An Overview of Cure

John Folkesson[1] and Patric Jensfelt[2]

[1] MIT/KTH, Email: johnfolk@mit.edu
[2] KTH, Email patric@nada.kth.se

Summary. Cure was meant to help new robotics researchers achieve critical mass in their robotics software libraries more quickly than if they had to re-invent each bit of code. It was hoped that certain common programing issues such as basic data representations could be settled upon and make code more portable between users of Cure. Programming issues common in robotics include matrices, transformations, geometry, communication, and data extraction from common sensor types. These are then used to provide simple working implementations for navigation such as, dead-reckoning, collision avoidance, SLAM and so on.

1 Introduction

[3] There is a certain Cure way of working. One avoids templet functions in general. We use no external libraries other than standard C++ libraries[4]. Data and descriptions of the data should be separate. Thus data is not easily human readable without a table lookup of what the data is. Data is the driver of robot apps. Nothing happens with no data input. The regularization and strict specification of data flow is a major aspect of Cure apps. An attempt is made at strong data typing to help prevent coding errors. That is a two edged sword and can be quite annoying when you have not bothered to specify correctly and mysteriously get no output from your app.

Cure can be useful on a several levels. Some only install the Math library to use the Matrix family of classes. Others only use the out of the box Cure apps such as the EKF SLAM implementation. This has proved useful to researchers who need robust reliable SLAM on their robots but SLAM is not the focus of their work. A third level is between these two and uses the toolbox libraries to create new cure modules.

[3] This document is meant to be read partially. Start at the begining and stop when you have gone deep enough.
[4] There is some libX11 that is optional

On the first level, the Math library adds useful classes for arrays of double, short, binary, and long that both help manage the allocation of memory and provide functionality on those data arrays. This is highly recommended and once you become comfortable with these classes, you will find that the 'a=new double[n];' will disappear from your code. It is simply much easier to have the Matrix class do that. As many function calls on Matrices are in-line, they are very efficient. When writing that class I actually compared to allocating doubles in the source code. I found, strangely, that execution time was significantly faster for using the Matrix class to do matrix multiplication than for loops over a double array in my source code. The Matrix class was compared to a some downloaded C++ library by Patric and he found the Cure class to be faster at matrix inversion for any matrix one might invert in a real-time app[5]. For huge matrices the use of the CPU specific code mght be better.

These classes all require you to specify the size. This is definitely better than the option of having the size adjust to your request for an element as in matlab. That option leads to hard to find bugs in your code. Be careful and get it right or it simply wont work at all.

The use of these classes provides debug messages when you try to access data out of bounds. This saves having such actions hang around unnoticed in your code for months sometimes.

Other useful basic classes can be found in Geometry and Transformations. The Pose3D class is another workhorse. It can do lots. So the days of figuring out the sign of the Jacobian for the Euler angles when subtracting two transformations is over. Its solved once and for all. The same is true for a bunch of other things like transforming 3D points and so on. One must understand the CovType of a Pose3D. It is fundamental to using the object effectively. The CovType is 6 binary flags coded as a 6 bit integer. The lsb corresponds to x and the msb corresponds to Euler angle psi. These specify how the pose coordinates are treated when calculating Jacobians. So if you are in 2D you might only want the xy and theta to be treated as variables while z, phi, and psi are considered fixed values. More specific information is given in the class documentation. This class has a learning curve that is well worth traversing for a robotics programmer.

The Pose3D class is also an example of a TimestampedData object. TimestampedData, TD, is the base class for all Cure data. It has a timestamp, ID, ClassType, and SubType. It also has the concept of SensorID and SensorType. The four integers ID, ClassType, SensorID, and SensorType form a data descriptor. This data descriptor is then tied to a string in the DataDescriptor class. So one can give your data a name but it is not stored and manipulated with it. You look up its unique DataDescriptor to find the name.

TD are of central importance as one gets deeper into using Cure. They are the objects that get passed between modules to make stuff happen. There

[5] The FBI has confiscated theses results and we have not bothered to redo the experiment. Conspiracy theory anyone?

are a number of subtypes defined in Transformation, SensorData, and AddressBank. TD can be cast to specific subclasses using virtual methods in the TimestampedData base class.[6] This is recommended practice even if you are sure its a SICKScan or whatever. The cast will return null if it is the wrong type.

Most useful TD classes so far are Pose3D, MeasurementSet, SmartData, DataSet,and GenericData. These can do most of the work you need done. SmartData can hold any of the other types of data. It is the object used in general operations like setting up queues and buffers.

All TD objects can be easily read from and written to files and sockets. This is done using FileAddress and SocketAddress classes. The format for writing to a file can be *.tdf or *.dat. The *.tdf files can be read later by a FileAddress with no loss of data. The *.dat formatting is user specified in a Cure configuration file *.ccf. These user formated data can not be read by Cure but may be more user readable and matlab loadable than the original data. Parsing from a mixed up tdf file of different types of data to a *.dat file can be done easily using the FileParse class which exists now outside of Cure but will soon be added.

The DataSlotAddress is a circular buffer of fixed length for TD objects. It can look up data in a number of ways based on the timestamp. It can interpolate between data if that is defined for the data type. Being of fixed length, it is not always so useful. There is a SmartDataList for more general situations but that lacks some of the functionality of the Slot.

We have begun to describe addresses here. That is a major concept in Cure. It is one for the higher level Cure users. An Address is an interface for transfer of data between modules. It has push/pull/read/write and a bit more defined. Addresses all exist in specified thread spaces and this can be used to control multi-threaded apps. So it is not possible to directly push data from one thread space to another. There are special classes to handle that which get the thread lock first. It is not neccesary to use this locking mechanism but it is an option[7].

Addresses can be deleted and created dynamically and the other addresses that are set to pull from or push to the address need not be notified of the deleted address. They will find out for themselves next time they try read/write.

The original Address concept is for within process communication. There is now a a SocketAddress that can handle pushing data through a socket to a module in another process. Pull has not been implemented. The nice thing here is that the SocketAddress is very easy to use. If it is set to be a client it will try to find the server periodically when it has data to send until it finds the server and sets up a permanent connection. So you don't need to start

[6] These are the getClassPointer(...) and narrow family of methods.

[7] That is, you don't have to do it our way but you do have to protect against thread collisions.

the server first. It just gets sorted out if you are pushing data from the client side. This sorting out does not involve contacting any central server process. One can even kill and restart the server or client without any communications problem.

The AddressMaker class is for the power Cure users. It is the ultimate Cure object and can create, start, stop and destroy an entire runtime configurable Cure app as specified in detail in a set of *.ccf files. It allows specifying what DevicesAddress, FilesAddress, DataFilter, ServiceAddress, and FilterBankAddress are created. Then how they are configured and hooked up to one another. One can then query it by name for specific addresses. So your app becomes a set of *.ccf files. Still to do is to include in Cure a simple example of using it. The FileParser is one such and will be added as time permits.

2 Address

The Cure Address concept is heart of the Cure paradigm. It is how Cure can help bring structure to the data flow in an application. This Address based data flow forces a great deal of discipline on the programmer to separate the functional blocks into modules that receive push/pull data only in the Cure format. This discipline is a good thing and welcome when the system gets larger and more complicated. Strange stuff can happen at demo time and one needs to have good program structure to have any hope of debugging on the fly. One can easily verify that good data is flowing on any Cure data path by inserting a DebugAddress or a FileAddress in the path. If good data is flowing in and bad data out the problem is isolated. Timing issues can also be debugged like this.

Specific Address objects are FileAddress, SocketAddress, DebugAddress, DeviceAddress, DataSlotAddress, ServiceAddress, and DataFilterAddress. One can make subclasses of Address for special needs like RoboLookAddress or whatever. All Addresses have DataDescriptors and thus names. The DataDescriptor can be used to both give the address a name and to specify what data it will handle. It is not necessary for the actual data to match the DataDescriptor but one can enforce such a requirement. That is one relatively weak data type check that is built into Cure. Stronger type checking can be done by the typeCheck method of the DataFilter class.

2.1 FileAddress

This class is easy to use and requires little explanation for writing and reading from *.tdf files. To read a file there has to be a thread that calls read on the FileAddress. That can be just a while loop in main that reads the canned data and writes it to the input of your app. Writing is easier. Just call push on the output address giving it the FileAddress and the data will be written to the file.

The *.dat files are quite a bit more complicated as there are many options on formating. This is not fully implemented either but works fine where implemented. The FileAddress needs to be configured

Here is an example of a ccf section:

FBNout.dat
m_Header FBN pose estimate (time x,y,z,theta,phi,psi, covMatrix (x 36))
m_MessageString x 1 0 0 6 1
m_MessageString Covariance 1 0 0 6 6

It starts with the name of the File. Somewhere else in the ccf file there was a section like this:

FILEADDRESS
Write FBNout.dat FBNDataName
Write XYZ.tdf XYZdataname
Read input.tdf inputdataname

that told the AddressMaker to look for this file and configure it for write. Then it calls configure on it with each of the lines after FBNout.dat above. The m_Header line will write a descriptive header at the top of the file. The names in the ccf file such as m_Header match the member names in the class FileAddress. That ties this to the html documentation for the member. The m_MessageString tells the FileAddress to call getMatrix with this string on the TD object it is writing from. Then it writes what the TD object returns. So the FileAddress doesn't understand what is after the m_MessageString the TD object does. In this case it is a Pose3D object that will return the 6 rows matrix with its coordinates in. Then a 6x6 Matrix with the Covariance. This is all written to one line of the file (in ASCII) which will start with the timestamp.

Much more complicated formats can be done such as selecting certain members of a DataSet. One has to look at how the TD object implement getMatrix to see what will happen. Not all TD objects have implemented getMatrix. There is lots more on ccf files later on.

2.2 SocketAddress

One creates the object and sets the few config items. You can set it to be either a server or a client and give it a port/host. Just watch out for your threads. This object creates a new thread. Then call startDevice and it goes. You can then push data to the socket or from the socket. The AddressMaker will do all that for you with a few lines in the ccf file. So you don't ever need to create these sockets in your source code at all.

2.3 DebugAddress

This is inserted in a data path by calling push/pull. It then prints to the screen information on the TD passing through it. It has various levels of information, m_Level. Higher levels give more information on data passing through it. It can even pause the program for you. One can set m_StartTime and m_StopTime to have the debug info turn on and off at various timestamps

2.4 ServiceAddress

This was meant to be a sort of backdoor around having to make a full blown DataFilter for every little thing. It is considerably easier to use. It does not have the DataFilter's multiple type checked inputs and outputs. It is one address which is used for both input and output and the subclass just implements read and write to make it do some work. There is one major difference with Service addresses in that they are registered with the AddressBank[8] as Services with a specific name. This allows other modules to find the ServiceAddress by querying the AddressBank with the name. DataFilter implements this with its getService method. Thus one can write modules that simply find one another at run time without the need to call push/pull.

2.5 DataFilter

The DataFilter class is where processing of data is to occur. It provides multiple input and output ports as well a trigger port. These are all DataFilterAddress's. There are lots of examples of DataFilters. The base class defines a sequence of virtual methods that are called on read and write and do such things as type checking, gathering the inputs by reading pull data, doing the calculation, and pushing the data from the output addresses. Getting one of these to work can be tricky and requires careful consideration of what will start the sequence and what will happen then. This class is more complicated than any other part of Cure. Once this one class is well understood much of the rest of Cure should be as well.

3 Threads

Cure provides optional help in managing multiple threads. One can do this outside of Cure and ignore this whole section. One should best ignore this section until one needs it. Threads are complicated.

Cure::Address's are assigned to thread spaces by an index called Thread. This allows addresses within the same address space to communicate without the need for lock and unlock. Calls to read/write/push/pull/isThreadSafe as

[8] I know I haven't told you what that is and you really don't need to know

well as add/remove on theAddressBank() all require the caller to have the lock. These methods will be referred to as 'locked methods'.

The convention is if an address has write/read/push/pull called on it then it has the lock on the thread space. It can then call the locked methods so long as it stays in its own thread space.

If an object needs to call a locked method outside its thread space it must first call getThread(thread) on theAddressBank() and releaseThread(thread) when done. An alternative is to call one of the readLocked ect. methods which get and release the lock for you.

This of course can in principal lead to deadlock if one thread holding its own lock then calls writeLocked on an address which in turn causes write lock on the first thread. That is how threads work and the programmer must be aware still of this. We haven't allowed the the programmer to forget about threads but we have set up a way to manage them with minimal locking.

Here is the main rule to avoid collisions:

'Any object that spawns a thread that then calls a locked method should get the lock and release the lock.'

So Device A has a thread checking some input. It gets data and wants to push it out to the world in another thread space. It should call pushDataLocked which gets the lock for it.

Exceptions can be made if the programmer is certain no other thread will be using the thread space in question. We have set this up as well using the Address::isThreadSafe, AddressBank::makePermanent, and ThreadSwitcher::setPushLocked. Explained a little later.

If Device A knows that the space is free it can just pushData with no lock. That is if it is using a ThreadMerger for instance to protect and queue. The input there is safe for multi-threading.

3.1 theAddressBank()

The process needs to tell the AddressBank how many Address spaces there are and how deep to make the hash table for each. The hash table is made in sizes of powers of 2. The size should be about the number of addresses in the address space for best effieciency. One specifies the power of two as a depth. This can look like this:

```
unsigned short depths[4];
depths[0]=7;
depths[1]=3;
depths[2]=2;
depths[3]=2;
Cure::AddressBank::theAddressBank(4,depths);
```

These lines of code should be the first Cure thing in main. Calling theAddressBank() the first time will create the static object with the arguments given. Subsequent calls to theAddressBank() just get the static object. If one is not setting up multi-thread spaces one does not worry about this or call anything special in main.

3.2 ThreadMerger

The ThreadMerger is another way to manage threads. It takes many threads on the inputs and creates a single thread output. Typically devices are set to push to the ThreadMerger inputs and the outputs are pushed from the ThreadMerger to the rest of the app.

If the ThreadMerger is not going to be deleted until after the device and no other Address is in the thread space then locking the thread space is not needed. The input threads are set to match those of the devices and the output threads match the input thread of the modules being pushed to.

The ThreadMerger then makes most of the mutex calls and organizes the queue of data. No other objects need to lock except under config and delete. A bunch of input devices pour data on the ThreadMerger inputs. The inputs are mutex protected by the ThreadMerger itself and the data put in the queue. The ThreadMerger pusher thread is awaken and it gets data from the queue and pushes out to its outputs, locked. It would even be possible to eliminate having the ThreadMerger lock its output space if one knew that only the ThreadMerger pusher thread was operating here. Then set ThreadMerger.m_PushLocked=0. Not really to be recommended since you may later start assuming the thread space in question is protected and write to it from some other thread getting the now pointless lock.

The ThreadMerger can write to different thread spaces on its different outputs. Each output pushes the data written to the corresponding input but the data is now sequenced by being passed through the single ThreadMerger queue.

3.3 Permanent Addresses

Device addresses often can have two way data flow. In those cases they would normally have to protect their data internally from multi-thread access. In that case again no lock is needed if the device is going to be permanent. It must be permanent in the sense that it will be assigned a special address Index that will not be affected by other threads adding or deleting addresses in its address space while a second address is trying to write to this thread. In principle that could cause a segmentation fault if this make permanent procedure is not followed.

This is how you set up such a no locking push/pull. The address, lets call it A, that doesn't need to have the address space locked must be set up as permanent. This is done by calling:

A->setCanPermit()

on the address A. This should only be done if one knows it will be safe either by internal protection in A or the way the program is set up. A can NOT now assume it has the lock on its thread space when written to.

Then if one uses an AddressMaker subclass to do the hookup from a ccf file one can just say

```
HOOKUP
Push B to A
```

Where B is some other address in some other Thread space. The AddressMaker will see the problem of pushing between two thread spaces and insert a ThreadSwitcher between them. The ThreadSwitcher will then do:

A->isThreadSafe(B)

on A when it gets its first data to push out. If A succeded in setCanPermit() it will now return true to isThreadSafe and record B's Thread index. After this the thread switcher will just call write on A without any locking. A can NOT assume it has the lock on its thread space when written to. (I know I just said that.)

When A is deleted it will lock B's Address space to be sure that B doesn't try to write to A during removing A from the AddressBank. Once A is removed B has no way of finding A anymore and will disconnect automatically.

So the trick is to just push between the two thread spaces as if you were not aware of it, after having setCanPermit. This can be set up as part of the configuration of A and done in the ccf file. Alternatively without an AddressMaker just do

```
A->setCanPermit();
ThreadSwitcher ts(B->Thread,A->Thread);
B->push(ts.in(0));
ts.out(0)->push(A);
```

If you want it to use the locks just leave out the A->setCanPermit().

If data is flowing both in and out of the process a separate ThreadMerger can be used for the output devices. This would work the same except the inner application thread computing the output data would write to the ThreadMerger queueing the data to be written to the output. This can also be handled by the device itself as in the SocketAddress which has no difficulty with multiple threads writing to it while it is reading its socket. So the Socket can be A and the application thread output address B above.

3.4 More Thread Advice

The addresses themselves need to be clear about their own threads. So if an address is on a filter and calls write on the filter then the filter has the lock and can do only things in that thread space. Most filters operate in only one thread space and can be totally indifferent to threads. If that is not the case the filter must set up critical regions and mutex protect its data. The filter has its own thread index which is the thread that its constructor and destructor runs in. It will try to get any other locks it needs to be made or deleted.

So if fiterX is made with thread=4 and it will first

make all its addresses on thread 4.

It then can do any setup on the ports.

Then if the filter subclass is multi-threaded (rarely the case): To change the Ports threads the subClass would call initPorts which gets locks as needed.

This is all handled in the DataFilter base class constructor/destructor/initPorts. When destroyed it will get the lock again on any addressess that are not in its thread space and change them to its thread space. Then delete them.

4 Cure Configuration and the AddressMaker

Cure give a lot of support for configuration of Cure apps. We have our older config style which is used by WrappedSLAM and other popular apps and we have since developed that into a language for creating custom applications at run time.

The configuration files are organized in sections with section names, (tags), followed by lines of ascii text. The apps can then ask the ConfigFileReader for a list of these lines which it must then parse and interpret.

The *.ccf files can include other *.ccf files and the ConfigFileReader will search for a tag until it finds it. So the search path matters. If the same tag appears twice in the path it is the first one that is used. This allows overriding of default settings. The search path is a recursive depth first on the include list.

The more detailed and powerful configuration uses the ConfigFileReader together with a subclass of AddressMaker to give run time configuration control on such aspects as which modules get created and how the individual module input and outputs are connected. For instance, one can created sockets or files and send specified data to them in the *.ccf files.

The main program might look like this:

1. SubclassOfAddressMaker am;
2. am.configure(configfile);
3. am.startDevices();
4. pthread_mutex_lock(&signal_mutex);
5. pthread_mutex_lock(&signal_mutex);
6. am.stopDevices();
7. am.deleteAll();

Lines 4 and 5 are just one way to have the main thread wait for an exit signal and of course require some more code elsewhere to unlock the mutex. We start with line 1. The SubclassOfAddressMaker must be written which is easy. It has to provide the makeDeviceSubClass(..), makeServiceSubClass(...), and makeFilterSubClass(...) methods which call the constructors of the modules you want to be able to make. The parent class, AddressMaker, will then know how to do the rest. One also has a place to add quick hacks to the configuration of these modules in this subclass if one wants to.

Line 2 sets in motion the parsing of the configfile which is a *.ccf file. This will now be explained.

AddressMaker::configure(const std::string &cfgFile):

1. checkDescriptors(cfgFile);
2. makeServices(cfgFile);
3. makeFiles(cfgFile);
4. makeDebugs(cfgFile);
5. makeBanks(cfgFile);
6. makeFilters(cfgFile);
7. makeDevices(cfgFile);
8. hookup(cfgFile);

It starts by examining the defined DATADESCRIPTORS. These are a list of 'Name Class ID SensorName'. That bind a Name to a DataDescriptor object. Then the rest of the *.ccf files can refer to the name instead of specifying the entire descriptor. These should be set up to be unique, one to one bindings. A warning is printed if that is not true.

The SensorName is looked up in another section of the *.ccf tagged SENSORS. This section binds a SensorName to a SensorType and SensorID. The sensor type is specified in:

getSensorType(const std::string &match,unsigned short &sensortype)

which can be found in SensorData.cc. As of writing this we have:

- "LongBaseLine" SensorData::SENSORTYPE_LONG_BASELINE;
- "Unknown" SensorData::SENSORTYPE_UNKNOWN;
- "Sick" SensorData::SENSORTYPE_SICK;
- "Camera" SensorData::SENSORTYPE_CAMERA;
- "Sonar" SensorData::SENSORTYPE_SONAR;

- "Contact" SensorData::SENSORTYPE_CONTACT;
- "GPS" SensorData::SENSORTYPE_GPS;
- "Inertial" SensorData::SENSORTYPE_INERTIAL;
- "RangeBearing" SensorData::SENSORTYPE_RANGEBEARING;
- "Compass" SensorData::SENSORTYPE_COMPASS;
- "Robot" SensorData::SENSORTYPE_ROBOT;
- "Odometry" SensorData::SENSORTYPE_ODOMETRY;
- "Position" SensorData::SENSORTYPE_POSITION;
- "Actuation" SensorData::SENSORTYPE_ACTUATION
- "Altitude" SensorData::SENSORTYPE_ALTITUDE
- "Depth" SensorData::SENSORTYPE_DEPTH

It is not really neccesary that the sensor type name string be the actual sensor used but the SensorData SENSORTYPE enumerator should be the same as the SensorType member of the data. So some TimestampedData have no SensorType at all such as Pose3D. Other such as SICKScan will always be a particular sensor type while others such as SensorPose might have any sensor type.

Each of the make methods work similarly[9]. Each has a section that contains a list of 'Create tag thread', where tag is a header to another section of the *.ccf files that specifies how to create and configure this object. The thread is the thread index of the address space to create this object in. The tag is looked up and the information in its section interpreted.

For example the Services are specified in the SERVICEADDRESS section of the *.ccf. That section might look like:

```
SERVICEADDRESS
Create LatLongService 0
Create PoseInitService 0
```

The Create line tells the AddressMaker that it must look for a section with tag LatLongService and run makeServiceSubClass(...) using what it finds there. The LatLongService section might look like this:

```
LatLongService
Create LatLongService
Global GPSOrigin LatLongOrigin
m_SomeClassMember 3.987
```

It is the Create line that is passed to makeSeviceSubClass. The Global line is used to set up any global parameters that this service might use. The values of these are specified in yet another section of the *ccf file, the GLOBAL section. In that section GPSOrigin is bound to a set of values and possibly a Covariance matrix. The string LatLongOrigin is defined in the LatLongService class and gives the binding internal to this particular service class. So another

[9] Except for FileAddresses which are slightly differently created, see section 2.1.

class might also want the GPSOrigin but it might be bound to something with another name internally. Global configuration can be done to DataFilters as well as Services and works the same way.

The AddressMaker assigns integer id's to all the global parameters which are passed to the objects along with the values. This id can then be used to match parameters being estimated by different modules. So one global might be air temperature. Two modules may need the initial value of this temperature. Later one of the modules may be able to estimate a better temperature. Having the id allows the other module to understand that this better value is for the temperature. The binding is made indirectly by using the same global initial value.

This particular Service[10] has a simple configuration. Any number of config lines can be added and each line will be passed to the object configService method which defines how it is interpreted. In this case the member called m_SomeClassMember will be set to 3.987.

Besides Global there is one other keyword that is understood by the ServiceAddress base class as requiring lookup in the *.ccf file. That is the Sensor keyword. This will look up the SENSOR section to bind a name to a particular Sensor that the class will be using. The offset to the sensor can be specified as a Global parameter. This also works the same for DataFilters.

The other types of addresses are configured in similar ways. The FileAddress allows detailed formating of the data written to the file. The DebugAddress allows one to specify the debug level and start and stop timestamps for debug output.

The MakeFilterBank looks up a section that might look like:

```
FILTERBANKADDRESS
Create FilterBank1 0
Create FilterBank2 3
```

This tells the AddressMaker to look for the named sections and make filter bank objects based on what is there.

```
FilterBank1
addPort 10 IMURaw
addPort 10 OdometryRaw
```

This says to add two ports to the FilterBank one that will buffer the IMURaw data and one that will buffer the OdometryRaw data. Those names would be defined in DATADESCRIPTORS. Now one can write a stream of data to the FilterBank and it will parse it (ie. send the differnt data to different ports) according to the DataDesecriptor definition. The ports can then be set to push to (or be pulled from) appropriate addresses. There is no limit on the number of ports. The buffer size is set here to 10 by the number after

[10] The actual LatLongService is not in Cure at all but is used for illustrating here.

addPort. It is implemented as a DataSlotAddress so the oldest data is simply overwritten by the newest data.

MakeFilters is the most complicated configuration sequence. It allows one to specify the DataDescriptor for each input and output, global parameters, sensors, as well as arbitrary configuration lines that are parsed by the individual DataFilter subclass. All the inputs and outputs should at least be given names. These names need not appear in the DATADESCRIPTORS section but if they do then the filter can set the entire DataDescriptor allowing some type checking on the port.

The names are used by the Hookup method to find the right address on the filter. So both DataFilters and FilterBanks have multiple ports and need to have strings assigned for specifying them in Hookup. The FilterBank takes its names from the DATADESCRIPTOR name. The DataFilter takes them from the InputData and OutputData lines of its config section.

DeviceAddress also has an OutputData configuration line but it has only one address so the DataDescriptors specified on that line are used differently. They are assigned to the various data being pushed out of the device and the order they are specified in the OutputData config line is used to bind them to a particular data being produced by this device. By assigning DataDescriptors to the output of the device one can parse the data in a FilterBank downstream. The data stream will by then have passed a ThreadMerger or some other protection.

The Devices are started in the order they are given in the DEVICEADDRESS section. They are stopped in the reverse order. The deleteAll method deletes the objects in the reverse order from that which they were created in.

Hookup will read the final section of the ccf:

HOOKUP
Push ThreadMerger MergeOut1 to FilterBank1
Push ThreadMerger MergeOut2 to FilterBank1
Push OdoSerial to ThreadMerger MergeIn1
Push IMUSock to ThreadMerger MergeIn2
Push FilterBank OdometryRaw to OdoFilter OdometryRaw
...

This specifies the address by name. The name is that already given and for FilterBank1's input it is its tag. The DataFilters need two strings so the tag of the filter and the input or output name. Pull can be done the same way. All the addresses that were created can be hooked up in this section. Interprocess communication consists of creating SocketAddresses and pushing the data to/from them. The Address Maker object will always be able to find a particular address so it would be easy to set up a simple name service that listens to a socket and sends data as requested. This allows one to write monitor programs that can get at any data flowing inside your app. We should do that.

5 Conclusion

Cure is a wonderful thing! Use it.