices.
2. ActuatorFamily, ActuatorDeviceList, ActuatorDelta, ModulationMethod, WaitFlag are
   not cell arrrays.
3. If ActuatorDeviceList is empty, then the entire family is change together.
4. Bipolar mode changes the actuator by +/- ActuatorDelta/2
5. Unipolar mode changes the actuator by ActuatorDelta
6. Return values are MonitorChange/ActuatorDelta (normalized)
7. When using cell array inputs don't mix structure data input with non-structure data

Examples:
1. 2x2 tune response matrix for QF and QD families (delay for tune matrix will need
   to be addjusted):
       TuneRmatrix = [measrespmat('TUNE',[1;2],'QF',[],2,'unipolar') ...
                      measrespmat('TUNE',[1;2],'QD',[],2,'unipolar')];

2. Orbit response matrix for all the horizontal correctors (+/-1 amp kick size):
   Smat = measrespmat({'BPMx','BPMy'}, {getlist('BPMx'),getlist('BPMy')}, 'HCM', ...
                                  getlist('HCM'),ones(size(getlist('HCM'),1),1), ...
                                  'bipolar',-2);
   The output is stored in a cell array.  Smat{1} is the horizontal plane and Smat{2}
   is the vertical cross plane.
```

```
For struct outputs,
Smat = measrespmat({'BPMx','BPMy'}, {getlist('BPMx'),getlist('BPMy')}, 'HCM',
       getlist('HCM'),ones(size(getlist('HCM'),1),1),'bipolar',-2,'struct');
```

The response matrix, Rmat, is stored in the following format:

Data:	[Response matrix]
Monitor:	[Data Structure for the Monitor]
Actuator:	[Data Structure for the Actuator]
ActuatorDelta:	[Delta change in the Actuator]
GeV:	Energy
TimeStamp:	[2003 6 17 20 27 25.0770]
DCCT:	Beam current
ModulationMethod:	'bipolar' or 'unipolar'
WaitFlag:	WaitFlag
ExtraDelay:	ExtraDelay
DataType:	'Response Matrix'
CreatedBy:	'measrespmat'

Every accelerator uses a number of response matrices for daily operation and physics shifts. (Note: dispersion and chromaticity also have response matrix like structures.) Since these matrices are generated many times a year, special functions have been created to force a consistent data format, deal with bad devices, and archiving of these matrices. The basic response matrix retrieval functions are the following.

- getbpmresp – BPM response matrices
- gettuneresp – TUNE response matrices
- getchromresp – Chromacity response matrices
- getdispresp – Dispersion response matrices (corrector magnets to dispersion)
- getrespmat – General response matrix retrieval

The general function for extracting saved response matrix data is *getrespmat*.

S = getrespmat(BPMFamily, BPMDevList, CorrFamily, CorrDevList, FileName, GeV)

This function is quite versatile at finding response matrix variables. The data will be extracted from file FileName. If no FileName is specified, this function will search through the list of default response matrix file names as specified in AcceleratorData.Resp.Files, e.g. AD.Resp.Files = {'BPMRespMatrix', 'TuneRespMatrix'}. *getrespmat* will then search through all variables in the file (and through each cell array variable if it exists) for the existence of a response matrix structure with the proper Monitor and Actuator field names. Data structure inputs are also allowed. For example, the following commands will get the orbit, corrector values, and response matrix for say an orbit correction function.

```
>> HCMsp = getsp('HCM', 'Struct');
>> BPMam = getam('BPMx', 'Struct');
>> R = getrespmat(BPMam, HCMsp);
```

10. HIGH LEVEL FUNCTIONS

The major reasons for developing the Middle Layer software is to make writing scripts and high level functions relatively easy. The following example is a horizontal global orbit correction routine for the ALS using a singular valued decomposition (SVD) method where only the first 24 singular values of the matrix are used.

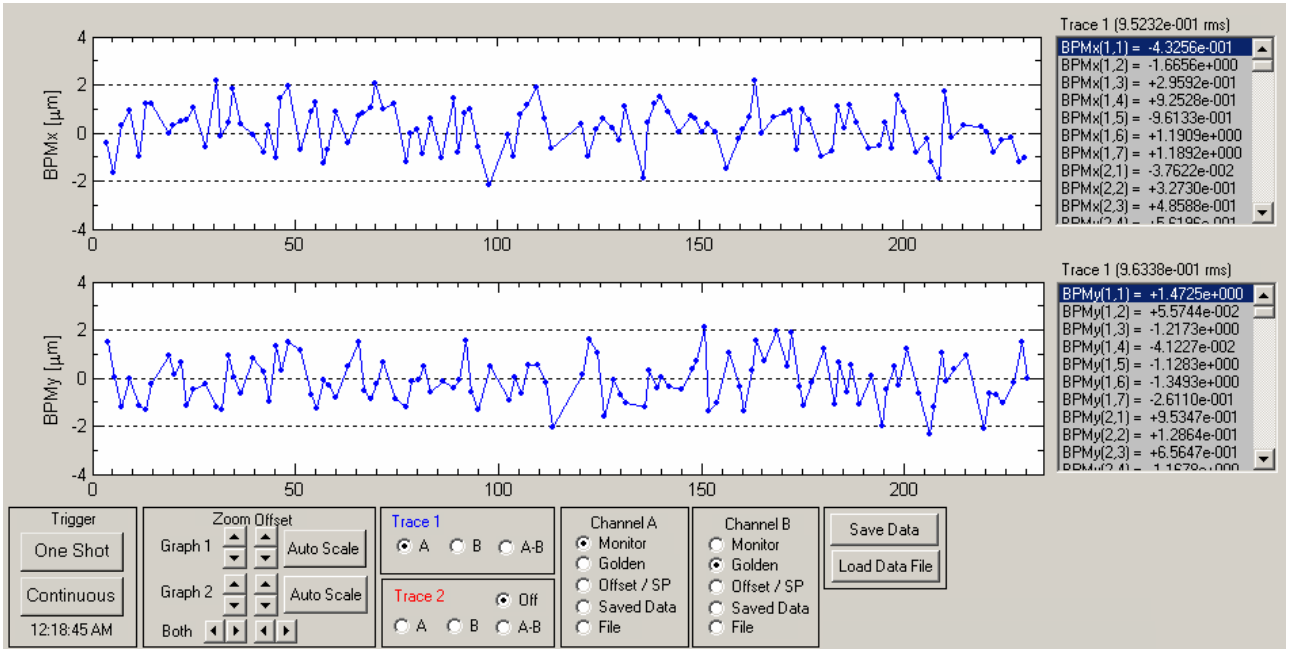
```
Sx=getrespmat('BPMx',[ ],'HCM',[ ]);                % Get the proper response matrix
X = getx;                                           % Gets all 96 horizontal BPMs (96x1 vector)
Ivec = 1:24;                                        % Use singular vectors 1 thru 24
[U, S, V] = svd(Sx);                                % Computes the SVD of the response matrix, Sx(96x94)
DeltaAmps = -V(:,Ivec)*((U(:,Ivec)*S(Ivec,Ivec))\X) ;  % Find the corrector changes (94x1 vector)
stepsp('HCM', DeltaAmps);                           % Changes the current in all 94 horizontal corrector magnets
plot(1:96, X, 'b', 1:96, getx, 'r');                % Plot new orbit
```

High level functions and applications

1. findrf – one method of finding an "optimal" RF frequency based on dispersion
2. findqfa – optimizes the setpoint of the quadrupole in the center of arcs.
3. goldenpage – displays the important settings and setpoints (like tune, chromaticity, etc)
4. plotfamily – general plotting GUI for families (see section 11 for details)
5. rmdisp – adjusts the RF frequency to remove the dispersion component of the orbit by fitting the orbit to the dispersion orbit (fitting the mean is optional).
6. setorbit – general purpose global orbit correction function
    - SVD (singular value selection based on user input vector or max/min ratio)
    - BPM weights
    - with or without RF (ie, dispersion included as a column of the response matrix)
    - measured or model response matrix
    - number of iterations user selectable
    - absolute or incremental orbit change
7. srcycle – cycles the storage ring magnets
8. srramp – energy ramping (with beam) of the storage ring
9. setlocalbump – general purpose local bump function
10. quadcenter, quadplot, quaderror – finds the quadrupole center of one magnet at a time
11. setorbitquadcenter – corrects the orbit to the quadrupole centers
12. srsetup – launch pad for setup applications
13. srcontrol – GUI for storage ring operations
14. scanbpms – local bump scan in the straight sections checking the linearity of the BPMs
15. scanpipe – used for scanning the electron beam in the straight sections and checking the lifetime (physical aperture)

11. HIGH LEVEL APPLICATIONS

1. DISPLAY (*PLOTFAMILY*)



2. ORBIT CORRECTION

(To be written)

3. BEAM BASED ALIGNMENT

a. Single magnet
b. All quadrupoles

(To be written)

12. ARCHIVED DATA RETRIEVAL

Retrieving archived or history buffer information (SPEAR only)

1. getrdbdata – basic call the Oracle rdb database
2. gethist – gets data from the history buffer
3. family2history – converts a family name to a history buffer name

13. MIDDLE LAYER FOR ACCELERATOR SETUP & OPERATIONS

(To be written)

APPENDICES

Appendix I: Software Installation

1. EPICS software must already be working on your computer.

2. Install the Middle Layer and AT software as well as applications like LOCO or orbit correction GUIs. Call the root directory location of these files MLPATH.

3. Set the Matlab path. One way to do it is put the following code in the *startup*.m file.
   run MLPATH\acceleratorcontrol\setpathspear3
   or setpathcls, setpathals depending which accelerator is being controlled/simulated. Some modifications to these files may need to be made depending on what applications are installed.

4. Running Matlab functions in standalone mode
   A few applications have been compiled to run standalone. In order for this to work the Matlab, AT, and MCA dll's must be on the computers path, hence,
   MLPATH \bin\win32
   needs to be put on the windows systems path.

5. Note: when running AT, the windows environmental variable ATROOT needs to be defined as well.

Appendix II: General Programming Guidelines

1. **Function Case**: All functions should be lower case. The fact that PC's are not case sensitive on function name but Unix is can cause confusion.

2. **Function Names:** Don't use too common a name for a new function and first check that it doesn't already exist (>> *which* FunctionName).

3. **Family Names:** Applications should try to be rewritten with generic families in mind. Hopefully families can be changed (or accelerators changed) without breaking the application.

4. **Directory Control:** The directory tree should not be hardcoded into an application. The root of the data directory can be found using *getfamilydata*('Directory', 'DataRoot'). New data should be saved to a subdirectory by type, date, and time.

5. **Online Help:** Just to keep some consistence to the online help, the recommended layout is the following.

```
%FUNCTIONNAME - Description
%
%  [Out1, Out2, …] = functionname(Input1, Input2, …)
%
```

```
% INPUTS
% 1.
% 2.
%
% OUTPUTS
% 1.
% 2.
%
% NOTES
% 1.
% 2.
%
% EXAMPLES
% 1.
% 2.
%
% Written by ____
```

6. **Error Handling:** Instead of using error flags, usually the Matlab *error* or *mexerror* functions have been used in the Middle Layer. This prevents having to error check after every function executes. However, when a more graceful error handle method is required, use the try/catch/end statements.

Appendix III: Creating Families

Although the four basic monitor and setpoint functions (*getam*, *getsp*, *setsp*, *stepsp*) are most commonly used with families, there are really only two things one really needs to do to a data channel—get and set. Hence, all Middle Layer functions eventually get routed threw two functions—*getpv* and *setpv*. PV stands for process variable. *getpv* and *setpv* in turn call MCA. All the information for these functions comes from a structure called the Accelerator Object (AO), which is stored in the application data of the command window. The AO has a number of sub-structures. The first field of the AO is the family name – AO.(Family). AO.(Family) is also a structure which has all the necessary information for *getpv*, and *setpv*. The format of the sub-structure is as follows.

Main family structure:

AcceleratorObject.(Family)

FamilyName:	Family Name ('BPMx', 'HCM', etc.) (must be unique)
FamilyType:	Type of hardware (BPM, COR, QUAD, BEND, etc)
IsMember:	(to be defined)
Status:	1 for good status, 0 for bad status
ElementList:	Column vector
Monitor:	Structure shown below
Setpoint:	Structure shown below
CommonNames:	String matrix of common names
Position:	Column vector of longitudinal position along the ring [meters]
AT:	Structure for the AT simulator (optional)
Golden:	Structure of “optimum” values (optional)

Sub-family structure for monitors:

AcceleratorObject.(Family).Monitor

DataType:	'Scalar' or 'Vector' depending on the EPICS type
DataTypeIndex:	Sub-indexing of the EPICS record for DataType='Vector' (optional)
Mode:	'Online', 'Simulator', 'Manual' or 'Special'
SpecialFunction:	Function name if Mode = 'Special'
Units:	What units to work in: 'Hardware' or 'Physics'
HW2PhysicsFcn:	Hardware to physics units conversion function (see Appendix VI)
HW2PhysicsParams:	Hardware to physics units conversion parameters
HWUnits:	String name of the hardware units
PhysicsUnits:	String name of the physics units
ChannelNames:	String matrix of monitor channel names
Handles:	Handle vector (NaN if a channel has not been opened yet)

Sub-family structure for setpoints:

AcceleratorObject.(Family).Setpoint

DataType:	'Scalar' or 'Vector' depending on the EPICS type
DataTypeIndex:	Sub-indexing of the EPICS record for DataType='Vector' (optional)
Mode:	'Online', 'Simulator', 'Manual' or 'Special'
SpecialFunction:	Function name if Mode = 'Special'
Units:	What units to work in: 'Hardware' or 'Physics'
Physics2HWFcn:	Physics to hardware units conversion function (see Appendix VI)
Physics2HWParams:	Physics to hardware units conversion parameters
HWUnits:	String name of the hardware units
PhysicsUnits:	String name of the physics units
ChannelNames:	String matrix of setpoint channel names
Handles:	Handle vector (NaN if channel has not been opened yet)
Range:	[Min Max] range for the setpoint (two column matrix)
Tolerance:	Tolerance column vector for SP-AM comparison

The number of rows of DeviceList, ElementList, CommonNames, Positions, Range, and Tolerance must be equal. The number of rows of ChannelNames and Handles must also equal DeviceList if DataType='Scalar'. For DataType='Vector', ChannelNames and Handles can only have one row but the output of mcaget (or DataTypeIndex, if used) must equal the number of rows of DeviceList. It is relatively easy to create a family. It is probably wise to agree on a set of family names for an accelerator. Otherwise sharing software becomes difficult.

The number for subfields in the AO (like Monitor and Setpoint) depends on the type of family. And any field name can be used. However, the Monitor and Setpoint names have reserved meaning for the functions *getam*, *getsp*, *setsp*, and *stepsp*. It is highly advised but it is not necessary to use these methods. *getpv* and *setpv* are very similar to *getam* and *setsp* except that the subfield name of the AO data structure is a required input. *getam, getsp, setsp* and *stepsp* are basically only shortcut functions to *getpv* and *setpv* where the field input is either Monitor or Setpoint. Usually it is desirable to hide this field name from the Matlab user. However, if it is appropriate to associate other channels with the family then more fields can be added to the AO.

For instance, an on/off control for a power supply could be added as AO.(Family).OnOffSetpoint. One would get the data by *getpv(Family, 'OnOffSetpoint').* It would not be accessible via *getsp* and *setsp*. If it is more desirable to create an On/Off family name, then one could create a separate family for on/off control (like HCMonoff) and use the standard *getam, getsp,* and *setsp* functions (or create new aliases). It's a matter of taste.

Additional field when using the AT simulator:

AcceleratorObject.(Family).AT (simulator only)

ATType:	'X', 'Y', 'BPMX', 'BPMY', 'HCM', 'VCM', else ATParameterGroup is used
ATParameterGroup:	Parameter group
ATIndex:	Column vector of AT indexes

The AT physics simulator and the online machine can exist together by setting up the accelerator object properly. It is not required to do this for the Middle Layer to function. It is required in order to have the Simulator mode or use any of the AT functions — *getmcf*('Model') or *modeltwiss*. *getpv* and *setpv* check if the Mode is 'online', 'simulate', 'manual', or 'special.' If in simulate mode, then ATType can be 'X', 'Y', 'BPMX', 'BPMY', HCM', 'VCM'; or the ATParameterGroup field is used (see *help setparamgroup* for details).

Notes about the simulator

1. Cavity must be on with THERING{ }.PassMethod = 'ThinCavityPass' for RF frequency related functions to work.
2. RF frequency does not change the tune.
3. The physics units must match AT units in order for the simulator to work properly.
4. Be careful with AT models with split magnets. The ATparameter group must be setup properly.
5. Simulator only works on Monitor and Setpoint fields. This limitation could be removed with better use of the parameter group in AT.
6. Channel and common name methods do not work in simulator mode. It is possible to search for channel names in the AO structure before setting and checking for simulate mode.

Appendix IV: GET and SET Functions

There are 4 main functions for getting and setting data by family.

1. getam – gets the monitor values for any family
2. getsp – gets the setpoint values for any family
3. setsp – sets the setpoint values for any family
4. stepsp – delta change in setpoint for any family

SP and AM functions were recreated to allow for pairing setpoints and monitors if the natural pairing exists. For instance, *getsp('HCM')* gets the current setpoint of the horizontal corrector magnets and *getam('HCM')* gets the monitors values. Keeping track of the channels names is done by the Middle Layer. Information for each function can be found using *help* in Matlab. Notice that there are three different input schemes for each function—family-device list, family-common name, and channel name.

The above function call the following more general functions.

1. getpv – gets the monitor values for any family
2. setpv – sets the setpoint values for any family
3. steppv – delta change in setpoint for any family

GETPV

```
>> help getpv

GETPV - Get an EPICS process variable (or AT simulated channel)

   FamilyName/DeviceList Method
   [AM, tout, ErrorFlag] = getpv(Family, Field, DeviceList, t, FreshDataFlag, TimeOutPeriod)
   [AM, tout, ErrorFlag] = getpv(DataStructure, t, FreshDataFlag, TimeOutPeriod)

   ChannelName Method
   [AM, tout, ErrorFlag] = getpv(ChannelName, t, FreshDataFlag, TimeOutPeriod)

   CommonName Method
   [AM, tout, ErrorFlag] = getpv(Family, Field, CommonName, t, FreshDataFlag, TimeOutPeriod)

   INPUTS
   1. Family = Family Name
               Data Structure
               Channel Name
               Accelerator Object
               For CommonNames, Family=[] searches all families
               (or Cell Array of inputs)
   2. Field = Accelerator Object field name ('Monitor', 'Setpoint', etc) {'Monitor'}
              Cell Array of fields
   3. ChannelName = Channel access channel name (scalar or vector outputs),
                    Matrix of channel names (scalar outputs only)
                    Cell array of channel names
   4. CommonName = Common name (scalar or vector outputs),
                   Matrix of common names (scalar outputs only)
                   Cell array of common names
   5. DeviceList = [Sector Device #] or [element #] list {default or empty list: whole family}
                   Cell array of DeviceLists
                   Note: if input 1 is a cell array then DeviceList must be a cell array
   6. t = Time vector (t can not be a cell) {default: 0}
   7. FreshDataFlag =  0   -> Return after first measurement {default}
                      else -> Return after FreshDataFlag number of new measurements have been read
                      ie, getpv('BPMx',[1 1],0,2) measures the orbit then continues to read the orbit
                          until 2 new orbits have been measured and returns the last measurement.
   8. TimeOutPeriod = Time-out period when waiting for fresh data {10 seconds}
   9. 'Struct' will return a data structure {default for data structure inputs}
      'Numeric' will return numeric outputs {default for non-data structure inputs}
   10. 'Physics'  - Use physics  units (optional override of units)
       'Hardware' - Use hardware units (optional override of units)
   11. 'Online' - Get data online (optional override of the mode)
       'Model'  - Get data from the model (optional override of the mode)
       'Manual' - Get data manually (optional override of the mode)

   OUTPUTS
   1. AM   = Monitor values (Column vector or matrix where each column is a data point in time)
   2. tout = Time when measurement was completed (row vector)
   3. ErrorFlag =  0   -> no errors
                  else -> error or warning occurred

   The output will be a data structure if the input is a data structure or the word 'struct'
   appears somewhere on the input line.

   NOTES
```

```
   1. For data structure inputs:
      Family     = DataStructure.FamilyName
      Field      = DataStructure.Field
      DeviceList = DataStructure.DeviceList
      Units      = DataStructure.Units  (Units can be overridden on the input line)
      (The Mode field is ignored!)

   2. diff(t) should not be too small.  If the desired time to collect the data is too
      short then the data collecting will not be able to keep up.  Always check tout.
      (t - tout) is the time it took to collect the data on each iteration.

   3. An easy way to measure N averaged monitors is:
      PVmean = mean(getpv(Family,DeviceList,1:N)')';   % 1 second between measurements

   4. Channel name method is always Online!

   5. For cell array inputs:
      a. Input 1 defines the size of all cells
      b. All of the cell array inputs must be the same size or size=[1 1]
      c. t (if used) can not be a cell!
      d. FreshDataFlag and TimeOutPeriod can be a cell but they don't have to be

   See also getam, getsp, getx, gety, setpv

   Written by Greg Portmann
```

SETPV

```
>> help setpv

SETPV - Absolute setpoint change via MATLAB channel access or AT simulator

   FamilyName/DeviceList Method
   ErrorFlag = setpv(FamilyName, Field, NewSP, DeviceList, WaitFlag)
   ErrorFlag = setpv(DataStructure, WaitFlag)

   ChannelName Method
   ErrorFlag = setpv(ChannelName, NewSP)

   CommonName Method
   ErrorFlag = setpv(FamilyName, Field, NewSP, CommonNames, WaitFlag)

   INPUTS
   1. Family = FamilyName
               Data Structure
               Channel Name
               AcceleratorObject
               Use Family=[] in CommonName method to search all Families
              (or Cell Array of inputs)

   2. ChannelName = Channel access channel name (scalar or vector inputs)
                    Matrix of channel names (scalar inputs only)
                    Cell array of channel names

   3. CommonName = CommonNames (scalar or vector inputs)
                   Matrix of CommonNames (scalar inputs only)
                   Cell array of CommonNames
                   Must use Family=[] in to search all Families

   4. Field = AcceleratorObject Field name ('Monitor', 'Setpoint', etc) {'Monitor'}
              If Family is a cell array then Field must be a cell array

   5. NewSP = New Setpoints or cell array of Setpoints

   6. DeviceList = ([Sector Device #] or [element #]) {default or empty list: whole family}
```

```
               Note: all numerical inputs must be column vectors

7. WaitFlag = 0    -> return immediately {SLAC default}
              > 0  -> wait until ramping is done then adds an extra delay equal to WaitFlag
              = -1 -> wait until ramping is done
              = -2 -> wait until ramping is done then adds an extra delay for fresh data
                      from the BPMs  {ALS default}
              = -3 -> wait until ramping is done then adds an extra delay for fresh data
                      from the tune measurement system
              = -4 -> wait until ramping is done then wait for a carriage return
              else -> wait until ramping is done
              Note: change WaitFlag default in setpv.m and BPM delay in the Accelerator
                    Data structure

8. ErrorFlag = 0 -> OK
              -1 -> MATLAB Channel Access error
              -2 -> SP-AM warning, i.e. setpoint minus analogmonitor not within
                                        tolerance (only if WaitFlag=1)

9. 'Physics'  - Use physics  units (optional override of units)
   'Hardware' - Use hardware units (optional override of units)

10. 'Online' - Set data online (optional override of the mode)
    'Model'  - Set data on the model (optional override of the mode)
    'Manual' - Set data manually (optional override of the mode)

NOTES
1. For data structure inputs:
   Family     = DataStructure.FamilyName
   Field      = DataStructure.Field
   NewSP      = DataStructure.Data
   DeviceList = DataStructure.DeviceList
   Units      = DataStructure.Units  (Units can be overridden on the input line)
   (The Mode field is ignored!)

2. The number of colomns of NewSP and DeviceList must be equal,
   or NewSP must be a scalar.  If NewSP is a scalar, then all
   devices in DeviceList will be set to the same value.

3. When using cell array all inputs must be the same size cell array
   and the output will also be a cell array.  Field and WaitFlag can be
   cells but they don't have to be.

4. For Familys and AcceleratorObject structures unknown devices or elements are ignored.

5. ChannelName method is always Online!

6. For cell array inputs:
   a. Input 1 defines the maximum size of all cells
   b. The cell array size must be 1 or equal to the number of cell in input #1
   c. WaitFlag can be a cell but it doesn't have to be

7. WaitFlag
   a. If no change is seen on the AM then an error will occur.  The tolerance for this
      may need to be changed depending on the accelerator (edit the end of this function
      to do so)
   b. The delay for WaitFlag = -2 is in the AD.  It is often better to use the
      FreshDataFlag when getting BPM data but the data must to noisy for this to work.

EXAMPES
1. setpv('HCM','Setpoint',1.23);              Sets the entire HCM family to 1.23
2. setpv({'HCM','VCM'},'Setpoint',{10.4,5.3}); Sets the entire HCM family to 10.4 and
                                                          VCM family to 5.3
3. setpv('HCM','Setpoint',1.23,[1 3]);    Simple DeviceList method
4. setpv('HCM','Setpoint',1.23, 3);       Simple ElementList method
5. setpv(AO('HCM'),'Setpoint',1.5,[1 2]);     If AO is a properly formatted
                                         AcceleratorObject Structure then this sets the
```

```
                                          1st sector, 2nd element to 1.5
6. setpv('HCM','Setpoint',1.23,'1CX3');   CommonName method with Family specified
                                                    (spear3 naming convection)
7. setpv([],'Setpoint',1.23,'1CX3');      CommonName method without Family

See also getam, getsp, getpv, setsp, steppv, stepsp

Written by Greg Portmann
```

There is error checking on the inputs to all functions. However, the error checking is not meant to complete. Basically, it would require too much computer time and makes the code less readable.

Tuning the Middle Layer for each accelerator:
1. *setpv* using the waitflag = -2: the BPM delay needs to be set properly in the AD.
2. *setpv* using the waitflag may not timeout properly. Edit *setpv* and change the waitflag if statement at the end of the function if it does not work properly.
3. When setting up the accelerator object it is not clear where certain data channel should go. For instance, should getdcct (current), lifetime, and RF be their own function or part of the accelerator object? If they're part of the accelerator object then should they be their own family or made part of a list of common name under a MachineParameter family?

Accelerator objects could be used to accomplish more involved tasks then just monitoring or setpoint changes. One could do local bumps, SVD orbit correction, etc. However, this is beyond the present scope of the Middle Layer goals.

Appendix V: Data Storage

The control system and physics data is stored in a number of different places.

1. The Accelerator Object (AO)
   Purpose: Store family information related to the control system
   Location: Stored in the application of the command window
   Get/Set: *getfamilydata* / *setfamilydata*

2. The Accelerator Data (AD)
   Purpose: Store Middle Layer setup variables
   Location: Stored in the application of the command window
   Get/Set: *getfamilydata* / *setfamilydata*

   1. AD.Machine = accelerator name, like 'ALS' or 'Spear'
   2. AD.Directory.DataRoot = the top of the data directory tree
   3. AD.Resp.Files = cell array of response matrix files, like
      {'respmatbpm_08-06-2002', 'respmattune'}
   4. AD.ATModel = AT lattice filename
   5. AD.BPMDelay = Time to wait before BPM data is fresh
   6. AD.TUNEDelay = Time to wait before TUNE data is fresh
   7. …

3. Physics Data

Purpose: Store physics related data
Location: Stored in a file
FileName = getfamilydata('OpsData','PhysDataFile');
Directory = getfamilydata('Directory','OpsData');
Get/Set: *getphysdata* / *setphysdata*

The physics data file contains a variable, PhysData. The name is not important unless there is more than one variable in the file. It is a structure where each subfield is a family name. Each subfield of family is a particular data type name. The data can be a scalar or a vector equal in length to the number of elements in the family. For instance,

1. PhysData.BPMx.Golden
2. PhysData.BPMx.Gain
3. PhysData.BPMx.Coupling
4. PhysData.BPMx.Offset
5. PhysData.BPMx.Sigma
6. PhysData.BPMx.PinCushion
7. PhysData.BPMx.Dispersion
8. PhysData.BPMx.DesignDispersion
9. PhysData.BPMx.DesignBeta

10. PhysData.HCM.Gain
11. PhysData.HCM.Offset
12. PhysData.HCM.Coupling

13. PhysData.Tune.Golden
14. PhysData.Chro.Golden

makephysdata will create a default data file with all BPMs and magnets. Beware, it also will overwrite an existing physics data file.

Appendix VI: Hardware and Physics Units

Process variables in EPICS typically communicate via Channel Access in hardware units. However, accelerators are typically designed using the physics units for a particular tracking code. The Middleware has been designed to conveniently switch between these two types of units and choose which units should be the default. This section will describe how to configure the AcceleratorObject with the necessary information to accomplish this.

Each family can be configured to operate in either mode by setting the Units field to 'Hardware' or 'Physics'.

AO.(Family).Monitor.Units = 'Hardware' or 'Physics'
AO.(Family).Setpoint.Units = 'Hardware' or 'Physics'

Although it is possible to operate in a mixed mode, it is advisable to pick one mode for all applications. Since there is only one AcceleratorObject per Matlab session all application running

in that session must be using the same units. Note: many functions allow for an override of the Units field on the input line.

Hardware Units

Hardware units are used for applications that manipulate accelerator parameters in terms of the units expected by the process variables (PV) in the EPICS database, like current in amperes for a quadrupole or corrector. Applications that get or set in hardware units require no unit conversions in *getpv* / *setpv*. Hardware units are commonly used for on-line applications like response matrix measurements or empirical orbit correction routines. *getpv* and *setpv* are the main functions that deal with units.

> When a call to *getpv* is made with AO.(Family).Monitor.Units = 'Hardware' the monitored value is returned by *getpv* in 'Hardware' units (like amperes) after *mcaget* is executed.
>
> When a call to *setpv* is made with AO.(Family).Setpoint.Units = 'Hardware' the setpoint value remains in Hardware units (like amperes) before *mcaput* is executed.

Physics Units

Physics units are used when applications calculate accelerator parameters in terms of physics quantities, e.g. K-values for a quadrupole or radians for a corrector, but the EPICS process variables communicate in hardware units. Application can get or set in physics units, however, the low level functions need to convert these values to values to hardware units before the control system PV is set. Once again, *getpv* and *setpv* are the main functions that deal with units conversion.

> When a MATLAB call to *getpv* is made with AO.(Family).Monitor.Units = 'Physics' the parameter to be monitored is converted in getpv from Hardware units (like amperes) to Physics units (like K value) after mcaget is executed.
>
> When a call to *setpv* is made with AO.(Family).Setpoint.Units = 'Physics' the setpoint value is converted from Physics units (like K value) to Hardware units (like amperes) in *setpv* before *mcaput* is executed.

Note that each AcceleratorObject has only one AO.(Family).Monitor.Units and one AO.(Family).Setpoint.Units setting. Individual components within an AcceleratorObject family/field cannot have different units. The different fields (like Monitor and Setpoint) can have different Units, but this is not recommended.

Middleware Conversion Functions

hw2physics and *physics2hw* are Middleware functions that convert between values in 'Hardware' or 'Physics' units for any family. They access family-specific data in the AcceleratorObject and apply the function specified in the HW2PhysicsFcn or Physics2HWFcn field to the values to be converted using parameters found in HW2PhysicsParams and Physics2HWParams. If the function

field (HW2PhysicsFcn or Physics2HWFcn) field does not exist, then it is assumed the conversion is just a gain specified by the parameter field (HW2PhysicsParams and Physics2HWParams). Note: when using the AT simulation with the Middle Layer the physics units must correspond to the units used in AT.

For example, when the AO is set in hardware units, *getsp* returns hardware units and *hw2physics* will convert the QF power supplies currents to physics units (k-value).

```
>> val  = getsp('QF');
>> pval = hw2physics('QF', 'Setpoint', val);
```

To make conversions for specific element within a family, one can specify their ElementList or DeviceList indices. In this case the number of values to convert must match the length of the list or be a scalar (ie, the same for all devices).

```
>> val  = getsp('QF',[1; 2; 4]);
>> pval = hw2physics('QF', 'Setpoint', val, [1; 2; 4]);
```

AcceleratorObjects Setup for Units Conversion

As discussed above, in order for the units conversion to work properly the necessary data must be added to the AcceleratorObject. As shown in Appendix III, the following fields must exist as part of the family description for each subfield (like, Monitor, Setpoint, etc).

1. HW2PhysicsFcn – string name or handle to a mapping function from 'Hardware' to 'Physics' to units. The mapping function itself is a separate m-file or an inline function.

2. HW2PhysicsParams – matrix or cell array of parameters needed by HW2PhysicsFcn.

3. Physics2HWFcn – string name or handle to a mapping function from Physics to 'Hardware' units.

4. Physics2HWParams – matrix or cell array of parameters needed by Physics2HWFcn.

5. PhysicsUnits – optional field with the string name of the physics units.

6. HWUnits – optional field with the string name of the hardware units.

Mapping Functions and Parameters

The mapping functions (or function handles) are stored in HW2PhysicsFcn and Physics2HWFcn fields. Basically, the physics2hw and hw2phyics uses feval with the parameter list to do the conversion. The function fields do not exist, then a simple gain conversion is done using the parameter list.

The parameters for the mapping function are stored in HW2PhysicsParams and Physics2HWParams fields. They must be consistent with the HW2PhysicsFcn and Physics2HWFcn calling syntax and the number of devices in the family. If there are M devices in the family and N parameters expected by the mapping function (in addition to the first argument – value to be converted) then HW2PhysicsParams and Physics2HWParams are either:

1. 1-by-N vector
2. M-by-N matrix
3. M row string matrix
4. N-element cell array whose elements are vectors of length M
5. Empty

For matrices, the number of rows must be equal to the number of devices in the family or equal to 1 (which implies all the devices have the same parameters); and each column gets passed as a separate input to the function specified by HW2PhysicsFcn and Physics2HWFcn. If the matrix is a string matrix, then the rows corresponding to each device is past as one input. If multiple, non-scalar inputs are required, a cell arrays must be used. The contents of each cell are passed to HW2PhysicsFcn or Physics2HWFcn as a separate input. (Cell matrices are fine to use but the added complication is rarely required.) If empty, then no parameters are passed.

Examples

The following examples illustrate a few common ways the AO can be setup for physics to hardware conversions.

1. If HW2PhysicsFcn or Physics2HWFcn do not exist, then HW2PhysicsParams and Physics2HWParams field can contain a constant scaling term. If the physics units for the BPM family is meters and mm for the hardware units, then following setup will do the conversion.

```
AO.(BPMx).FamilyName               = 'BPMx';
AO.(BPMx).Monitor.Units            = 'Hardware';
AO.(BPMx).Monitor.HW2PhysicsParams =  1e-3;
AO.(BPMx).Monitor.Physics2HWParams =  1000;
AO.(BPMx).Monitor.HWUnits          = 'mm';
AO.(BPMx).Monitor.PhysicsUnits     = 'm';
```

2. HW2PhysicsFcn can be an inline function. Using the same example, the following setup will convert mm to meters with a option to add a offset correction.

```
AO.(BPMx).FamilyName               = 'BPMx';
AO.(BPMx).Monitor.Units            = 'Hardware';
AO.(BPMx).Monitor.HW2PhysicsFcn    = inline('P1.*x+P2', 2);
AO.(BPMx).Monitor.Physics2HWFcn    = inline('P1.*x+P2', 2);
AO.(BPMx).Monitor.HW2PhysicsParams =  [1e-3 0];
AO.(BPMx).Monitor.Physics2HWParams =  [1000 0];
AO.(BPMx).Monitor.HWUnits          = 'mm';
AO.(BPMx).Monitor.PhysicsUnits     = 'm';
```

3. HW2PhysicsFcn can be a function (more details on writing map function given below). If the functions amp2k and k2amp convert between K-value and current basic on a polynomial (input 1) with a gain correction (input 2), then the following setup can be used. Note that amp2k and k2amp must be on the path.

```
AO.(QF).FamilyName               = 'QF';
AO.(QF).Monitor.Units            = 'Hardware';
AO.(QF).Monitor.HW2PhysicsFcn    = @amp2k
AO.(QF).Monitor.Physics2HWFcn    = @k2amp;
AO.(QF).Monitor.HW2PhysicsParams = {[-0.06 .3 0], 0};
AO.(QF).Monitor.Physics2HWParams = {[-0.06 .3 0], 0};
```

```
AO.(QF).Monitor.HWUnits          = 'amperes';
AO.(QF).Monitor.PhysicsUnits     = 'K';
```

If the polynomial coefficients were different for each magnet in the family, then the coefficient row vector would need to be expanded to a matrix with equal number of rows to the number of magnets.

Writing a Mapping Function

The mapping function (like k2amp and amp2k in example 4 above) have the following properties:

- Standalone mapping functions are independent from Middleware
- Mapping functions are the same for all devices in the same family – only different parameters to the function are allowed within a family
- All the parameters necessary for conversion are passed as input arguments to the mapping function
- Mapping functions must handle vector inputs if multiple devices exist in the family.

The syntax for a mapping function is

```
myhw2physicsfcn(Val, Param1, Param2, …, ParamN)
```

Where Val comes from the input in hw2physics and the parameters comes from the HW2PhysicsParams field in the accelerator object.

Mapping Function Examples

Consider the following mapping from x to y

$$y = s\left(c_o + c_1 x + c_2 x^2\right)$$

where S is a scaling coefficient and c0, c1 and c2 are offset, linear and quadratic terms of a second order polynomial mapping.

```
function Y = myhw2physicsfcn(X, s, c0, c1,c2)
Y = s * (c0 + c1*X + c2*X^2);
```

This function can be called from command line

```
>> myhw2physicsfcn(1,2,3,4,5)
ans = 25
```

A vectorized version of this function will accept vector arguments as long as they are the same length.

```
function Y = myhw2physicsfcn(X,s,c1,c2);
Y = s(:) .* (c0(:) + c1(:).*X(:) + c2(:).*X(:).^2);
```

Command line call could look like this.

```
>> S = [1; 0.99; 1.01];
>> C0 = [1; 2; 3];
>> C1 = [4; 5; 6];
>> C2 = [7; 8; 9];

>> myhw2physicsfcn( [pi; exp(1); sqrt(2)], S, C0; C1, C2)

ans = [82.6536
       73.9568
       29.7801]
```

As a consistency check for myhw2physicsfcn and HW2PhysicsParams, use the feval statement in the following way.

If HW2PhysicsParams is a matrix, then

```
>> feval(HW2PhysicsFcn,X,HW2PhysicsParams(:,1), … HW2PhysicsParams(:,N))
```

If HW2PhysicsParams is a cell array, then

```
>> feval(HW2PhysicsFcn,X,HW2PhysicsParams{:})
```

A more flexible mapping function that does not restrict the length of the polynomial is shown below. For Spear, a slightly expanded version of this function is used to map the magnet hysteresis. The scale factor (calibration constant) is multiplied to the polynomial in amp2k and divided in k2amp. The figure below shows a more detailed information flow diagram for the full amp2k and k2amp functions.

$$y = s\left(c_0 + c_1 x + c_2 x^2 + \ldots c_N x^N\right)$$

```
function k = amp2k(Amps, C, ScaleFactor)
% C = [Cn … C2 C1 C0]
Amps = Amps ./ ScaleFactor;
brho = (10/2.998);
for i = 1:length(Amps)
    if size(C,1) == 1
        k(i,1) = polyval(C, Amps(i)) / brho;
    else
        k(i,1) = polyval(C(i,:), Amps(i)) / brho;
    end
end
```

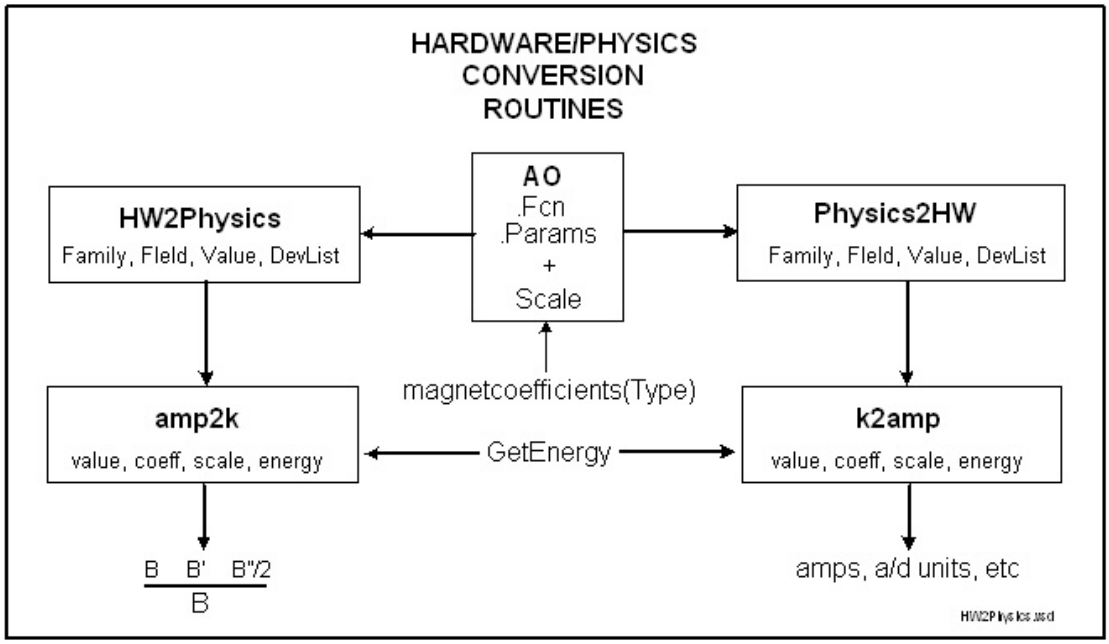


Fig. Information flow diagram for the amp2k and k2amp

Appendix VII: Matlab Channel Access (MCA)

The details of MCA (written by Andrei Terebilo, [2]) will not be discussed in this document; however, here is the basic list of MCA functions.

1. mcastat
2. mcainfo
3. mcaopen
4. mcaisopen
5. mcaget
6. mcaput
7. mcaclose

The *mcaget* and *mcaput* access the EPICs value field unless that the full EPIC's field is stated in the channel name.

REFERENCES

[1] Andrei Terebilo, "AT Users Manual."
[2] Andrei Terebilo, "MCA."
[3] A. Terebilo, G. Portman, J. Safranek, "Linear Optic Correction Algorithm in MATLAB", 2003 IEEE Particle Accelerator Conference, Portland, May 2003.
[4] G. Portmann, "Slow Orbit Feedback at the ALS Using MATLAB," Particle Accelerator Conference 1999, March 1999, New York, New York.
[5] G. Portmann, "Recipe for ALS Storage Ring Operation," LSAP Note #249, May 1998.
[6] G. Portmann, "ALS Storage Ring Setup and Control Using Matlab," LSAP Note #248, June 1998.

[7] G. Portmann, D. Robin, and L. Schachinger, “Automated Beam Based Alignment of the ALS Quadrupoles,” 1995 IEEE Particle Accelerator Conference, LBL-36434, May 1995, Dallas TX.
[8] J. Safranek, G. Portmann, A. Terebilo, C. Steier, “Matlab-Based LOCO,” European Particle Accelerator Conference, Paris, France, June 2002.