LSAP-302

MIDDLE LAYER SOFTWARE MANUAL FOR ACCELERATOR CONTROL

Gregory J. Portmann, Jeff Corbett, and Andrei Terebilo
August, 2006[1]

TABLE OF CONTENTS

1. Introduction
2. Middle Layer Nomenclature
3. Middle Layer Families
4. Basic Middle Layer Functions
5. Shortcut Functions
6. Special Functions
7. Machine Physics Functions
8. Data Management
9. Response Measurement/Save/Restore
10. High Level Functions
11. High Level Applications
12. Middle Layer Setup for Operations
13. Archived Date Retrieval

APPENDICES

I. Installing Software
II. General Programming Guidelines
III. Creating Families
IV. GET and SET Functions
V. Data Storage
VI. Hardware and Physics Units
VII. Tuning the Matlab Middlelayer
VIII. Matlab-to-Control System Access Libraries
IX. MML-to-EPICS Timing

REFERENCE

[1] Most recent document in …\mml\docs\MatlabMiddleLayerManual
Text in red is work-in-progress.

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.

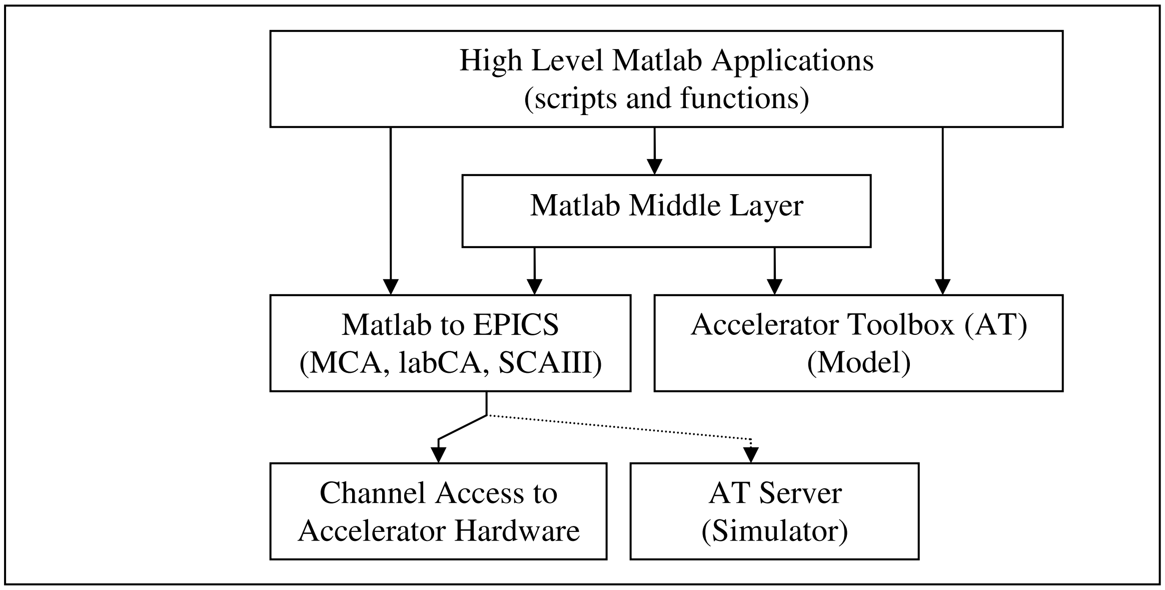

High Level Matlab Applications (scripts and functions)
Matlab Middle Layer
Matlab to EPICS (MCA, labCA, SCAIII)
Accelerator Toolbox (AT) (Model)
Channel Access to Accelerator Hardware
AT Server (Simulator)

Fig. 1. Data 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), 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 either the online hardware or simulator. The two main functions in this class are *getpv* and *setpv*. Both functions accept a variety of input formats including multiple Families and timing information. On the original coding the online system was EPICS and the model was AT (Accelerator Toolbox), however, the functions that communicate directly with the hardware (*getpvonline*, *setpvonline*) and model (*getpvmodel*, *setpvmodel*) have been separated from the main *getpv*/*setpv* function to make changing to other systems straightforward. To date, the middle layer has also been connected to Tango and the Brookhaven UCode control system. For more information on these functions refer to the Appendix or type *help* and the appropriate name, like *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 by Family, Field, DeviceList
2. setpv – set by Family, Field, DeviceList
3. steppv – step by Family, Field, DeviceList

4. getpvonline – get online channels
5. setpvonline – set get online channels
6. getpvmodel – get model data by Family, Field, DeviceList
7. setpvmodel – set model data by Family, Field, DeviceList
8. getam – get by Family and DeviceList (Field= 'Monitor')
9. getsp – get by Family and DeviceList (Field= 'Setpoint')
10. setsp – set by Family and DeviceList (Field= 'Setpoint')
11. stepsp – step by Family and DeviceList (Field= 'Setpoint')
12. switch2sim – changes family in online mode to simulate mode
13. switch2online – changes family in simulate mode to online mode
14. switch2physics – get/set family in physics units
15. switch2hardware or 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. family2channel – convert family to channel names
12. family2common – convert family name to common names

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. family2datastruct – returns a data structure for a Family, Field, DeviceList
2. family2dev[2] or getlist – returns the DeviceList for a Family
3. family2handle – returns the MCA handles for a Family, Field, DeviceList (if using MCA)
4. family2mode – returns the mode for a given Family and Field
5. family2units – returns units for a given Family and Field
6. family2status – returns the status information about a device (1-in operation, 0-removed from service)
7. family2tol – returns the tolerance field for a given Family, Field, DeviceList
8. findmemberof – find all families that are part of a member group.
9. getdata – get data structure from a file
10. getfamilylist – returns the list of families
11. getfamilydata – get specified data field for a family
12. getgolden – get the set of golden values for a family

[2] The default behavior of family2dev is very important because many functions (including *getpv*/*setpv*) use this function to get the default DeviceList if one is not provided as an input.

13. getoffset– get the offset value for a family
14. getnumberofsectors – returns the number of sectors
15. getphysdata – get data from a special "physics" data file.
16. getramprate – get the ramp rate for a device (usually a power supply)
17. getrespmat – get response matrix data from a file
    - getbpmresp
    - gettuneresp
    - getchroresp
    - getdispresp
18. getspos – get s-position in the ring for specified set of elements
19. getsigma – gets the standard deviation of the monitor (pre-measured)
20. isfamily – check for valid family name
21. ismemberof – check if a family is a member of a member list
22. minpv/maxpv – get minimum/ maximum value for family/field
23. minsp/maxsp – get minimum/ maximum Family and DeviceList
24. minpv/maxpv – get minimum/ maximum by Family, Field, DeviceList
25. setphysdata – set data from a special "physics" data file.
26. 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)

The families get/set by these functions is determined by *findmemberof*('MachineConfig'). See "Appendix III Creating Families" for more information on "memberof" groupings.

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
7. tune2manual, tune2online – switch just the TUNE family

Note: some of these shortcut functions many belong in the "special" functions category which is discussed in the next section. For instance, if DCCT is a family then *getdcct* is basically an alias to *getam*('DCCT') . However, making the DCCT 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. That said, I would highly recommend making RF, DCCT, and TUNE families. Some functions expects them to be families and it makes it easier to monitor these channels together with cell arrays inputs to *getam*.

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. Temperatures and vacuum are often special 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. getbpmaverages – returns the number of averages used in the BPM processor
7. getrfcavitytemperature / setrfcavitytemperature
8. getscrap / setscrap – get/set the scraper position
9. getbpmv – get the raw BPM button voltages
10. setbpmaverages – sets the number of averages used in the BPM processor

Power supply functions like on/off, ready, and reset can be a special function but it's better to make these controls a separate field in the power supply family. For instance, at the ALS getpv('HCM','Reset') returns the reset channel for all horizontal corrector magnets.

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. amp2mm / mm2amp – converts a change in a corrector magnet from hardware units (usually amperes) to max orbit change (usually millimeters) (based on the BPM 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
4. buildlocoinput – assembles a LOCO input file
5. bumpinj – creates an injection bump
6. findrf, findrf1 – finds a new RF frequency setting

7. getchro – get the data from a chromaticity measurement
8. checklimits – check that a setpoint change in within limits
9. getdisp – get the data from a dispersion measurement (or get the default dispersion)
10. getenergy – returns the beam energy or desired beam energy (also calculates the energy shift due to the correctors)
11. getmcf – return the momentum compaction factor
12. gev2bend – converts electron beam energy to bend magnet current
13. hw2physcis – convert between hardware and physics units
14. measlifetime – computes the lifetime using beam current measurements (lease squares fit)
15. measchro – measure the storage ring chromaticity (uses SF & SD)
16. measdisp – measure the dispersion function
17. monbpm – monitor, plot, and compute basic statistics like standard deviations on the BPMs
18. monmags – monitor, plot, and compute basic statistics like standard deviations on the storage ring magnets
19. physcis2hw – convert between physics and hardware units
20. plotchro – plot a chromaticity measurement
21. plotcm – plots the corrector magnets & energy change due to the horizontal correctors
22. plotdisp – plot a dispersion measurement
23. plotorbit – plots the current orbit
24. plotgoldenorbit – plots the golden orbit
25. plotoffsetorbit – plots the offset orbit
26. plotorbitdata – plots orbit data from a file (orbit, dispersion, sigma, etc)
27. ramppv – ramp a setpoint change
28. raw2real – converts control system data (raw) to calibrated data (real)
29. real2raw – converts calibrated data (real) to control system data (raw)
30. rmdisp – fits the RF frequency to minimize the correlation with the dispersion
31. setorbit – general orbit correction function
32. setorbitgui – GUI to call setorbit
33. setorbitbump – general orbit bump function
34. setorbitbumpgui – GUI to call setorbitbump
35. setchro – sets the storage ring chromaticity
36. stepchro – steps the storage ring chromaticity
37. setgolden – set a golden value by Family, Field, DeviceList
38. settune – sets the storage ring tune (uses quadrupoles and tune measurement)
39. steptune – steps the storage ring tune (uses quadrupoles)
40. 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. getdispresp – get a dispersion response from a file
6. getrespmat – general response matrix retrieval
7. measrespmat – measure a response matrix (general function)
8. measbpmresp – measure a response matrix for the BPM family

9. measdispresp – measure the dispersion response matrix
10. measchroresp – measure the chromaticity response matrix
11. meastuneresp – measure a response matrix for the quadrupole family
12. plotlattice – plot the lattice magnets
13. plotorbitdata – plot the response matrix
14. plotbpmresp – plot a orbit response matrix
15. plotbpmrespsym – symmetry plot for an orbit response matrix

Insertion Device Compensation Functions
1. measidfftable – measures a insertion device vertical feed forward table
2. measepufftable – measures a EPU 2-dimensional feed forward table
3. plotidfftable – analyzes an existing feed forward table
4. testidfftable – tests the current feed forward table

System Checking
1. monrate – 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 magnet
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 accelerator simulator. However, it is can be very useful to use the model with the Middle Layer. *Switch2sim/switch2online* and the mode flag are often used to access the model from the Middle Layer. It is helpful to have commands that directly use the AT model. Note that functions like *modeltwiss* and *modeldisp* use the coordinate system of the model whereas the middle layer coordinate system may also include gain and roll errors as part of the model. For instance, if BPMx has a roll in the middle layer then

getam('BPMx',[1 1], 'Physics', 'Model')

will include the effects of the roll whereas

modeltwiss('x','BPMx',[1 1])

will not. A partial list of model functions include:
1. atsummary
2. drawlattice
3. family2atindex – returns the at index for a family
4. getcavity
5. getenergymodel
6. getharmonicnumber
7. getkleff
8. getleff

9. gettwiss
10. golden2sim – sets the golden lattice to the simulator
11. modelbeta – beta function of the model
12. modelchro – chromaticity function
13. modelcurh
14. modeldisp – dispersion function
15. modelmcf – returns the momentum compaction factor of the model
16. modeltune – returns the model tune
17. modeltwiss – returns model twiss functions
18. machine2sim – copies the machine setpoints to the simulator
19. plotmodelorbit
20. plotcod – plots the closed orbit
21. plottwiss – plots the twiss parameters for a sector
22. printlattice – simple printout of the elements of the model
23. sim2machine – copies the simulator setpoints to the machine
24. setcavity – sets the RF cavities in the model
25. setenergymodel – sets the model energy
26. setradiation – sets the radiation on/off methods in the model

Miscellaneous Functions
1. addlabel – adds a label to an arbitrary location on a figure window
2. appendtimestamp – appends a date and 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. showfamily – prints family information to the screen
6. sleep – delay in seconds
7. xaxis – just changes the horizontal axis
8. xaxiss – change all the horizontal axis in a figure
9. yaxis – just changes the vertical axis
10. yaxiss – change all the vertical axis in a figure
11. zaxis – just changes the z-axis in a 3d plot

(See the commons directory for a more complete list)

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
5. …

Machine data that needs to be periodically updated
1. Offset (based on magnet centers)
2. Model calibration data (LOCO output)
3. Golden parameters:
   - Orbit (based on the offset orbit plus user requests)
   - Tune
   - Chromaticity
   - 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 "Appendix: 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 (Matlab clock) (vector)
tout:	time when the measurement completed (Matlab clock) (vector)
TimeStamp:	Time (Matlab *clock*) at the start of the function
DataTime:	Hardware time stamps in Matlab serial date number format
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.

Response matrix data as return *measrespmat, measbpmresp,measdisp, 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).

Saving Experimental 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'). The optional input following 'Archive' is the filename. If filename is '', a browser will prompt the user for a filename (the default name and location will be suggested).

Saving Operational Data
Operational data for lattice save, response matrices, dispersion, etc go in a special directory for each operational mode – OpsDataRoot = getfamilydata('Directory','OpsData'). Magnet lattice saves need to be in this directory. If response matrix files are not found, then the accelerator model will be used to general the appropriate data.

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
 MEASRESPMAT - Measure a response matrix

   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
   1. MonitorFamily       - AcceleratorObjects family name for monitors
      MonitorDeviceList   - AcceleratorObjects device list for monitors (element or device)
                            (MonitorFamily and MonitorDeviceList can be cell arrays)
      or
      MonitorStruct can replace MonitorFamily and MonitorDeviceList
```

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

3. ActuatorDelta    - Change in actuator
                      {Default: getfamilydata('ActuatorFamily','Setpoint','DeltaRespMat')}
4. ModulationMethod - Method for changing the ActuatorFamily
                     'bipolar' changes the ActuatorFamily by +/- ActuatorDelta/2
                               on each step {Default}
                     'unipolar' changes the ActuatorFamily from 0 to ActuatorDelta on each step
5. WaitFlag - (see setpv for WaitFlag definitions) {Default: []}
           WaitFlag = -5 will override gets to manual mode

6. ExtraDelay - Extra time delay [seconds] after a setpoint change

7. 'Struct'  - Output will be a response matrix structure {Default for data structure inputs}
   'Numeric' - Output will be a numeric matrix        {Default for non-data structure inputs}

8. Optional override of the units:
   'Physics'  - Use physics  units
   'Hardware' - Use hardware units

9. Optional override of the mode:
   'Online' - Set/Get data online
   'Model'  - Set/Get data on the model (same as 'Simulator')
   'Manual' - Set/Get data manually

10. 'Display'    - Prints status information to the command window {Default}
    'NoDisplay'  - Nothing is printed to the command window

OUTPUTS
1. S = Response matrix

   For stucture outputs:
   S(Monitor, Actuator).Data - Response matrix
                       .Monitor - Monitor data structure (starting orbit)
                       .Monitor1 - First  data point matrix
                       .Monitor2 - Second data point matrix
                       .Actuator - Corrector data structure
                       .ActuatorDelta - Corrector kick vector
                       .GeV - Electron beam energy
                       .ModulationMethod - 'unipolar' or 'bipolar'
                       .WaitFlag - Wait flag used when acquiring data
                       .ExtraDelay - Extra time delay
                       .TimeStamp
                       .CreatedBy
                       .DCCT

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 inputs with non-structure data

EXAMPLES
1. 2x2 tune response matrix for QF and QD families:
   TuneRmatrix = [measrespmat('TUNE',[1;2],'QF',[],.5,'unipolar') ...
                  measrespmat('TUNE',[1;2],'QD',[],.5,'unipolar')];

2. Orbit response matrix for all the horizontal correctors (+/-1 amp kick amplitude):
   Smat = measrespmat({'BPMx','BPMy'}, {getlist('BPMx'),getlist('BPMy')}, 'HCM', ...
                                  getlist('HCM'),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.

3. Orbit response matrix for all the horizontal correctors (Default kick amplitude):
   Smat = measrespmat(getx('Struct'), getsp('HCM','struct'));

Written by Greg Portmann
```

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:	Time (Matlab *clock*) on exist of the function
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 *getfamilydata ('OpsData','RespFiles')*, e.g. {'GoldenBPMResp', 'GoldenTuneResp'}. *getrespmat* will then search through all variables in the file (and through each cell array and structure array if they exist) for the existence of a response matrix structure with the proper Monitor and Actuator field names. As a last resort, the accelerator model will be use to calculate a response matrix. Data structure inputs are also allowed. For example, the following commands will get the orbit, corrector values, and response matrix to be used in an application like orbit correction.

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

During commissioning of an accelerator it is often interesting to compare the response matrix of the model compared to the actual accelerator.

```
RespMatModel = measrespmat(BPMam, HCMsp, 'Model');
for i = 1:size(RespMatMeas);
```

```
          subplot(2,1,1);
          plot([RespMatMeas(:,i) RespMatModel(:,i)]);
          title(sprintf('Column Number %d',i));
          subplot(2,1,2);
          plot(RespMatMeas(:,i)-RespMatModel(:,i));
          ylabel('Error');
          xlabel('BPM Number');
          pause;
      end
```

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.

```
                                                      % ALS orbit correction example
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. finddispquad – optimizes the setpoint of the quadrupole that sets the dispersion in the straight sections.
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. srcycle – cycles the storage ring magnets (machine specific)
7. srramp – energy ramping of the storage ring
8. setorbit – general purpose global orbit correction function (see below for details)
9. setorbitbump – general purpose local bump function (see below for details)
10. quadcenter, quadplot, quaderror – finds the quadrupole center of one magnet at a time
11. setorbitquadcenter – corrects the orbit to the quadrupole centers
12. scanaperture – used for scanning the electron beam in the straight sections and checking the lifetime (physical aperture)
13. scantune – scan in tune space and record the lifetime (or loss monitors)

Orbit Correction (*setorbit & setorbitdefault*)

(Still writing this section)

Orbit Correction without RF adjustments

1. SVD (singular value selection based on user input vector or max/min ratio)
2. BPM weights
3. Measured or model response matrix
4. Number of iterations user selectable
5. Absolute or incremental orbit change

Orbit Correction with RF adjustments

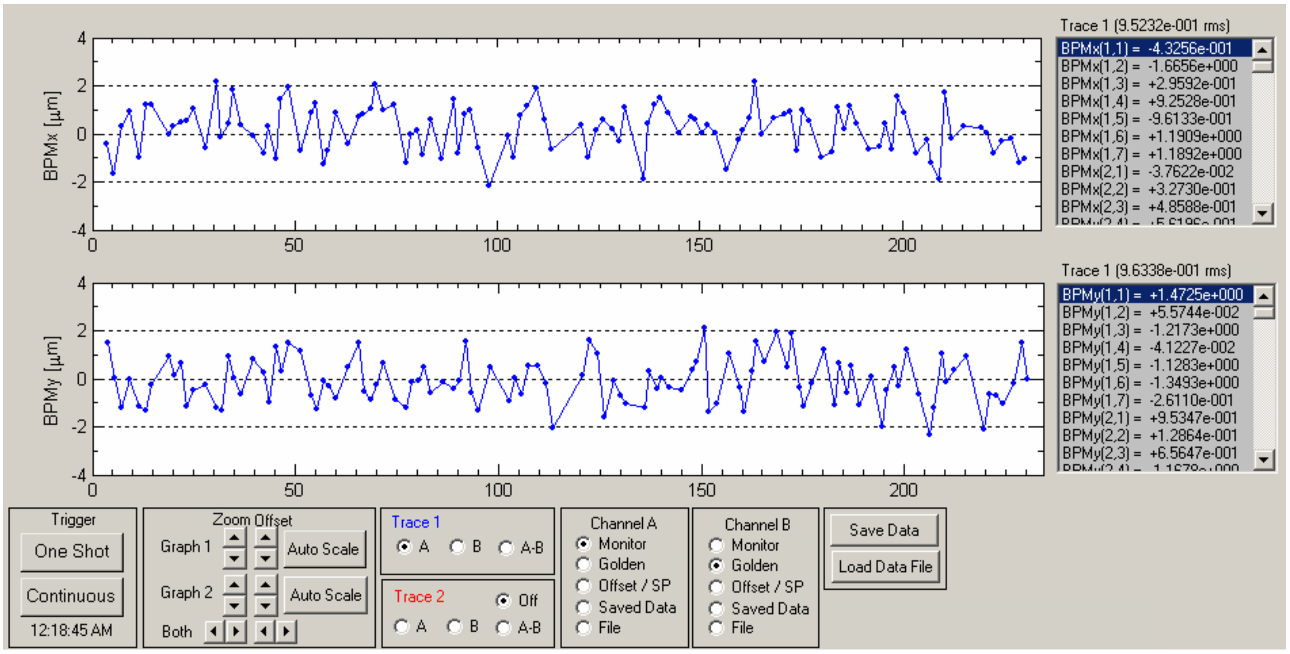
1. Included the dispersion function as a column of the response matrix. What dispersion function to use is determined by the user. The choices are,
    a. User input
    b. Measure dispersion
    c. Model dispersion
    d. Golden dispersion
2. SVD orbit correct (same inputs as in Orbit Correction without RF adjustments)

Instead of fitting the RF frequency one could also remove the dispersion component of the orbit use *rmdisp* or *findrf* before calling *setorbit*. At the ALS the RF is manually adjusted to fix energy shift of the horizontal correctors to zero by iterating orbit correction and RF frequency adjustment. Accomplishing this as an energy constraint within the orbit correction algorithm is a planned future improvement.

Local Orbit Correction (*SETORBITBUMP*)
Selecting the proper corrector magnets for a local bump can be cumbersome. This function makes an attempt to overcome this.

11. HIGH LEVEL APPLICATIONS

1. DISPLAY (*PLOTFAMILY*)



```
Trace 1 (9.5232e-001 rms)
BPMx(1,1) =  -4.3256e-001
BPMx(1,2) =  -1.6656e+000
BPMx(1,3) =  +2.9592e-001
BPMx(1,4) =  +9.2528e-001
BPMx(1,5) =  -9.6133e-001
BPMx(1,6) =  +1.1909e+000
BPMx(1,7) =  +1.1892e+000
BPMx(2,1) =  -3.7622e-002
BPMx(2,2) =  +3.2730e-001
BPMx(2,3) =  +4.8588e-001
```

```
Trace 1 (9.6338e-001 rms)
BPMy(1,1) =  +1.4725e+000
BPMy(1,2) =  +5.5744e-002
BPMy(1,3) =  -1.2173e+000
BPMy(1,4) =  -4.1227e-002
BPMy(1,5) =  -1.1283e+000
BPMy(1,6) =  -1.3493e+000
BPMy(1,7) =  -2.6110e-001
BPMy(2,1) =  +9.5347e-001
BPMy(2,2) =  +1.2864e-001
BPMy(2,3) =  +6.5647e-001
```

Channel A: [✓ Monitor] Golden Offset / SP Saved Data File
Channel B: Monitor [✓ Golden] Offset / SP Saved Data File
Trace 1: [✓ A] B A-B
Trace 2: [✓ Off] A B A-B

The family menus in *plotfamily* come from findmemberof('PlotFamily'). If empty, findmemberof('MachineConfig') is tried, then all families are used.

2. BEAM BASED ALIGNMENT

quadcenter
quadcenterall
quadplot
quaderrors
bpm2quad & quad2bpm

(To be written)

12. MIDDLE LAYER SETUP FOR OPERATIONS

SETUP FILES

The large attempt was made to make the MiddleLayer accelerator independent. However, the basic setup file are specific to the accelerator hardware and control system. A separate directory for each accelerator is used to store all the machine dependent functions. The following is a brief list of functions that must exist.

1. Middle layer initialization file (like alsinit)
2. setoperationalmode
3. srinit
4. srcycle

5. getbpmaverages/setbpmaverages
6. bend2gev / gev2bend
7. setorbitdefault
8. buildlocoinput / setlocogains
9. setquad / getquad / quadplotall / quadcenterall

MIDDLELAYER OPERATIONAL DATA FILES

There are 8 data files that must exist on in the operations data directory to get the full functionality of the MiddleLayer. These data files often need to be regenerated and copied to a specific located with a specific name. A number of functions have been written to make this task easier and reduce file deletion and name mistakes as well as automatic backups. If the file does not exist, then the model will be used to general the needed data. A message will be printed to the screen if this happens. The file name is chosen as part of the middle layer setup; however, most people use a similar naming scheme. The file name usually has two parts, the root name and a suffix. Usually accelerators keep the same root name and change the suffix with the mode name. Names are usually set in setoperationalmode function. The example names below are for an operational mode called "user."

1. Machine save - machine configuration file (getmachineconfig) (GoldenConfig_User)
2. Injection save - injection configuration file (getmachineconfig) (InjectionConfig_User)
3. BPM Sigma (monbpm) (GoldenBPMSigma_User)
4. Dispersion function (measdisp) (GoldenDisp_User)
5. Corrector-to-BPM response matrix (measbpmresp) (GoldenBPMResp_User)
6. Quad-to-TUNE response matrix (meastuneresp) (GoldenTuneResp_User)
7. Sext-to-Chromaticity response matrix (measchroresp) (GoldenChroResp_User)
8. Corrector-to-Dispersion response matrix (measdispresp) (GoldenDispResp_User)

To replace the above files it's best to use one of the functions below.

1. copymachineconfigfile – Copy a machine configuration file to the default file
2. copyinjectionconfigfile – Copy an injection configuration file to the default file
3. copybpmsigmafile – Copy a measured BPM sigma file to the default file
4. copydispersionfile – Copy a measured dispersion file to the default file
5. copybpmrespfile – Copy a measured BPM response file to the default file
6. copytunerespfile – Copy a measured tune response file to the default file
7. copychrorespfile – Copy a measured chromaticity response file to the default file
8. copydisprespfile – Copy a measured dispersion response file to the default file

These functions are also a menu item in plotfamily.

13. ARCHIVED DATA RETRIEVAL

Spear Only

Retrieving archived or history buffer information

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

ALS Only
Retrieving archived or history buffer information
1. arread – read an archived day
2. arselect – get the data for a given channel(s)
3. arplot – plot the data for a given channel(s)

APPENDICES

Appendix I: Software Installation

1. Install the Matlab Middle Layer and AT software as well as applications like LOCO or orbit correction GUIs. <ROOT> is the root directory location of these files.

2. Set the Matlab path. One way to do it is put the following code in the *startup*.m file.
   ```
   run <ROOT>\mml\setpathmml
   ```
   or a set path function for a particular machine like setpathals.

   The Middle Layer directory tree is the following
   Matlab MiddelLayer Toolbox (MML)
   ```
   <ROOT>\MML
   <ROOT>\MML\storagering
   <ROOT>\MML\booster
   <ROOT>\MML\linac
   <ROOT>\MML\transport
   <ROOT>\MML\links
   <ROOT>\MML\at
   <ROOT>\MML\at\storagering
   <ROOT>\MML\at\transport
   ```

   Machine Directories (using ALS as an example)
   ```
   <ROOT>\machine\ALS\Booster
   <ROOT>\machine\ALS\BoosterData
   <ROOT>\machine\ALS\BoosterOpsData
   <ROOT>\machine\ALS\BTS
   <ROOT>\machine\ALS\BTSData
   <ROOT>\machine\ALS\BTSOpsData
   <ROOT>\machine\ALS\LTB
   <ROOT>\machine\ALS\LTBData
   <ROOT>\machine\ALS\LTBOpsData
   <ROOT>\machine\ALS\Ring
   <ROOT>\machine\ALS\RingData
   <ROOT>\machine\ALS\RingOpsData
   ```

   Matlab to control system link libraries
   ```
   <ROOT>\links\mca
   ```

<ROOT>\links\mca_asp
<ROOT>\links\sca
<ROOT>\links\ucode
<ROOT>\links\tango

Applications
<ROOT>\applications\common
<ROOT>\applications\loco
<ROOT>\applications\naff
<ROOT>\applications\orbit
<ROOT>\applications\datebase\mym
<ROOT>\applications\m2html

AT - Accelerator Toolbox
<ROOT>\at

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. **Accessing MiddleLayer Data**: Applications should not use the AcceleratorObject (AO) or AcceleratorData (AD) structures directly. Use getfamilydata or the appropriate function instead. In the future how the data is stored may change.

5. **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.

6. **Error Handling:** Instead of functions returning an 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. However, some middle layer users prefer Setpoint – Monitor errors to behave differently. Hence a middle layer family variable, AD.ErrorWarningLevel, can be set to change what happen on a SP-AM error.

7. **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.
%
% See also …
%
% Written by ____
```

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:

<u>AcceleratorObject.(Family)</u>

FamilyName:	Family Name ('BPMx', 'HCM', etc.) (must be unique)
MemberOf:	Cell array of strings, for instance {'MachineConfig'; 'PlotFamily'; 'QUAD'; 'Magnet'}
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)

Sub-family structure for monitors:

<u>AcceleratorObject.(Family).Monitor</u>

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'
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

Optional monitor fields:

SpecialFunctionGet:	Function name if Mode = 'Special'
Golden:	Vector of golden values (usually for BPMs) (hardware units)
Offset:	Vector of offset values (usually for BPMs or magnets) (hardware units)
Gain:	Vector of gains (usually for magnets) (hardware units)

Sub-family structure for setpoints:
<u>AcceleratorObject.(Family).Setpoint</u>

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'
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
Range:	[Min Max] range for the setpoint (two columns) (hardware units)

Optional setpoint fields:

SpecialFunctionSet:	Function name if Mode = 'Special'
DeltaRespMat:	Delta setpoint for measuring response matrices (hardware units)
RampRateFcn:	Function to call in order to get the ramp rate (hardware units)
	or
RampRate:	Ramp rates (hardware units)
RunFlagFcn:	Function to call to
	or
Tolerance:	Tolerance column vector for SP-AM comparison (hardware units)
Offset:	Vector of offset values (usually for BPMs or magnets) (hardware units)
Gain:	Vector of gains (usually for magnets) (hardware units)

Note: Fields Range, DeltaRespMat, RampRate, Tolerance, Offset, or Gain must be in hardware units and RampRateFcn must return hardware units. A middle layer user can switch between hardware and physics units at will, but the initialization data must only be in hardware units so that all functions who what to expect.

The number of rows of DeviceList, ElementList, CommonNames, Positions, Range, and Tolerance must be equal. The number of rows of ChannelNames must equal DeviceList if DataType='Scalar'. For DataType='Vector', ChannelNames can only have one row (channel name) but the output of getpvonline (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.

Note: Do not create a sub-family structure with the name "Field." One way to tell the difference between an AO structure and a data structure is to look if the structure field name "Field" exists.

Certain functions, like *getmachineconfig* or *settune*, need to know what families to use for that function. For instance, the tune correction for one accelerator may be done with QF and QD families and QFA and QFB for another. To make the software more machine independent, the memberof field is used. Below is a table of memberof strings and which function uses it. The MemberOf field can exist at the family level or at the field level. For instances, if one want to include the HCM.Setpoint and HCM.FeedForwardField in the machine configuration, then put the 'MachineConfig' string at the field level. When one wants to save/restore just the .Setpoint field then either location will work.

memberof string	**Functions that uses it**
MachineConfig	getmachineconfig, setmachineconfig, getgolden
Magnet	monmags
BPM	monbpm, monmags, …
BPMx or HBPM	gethbpmfamily, …
BPMy or VBPM	getvbpmfamily, …
COR	getbpmresp, measbpmresp, …

HCM	gethcmfamily, …
VCM	getvcmfamily, …
QUAD	
SKEWQUAD	
SEXT	
BEND	
Tune Corrector	meastuneresp, settune, steptune, getgolden
Chromaticity Corrector	measchroresp, setchro, stepchro
PlotFamily	plotfamily

Table. Reserved MemberOf Words.

Ideally the use of the Member of field could eliminate hardcoding family names. However, the Middle Layer code has not been made that flexible. For one, always searching the MemberOf field for the default BPM or corrector is a little time consuming. And more honestly, certain names were so often used in the original ALS code and it would be time consuming to generalize to any naming scheme. The following table shows the family names which probably should exist.

Family Name	**Functions that uses it**
BPMx, BPMy	To numerous to list
HCM	“
VCM	“
BEND	“
TUNE	“
RF	“

Table. Family Names That Must Exist.

Notes

1. Sometimes the channel name is only available for some devices and not others. The channel name field can be filled with spaces if one is not available. getpv would return NaN for that channel and setpv would do nothing.

Additional field when using the AT simulator:

<u>AcceleratorObject.(Family).AT (simulator only)</u>

ATType:	'X', 'Y', 'BPMX', 'BPMY', 'HCM', 'VCM', else ATParameterGroup is used
ATParameterGroup:	Parameter group
ATIndex:	Column vector of AT indexes (if using split magnets, then add a column for every split)

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'), *modeltwiss*, etc. *getpv* and *setpv* check if the Mode is 'online', 'simulate', 'manual', or 'special.' If in simulate mode, then the simulator functions *getpvmodel* and *setpvmodel*. The middle layer families 'TUNE', 'DCCT', and 'RF' are automatically simulated. Other families require the AT fields (above) be setup properly. If the ATType is 'X', 'Y', 'BPMX', 'BPMY', HCM', 'VCM', 'QUAD', 'SkewQuad', 'SEXT', or 'BEND' only the ATIndex field needs to be complete. All other devices require the ATParameterGroup field (see *help setparamgroup* for details).

Notes about the simulator

1. The AT field can be at different level in the AO family tree. The simulator programs first look in AO.(Family).(SubField).AT then AO.(Family).AT. That way one AT field can work for multiple subfields, like 'Setpoint' and 'Monitor'.
2. The physics units must match AT units in order for the simulator to work properly.
3. The middle layer physics units are AT units. Hence for correctors KLeff/brho (radians) are used and other magnets (quads, sext, etc.) use "K" (ie, B, B', B'').
4. Be careful with AT models with split magnets. Each column of ATIndex implies a split magnet. Otherwise, clever uses of the ATparameter group can be used to split magnets. The multiple column ATIndex method only works as long as K is being varied, not K*Leff. So, if you split a corrector it's not going to work properly. But all other magnets work fine (and who splits a corrector magnet?).
5. Channel and common name methods do not work in simulator mode. Convert them to family, field, device list first.
6. Changing the BEND family in the simulator changes the energy in AT and changes all other lattice magnets (returned by *getmachineconfig*) to reflex focusing changes at the new energy. It doesn't change the BEND family. Changing a subset of the BEND magnet family in the model is not programmed. Use AT commands directly to do this.

Function names

The setup functions are usually separated into three tasks with the same names for all accelerators.

1. *aoinit* – creates the Accelerator Object (AO) structure (also calls *setoperationalmodel*).
2. *setoperationalmodel* – creates the Accelerator Date (AD) structure and may make some changes to to AO for different lattice configuration (also calls *updateatindex*).
3. *updateatindex* – adds the AT sub-structure to AO.

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 MML. 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

Use help getpv or help setpv to see update to date information on these functions.

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.

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', 'Spear3', or 'CLS'
   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.TUNEDelay = Time to wait before TUNE data is fresh
   6. …

3. Physics Data
   Purpose: Store physics related data (Note: we may be fazing out the physdata file since it is a little confusing. Putting all the data in the AO using your method of choice is the other option.)
   Location: Stored in a file
   FileName = getfamilydata('OpsData','PhysDataFile');
   (Directory must be included in the filename)
   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

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

There is not a unique way to store data. For instance, some machine may put golden data in the AO and others put it in PhysData. The functions that get data often search in different places if the data is not found on the first attempt. The reason behind this is to give flexibility to the user.

A. *getramprate* is a basic function used by setpv and setpv. For some accelerators the rate rate is fixed and the AO might be a good place to store the data. For others, the ramp rate is variable hence a channel is used. And others have oddball ways of getting the ramp rate so a generic function is can be used. *getramprate* looks for the data in the following order:
   a. AO.(Family).(Field).RampRateFcn
   b. if Field = 'Setpoint', looks for ramp rate channel names, ie, AO.(Family).RampRate.ChannelName
   c. Constant in the AO
   d. Physdata file

B. *getrunflag* is a basic function used by setpv and setpv. For simple Setpoint-Monitor comparisons to determine to a devices has reached it's setpoint, use the .Tolerance field. For more general behavior use the .RunFlagFcn special function. *getrunflag* "executes" in the following order:
   a. AO.(Family).(Field). RunFlagFcn
   b. if Field = 'Setpoint', does a Setpoint-Monitor comparison based on the AO.(Family).(Field). Tolerance field (in hardware units)

C. *getgolden* looks for the data in the following order:
   a. Look in the AO and AD
   b. Look in the MachineConfig if the Family is a member of 'MachineConfig'
   c. Look in PhysData

D. *getoffset* and *getgain* look for the data in the following order:
   a. Look in the AO and AD
   b. Look in PhysData

E. *getgain* looks for the data in the following order:
   a. Look in the AO and AD
   b. Look in PhysData

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
```

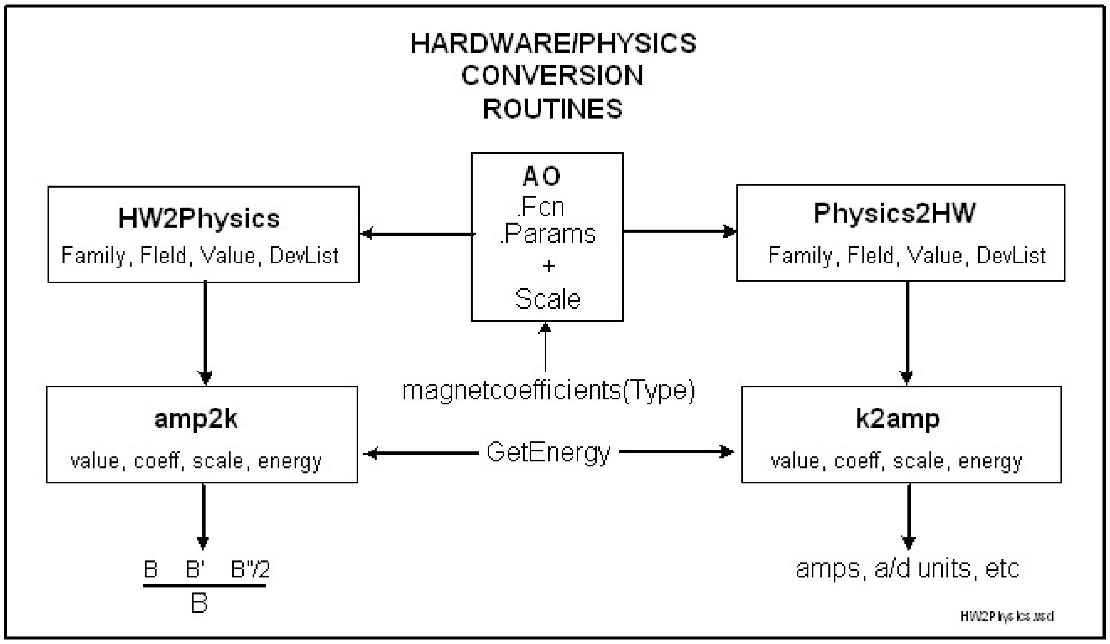

HARDWARE/PHYSICS CONVERSION ROUTINES
HW2Physics
Family, Field, Value, DevList
AO
.Fcn
.Params
+
Scale
Physics2HW
Family, Field, Value, DevList
magnetcoefficients(Type)
amp2k
value, coeff, scale, energy
GetEnergy
k2amp
value, coeff, scale, energy
B B' B"/2
B
amps, a/d units, etc
HW2Physics.vsd

Fig. Information flow diagram for the amp2k and k2amp

Appendix VII: Tuning the Matlab Middlelayer

After the MML is basically setup (families established, channel names, etc.) a number of parameters need to be determined and set in order for all functions to work properly. Tuning the MML for each accelerator can be a bit time consuming but it is absolutely crucial and it should be periodically tested as the accelerator ages.

1. **BPM Timing**
   It is very important to get the BPM timing correct. Every function that uses a *setpv* with a WaitFlag = -2 relies on the *getbpmaverages* function to return the proper delay. Often the delay is a function of both control system delay and the amount of averaging done the BPM hardware or software.

2. **Corrector Magnet Timing**
   When using *setpv* with the WaitFlag < 0, it is crucial that the function does not return until the actuator (using a power supply) has reached it's setpoint and any remained dynamics are damped out. The basic run flag is to base the decision on a Setpoint-Monitor tolerance. The tolerances can be measured using *monmags* and included in the .Tolerance field in the MML (make sure it's in hardware units). If run flag are provided by the control system (a rare luxury), they can be used directly. All other cases be handled using a run flag function that you provide (.RunFlagFcn).

   How run flag errors are handled is a MML setup variable. The *setpv* function can either return an error flag, ignore an error, or interactively prompt the user for action. Use the AD.ErrorWarningLevel to change what is done on a RunFlag error.

A good test is to run magstep. This not a polished function. It's a simple Matlab script that will need to be edited for your accelerator. Since response matrices are used a lot, another good test is take two response matrices back-to-back but change the delay (I would double it) in the *getbpmaverages* function. The difference of the matrices should be comparable to the BPM sigma (which can be found with *monbpm*).

It can be a little tricky to get the timeout properly code in *setpv* when using the WaitFlag. Contact the author if it's does not work properly.

3. **Tune Measurement Delays**
   When using setpv with the WaitFlag = -3, the tune delay most be know to the MML. Presently, it's set to a constant in AD.TuneDelay.

Appendix VIII: Matlab-to-Control System Access Libraries

The MML has been connected to a number of different control systems – EPICS, Tango, UCode, OPC.

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.

LABCA
LabCA was written on top of EZCA by Till Straumann at SSRL. The version I test with comes with the Matlab Middle Layer. The latest version is available on SLAC's website.

SCAIII
SCAIII is yet another Matlab to EPICS link. This one was written by Greg Portmann and it sits on top of SCAIII. It quite fast but it does not have the functionality that LabCA has.

Appendix IX: Matlab-to-EPICS Timing

As an example of what performance one might expect to see using the MML, the below table shows timing results for the ALS taken from a sun workstation. There were 122 BPM and 95 HCM when this data was taken.

Command	# of elements	SCAIII (Seconds)	LABCA (Seconds)	OLD ALS ML (Sec.)
Get By Family				
getam('BPMx'), [1 2])	1	.008	.025	.004
getam('BPMx')	122	.045	.060	.016
Get By ChannelName				
getpv(BPMxName)	1	.010	.025	N/A
getpv(BPMxNames)	122	.140	.160	N/A
getpvonline(BPMxName)	1	.003	.010	.001[3]
getpvonlilne(BPMxNames)	122	.019	.030	N/A
Set By Family				
setsp('HCM', hcm(1)), [1 2])	1	.007	.007 / N/A	
setsp('HCM', hcm, DevList)	95	.014	.017 / .070[5]	
Set By ChannelName				
setpv(HCMName, hcm(1))	1	.008	.007 / N/A	N/A
setpv(HCMNames, hcm)	95	.120	.100 / N/A	N/A
setpvonline(HCMName, hcm(1))	1	.0008	.0008 / N/A	.0005[4]
setpvonline(HCMNames, hcm)	95	.0025	.0055 / .050[5]	N/A

Table. IX. Get/Set Time Results

The old ALS middle layer is not very flexible but more of the lower level code is compiled c-code. This makes it quite a bit faster but it's very difficult to port to other accelerators, hence the start of the new MML.

The timing of LabCA depends on how it's initialized. I used lcaSetRetryCount(1000), lcaSetTimeout(.005), and lcaSetSeverityWarnLevel(4). I'm not sure if these are the best settings but they are fast than the default settings. A warning level of 4 is needed at the ALS because our EPICS crates often don't define data on resets (UDF errors). I also used lcaPutNoWait for these tests.

Things to think about if "fast" timing is important:

- Use the device list in all getpv/setpv calls (family2dev('BPMx') took .01 seconds in the examples in Table IX).
- Group channels together into one getpv/setpv call as much as possible.
- If working strictly online, getpvonline/setpvonline are faster. However, these functions are low level MML calls that the author would like to retain the right to change if necessary.

[3] Using the compiled scaget function.
[4] Using the compiled scaput function.
[5] Timing for lcaPutNoWait / lcaPut

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.