

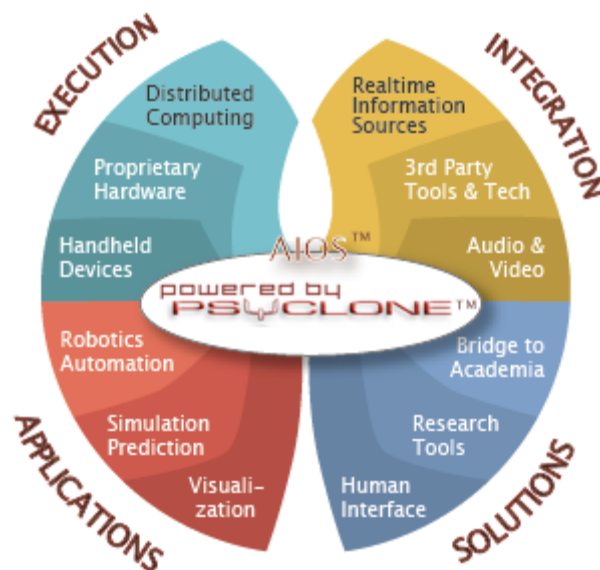


Psyclone Platform

System documentation

Communicative Machines

Last updated: 19 July 2018



Contents

Introduction	5
Licensing.....	6
Supported Platforms and Programming Languages	6
Prerequisites	6
Psyclone overview.....	7
What is Psyclone	7
What can you do with it.....	7
Running Psyclone	7
Running Psyclone on Windows.....	8
Running Psyclone on UNIX.....	8
Building Psyclone from source.....	9
Using Visual Studio 2015 (Windows)	9
Using Make (Linux).....	10
The Psyclone system and the PsySpec XML file	11
A simple PsySpec file: pingpong.....	11
Message types and wildcards	12
Component cranks and libraries	14
One-shot vs continuous components	15
Local Persistent (Private) Data	16
Component parameters.....	16
Providing custom configuration data.....	18
PsySpec variables	18
PsySpec including other files.....	19
Passthrough modules.....	20
Messages and signals.....	21
Dataflow parameters	21
After/delay	21
Interval	21
To	22
Filters.....	22
Maxage.....	22
From	22
To	22
Tag.....	23

More advanced filtering.....	23
Tagging.....	23
System parameters.....	24
Main system port.....	24
Verbose and debug output.....	24
PsyProbe HTML files location.....	25
PsyProbe port.....	26
Contexts.....	27
Overview.....	27
Python modules.....	30
Inline Python modules.....	30
Python code file modules.....	31
Python import libraries and paths.....	31
Components: Modules, Whiteboards and Catalogs.....	32
Modules.....	32
Whiteboards.....	32
Catalogs.....	34
The File Catalog.....	34
The Data Catalog.....	35
The Replay Catalog.....	36
The Request Store Catalog.....	37
Creating custom Catalogs.....	38
Data Messages.....	41
Types.....	41
User contents.....	41
PsyProbe Web Interface.....	42
System overview.....	42
Tab descriptions.....	43
Using PsyProbe.....	44
Messaging activity.....	45
Communication statistics.....	47
Custom views.....	47
Creating custom tabs.....	48
Working with private data.....	49
Use a RequestStore Catalog.....	49
Use a custom PsyProbe subsite.....	51

Services and interfaces	52
Distributed Psyclone systems using Nodes.....	54
Nodes	54
Process separation using Spaces.....	55
Spaces	55
External spaces	55
Communication between separate Psyclone systems	57
Remote requests.....	57
Recognising a remote query	57
The CMSDK library	58
API documentation	58
Compiling the CMSDK library.....	58
Using Visual Studio 2015 (Windows)	58
Using Make (Linux).....	59
Linking with the CMSDK library.....	59
Using Visual Studio 2015 (Windows)	59
Using Make (Linux).....	60
Using the CMSDK library	60
CMSDK Licensing.....	61
Use Cases	62
Integration of third-party applications	62
Grid data processing	62
Agent-based simulation	63
Interactive robots.....	63
Tutorials	64
Creating your first Psyclone system.....	64
Adding your own modules	66
Creating Python modules.....	68
Inline Python modules	68
Python code file modules	69
Python import libraries and paths	69
Creating external modules.....	70
Creating your own catalogs	72
Add custom data in PsyProbe	74
Add a custom tab in PsyProbe	75
Example of a PsySpec for a large system	76



Introduction

Psyclone is a platform engineered to support large-scale modular applications and integration of disparate software and hardware products. It provides developers with the ability to create a few or thousands of modules, running on a single computer or a grid of heterogeneous systems, communicating with each other using messaging and signalling in a context sensitive publish/subscribe (PubSub) environment. And it provides admins and integrators with a production-ready environment for integrating own and third-party software components in a near real-time system with live monitoring and built-in load balancing.

Examples of applications of the Psyclone platform are:

Grid-based live data analysis (more details) – the system read live data from the network and applied a pipeline of algorithms with branching to processing the data in near real-time. The data flowed between modules using messages and the flow itself was adjusted at runtime based on a set of global contexts. Each module applied a specific algorithm to the data, but during times of heavier resource use lower power algorithms were applied to keep up with the incoming dataflow. The output data was sent to a database and also made available as a stream of data over the network.

Agent-based simulations (more details) – a large system with hundreds of modules was distributed across a number of computers, but configured by a single configuration file and monitored via the PsyProbe web interface. To synchronise the processing of all the modules signals were used to inform the whole system of the next time step in the simulation which made it possible to allow some steps to take longer and use more accurate algorithms. Each entity in the simulation was represented by around 10 modules which worked together as a mini system within the larger architecture. Custom visualizers were created in PsyProbe to show the state of each entity rather than just the modules.

Interactive robotics (more details) – the CoCoMaps project used the Psyclone platform to create robots that could communicate with multiple humans and collaborate with the humans and other robots on completing dynamically specified tasks. Each robot ran its own Psyclone system which included a ROS interface, multi-camera vision, face and emotion recognition, speech recognition and generation, dialogue and task planning and navigation control. Multiple independent Psyclone systems used an external Psyclone Catalog to negotiate about observations, roles and tasks.

Some of the unique selling points of Psyclone are:

- A unique publish/subscribe architecture with
 - Single point of start-up configuration
 - Dynamic add/change/remove at runtime
- Multi-OS (Windows, UNIX) and multi-language (C++, Python, Java) in the same system
- Near real-time rerouting of local or global dataflow based on context, data and resources
- Dynamic switching of algorithms based on context, data and resources
- Web-based operator and monitoring interface with custom plug-ins



Licensing

Psyclone is released under a dual licence – LGPL Open Source for academic and commercial use alike and a Commercial licence which includes support and priority of change requests. Psyclone includes the CMSDK library which is released separately under a BSD licence. Both are standard for these licenses in general terms except for the addition of the **CADIA Clause**:

- **CADIA Clause:** The license granted in and to the software under this agreement is a limited-use license. The software may not be used in furtherance of: (i) intentionally causing bodily injury or severe emotional harm to any person; (ii) invading the personal privacy or violating the human rights of any person; or (iii) committing or preparing for any act of war.

Supported Platforms and Programming Languages

Psyclone currently runs on Windows 7 and later and compiles in Visual Studio 2015 and later. It runs on recent Linux and other X86 compliant UNIX systems and compiles with GCC 5+ and GLIBC 2+. The roadmap includes supporting the Mac OSX platform as well.

Prerequisites

Psyclone itself doesn't require any third-party libraries (other than the included CMSDK) to compile or to run. On Windows the standard Visual Studio libraries are dynamically linked and on Linux it uses pthread, dl and rt.



Psychone overview

What is Psychone

Psychone is a platform which allows developers to create and execute modules which subscribe to input data and produce output data in a publish/subscribe system. A module can be very simple or run hundreds of lines of code using multiple threads and it can even be embedded into another existing application. A module can subscribe to input data based on type and its subscription can change dynamically over time depending on global contexts or even be rewritten completely at runtime.

A Psychone system can contain hundreds or thousands of modules and each module will contain one or more crank functions which are called to process input data and optionally output results of this processing. Other more advanced modules (called Catalogs) can interface with other systems and/or store data which can be queried by any other module using a simple query language.

A full Psychone system can run on a single Windows or UNIX computer or span many (mixed OS) computers where its components (modules and catalogs) are distributed either manually or automatically – and the components can even be moved to other computers at runtime for load balancing.

Psychone is a commercial grade platform which offers process separation and segmentation which means that it is often used to mix commercial grade modules with experimental and/or academic code. If one module misbehaves or crashes the rest of the system can be protected from this event and the unfortunate module can simply be restarted to allow the system to continue operating as normal.

What can you do with it

The Psychone platform specifically targets large modular and scalable applications with a high volume throughput of heterogeneous data. Application types include

- Single computer standalone applications
- Single computer integration systems linking multiple and otherwise incompatible third-party applications
- Dynamically scalable modular systems with thousands of modules being load balanced across many heterogeneous physical and/or virtual computer systems
- Near real-time systems requiring awareness of real and CPU time with the ability to dynamically adapt to resource shortages or dataflow delays

Psychone supports incremental development, debugging and testing of large systems where whole areas of functionality are implemented in draft initially and gradually fleshed out as the system complexity grows.

The PsyProbe web interface provides the ability to monitor the dataflow of the whole system or individual modules alike.

Special built-in modules allow the developer to record parts of or the whole dataflow from a live system and play it back in offline mode while developing new functionality or debugging existing modules.

Running Psychone

Psychone currently runs on X86-based Windows and Posix-compatible UNIX platforms. The roadmap includes supporting Mac OSX as well.

Psychlone overview -- Running Psychlone

Running Psychlone on Windows

On Windows you would normally run Psychlone on the command line like this:

```
<path>\Psychlone.exe spec=<PsySpec XML file>
```

The PsySpec file* contains the full system specification including all the modules and system parameters. Some of these can for convenience also be specified on the command line:

```
port=<network port>      defaults to 10000
html=<path to html dir>  defaults to ./html
```

To run a Psychlone in Satellite mode (as a Node to a master Psychlone on another computer) just start up Psychlone on the command line without any parameters or optionally specify the port if different from port 10000:

```
<path>\Psychlone.exe [port=<network port>]
```

Running Psychlone on UNIX

On UNIX you would normally run Psychlone on the command line like this:

```
<path>/Psychlone spec=<PsySpec XML file>
```

The PsySpec file* contains the full system specification including all the modules and system parameters. Some of these can for convenience also be specified on the command line:

```
port=<network port>      defaults to 10000
html=<path to html dir>  defaults to ./html
```

To run a Psychlone in Satellite mode (as a Node to a master Psychlone on another computer) just start up Psychlone on the command line without any parameters or optionally specify the port if different from port 10000:

```
<path>/Psychlone [port=<network port>]
```

* For more information about the content of the PsySpec file please see the PsySpec section.

Building Psychone from source

Psychone itself consists of 5 projects:

- CMSDK is the core library providing OS dependent functionality and lots and lots of the base code.
- Psychone is the main executable, links with CMSDK
- System provides components such as Whiteboards, Catalogs, etc. which are loaded by Psychone when the PsySpec defines one or more of these. It only links with CMSDK and builds a single DLL.
- Examples provides example modules which are loaded by Psychone when the PsySpec defines one or more of these. It only links with CMSDK and builds a single DLL.
- External Modules shows how third-party programs can link with CMSDK and connect to a running Psychone system - i.e. these modules actually run inside the third party executable.

The Python2 and Python3 projects are used to port important parts of CMSDK to Python 2.x and 3.x using SWIG. These libraries can then be used

- a) for accessing CMSDK functionality from within a Python program
- b) by the PythonLink DLLs which Psychone loads to enable users to write whole modules in just Python 2 or 3.

You need to have Python 2.7 and/or 3.5 installed for either or both 32-bit and 64-bit versions to compile these.

Using Visual Studio 2015 (Windows)

In the Open Source distribution of Psychone find the Visual Studio solutions file in the top directory called **Psychone.vs2015.sln**. Open this in Visual Studio 2015 and compile the whole project either in Debug or Release mode. To build the SSL version of Psychone select either Release SSL or Debug SSL. For this to work the following path must exist and contain the OpenSSL include files:

```

..\..\_libs\OpenSSL\openssl-1.0.2g-vs2015\include           for 32 bit
..\..\_libs\OpenSSL\openssl-1.0.2g-vs2015\include64       for 64 bit

```

and the library files

```

..\..\_libs\OpenSSL\openssl-1.0.2g-vs2015\lib             for 32 bit
..\..\_libs\OpenSSL\openssl-1.0.2g-vs2015\lib64           for 64 bit

```

To use modules written in the Python language you will need to build either the Python2Link and/or the Python3Link DLLs. You do this by selecting them in the Solution Explorer, right click and select Build or Rebuild. For this to work you will need Python 2 and/or 3 installed in either 32 and/or 64-bit versions in the following directories:

```

C:\Python27          64-bit version of Python 2.7
C:\Python27-32       32-bit version of Python 2.7
C:\Python35          64-bit version of Python 3.5
C:\Python35-32       32-bit version of Python 3.5

```

Other versions and locations can be used by editing the Visual Studio CMSDKPY2, CMSDKPY3, Python2Link and Python3Link project files.

Psychlone overview -- Building Psychlone from source

Using Make (Linux)

In the Open Source distribution of Psychlone find the main Makefile in the top directory called **Makefile**.

On the command from this directory line issue the command

```
make           to build everything for release
make debug     to build everything for debug
```

To build the SSL version of Psychlone run either

```
make ssl           to build everything for release including SSL support
make ssldebug      to build everything for debug including SSL support
```

For this to work the SSL development libraries must be installed on the computer (used via `lssl` and `lcrypto`).

To use modules written in the Python language you will need to build either the Python2Link and/or the Python3Link DLLs. You do this by running

```
make python2           to build all files needed for Python 2 integration in release
make python2 debug     to build all files needed for Python 2 integration in debug
make python3           to build all files needed for Python 3 integration in release
make python3 debug     to build all files needed for Python 3 integration in debug
```

For this to work you will need Python 2 and/or 3 packages installed in the following directories:

```
PYTHON2INCLUDE=-I/usr/include/python2.7
PYTHON3INCLUDE=-I/usr/include/python3.4m
PYTHON2LIB=-lpython2.7
PYTHON3LIB=-lpython3.4m
```

Other versions and locations can be used by editing the CMSDK/Makefile.sys file entries for these.

The Psyclone system and the PsySpec XML file

A running Psyclone system will usually contain a number of components (modules, Whiteboards, Catalogs), some of these provided as part of the system and some created using the SDK by users of the system. When starting up Psyclone an XML-based configuration file called a PsySpec can be specified on the command line which tells Psyclone which components to start up and how the data should flow between them.

The PsySpec file is written in XML and forms the blueprint of the system architecture at start-up time including which components to run and the dataflow between them. Once Psyclone is running the system can dynamically create new modules without these being specified in the PsySpec.

A simple PsySpec file: pingpong

A simple pingpong PsySpec could look like this:

```
<psySpec>
  <module name="Ping">
    <trigger name="Ready" type="Psyclone.Ready" />
    <trigger name="Ball" type="ball.1" />
    <crank name="Ping" function="Ping" />
    <post name="Ball" type="ball.2" />
  </module>
  <module name="Pong">
    <trigger name="Ball" type="ball.2" />
    <crank name="Pong" function="Ping" />
    <post name="Ball" type="ball.1" />
    <post name="Done" type="Psyclone.Shutdown" />
  </module>
</psySpec>
```

This spec creates two simple modules, one called Ping and the other called Pong. They are both created the built-in crank function called Ping which means that no user code is required to run this system.

The dataflow between the modules goes as follows:

- 1) On system startup the Psyclone system itself posts the Psyclone.Ready message
- 2) Immediately the Ping module is triggered with this message as input
- 3) Once this message has been processed Ping posts an output message with type ball.2
- 4) Immediately the Pong module is triggered with the ball.2 message as input
- 5) Once this message has been processed Pong posts an output message with type ball.1
- 6) Immediately the Ping module is triggered with the ball.1 message as input
- 7) Once this message has been processed Ping posts an output message with type ball.2

And this back-and-forth flow continues until 100,000 messages have been received after which the Pong module posts a message with type Psyclone.Shutdown and the system shuts down. Every 10,000 messages the current throughput performance it logged to the console in lines like:

```
[1] 28/06/2018 08:44:11.323.934 [Ping] Got msg 40000, avg msg time: 89.4us, avg msg age: 89.0us (1.1us faster)...
```

To run this in Psyclone use the following command line:

```
Psyclone spec=pingpong.xml
```

The Psyclone system and the PsySpec XML file -- Message types and wildcards
The system would run for a while and then exit, producing the following console output:

```
28/06/2018 08:44:03.947.650 Psyclone Node starting on port 10000...
[0] 28/06/2018 08:44:04.047.825 Parsing configuration and setting up system...
[0] 28/06/2018 08:44:04.051.347 *** Single Node ready, continuing system setup ***
[1] 28/06/2018 08:44:04.051.455 Success synchronising ID Manager...
[1] 28/06/2018 08:44:04.051.502 Configuring local node...
[1] 28/06/2018 08:44:04.053.195 Creating Module 'Ping'...
[1] 28/06/2018 08:44:04.053.286 SetupQ need more memory size: 5378344 use: 5377048 need: 2352
[1] 28/06/2018 08:44:04.058.505 PsySpace 'Root' (1) starting up...
[1] 28/06/2018 08:44:04.161.087 Configured Module 'Ping' (1) successfully
[1] 28/06/2018 08:44:04.161.204 Creating Module 'Pong'...
[1] 28/06/2018 08:44:04.162.020 Configured Module 'Pong' (2) successfully
[1] 28/06/2018 08:44:04.162.130 Sending config to all other nodes...
[1] 28/06/2018 08:44:04.162.186 *****
[1] 28/06/2018 08:44:04.162.232 ***** SYSTEM READY *****
[1] 28/06/2018 08:44:04.162.274 *****
[1] 28/06/2018 08:44:04.163.309 Space 'Root' connected
[1] 28/06/2018 08:44:04.163.233 [Ping] Started running (internal)...
[1] 28/06/2018 08:44:04.163.288 [Ping] Starting 10 cycles test...
[1] 28/06/2018 08:44:04.163.577 [Pong] Started running (internal)...
[1] 28/06/2018 08:44:05.937.545 [Ping] Got msg 10000, avg msg time: 88.7us, avg msg age: 88.4us...
[1] 28/06/2018 08:44:07.724.488 [Ping] Got msg 20000, avg msg time: 89.3us, avg msg age: 88.7us (0.7us slower)...
[1] 28/06/2018 08:44:09.535.502 [Ping] Got msg 30000, avg msg time: 90.5us, avg msg age: 90.4us (1.2us slower)...
[1] 28/06/2018 08:44:11.323.934 [Ping] Got msg 40000, avg msg time: 89.4us, avg msg age: 89.0us (1.1us faster)...
[1] 28/06/2018 08:44:13.051.856 [Ping] Got msg 50000, avg msg time: 86.4us, avg msg age: 86.1us (3.0us faster)...
[1] 28/06/2018 08:44:14.799.030 [Ping] Got msg 60000, avg msg time: 87.4us, avg msg age: 87.2us (1.0us slower)...
[1] 28/06/2018 08:44:16.566.469 [Ping] Got msg 70000, avg msg time: 88.4us, avg msg age: 88.1us (1.0us slower)...
[1] 28/06/2018 08:44:18.309.929 [Ping] Got msg 80000, avg msg time: 87.2us, avg msg age: 86.9us (1.2us faster)...
[1] 28/06/2018 08:44:20.100.120 [Ping] Got msg 90000, avg msg time: 89.5us, avg msg age: 89.2us (2.3us slower)...
[1] 28/06/2018 08:44:21.848.279 [Pong] Got msg 100000, avg msg time: 87.4us, avg msg age: 88.0us (2.1us faster)...
[1] 28/06/2018 08:44:21.848.336 *****
[1] 28/06/2018 08:44:21.848.526 ***** SYSTEM SHUTDOWN *****
[1] 28/06/2018 08:44:21.848.576 *****
[1] 28/06/2018 08:44:21.848.620 Module Pong requested system shutdown, shutting down...
[1] 28/06/2018 08:44:21.898.589 Local Node is preparing to shutdown...
[1] 28/06/2018 08:44:21.899.068 PsySpace 'Root' (1) shutting down...
[1] 28/06/2018 08:44:22.001.197 PsySpace 'Root' has shut down (1, 00000000057D8B0)
28/06/2018 08:44:22.959.571 Node Networking is shutting down...
28/06/2018 08:44:23.379.153 Psyclone Node on port 10000 has shutdown successfully...
```

The first part of the output up until SYSTEM READY shows the system starting up and configuring itself according to the PsySpec. After this the Psyclone.Ready message is posted and the system starts running, outputting the stats after each batch of 10,000 messages. Then the system receives the message to shut down and it goes through the process of doing just that.

Message types and wildcards

All messages in Psyclone have a type in dot notation. The names of each part or level are ASCII strings and can have any length. They would usually be application dependent and be used to semantically group different messages, such as

```
input.video.raw
input.audio.raw
input.audio.normalised
```

Types are normally applied to output messages posted by components and used in triggers as part of the subscriptions for Psyclone to deliver a copy of the message to any other component which has asked for it. If no other component has a matching trigger then the message will be discarded (or saved in shared memory if it has a time-to-live (ttl) specification).

A trigger specification can be either for the complete type name `input.audio.raw` or it can use wildcards like `input.audio.*` or `input.*.raw`. The trigger specification also needs a name so the code can identify the trigger in case a module has more than one. The reason for this separation of type and name is so the dataflow can be altered by only editing the PsySpec file and without requiring the actual code to be changed. For example, a video processing module may initially receive raw video frames:



```
<trigger name="VideoFrame" type="input.video.raw" />
```

but later the developer wants to test the same code on smoothed video frames, so all they need to do is change the spec to

```
<trigger name="VideoFrame" type="input.video.smoothed" />
```

and the system will simply feed the output of different module in instead. This is because the code for the component only ever refers to the name and not the type:

```
if ( inMsg = api->waitForNewMessage(100, triggerName)) {
    if (strcmp(triggerName, "Ready") == 0) {
        ...
    }
}
```

Similarly goes for output messages

```
<post name="OutputFrame" type="input.video.filtered" />
```

where the post type can be changed in the spec without having to change the code as the code only ever refers to the name:

```
api->postOutputMessage("OutputFrame", outMsg);
```

which returns an integer value informing the calling code how many messages were posted. If this integer is negative the post failed for the following reason:

```
POST_FAILED -1      Error occurred while trying to post the message
POST_NOSPEC -2      The crank tried to post a message before a trigger arrived
POST_OUTOFCONTEXT -3 The crank is no longer active because the context has changed
                    and is no longer active
```

The post spec in the PsySpec can have the following parameters

name=	A module-internal alias specified by the crank of the module. Any crank connected to Triggers with that name will be activated
type=	This is the triggertype associated with the post. The type is the relevant feature that the module subscribes to
maxage=	In addition to having the correct type, a post's age has to be below this number to wake up the module. This can be used to skip triggers if the system is lagging behind
after=	determines the time that the module waits after receiving the trigger to activate the crank



Component cranks and libraries

Each component in the PsySpec will have at least one crank function to run. Module can specify one or more crank functions as part of their subscription, Whiteboards use a built-in crank function and Catalogs specify their crank function as the Type of the Catalog.

A crank function is the function name in the code library (a DLL or SO) which will be called when a trigger message arrives. Psyclone contains a number of built-in cranks such as the Ping crank above and these do not need to specify the library. For all other components a library needs to be created using the CMSDK, compiled and specified to inform Psyclone about which file on disk contains the compiled code for the crank.

Libraries containing crank functions are specified using one or more library entries in the spec:

```
<library name="MyExamples" library="Examples" />
```

The library name is how subsequent cranks refer to the library and the library entry tells Psyclone which actual file to load. On Windows Psyclone would look for the file Examples.dll in the current directory and on UNIX Psyclone would look for the file libExamples.so. (In addition, when running a debug build of Psyclone it will first look for the debug version of the library by appending Debug to the filename such as ExamplesDebug.dll or libExamplesDebug.so).

Once the library has been defined cranks can refer to the library names:

```
<crank name="Ping" function="MyExamples::Ping" />
```

A custom library can be created using just the CMSDK Open Source library, i.e. Psyclone does not need to be installed on the computer to compile a library, nor does it need the Open Source version of Psyclone to compile or run.

A crank function is normally defined in a C++ header file:

```
#include "PsyAPI.h"
namespace cmlabs {
extern "C" {
    DllExport int8 MyCrankFunction(PsyAPI* api);
    ... more crank function definitions ...
}
} // namespace cmlabs
```

and implemented in a C++ source file:



```
#include "MyCranks.h"

namespace cmlabs {

int8 MyCrankFunction(PsyAPI* api) {
    DataMessage* inMsg, *outMsg;
    const char* triggerName;
    if (api->shouldContinue()) {
        if (inMsg = api->waitForNewMessage(100, triggerName)) {
            api->logPrint(1, "Received trigger message: %s", triggerName);
            outMsg = new DataMessage();
            ... add custom data entries to outMsg ...
            outMsg->setInt("mycount", 3);
            outMsg->setString("mytext", "my data");
            api->postOutputMessage("MyPost", outMsg);
        }
    }
    return 0;
}
} // namespace cmlabs
```

(For Python-based modules please refer to the Python sections.)

A crank function takes one input parameter which is a pointer to a PsyAPI object. This object can be used by the function to communicate with the Psyclone system including waiting for new messages and posting output messages, but also for reading and changing parameters, retrieving or querying messages from other components, etc.

The definitions of crank functions need to have the `DllExport` classifier which makes them visible to Psyclone when loading the DLL/SO.

One-shot vs continuous components

This particular crank function runs once and returns, to be called again when the next trigger arrives. If you change the ***if shouldContinue()*** line to a while instead you can then keep the component running and do other stuff while it waits for more messages to arrive:

```
while (api->shouldContinue()) {
    if (inMsg = api->waitForNewMessage(100, triggerName)) {
        ... handle incoming message ...
    }
    else {
        ... do something else every 100ms ...
    }
    ... do something on every cycle even if a message arrived ...
}
```

The main difference between the two is that the continuous component keeps hold of its OS thread until the module is not supposed to run anymore whereas the thread running the one-shot component is returned to the thread pool to be used again for triggering other components. Psyclone will automatically create enough threads to always have a few spares in the thread pool and with a large number of components in a system a large number of threads could be used which might affect performance.

However, for components with significant amounts of initialisation work or large volumes of local persistent data it may be beneficial to make them continuously running.



Local Persistent (Private) Data

Any component in Psyclone can optionally choose to save its local data to persistent storage. This is useful for one-shot modules which exit in between processing trigger messages and for modules which might be migrated to another computer. It also offers the component the ability to show its local data in the Data tab in the PsyProbe web interface, if a mimetype is provided when saving the data.

Data is saved as a binary blob. To save local data to persistent storage a component can use the PsyAPI object:

```
api->setPrivateData("My Data", str.c_str(), str.length());
```

To read the data back in at a later time:

```
data = api->getPrivateDataCopy("My Data", size);
```

And to save the data and make it visible in PsyProbe:

```
api->setPrivateData("My Data", str.c_str(), str.length(), "text/html");
```

Component parameters

The PsySpec allows the designer to define one or more parameters for each component which are available to the crank code via the PsyAPI object. These can be static strings, integers, floats and collections of these and each parameter is interactive in that the local component or other components can tweak the value of a parameter, much like a knob on a control panel.

Simple parameters are defined like this:

```
<parameter name="MyStringValue" type="String" value="Some value" />
<parameter name="MyIntegerValue" type="Integer" value="5" />
<parameter name="MyFloatValue" type="Float" value="0.2" />
```

and accessed like this:

```
const char* str = api->getParameterString("MyStringValue");
int64 val = api->getParameterInt("MyIntegerValue");
float64 fval = api->getParameterInt("MyFloatValue");
```

The code can check if a named parameter exists

```
int64 exists = api->hasParameter("SomeParameterName");
```

and the code can change a parameter value using the setParameterXXX versions of these functions:

```
bool setParameter(const char* name, const char* val)
bool setParameter(const char* name, int64 val)
bool setParameter(const char* name, float64 val)
```

Any parameter can be reset to its initial value using

```
bool resetParameter(const char* name)
```




The crank code can ask about the type of a parameter

```
uint8 type = api->getParameterDataType("SomeParameterName")
```

which will return a number from this list:

```
#define PARAM_STRING      1
#define PARAM_INTEGER     2
#define PARAM_FLOAT       3
#define PARAM_STRING_COLL 4
#define PARAM_INTEGER_COLL 5
#define PARAM_FLOAT_COLL  6
```

The numeric parameters can specify a valid range:

```
<!-- any integer in range, initial value lowest value -->
<parameter name="MaxCount" type="Integer" value="[0:110]"/>

<!-- any integer in range, initial value 55 -->
<parameter name="MaxCount" type="Integer" value="[0:110]=55"/>

<!-- any integer in range, random initial value -->
<parameter name="MaxCount" type="Integer" value="[0:110]=*"/>

<!-- any float in range -->
<parameter name="Interval" type="Float" value="[-2.5:2.5]"/>
```

and even specify a step value

```
<!-- any integer in range, in steps of 2 -->
<parameter name="MaxCount" type="Integer" value="[0:110:2]"/>
```

Step values can also be specified on non-interval numeric parameters

```
<!-- any integer, start value 0, in steps of 2 -->
<parameter name="MaxCount" type="Integer" interval="2" value="10"/>
```

For all parameters with step values the code can tweak the parameter up or down instead of setting the new value outright:

```
api->tweakParameter("MaxCount", 2);
```

where the second parameter can be a positive or negative integer.

The following list types are supported:

```
<!-- any integer in list -->
<parameter name="Interval2" type="Float" value="[-2;1;3;9]"/>

<!-- any float in list -->
<parameter name="Interval2" type="Float" value="[-2.5;1.5;1.8;2.5]"/>

<!-- any string in list -->
<parameter name="Category" type="String" value="['str1';'str2';'str3']"/>
```



The PsyCone system and the PsySpec XML file -- Providing custom configuration data
Any of these can be queried for the current value using the normal typed `getParameter` API, tweaked up or down using the `tweakParameter` API and set outright using the `setParameter` API.

Providing custom configuration data

Part of a component's PsySpec configuration is an optional setup specification:

```
<module name="MyModule">
  ... triggers, posts, cranks, etc.
  <setup>
    ... custom data in valid XML format
  </setup>
</module>
```

The crank code can then read this data using:

```
std::string xml = api->getParameterString("componentsetup");
```

The CMSDK provides an easy to use XML parser originally created by Frank Vanden Berghen which you can use like this:

```
XMLResults xmlResults;
XMLNode node, subNode, mainNode = XMLNode::parseString(xml.c_str(),
    "setup", &xmlResults);

if (xmlResults.error != eXMLErrorNone) {
    if (xmlResults.error == eXMLErrorFirstTagNotFound) {
        api->logPrint(0, "Setup error, main tag '<setup>' not found");
        return -1;
    }
    else {
        api->logPrint(0, "setup parsing error at line %i, column %i:\n    %s\n",
            xmlResults.nLine, xmlResults.nColumn,
            XMLNode::getError(xmlResults.error));
        return -1;
    }
}
```

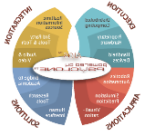
After this the XMLNode object can be queried using functions like `getChildNode` and `getAttribute`.

PsySpec variables

The PsySpec usually contains lots of information about components and lots of component variables. It is often beneficial to the developer to be able to specify certain data globally at the top of the file so it can be changed in one place and used universally.

So for example if two modules have parameters which contain the same values

```
<module name="MyModule1">
  <parameter name="SystemID" type="Integer" value="2" />
  <parameter name="SystemName" type="String" value="Bill" />
  <parameter name="SystemDir" type="String" value="/home/user/M1" />
  ...
```



```
<module name="MyModule2">
  <parameter name="SystemID" type="Integer" value="2" />
  <parameter name="SystemName" type="String" value="Bill" />
  <parameter name="SystemDir" type="String" value="/home/user/M2" />
  ...

```

it could be easier to just define the common parts at the top of the file

```
<variable name="SystemID" value="2" />
<variable name="SystemName" value="Bill" />
<variable name="MainDir" value="/home/user" />

```

and then use these variables in the PsySpec

```
<module name="MyModule1">
  <parameter name="SystemID" type="Integer" value="%SystemID%" />
  <parameter name="SystemName" type="String" value="%SystemName%" />
  <parameter name="SystemDir" type="String" value="%MainDir%/M1" />
  ...

<module name="MyModule2">
  <parameter name="SystemID" type="Integer" value="%SystemID%" />
  <parameter name="SystemName" type="String" value="%SystemName%" />
  <parameter name="SystemDir" type="String" value="%MainDir%/M2" />
  ...

```

The variable extraction is done using a simple string search and replace so it can be used for any part of the spec, not just variables, but also component names, crank information, triggers, etc.

Just make sure that the content of the variables fit into the XML at the point where the variable is specified – as the full XML will not be parsed until after all variables have been replaced with their actual values.

PsySpec including other files

A PsySpec file can grow to become very large and/or multiple PsySpec files may have sections in them which are common to them all. In this case it can make managing the specs easier to simply include one or more external files into a PsySpec as if it had been one big file in the first place.

This works similarly to include directives in other products such as C language code and to do this simply add the include line in a PsySpec file like this:

```
<include file="system.xml" />

```

Just make sure that the content of the included file fits into the XML at the point where the include is specified – as the full XML will not be parsed until after all files have been included and all variables have been replaced with their actual values.

Passthrough modules

Modules would normally specify at least one crank function in their spec, but if they do not a very simple default crank function will be used instead which simply passes through data messages. For every incoming trigger every post is activated with a copy of the incoming message.

This is very useful if one simply wants to change the type of a message:

```
<module name="DefaultRoleDetector">
  <trigger name="NowDefaultRole" type="Self.Role.Searcher" />
  <post name="DialogOff" type="dialog.off"/>
</module>
```

or to split a message into multiple messages, all containing the same data:

```
<module name="DefaultRoleDetector">
  <trigger name="NowDefaultRole" type="Self.Role.Searcher" />
  <post name="DialogOff" type="dialog.off"/>
  <post name="AutoNavigationOn" type="navigation.auto.on" />
  <post name="StopMicrophone" type="cmd.input.audio.off" />
</module>
```

It can also be used to introduce a delay in a message path:

```
<module name="FaceFinder">
  <trigger name="Start" type="Psyclone.Ready" after="5000" />
  <post name="Face" type="Input.Face.Start" />
</module>
```

or to post messages at a regular interval:

```
<module name="PostTester">
  <trigger name="Ready" type="Psyclone.Ready" />
  <trigger name="Post" type="Regular.Post" interval="1000" />
  <post name="Post" type="Post.Data.1" />
</module>
```

Custom data entries can be added to the output messages as well, either as a quick string key and value:

```
<module name="FaceFinder">
  <trigger name="Start" type="Psyclone.Ready" after="5000" />
  <post name="Face" type="Input.Face.Start" contentkey="MyKey" content="MyVal" />
</module>
```

or as a more elaborate entry adding several different types of content:

```
<module name="FaceFinder">
  <trigger name="Start" type="Psyclone.Ready" after="5000" />
  <post name="Face" type="Input.Face.Start">
    <content key="MyKey" type="String" value="MyVal" />
    <content key="MyKey" type="Integer" value="5" />
    <content key="MyKey" type="Float" value="3.21" />
  </post>
</module>
```

Messages and signals

Most commonly components communicate via subscriptions to Messages by type. However, they can also communicate via Signals which are generally faster and do not use complicated routing mechanisms.

Signals can be imagined like a music conductor stick - it is a 'beat' which a module sends out to make other modules do things at (approximately) the same time. So any module subscribed to the signal A will be triggered very soon after the 'conductor' module sends out the signal A. No subscription checking is involved so signals are generally faster (and less flexible) than trigger messages. Signals are also independent from contexts.

This is often used in simulations where modules have to do things for the next simulation timestep - and then wait for the next step before continuing.

The following shows an example like the Ping/Pong messaging system above, but this time using signals:

```
<module name="Ping">
  <trigger name="Start" type="Psyclone.Ready" />
  <crank name="signal" function="Signal" />
  <signal name="Output" type="My.Signal.1" />
  <signal name="Input" type="My.Signal.2" />
</module>
<module name="Pong">
  <trigger name="Ready" type="Psyclone.Ready" />
  <crank name="signal" function="Signal" />
  <signal name="Input" type="My.Signal.1" />
  <signal name="Output" type="My.Signal.2" />
  <post name="Done" type="Psyclone.Shutdown" />
</module>
```

Dataflow parameters

The subscription entry for a component in the PsySpec can specify additional parameters to control the flow of data beyond the publish/subscribe mechanism.

After/delay

A module can ask for a trigger message to be delayed for a set amount of time before the trigger is delivered like normal:

```
<trigger name="Start" type="Psyclone.Ready" after="5000" />
```

This trigger will be delivered after a 5 seconds pause.

Interval

A component specification can ask for an artificial trigger message to be sent at a regular interval:

```
<module name="PostTester">
  <trigger name="Ready" type="Psyclone.Ready" />
  <trigger name="Post" type="Regular.Post" interval="1000" />
  <post name="Post" type="Post.Data.1" />
</module>
```

The Psychone system and the PsySpec XML file -- Filters

Please note that the module must be triggered by a real message first before the timed simulated trigger will start.

To

Usually posted messages are delivered based on other modules' subscriptions, but the output module can specify that a copy of the message be delivered directly to a named component regardless of whether that component has asked for it or not:

This is most commonly done for modules that want to post output messages to a specific Whiteboard:

```
<whiteboard name="WB1" />

<module name="PostTester">
  <post type="Data.Web" to="WB1" />
</module>
```

In this case, the output message will be delivered directly to the Whiteboard, but will also be delivered to any other component subscribing to it.

The same result could be achieved by posting without the to entry, but then in the Whiteboard subscribing to the message by type instead:

```
<whiteboard name="WB1">
  <trigger type="Data.Web" />
</whiteboard>

<module name="PostTester">
  <post type="Data.Web" />
</module>
```

Filters

Triggers can employ various filters to be more selective about the messages it is triggered with.

Maxage

In a time sensitive system a trigger can specify a maximum age to prevent being triggered by a message which was posted too long ago:

```
<trigger name="input" type="msg.1" maxage="20" />
```

This example tells the subscription engine to not trigger the module if the incoming message is more than 20ms old by filtering it out.

From

A trigger can filter out any message which does not come from a particular named component in the system:

```
<trigger name="input" type="msg.1" from="MyOtherModule" />
```

This example tells the subscription engine to not trigger the module if the incoming message was not posted by the component called MyOtherModule by filtering it out.

To

A trigger can filter out any message which was not addressed to a specific named component which used the to parameter in its post:

The Psyclone system and the PsySpec XML file -- Tagging

```
<trigger name="input" type="msg.1" to="SomeModule" />
```

This example tells the subscription engine to not trigger the module if the incoming message was not posted with a to specification of SomeModule by filtering it out.

Tag

A trigger can filter out any message which was not tagged with a specific named tag when being posted:

```
<trigger name="input" type="msg.1" tag="SomeTag" />
```

This example tells the subscription engine to not trigger the module if the incoming message was not tagged with SomeTag by filtering it out.

See the section on Tagging for more information about this topic.

More advanced filtering

More advanced trigger filters can be specified on a trigger to filter out based on other aspects of the incoming message:

```
<module name="Simple" node="Node0">
  <trigger name="t1" type="blat" maxage="500">
    <!-- additional filtering such as -->
    <filter type="equals" key="MyKey" value="Val1" />
    <filter type="haskey" key="SomeKey" />
  </trigger>
  ...
</module>
```

Filters can specify the following types:

Haskey – this type checks if the incoming message contains a user defined entry with that name.

Equals – this type checks if the string value of this entry is exactly equal to this value specified. This comparison is case sensitive and does not use wildcards.

Notequals – this type is the opposite of the Equals type.

Equalsnumeric – this type checks if the float value associated with the key is equals to the float value specified in the filter. If either value is not a floating point number Psyclone will attempt to convert the value (from a string or integer) to a floating point number first.

Notequalsnumeric – this type is the opposite of the Equalsnumeric type.

Greaterthan – this type checks if the float value associated with the key is greater than the float value specified in the filter. If either value is not a floating point number Psyclone will attempt to convert the value (from a string or integer) to a floating point number first.

Lessthan – this type checks if the float value associated with the key is less than the float value specified in the filter. If either value is not a floating point number Psyclone will attempt to convert the value (from a string or integer) to a floating point number first.

Tagging

Output messages can in addition to a type have an optional tag. This is a way of grouping messages relating to a specific topic or identity across all message types. So if a system has several modules

The Psyclone system and the PsySpec XML file -- System parameters
outputting a specific message type all messages produced as a consequence of this message at any point in the forward chain of components will also carry this tag.

```
<module name="TagPost">
  ...
  <post name="p1" type="bla4" tag="mytag" />
</module>
```

And at any point a filter can be put onto a trigger so only matching tags will be allowed through:

```
<module name="TagTrigger">
  <trigger name="t1" type="bla4" tag="mytag" />
  ...
</module>
```

Separately, a component can set its own tag on output messages. This tag is a 32-bit unsigned integer value and if set on a message every module receiving this message will automatically carry this tag over to the output messages posted as a consequence of this input message.

So if module A sets a tag = 10 on its output message and if modules B and C are both triggered by this message, any messages which B and C post will automatically carry tag = 10 too – and any posts by modules triggered by B and C's output messages will too, etc.

System parameters

Main system port

You can change the main network port of a Psyclone instance by specifying the new port on the command line (... port=6789 ...). You can, however, also program this into the PsySpec by adding the port number to the top PsyProbe XML tag:

```
<psySpec port="6789">
```

This changes the port which other Psyclone instances or systems use to communicate with this instance and it also sets the port number to use for accessing the PsyProbe web interface via a browser:

```
http://<hostname>:6789/
```

Verbose and debug output

The default verbose and debug output levels are both set to 1 for all topics. This will produce a minimal, but useful set of output logging to the console and to the log file. For less output which only contains errors use a level of 0 and for more output levels can go up to 9.

Increasing levels of output will output increasing amount of logging and this can be done per topic or for all topics. Furthermore, it can be done system-wide and for individual components. To set the output levels for an individual component add the verbose and/or the debug parameters in the PsySpec for that component:

```
<module name="Simple" verbose="3" debug="2" logfile="../log2.txt">
```


The Psyclone system and the PsySpec XML file -- System parameters
System-wide it can be set on the main PsyProbe XML tag:

```
<psySpec verbose="5" debug="0" logfile="../log.txt">
```

Levels can also be specified on the command line and can either be for all topics:

```
... verbose=5 debug=0 ...
```

or per topic:

```
... verbose-network=5 debug-memory=0 ...
```

The list of topics are:

<i>Topic name</i>	<i>Verbose cmdline entry</i>	<i>Debug cmdline entry</i>
<i>System</i>	verbose-system	debug-system
<i>Network</i>	verbose-network	debug-network
<i>Memory</i>	verbose-memory	debug-memory
<i>Process</i>	verbose-process	debug-process
<i>Sync</i>	verbose-sync	debug-sync
<i>Node</i>	verbose-node	debug-node
<i>Space</i>	verbose-space	debug-space
<i>Component</i>	verbose-component	debug-component
<i>Subscriptions</i>	verbose-subscriptions	debug-subscriptions
<i>Triggers</i>	verbose-triggers	debug-triggers
<i>Timing</i>	verbose-timing	debug-timing

PsyProbe HTML files location

Psyclone will attempt to auto detect the location of the HTML files required for running the PsyProbe web interface. Usually this would be in a directory called 'html' below the location of the Psyclone executable and Psyclone will look for this directory in the current and parent directories.

If the location is not automatically found the user can specify the location either on the command line:

```
... html=<dir>/html ...
```

or in the PsySpec by adding a PsyProbe entry:

```
<psyprobe location="some/dir/html" />
```



PsyProbe port

Normally the PsyProbe network port is the same as the main Psyclone port (with auto detection of the protocol). To change the PsyProbe port to a different port (and different from the main messaging port) this can be specified in the PsyProbe entry:

```
<psyprobe port="6789" />
```

The PsyProbe entry can contain other information such as aliases and posts – please see the PsyProbe section for more information.

For more advanced system configuration topics such as using Nodes and Spaces, please see the relevant sections for more information.



Contexts

Contexts are used to make a module behave differently in different situations. For example, a vision module would want to use a different algorithm if the scene is bright from if the scene is dark. Instead of managing this manually inside the code of the module, contexts allow a module to change from one type of operation to another by listening to an external signal.

Overview

The way contexts are typically used is that in each Context a module will have a separate set of inputs and outputs. By looking at currently active context and a module's spec, anyone, including the system designer and other runtime modules, can become aware of what the module is doing.

The system as a whole will usually have a hierarchy of contexts and many of these may be active at the same time.

Contexts can manage a collection of modules by simultaneously turning them on and off and to make them behave differently in different situations. Contexts are very helpful to manage the complexity of a large number of modules that typically are not all relevant for the operation of a system at the same time. Psyclone modules can be context-driven in that each module can be set up to have a specific context in which it can run. If that context is not active, the module will not be woken up with any messages it has subscribed to.

If no context is specified for a module their subscription default to belong to the context Psyclone.Ready which is always active as long as the system is running normally.

Contexts are hierarchically defined, forming a tree. For example, this is a context tree:

```
SoB.[Alive, Dead]
SoB.Alive.[Awake, Asleep]
```

In this context tree, the root is called SoB (short for state of being). The root has two branches, Alive and Dead. The branch Alive has two leafs, Awake and Asleep. (Dead has no branches.) To see how this tree is used at run-time, let's look at a hypothetical system that uses this context tree. When the system starts up the context SoB is posted by some bootstrapping module. When this happens another module (called e.g. BeingAlive.100) posts SoB.Alive if the system is considered "alive", otherwise it posts SoB.Dead (the module would presumably run some tests to see if the system actually qualifies for the "alive" label). If SoB.Alive is posted, then another module (called e.g. BeingAwake.200) posts SoB.Alive.Awake if it considers the system to be "awake" (whatever that means in this hypothetical system).

Now we can look at how modules use these contexts to specify when they should be active and when not. For example, we might have a module called LookAround.150, whose job it is to move the sensors of a hypothetical being in the direction of a sound when it receives one. This module should not be running if the being is dead or if the being is asleep. So when we choose a context for this module, we choose the context in which it should be active, which is in this case SoB.Alive.Awake.

In the psySpec you can then have a module which is only active, i.e. should only run and receive trigger messages when the system is Awake:

Contexts -- Overview

```
<module name="Looking">
  <context name="SoB.Alive.Awake">
    <trigger name="t1" type="bla1" />
    <crank name="c1" function="lib::function" />
    <post name="p1" type="bla4" />
  </context>
</module>
```

and another module which is active when the system is Alive:

```
<module name="Breathing">
  <context name="SoB.Alive">
    <trigger name="t1" type="bla1" />
    <crank name="c1" function="lib::function" />
    <post name="p1" type="bla4" />
  </context>
</module>
```

When a module goes out of context it should stop running. This will be signalled to the code in that the function call to `api->shouldContinue()` returns false.

One module can also do different things in different contexts:

```
<module name="Metabolism">
  <context name="SoB.Alive.Awake">
    <trigger name="t1" type="bla1" />
    <crank name="c1" function="lib::function1" />
    <post name="p1" type="bla4" />
  </context>
  <context name="SoB.Alive.Asleep">
    <trigger name="t2" type="bla1" />
    <crank name="c2" function="lib::function2" />
    <post name="p2" type="bla4" />
  </context>
</module>
```

which practically means that a different crank is called for the other context, but both cranks have access to the module's private data.

Multiple roots are allowed. If you post for example the context `SoB.Alive.Asleep` and then post the context `System.Ok`, the two context trees have different roots, and will thus co-exist with no interference.

Notice that context trees are designed by the user. The only restrictions on them is that they be dot-delimited to designate branches.

Contexts are changed (i.e. announced to the system) by using a special post flag:

```
<post name="done" context="SoB.Alive.Asleep" />
```

which immediately makes causes any module and subscriptions to the other contexts with the same higher root to become inactive, such as

Contexts -- Overview

- SoB.Dead
- SoB.Alive.Awake
- SoB.Alive.Asleep.Snoring

but the higher up contexts will still be active, such as

- SoB
- SoB.Alive

A module can also subscribe to be triggered by a context change:

```
<trigger name="t1" context="SoB.Alive.Asleep" />
```

When a context is posted, the context gets "switched" only if a current branch in the tree gets replaced by a new branch. For example, if the current context is SoB.Alive.Asleep and SoB.Alive.Awake gets posted, the context will be switched. And if SoB.Alive.Asleep.REMsleep gets posted, this context will be switched on for any module which had listed this as their context. But if SoB.Alive gets posted while SoB.Alive.Asleep is active, nothing happens.

Internal modules will automatically be using the correct crank in the correct phase in the currently active context. If they go out of context they will stop running and wait to be in context again.

External modules will have to detect whether they are in context at all, and if so, which one. The PsyAPI object has several functions such as:

```
PsyContext getCurrentTriggerContext();
std::string contextToText(PsyContext context);
```

If a component attempts to post a message while its crank is no longer in context the function

```
int32 postOutputMessage(const char* postName = NULL, DataMessage* msg = NULL);
```

returns -3 (POST_OUTOFCONTEXT).



Python modules

In addition to modules written in C++ Psyclone also supports running modules written in the language of Python, both version 2.x and 3.x. This is done by allowing the crank function to be written in and loaded from Python.

Crank functions for C++ modules use the PsyAPI object from the CMSDK library to communicate with Psyclone and this in turn makes use of a large proportion of the other objects and functions in the CMSDK library. Full support for the CMSDK library has been added to Python by adding a SWIG interface – which means that any Python code can use the PsyAPI and other objects and functions just like a C++ module would.

Where a C++ module's crank function is specified in the PsySpec like this:

```
<module name="Ping">
  <trigger name="Ready" type="Psyclone.Ready" />
  <trigger name="Ball" type="ball.1" />
  <crank name="Ping" function="Ping" />
  <post name="Ball" type="ball.2" />
</module>
```

a Python module is almost the same except for the crank entry now referring to a Python code file:

```
<module name="Ping">
  <trigger name="Ready" type="Psyclone.Ready" />
  <trigger name="Ball" type="ball.1" />
  <crank name="Ping" language="python2" script="path/to/ping.py" />
  <post name="Ball" type="ball.2" />
</module>
```

The language can be either python2 or python3.

Inline Python modules

For simple Python modules it is even possible to write the crank code directly in the PsySpec file:

```
<crank name="p1" language="python2">
  <![CDATA[
    name = api.getModuleName()
    print("Module name: %s" % name)
    key = api.getParameterInt("Key")
    while (api.shouldContinue()):
      msg = api.waitForNewMessage(20)
      if msg != None:
        mykey = msg.getInt("MyKey")
        if mykey == 0:
          mykey = key
        triggerName = api.getCurrentTriggerName()
        api.logPrint(1, "Got Key: " + str(mykey))
        outMsg = cmsdk.DataMessage()
        outMsg.setInt("MyKey", mykey + 1)
        api.postOutputMessage("Output", outMsg)
  ]]>
</crank>
```

The inline Python code will automatically load all the libraries needed and create the PsyAPI object called api.

Python modules -- Python code file modules

Please note: Because Python is very sensitive about indentation, please take care that the inline Python code keeps these intact. Psyclone will when loading the module attempt to adjust the indentation.

Python code file modules

The equivalent simple Python module written in external files looks like this:

```
def PsyCrank(apilink):
    api = cmsdk.PsyAPI.fromPython(apilink);
    name = api.getModuleName();
    print("Module name: %s" % name)
    key = api.getParameterInt("Key");
    while (api.shouldContinue()):
        msg = api.waitForNewMessage(20)
        if msg != None:
            mykey = msg.getInt("MyKey")
            if mykey == 0:
                mykey = key
            triggerName = api.getCurrentTriggerName()
            api.logPrint(1, "Got Key: " + str(mykey))
            outMsg = cmsdk.DataMessage()
            outMsg.setInt("MyKey", mykey + 1)
            api.postOutputMessage("Output", outMsg)
```

The first two lines are needed in this case (they are added automatically for inline modules) and there is no need to Import the CMSDK library as Psyclone will do this automatically.

Python import libraries and paths

When Psyclone runs any Python, both inline and in a code file, module it needs to have access to

- 1) The correct version of Python installed correctly
 - a. Correct major version, either 2.x or 3.x
 - b. Correct bitness, either 32 or 64-bit
 - c. The python DLLs in the search path
- 2) The CMSDK binary library and its Python interface file
- 3) Any libraries which the Python module may import in addition

The CMSDK binary library is a file called _cmsdk.pyd on Windows and _cmsdk.so on Linux. This file is accompanied by the Python interface file called cmsdk2.py for Python2 and cmsdk3.py for Python3. In addition, for Debug builds of Psyclone debug versions of these file will be used (_cmsdkdebug.pyd/so and cmsdk2/3debug.py).

These files will either need to be

- a) in the same directory as the Python program as specified in the crank:


```
<crank name="Ping" language="python2" script="path/to/ping.py" />
```
- b) in the same directory as the Psyclone binary used to run the system
- c) and if not, an additional parameter can be specified on the module to tell Psyclone where to find these:

```
<parameter name="libpath" type="String" value="./CMSDK/bin/Win32" />
<crank name="Ping" language="python2" script="path/to/ping.py" />
```

Components: Modules, Whiteboards and Catalogs

The PsySpec can specify three types of components:

Modules are the standard component which predominantly is triggered by new messages which are processed, producing output messages for posting and optionally can remain in the crank function to do other work in between.

Catalogs can do exactly the same, but are designed to in addition receive synchronous queries from other components to which they will respond with an answer. The most common types of Catalogs are for data storage and queries (like a database or datawarehouse), but they could also be conduits to other systems such as forwarding queries to a Google Search or Face Recognition server and providing synchronous replies to the component which requested the data.

Whiteboards are a special type of Catalog which store and work with Messages only. They receive messages by trigger (or direct messaging), store these in memory and other component can then use a sophisticated query language to retrieve copies of stored message based on a set of filters. In addition, the PsyProbe interface for Whiteboards provides the user with a nice visual way of viewing messages with filtering capabilities.

Modules

Modules are specified in the PsySpec like seen above:

```
<module name="Ping">
  <trigger name="Ready" type="Psyclone.Ready" />
  <trigger name="Ball" type="ball.1" />
  <crank name="Ping" function="Ping" />
  <post name="Ball" type="ball.2" />
</module>
```

Specifically, the function code they call are specified as part of the subscription and as seen later a module can specify multiple crank functions and subscriptions for different contexts.

Whiteboards

Whiteboards have their own syntax in the PsySpec and can be as simple as:

```
<whiteboard name="WB1" />
```

This defines the Whiteboard component without any triggers so it will only ever receive messages addressed specifically to it, such as:

```
<module name="Ping">
  ...
  <post name="Ball" to="WB1" type="ball.2" />
</module>
```

This way of directly messaging another component happens in addition to the subscription-based routing of the message so in addition to delivering the message to the Whiteboard WB1 Psyclone will still look for components who have subscribed to this message type.

Most commonly, however, Whiteboards will have a long list of triggers to hover up messages without the posting components needing to be aware of it:



```
<whiteboard name="WB2">
  <trigger name="Ball" type="ball.1" />
</whiteboard>
```

When the Whiteboard stores messages in its internal database it automatically indexes based on message creation time, so other components can retrieve message copies based on time. The retrieve specification is usually specified mostly in the PsySpec like this:

```
<module name="Test">
  <trigger name="Ready" type="Psychone.Ready" />
  <retrieve name="r1" source="WB1" maxcount="10" />
```

and Test module crank can then execute this retrieve by name:

```
std::list<DataMessage*> retrievedMsgs;
uint8 status = api->retrieve(retrievedMsgs, "r1");
```

The std::list will contain a copy of the retrieved messages and the status reply will indicate the success or failure of the operation:

```
#define QUERY_FAILED          1
#define QUERY_TIMEOUT         2
#define QUERY_NAME_UNKNOWN    3
#define QUERY_COMPONENT_UNKNOWN 4
#define QUERY_QUERYFAILED     5
#define QUERY_SUCCESS         6
#define QUERY_NOT_AVAILABLE   7
#define QUERY_NOT_REACHABLE   8
```

This retrieve operation retrieves the last messages received regardless of type, up to a maximum of 10 messages.

Retrieves can be done based on keys other than time. If the Messages contain a user entry called Count of type Integer the Whiteboard can specify this as a searchable key:

```
<whiteboard name="WB2" key="count" keytype="integer">
  <trigger name="Ball" type="ball.1" />
</whiteboard>
```

which creates a separate searchable index (in addition to time).

The module can now specify a retrieve based on this key:

```
<retrieve name="r2" source="WB1" key="count" keytype="integer" />
```

and execute it in its code, providing a from (2) and to (8) value, returning maximum of 4 messages:

```
status = api->retrieveIntegerParam(retrievedMsgs, "r2", 2, 8, 4);
```

This could additionally also contain a maximum message age in microseconds:

```
status = api->retrieveIntegerParam(retrievedMsgs, "r2", 2, 8, 4, 100000);
```



Catalogs

Catalogs are more flexible in terms of how they can be queried and what type of data they can return. Catalogs are specified in the PsySpec like this:

```
<catalog name="MessageDataCatalog" type="MessageDataCatalog">
```

where the type in this case refers to the Catalog's crank function.

Psychone has a number of built-in Catalogs with useful features. These are described next.

The File Catalog

The File Catalog allows any module to read and write files from/to a central location without having to worry about which computer the module is running on.

A File Catalog is defined in the PsySpec like this:

```
<catalog name="MyFiles" type="FileCatalog" root=".">
  <parameter name="ReadOnly" type="String" value="no" />
</catalog>
```

The optional parameter ReadOnly defaults to 'no', but can be set to 'yes' which will prevent write operations.

Any crank code can access the catalog by defining a named query in the PsySpec like this:

```
<module name="Test">
  ...
  <query name="MyFiles" source="MyFiles" subdir="test" ext="txt" binary="yes" />
  ...
</module>
```

And the crank code can execute the query this way:

```
uint32 datasize = 0;
char* result = NULL;
uint8 status = api->queryCatalog(&result, datasize, "MyFiles", "test", "read");
if (status == QUERY_SUCCESS)
  api->logPrint(1, "Successfully retrieved %u bytes from file catalog...", datasize);
else if (status == QUERY_TIMEOUT)
  api->logPrint(1, "Retrieve 'read' timed out for file catalog...");
else
  api->logPrint(1, "Retrieve 'read' failed (%u) for file catalog...", status);
... use binary data in result ...
delete[] result;
```

which will read a file from the root dir ./, sub dir test called test.txt, i.e. ./test/test.txt.



Likewise a file can be written by

```
char* data = utils::StringFormat(datasize, "Hello World");
status = api->queryCatalog(&result, datasize, "MyFiles", "test", "write", data, datasize);
if (status == QUERY_SUCCESS)
    api->logPrint(1, "Successfully wrote %u bytes to file catalog...", datasize);
else if (status == QUERY_TIMEOUT)
    api->logPrint(1, "Retrieve 'write' timed out for file catalog...");
else
    api->logPrint(1, "Retrieve 'write' failed (%u) for file catalog...", status);
delete[] data;
```

The Data Catalog

The Data Catalog allows any module to read and write data from/to a central datastore without having to worry about which computer the module is running on.

A Data Catalog is defined in the PsySpec like this:

```
<catalog name="MyData" type="DataCatalog" interval="2000" root="./mydata.dat" />
```

which puts the central datastore in the file ./mydata.dat and writes from memory to file every 2 seconds.

Any crank code can access the catalog by defining a named query in the PsySpec like this:

```
<module name="Test">
    ...
    <query name="MyData" source="MyData" />
    ...
</module>
```

And the crank code can execute the query this way:

```
uint32 datasize = 0;
char* result = NULL;
status = api->queryCatalog(&result, datasize, "MyData", "test", "read");
if (status == QUERY_SUCCESS)
    api->logPrint(1, "Successfully read %u bytes from data catalog...", datasize);
else if (status == QUERY_TIMEOUT)
    api->logPrint(1, "Retrieve 'read' timed out from data catalog...");
else
    api->logPrint(1, "Retrieve 'read' failed (%u) from data catalog, ok if starting with a
fresh catalog ...", status);
delete[] result;
```

which will read an entry from the datastore.



Likewise data can be written by

```
char* data = utils::StringFormat(datasize, "Hello2 World");
status = api->queryCatalog(&result, datasize, "MyData", "test", "write", data, datasize);
if (status == QUERY_SUCCESS)
    api->logPrint(1, "Successfully wrote %u bytes to data catalog...", datasize);
else if (status == QUERY_TIMEOUT)
    api->logPrint(1, "Retrieve 'write' timed out to data catalog...");
else
    api->logPrint(1, "Retrieve 'write' failed to data catalog (%u)...", status);
delete[] data;
delete[] result;
```

The Replay Catalog

The Replay Catalog can be used in one Psyclone system to simply record certain message activity to disk - and later in a different Psyclone system it can be used to playback these messages as they happened in the first system.

One example could be to record all perception from a robot over a 2 minute period, such as video, audio, sensors, etc. and potentially even the output of some of the processing of these. On a different computer these recorded messages can be used in a simulated system to get exactly the same messages without actually having to have the robot present and running.

The recording phase can be done by simply defining the catalog in the PsySpec and telling it what and how much to record:

```
<catalog name="Replay1" type="ReplayCatalog" root="./replay1" maxsize="10240000" maxcount="2000">
  <trigger name="Video" type="robot.sensor.video" />
  <trigger name="Bumper" type="robot.sensor.bumper" />
</catalog>
```

This will receive all messages of types 'robot.sensor.video' and 'robot.sensor.bumper' and store them to disk in ./replay1, only keeping the last 2000 messages and only keeping up to 10kb of data, whichever limit is hit first.

Then the files can be copied to another computer and to replay the messages you simply define the same catalog with different parameters:

```
<catalog name="Replay1" type="ReplayCatalog" root="./replay1">
  <post name="Video" type="robot.video" />
  <post name="Bumper" type="robot.bumper" />
</catalog>
```

This will play back the messages at the same interval they were recorded, but this time with different message types, just the trigger/post names have to match.



Two additional parameters can be used on playback

```
<catalog name="Replay1" type="ReplayCatalog" root="./replay1" interval="1000" rotate="yes">
  <post name="Video" type="robot.video" />
  <post name="Bumper" type="robot.bumper" />
</catalog>
```

```
interval="1000"    overwrites the posting interval to 1 second
                   regardless of the recorded message timing
rotate="yes"       when the last message has been replayed go back
                   and start playing from the beginning
```

The Request Store Catalog

The RequestStore Catalog can be used to collect messages from the whole system, keep the most recent versions of them and allow anyone to request parts of or the whole message from a web browser by easy to remember names. This means no custom programming or data in the modules at all.

An example of a RequestStore Catalog entry in the PsySpec could be:

```
<catalog name="RequestStore" type="RequestStore">
  <trigger name="VideoFrame" type="video.output.frame" />
  <trigger name="SampleXML" type="video.output.frame" />
  <trigger name="SampleText" type="video.output.frame" />
  <trigger name="SampleHTML" type="video.output.frame" />-->
  <setup>
    <store name="VideoFrame" trigger="VideoFrame" key="VideoDataFrame" keytype="data"
           datatype="raw" mimetype="image/bmp" maxkeep="1" maxfrequency="1" />
    <store name="SampleJSON" trigger="VideoFrame" mimetype="json" maxkeep="1"
           maxfrequency="1" />
    <store name="SampleXML" trigger="VideoFrame" mimetype="xml" maxkeep="1"
           maxfrequency="1" />
    <store name="SampleText" trigger="VideoFrame" mimetype="text" key="content"
           maxkeep="1" maxfrequency="1" />
    <store name="SampleNum" trigger="VideoFrame" mimetype="text" key="VideoWidth"
           maxkeep="1" maxfrequency="1" />
    <store name="SampleSize" trigger="VideoFrame" mimetype="text" key="size"
           maxkeep="1" maxfrequency="1" />
    <store name="SampleType" trigger="VideoFrame" mimetype="text" key="type"
           maxkeep="1" maxfrequency="1" />
    <store name="SampleTime" trigger="VideoFrame" mimetype="text" key="time"
           maxkeep="1" maxfrequency="1" />
    <store name="SampleTimeText" trigger="VideoFrame" mimetype="text" key="timetext"
           maxkeep="1" maxfrequency="1" />
    <!--<store name="SampleHTML" mimetype="html" key="html" maxkeep="1"
           maxfrequency="1" />-->
  </setup>
</catalog>
```

Here we see the catalog subscribing to messages like any other module, each with a trigger name as usual.

The custom setup XML part then defines named entries ('VideoFrame', 'SampleJSON', etc.) which can be queried by anyone like this:

```
GET /api/query?from=RequestStore&query=VideoFrame
```

The return content-type is specifically specified in the setup XML so the one calling the URL doesn't have to know.



Each store entry can have the following parameters:

name	The name used in the request	Required
trigger	Which trigger message to get the information from	Required
key	The name of the user entry inside the message - can be header fields such as 'time', 'type' too. 'time' will return the number of microseconds since year 0 - 'timetext' will return a nicer textual description of the date and time	If not specified the whole message is returned as either JSON, XML, HTML or TEXT, depending on the mimetype
keytype	The type of key	Not currently used
datatype	The type of binary data, mainly used for bitmaps	If set to 'raw' the code knows to expect raw pixels and need the message to contain width and height information too - so construct a valid bitmap format to return. In the future we may support other types of data types.
mimetype	The mime type used to help the browser display the data correctly	Required, mime types are standard browser mime types such as 'application/json', 'text/html', 'text/xml', etc. Can also be short versions, 'json', 'xml', 'text', etc.
maxkeep	How many messages (trigger history) to keep	Default is 1, later may support returning vectors of keys from several historical messages
maxfrequency	The maximum number of messages to keep per second	If set to 1 it will only keep 1 message every second and ignore new messages for that trigger until another 1 second has passed

Creating custom Catalogs

In addition to the built-in Catalogs users can create custom Catalogs which can be queried from any other component using either a string-based or message-based interface.

This section first describes how another component can query a Catalog and after that how a custom Catalog can be created to handle such queries.

The string-based query interface is:

```
uint8 queryCatalog(char** result, uint32 &resultsize, const char* name,
    const char* query, const char* operation = NULL, const char* data = NULL,
    uint32 datasize = 0, uint32 timeout = 5000);
```

Components: Modules, Whiteboards and Catalogs -- Catalogs

and can be called like this:

```
status = api->queryCatalog(&result, datasize, "MyFiles", "test", "read");
```

The message-based interface allows more flexibility in that a custom DataMessage can be used to hold the query (including any number of user entries and data types such as images, etc.) and the reply is returned as an equally flexible DataMessage:

```
uint8 queryCatalog(DataMessage** resultMsg, const char* name,
    DataMessage* msg, uint32 timeout = 5000);
```

This can be used like this:

```
status = api->queryCatalog(&resultMsg, "MyQueryName", queryMsg, 5000);
```

Status can be one of

```
#define QUERY_FAILED            1
#define QUERY_TIMEOUT          2
#define QUERY_NAME_UNKNOWN     3
#define QUERY_COMPONENT_UNKNOWN 4
#define QUERY_QUERYFAILED      5
#define QUERY_SUCCESS          6
#define QUERY_NOT_AVAILABLE    7
#define QUERY_NOT_REACHABLE    8
```

Custom Catalogs can be created by writing a custom Catalog crank function. This crank function works exactly like the module crank function and a Catalog can receive and process subscription trigger messages and post output messages. In addition it can also receive synchronous query messages to which it should reply.

```
while (api->shouldContinue()) {
    if (inMsg = api->waitForNewMessage(100, triggerName)) {
        if (triggerName && strcmp(triggerName, "SomeTrigger") == 0) {
            ... process normal trigger message ...
        }
        else if (inMsg->getType() == PsyAPI::CTRL_QUERY) {
            ... process query message ...
            ... and then reply to the query ...
        }
    }
}
```

To reply to a query not requiring return data the query reference number is required and a status flag providing information about whether the query succeeded or not.

```
api->queryReply((uint32)inMsg->getReference(), status);
```

Components: Modules, Whiteboards and Catalogs -- Catalogs

The incoming query message for string based queries will contain the textual parameters inside and the Catalog crank can read them like this:

```
if (str = inMsg->getString("Subdir"))
    subdir = str;
if (str = inMsg->getString("Ext"))
    reqext = str;
if (str = inMsg->getString("Binary"))
    reqbinary = (strcmp(str, "yes") == 0);
if ((str = inMsg->getString("Operation"))
    reqwrite = (strcmp(str, "write") == 0);
```

and the reply needs to be done like this:

```
bool queryReply(uint32 id, uint8 status, char* data, uint32 size, uint32 count);
```

such as

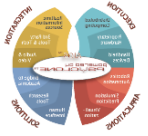
```
api->queryReply((uint32)inMsg->getReference(), QUERY_SUCCESS, data, size, 0);
```

For message-based queries the incoming query message can contain any number of custom data entries such as text, numbers, raw data and attached messages. Replies to these should be a Data Message too which similarly can contain any number of data entries:

```
bool queryReply(uint32 id, uint8 status, DataMessage* msg = NULL);
```

such as

```
api->queryReply((uint32)inMsg->getReference(), status, replyMsg);
```

Data Messages

Types

Message types should follow the convention to use a dot-delimited string where the first part indicates the root namespace, and the subsequent segments describe the type of content that the message relates to, of increasing specificity from left to right.

Messages usually contain user data entries that contain information for the recipient of the messages and can contain many entries of small or very large amounts of data (from integers to binary data blobs)

In the PsySpec:

```
<post name="exampleMsg" type="Robot.Status" />
```

In the code of the crank:

```
msg = new DataMessage();
msg->setString("RobotStatus", "Almost ready");
msg->setInt("BatteryLevel", 78);
api->postOutputMessage("exampleMsg", msg);
```

User contents

Data Entries can be of types String, Int, Float, Time, Binary and Message.

You can also set entries of type arrays (integer indexes) and maps (string indexes):

```
msg->setInt(1, "myarray", 2);
msg->setInt(2, "myarray", 2);
msg->setInt(3, "myarray", 2);
msg->setInt(4, "myarray", 2);
msg->setInt(5, "myarray", 2);

int val = msg->getInt(3, "myarray");
```

Or the whole array:

```
std::map<int64, int64> arr = msg->getIntArray("myarray")

msg->setIntArray("myarray2", arr)
```

This works for all types (int, float, string, time, data, messages) and the same for maps

```
msg->setInt("in", "mymap", 1);
msg->setInt("out", "mymap", 2);
msg->setInt("middle", "mymap", 3);
msg->setInt("off", "mymap", 4);
int val = msg->getInt("out", "mymap");
```

Or the whole array:

```
std::map<std::string, int64> arr = msg->getIntMap("mymap")
msg->setIntMap("myarray2", arr)
```

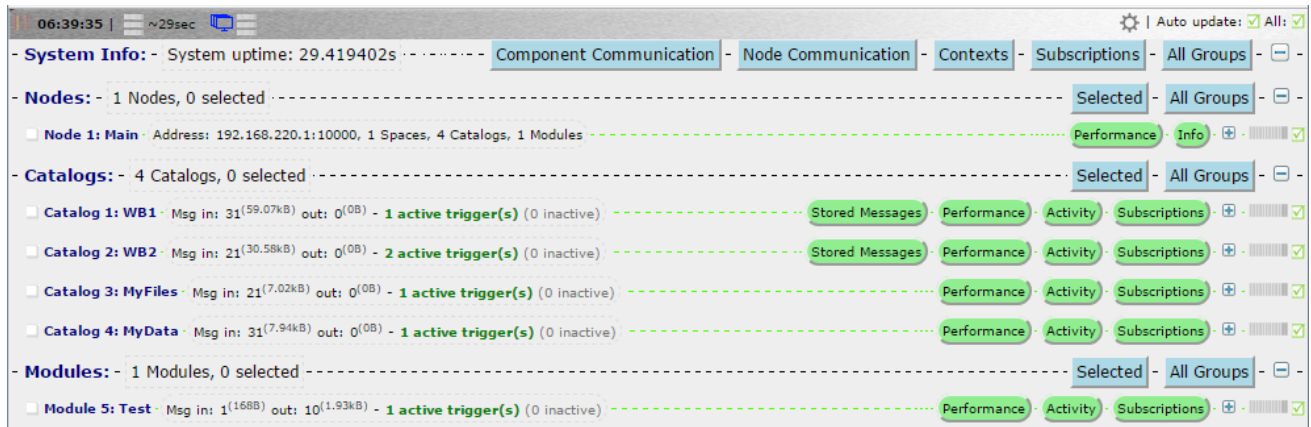


PsyProbe Web Interface

PsyProbe is the main visual interface into what is happening inside a Psyclone system. Point any standard web browser at the main Psyclone port like this

<http://localhost:10000/>

to get a dynamic live-updating view of all nodes, spaces and components in the system:



System overview

The very top grey line shows the time and the current system uptime and a horizontal list of the nodes with their activity levels to the left.

Each section below is hierarchical and part can be expanded and closed using the tabs and the (+) and (-) buttons.

The System Info section has tabs for component and node communication, list of currently active and inactive contexts and subscriptions of all components in the system.

The Nodes section lists the currently connected nodes (computers) with tabs for performance stats and general information. The node activity level is indicated to the far right on each node line.

The Catalogs section shows a list of the catalogs in the system. Catalogs can be Whiteboards that has tabs for stored messages, performance, recent activity and subscriptions. Other catalogs (either built-in or custom catalogs) have tabs for performance, activity and subscriptions and these may have custom tabs too.

Finally, the Modules section shows a list of modules in the system. Each has tabs for performance, activity and subscriptions and these may have custom tabs too.



Tab descriptions

System Component Communication

This tab shows a matrix of the amount and speed of communication for between all the components of the system.

System Node Communication

This tab shows a matrix of the amount and speed of communication for between all the nodes of the system.

System Contexts

This tab shows a list of all the active and inactive contexts currently known to the system. Each context can be expanded to show the components subscription triggers and associated posts for that context.

System Subscriptions

This tab shows a hierarchical list of all the message types known to the system and for each the components subscription triggers and associated posts. Each trigger shown can be manually activated with the [test] button to the right of the line, as can each post using the post line's [test] button.

Node Performance


This tab shows detailed performance stats for data input, output, queue sizes and CPU and memory usage for the last 1 second, 10 seconds and 30 seconds.

Node Info

This tab shows the list of spaces in each node with status and stats.

Whiteboard Stored Messages

This tab shows the messages currently stored on the Whiteboard. A line of filter options is available

Filter (0 of 0) | MaxCount msgs | MaxSize bytes (1.00MB) | MaxAge ms (~10sec) 

to specify which and how many messages to see, filtering on any textual content, count, size and age.

Component Performance

This tab shows detailed performance stats for data input, output, queue sizes and CPU and memory usage for the last 1 second, 10 seconds and 30 seconds.

Component Activity

This tab shows the last 10 messages sent and received by the component. If you hover over the message entry you will see as much of the message as is available to show and if clicked a new window is opened to keep this content visible.

Component Subscriptions

This tab shows either the active or all subscriptions for the component, shown hierarchically based on message type. Each trigger shown can be manually activated with the [test] button to the right of the line, as can each post using the post line's [test] button.

Custom tabs

Each component can create and register their own custom tabs. An example of this is the Stored Messages tab for Whiteboards and details on this can be found in the section on customising PsyProbe.



PsyProbe in the PsySpec

The user can override PsyProbe settings in the PsySpec, including the location of the built-in PsyProbe HTML dir, adding additional ports for PsyProbe to listen to and removing PsyProbe from listening to the main Psyclone port:

```
<psyprobe location="otherdir/html" port="8080" defaultport="no" />
```

- location overwrites htmldir, if present
- port adds a port (in addition to the main and other ports), if present
- defaultport="no" removes PsyProbe from the main port, if present

Custom subsites

Custom location aliases can be added to PsyProbe for application specific use:

```
<psyprobe>
  <alias name="robot" location="/somedir/html" default="index.html" />
</psyprobe>
```

- alias subnodes adds custom dirs to PsyProbe, if present

which means that when a browser requests the url

```
http://computer:port/robot/anydir/anyfile
```

PsyProbe will return the file found in /somedir/html/anydir/anyfile.

This can be any path on the local computer.

Using PsyProbe

The main purpose of PsyProbe is to be able to track data as it flows through the system. The data traditionally travels inside DataMessages which all have a header (much like an email) with standard fields for time, from, to, etc. and optionally can contain user data in the form of zero or more <text key> = <value>. The values can be either

- Text
- Integer
- Float
- Time
- Raw binary data
- Another DataMessage

PsyProbe Web Interface -- Messaging activity

Messaging activity

A component in Psychone (either a Module or a Catalog) can post output messages based on that component's post entries in the PsySpec. The code will post using a name (i.e. a label) based on which the Psychone system will set the message type. The type is then used to figure out which other components have registered (i.e. subscribed) to receive messages like this in the component's trigger entries in the PsySpec.

Using PsyProbe a human operator can keep an eye on the messages flowing into and out of a component. This is done using the component's Activity tab:

08:53:06 | ~33sec

System Info: System uptime: 32.522605s Log Component Communication Node Communication Contexts Subscriptions Dataflow Components

Nodes: 1 Nodes, 0 selected

Catalogs: 1 Catalogs, 0 selected

Modules: 1 Modules, 0 selected

Module 2: PostTester: Msg in: 33(6.45kB) out: 33(6.45kB) - 2 active trigger(s) (0 inactive)

Recent input messages:

Type	From	Size	Age
Regular.Post	System	200B	Sent 262.745ms ago
Regular.Post	System	200B	Sent ~1sec ago
Regular.Post	System	200B	Sent ~2sec ago
Regular.Post	System	200B	Sent ~3sec ago
Regular.Post	System	200B	Sent ~4sec ago
Regular.Post	System	200B	Sent ~5sec ago
Regular.Post	System	200B	Sent ~6sec ago
Regular.Post	System	200B	Sent ~7sec ago
Regular.Post	System	200B	Sent ~8sec ago
Regular.Post	System	200B	Sent ~9sec ago

Recent output messages:

Type	From	Size	Age
Post.Data.1	PostTester	200B	Sent 262.745ms ago
Post.Data.1	PostTester	200B	Sent ~1sec ago
Post.Data.1	PostTester	200B	Sent ~2sec ago
Post.Data.1	PostTester	200B	Sent ~3sec ago
Post.Data.1	PostTester	200B	Sent ~4sec ago
Post.Data.1	PostTester	200B	Sent ~5sec ago
Post.Data.1	PostTester	200B	Sent ~6sec ago
Post.Data.1	PostTester	200B	Sent ~7sec ago
Post.Data.1	PostTester	200B	Sent ~8sec ago
Post.Data.1	PostTester	200B	Sent ~9sec ago

Open in new window

In this example we see a system with two components; one Catalog (called CCMMaster) and one Module (called PostTester). When we click the Activity tab for the module we see the 10 most recent input messages and the 10 most recent output messages. This list will continuously update itself unless you untick the Auto update box for either the module itself (green tick to the far right) or for all (the green tick marked All in the top right corner).

If you use the mouse to hover over a message you will see a detailed view of that message including all its user content entries. If you click the message that detailed view will open in a new window and remain static, i.e. will not update again.

The list above will show you the source of any incoming messages, i.e. which other module or catalog created it. In this case the Regular.Post message was generated automatically on a timer by the system, but if we look at another component we will see the messages from the module flowing into the catalog:



09:00:27 | ~7min | [Log](#) | [Component Communication](#) | [Node Communication](#) | [Contexts](#) | [Subscriptions](#) | [Dataflow](#) | [Component](#)

Nodes: - 1 Nodes, 0 selected - Selected - All Groups - [Performance](#) - [Info](#) - [Cognitive Map](#) - [Log](#) - [Data](#) - [Performance](#) - [Activity](#) - [Subscriptions](#) - [Open in new window](#)

Catalogs: - 1 Catalogs, 0 selected - Selected - All Groups - [Performance](#) - [Info](#) - [Cognitive Map](#) - [Log](#) - [Data](#) - [Performance](#) - [Activity](#) - [Subscriptions](#) - [Open in new window](#)

Node 1: Main - Address: 10.239.67.21:10000, 1 Spaces, 1 Catalogs, 1 Modules - [Performance](#) - [Info](#) - [Cognitive Map](#) - [Log](#) - [Data](#) - [Performance](#) - [Activity](#) - [Subscriptions](#) - [Open in new window](#)

Catalog 1: CCMMaster - Msg in: 475(92.77kB) out: 0(0B) - 2 active trigger(s) (0 inactive) - [Cognitive Map](#) - [Log](#) - [Data](#) - [Performance](#) - [Activity](#) - [Subscriptions](#) - [Open in new window](#)

Recent input messages:				Recent output messages:			
Type	From	Size	Age	Type	From	Size	Age
Post.Data.1	PostTester	200B	Sent 260.489ms ago	Post.Data.1	PostTester	200B	Sent 260.489ms ago
Post.Data.1	PostTester	200B	Sent ~1sec ago	Post.Data.1	PostTester	200B	Sent ~1sec ago
Post.Data.1	PostTester	200B	Sent ~2sec ago	Post.Data.1	PostTester	200B	Sent ~2sec ago
Post.Data.1	PostTester	200B	Sent ~3sec ago	Post.Data.1	PostTester	200B	Sent ~3sec ago
Post.Data.1	PostTester	200B	Sent ~4sec ago	Post.Data.1	PostTester	200B	Sent ~4sec ago
Post.Data.1	PostTester	200B	Sent ~5sec ago	Post.Data.1	PostTester	200B	Sent ~5sec ago
Post.Data.1	PostTester	200B	Sent ~6sec ago	Post.Data.1	PostTester	200B	Sent ~6sec ago
Post.Data.1	PostTester	200B	Sent ~7sec ago	Post.Data.1	PostTester	200B	Sent ~7sec ago
Post.Data.1	PostTester	200B	Sent ~8sec ago	Post.Data.1	PostTester	200B	Sent ~8sec ago
Post.Data.1	PostTester	200B	Sent ~9sec ago	Post.Data.1	PostTester	200B	Sent ~9sec ago

Modules: - 1 Modules, 0 selected - Selected - All Groups - [Log](#) - [Data](#) - [Performance](#) - [Activity](#) - [Subscriptions](#) - [Open in new window](#)

Module 2: PostTester - Msg in: 474(92.58kB) out: 474(92.58kB) - 2 active trigger(s) (0 inactive) - [Log](#) - [Data](#) - [Performance](#) - [Activity](#) - [Subscriptions](#) - [Open in new window](#)

Recent input messages:				Recent output messages:			
Type	From	Size	Age	Type	From	Size	Age
Regular.Post	System	200B	Sent 260.489ms ago	Post.Data.1	PostTester	200B	Sent 260.489ms ago
Regular.Post	System	200B	Sent ~1sec ago	Post.Data.1	PostTester	200B	Sent ~1sec ago
Regular.Post	System	200B	Sent ~2sec ago	Post.Data.1	PostTester	200B	Sent ~2sec ago
Regular.Post	System	200B	Sent ~3sec ago	Post.Data.1	PostTester	200B	Sent ~3sec ago
Regular.Post	System	200B	Sent ~4sec ago	Post.Data.1	PostTester	200B	Sent ~4sec ago
Regular.Post	System	200B	Sent ~5sec ago	Post.Data.1	PostTester	200B	Sent ~5sec ago
Regular.Post	System	200B	Sent ~6sec ago	Post.Data.1	PostTester	200B	Sent ~6sec ago
Regular.Post	System	200B	Sent ~7sec ago	Post.Data.1	PostTester	200B	Sent ~7sec ago
Regular.Post	System	200B	Sent ~8sec ago	Post.Data.1	PostTester	200B	Sent ~8sec ago
Regular.Post	System	200B	Sent ~9sec ago	Post.Data.1	PostTester	200B	Sent ~9sec ago

There are lots of other tabs to investigate. For each component you can look at that component's subscriptions (the messages it has asked to be triggered by):

Catalogs: - 1 Catalogs, 0 selected - Selected - All Groups - [Cognitive Map](#) - [Log](#) - [Data](#) - [Performance](#) - [Activity](#) - [Subscriptions](#) - [Open in new window](#)

Catalog 1: CCMMaster - Msg in: 601(117.38kB) out: 0(0B) - 2 active trigger(s) (0 inactive) - [Cognitive Map](#) - [Log](#) - [Data](#) - [Performance](#) - [Activity](#) - [Subscriptions](#) - [Open in new window](#)

Current Component Subscriptions [[all](#) | [active](#)]

- Psychone²
 - Ready³
 - Trigger 'DefaultComponentTrigger' on type *Psychone.Ready* to *CCMMaster¹* context *Psychone.Ready* [test](#)
 - Post 'Post' type *Posted.Data.2* context *Psychone.Ready* [test](#)
- Post⁹
 - Data¹⁰
 - 1¹¹
 - Trigger 'Post' on type *Post.Data.1* to *CCMMaster¹* context *Psychone.Ready* [test](#)
 - Post 'Post' type *Posted.Data.2* context *Psychone.Ready* [test](#)

and here you can even manually activate one of the triggers by clicking the 'test' buttons. This will result in that component receiving an empty message with that trigger name.

PsyProbe Web Interface -- Communication statistics

Communication statistics

The performance tab will describe the volume of data going through the module over the last 1, 10 and 30 seconds.

The Log tab will show the components log outputs (made in the code by calling the `logPrint()` method on the api object) filtered for just that one component and the Data tab will show the private data saved by the object (by calling the api method `setPrivateData()`).

The very top line in PsyProbe contains a number of system level views:

- **System Info** - System uptime: 11m 26s 419889us ----- **Log** - Component Communication - Node Communication - Contexts - Subscriptions - Dataflow - Components

The Log entry shows the full log output for all components, but you can filter based on a free-text search to show only what you need.

The Subscriptions tab shows the subscriptions for all components in the system.

The Dataflow tab shows this information in a graphical form, i.e. all the components and the data flowing between them based on their subscriptions. For big systems this becomes very cluttered, so you can use two filters to only see the messages or the components that you are interested in.

Custom views

There are four ways to view custom information within the PsyProbe framework:

1. Add a custom tab for individual components

2. Viewing private data and retrieving from the module itself



```
api->setPrivateData("My Data", str.c_str(), str.length(), "text/html");
```

```
GET /api/getcomponentdata?compid=15&name=MyDataName&format=text
```

3. Use a RequestStore Catalog to centrally collect and present data from messages

```
GET /api/query?from=RequestStore&query=FrameCounter
```

4. Use a custom PsyProbe subsite

You can create your own custom HTML pages at any time - just put the file(s) into the HTML tree and load them by name

```
http://localhost:10000/mycustompage.html
```

Creating custom tabs

This option allows the user to add custom tabs to the component entry on the main PsyProbe page. It involves creating a new template html file (i.e.), putting this file somewhere within the HTML directory structure and then telling Psyclone about it from within your crank via the PsyAPI:

```
api->addPsyProbeCustomView("Stored Messages", "elements/whiteboardmessages.html");
```

The content of the template html file needs to be as a minimum

```
<script>
var TemplateCompID = 0;
var TemplateCompName = "";

function loadTemplate($target, compID) {
    // called once when the template is loaded
    var comp = componentMap[compID];
    TemplateCompID = compID;
    TemplateCompName = comp.name;
    updateTemplate($target, compID);
    ... your custom code
}

function updateTemplate($target, compID) {
    // called regularly for every update cycle
    ... your custom code
}
</script>

... HTML, styles, more scripts, etc.
```

The custom code can then dynamically retrieve information from Psyclone retrieving either private data from the module itself (see below) or querying a RequestStore Catalog (see below).

An example of the template for the whiteboard custom tab can be seen in the template file

```
elements/whiteboardmessages.html
```




Working with private data

If any component saves private data in the crank using the PsyAPI

```
bool setPrivateData(const char* name, const char* data, uint64 size, const char* mimetype = NULL)
```

API, this data will show up in the component's Data section in the PsyProbe web interface if a mimetype has been specified. The mimetypes are the standard browser types and could be textual

```
api->setPrivateData("JSON", json.c_str(), json.length(), "application/json");
api->setPrivateData("LastSentence", utterance.c_str(), utterance.length(), "text/plain");
api->setPrivateData("LastSentenceTime", time.c_str(), time.length(), "text/plain");
```

or binary

```
api->setPrivateData("Face_2", bmpData, size, "image/bmp");
```

This data can also be retrieved either from a custom tab template page (see above) or by any browser page by the following HTTP GET request:

```
GET /api/getcomponentdata?compid=15&name=MyDataName&format=text
```

You just need to know the component ID of the module to query, the name of the private data entry and the format that the crank stored the data in ('text', 'xml', 'json', 'html' or 'binary').

You can enter this URL manually in a browser and see the result directly or you can retrieve this information via AJAX in any HTML page you create.

Use a RequestStore Catalog

The RequestStore Catalog can be used to collect messages from the whole system, keep the most recent versions of them and allow anyone to request parts of or the whole message by easy to remember names. This means no custom programming or data in the modules at all.

An example of a RequestStore Catalog entry in the PsySpec could be:

```
<catalog name="RequestStore" type="RequestStore">
  <trigger name="VideoFrame" type="video.output.frame" />
  <trigger name="SampleXML" type="video.output.frame" />
  <trigger name="SampleText" type="video.output.frame" />
  <trigger name="SampleHTML" type="video.output.frame" />-->
  <setup>
    <store name="VideoFrame" trigger="VideoFrame" key="VideoDataFrame" keytype="data"
      datatype="raw" mimetype="image/bmp" maxkeep="1" maxfrequency="1" />
    <store name="SampleJSON" trigger="VideoFrame" mimetype="json" maxkeep="1"
      maxfrequency="1" />
    <store name="SampleXML" trigger="VideoFrame" mimetype="xml" maxkeep="1"
      maxfrequency="1" />
    <store name="SampleText" trigger="VideoFrame" mimetype="text" key="content"
      maxkeep="1" maxfrequency="1" />
    <store name="SampleNum" trigger="VideoFrame" mimetype="text" key="VideoWidth"
      maxkeep="1" maxfrequency="1" />
    <store name="SampleSize" trigger="VideoFrame" mimetype="text" key="size"
      maxkeep="1" maxfrequency="1" />
    <store name="SampleType" trigger="VideoFrame" mimetype="text" key="type"
      maxkeep="1" maxfrequency="1" />
```

PsyProbe Web Interface -- Use a RequestStore Catalog

```
<store name="SampleTime" trigger="VideoFrame" mimetype="text" key="time"
maxkeep="1" maxfrequency="1" />
<store name="SampleTimeText" trigger="VideoFrame" mimetype="text" key="timetext"
maxkeep="1" maxfrequency="1" />
<!--<store name="SampleHTML" mimetype="html" key="html" maxkeep="1"
maxfrequency="1" />-->
</setup>
</catalog>
```

Here we see the catalog subscribing to messages like any other module, each with a trigger name as usual.

The custom setup XML part then defines named entries ('VideoFrame', 'SampleJSON', etc.) which can be queried by anyone like this:

```
GET /api/query?from=RequestStore&query=VideoFrame
```

The return content-type is specifically specified in the setup XML so the one calling the URL doesn't have to know. Each store entry can have the following parameters:

Name	The name used in the request	Required
trigger	Which trigger message to get the information from	Required
key	The name of the user entry inside the message - can be header fields such as 'time', 'type' too. 'time' will return the number of microseconds since year 0 - 'timetext' will return a nicer textual description of the date and time	If not specified the whole message is returned as either JSON, XML, HTML or TEXT, depending on the mimetype
keytype	The type of key	Not currently used
datatype	The type of binary data, mainly used for bitmaps	If set to 'raw' the code knows to expect raw pixels and need the message to contain width and height information too - so construct a valid bitmap format to return. In the future we may support other types of data types.
mimetype	The mime type used to help the browser display the data correctly	Required, mime types are standard browser mime types such as 'application/json', 'text/html', 'text/xml', etc. Can also be short versions, 'json', 'xml', 'text', etc.
maxkeep	How many messages (trigger history) to keep	Default is 1, later may support returning vectors of keys from several historical messages
maxfrequency	The maximum number of messages to keep per second	If set to 1 it will only keep 1 message every second and ignore new messages for that trigger until another 1 second has passed

Use a custom PsyProbe subsite

Custom location aliases can be added to PsyProbe for application specific use:

```
<psyprobe>
  <alias name="robot" location="/somedir/html" default="index.html" />
</psyprobe>
```

- alias subnodes adds custom dirs to PsyProbe, if present

which means that when a browser requests the url

`http://computer:port/robot/anydir/anyfile`

PsyProbe will return the file found in `/somedir/html/anydir/anyfile`.

This can be any path on the local computer.

Services and interfaces

Psyclone has a number of services built in which are automatically created at startup unless specifically disabled.

PsyProbe is the main system web interface which by default listens to the main system network port interface:

```
<!-- Default services on all Psyclone systems, don't need to be added -->
<service name="PsyProbe" root="../html" />
<service name="Console" />
```

and

```
<!-- Default interfaces on all Psyclone systems, don't need to be added -->
<interface name="PsyProbe" protocol="HTTP" service="PsyProbe" />
<interface name="Console" protocol="Telnet" service="Console" />
```

These two standard interfaces can be disabled completely:

```
<interface protocol="HTTP" service="none" />
<interface protocol="Telnet" service="none" />
```

Other services can be set up in addition to the default ones:

```
<!-- Simple Web Service that just posts its web input and replies
with what comes back as triggers -->
<service name="MyWebService">
  <trigger type="Data.Web.Reply" from="wb" />
  <post type="Data.Web" to="wb" />
</service>

<interface name="MyWebInterface" port="6789" protocol="HTTP" service="MyWebService" />
```

The same can be set up for a telnet service:

```
<service name="MyTelnetService">
  <trigger type="Data.Telnet.Reply" from="wb" />
  <post type="Data.Telnet" to="wb" />
</service>

<interface name="MyTelnet" port="6789" protocol="Telnet" service="MyTelnetService" />
```

and for a Messaging service (remote end needs CMSDK):

```
<!-- Use incoming types already in messages -->
<service name="MyMessageService">
  <trigger type="Data.Message.Reply" from="wb" />
</service>

<!-- Overwrite types before posting -->
<service name="MyMessageService">
  <trigger type="Data.Message.Reply" from="wb" />
  <post type="Data.Message" to="wb" />
</service>

<interface name="MyMessage" port="6789" protocol="Message" service="MyMessageService" />
```

Services and interfaces -- Use a custom PsyProbe subsite

Like the main Psychone network port a new interface can be set up to autodetect the incoming protocol.

This way multiple interfaces speaking different protocols can share the same port:

```
<!-- Additional autodetect interfaces -->
<interface name="MyWebInterface" node="Node0" port="1234" protocol="HTTP"
  service="MyWebService" timeout="3000" default="yes" />
<interface name="MyTelnet" node="Node0" port="1234" protocol="Telnet"
  service="MyTelnetService" timeout="3000" />
<interface name="MyMessage" node="Node0" port="1234" protocol="Message"
  service="MyMessageService" timeout="3000" />
```

In this example, the port will receive the incoming connection and from the client response attempt to detect which type of protocol to speak (HTTP/Telnet/Message). If after 3 seconds nothing has been detected the interface will assume that HTTP is the protocol and that MyWebService is the service.

Distributed Psyclone systems using Nodes

A single Psyclone system started up on a single computer with a PsySpec specified on the command line will run one instance of Psyclone on the local computer. An instance will always start up a local Node through which manages all communication and information in the system including subscriptions, services and interfaces, local components and local process spaces.

Nodes

A Psyclone system can be easily distributed across more than one computer (and more than one operating system such as Linux and Windows) by starting up Nodes on different computers and pointing the PsySpec on the startup computer to these Nodes. To start up an idle Node on another computer just run Psyclone on the command line without specifying a PsySpec and this instance will now listen on its main port for the startup Node connection. The main port number can be changed by specifying port=6789 as a command line parameter.

The PsySpec on the startup computer can now refer to and distribute components to these idle Nodes. First the spec has to define each additional node by name:

```
<node name="Node1" address="localhost" port="11000" />
<node name="Node2" address="otherhost" port="10000" />
```

The local node doesn't need to be specified and is always referred to in the spec as 'Main'.

To start up a component on the local (Main) Node the component simply adds the name:

```
<module name="Ping" node="Main">
...
</module>
```

To start up a component on another Node the component adds its name:

```
<module name="Pong" node="Node1">
...
</module>
```

This works for all types of components:

```
<whiteboard name="WB1" node="Node1" key="count" keytype="integer" />

<whiteboard name="WB2" node="Node2" key="count" keytype="integer">
  <trigger name="Ball" type="ball.4" />
</whiteboard>

<catalog name="MyFiles" node="Node1" type="FileCatalog" root="." />
  <parameter name="ReadOnly" type="String" value="no" />
</catalog>

<catalog node="Main" name="MyData" type="DataCatalog" interval="2000" root="./mydata.dat" />
```

If no Node is specified the component will be created on the startup Node.

Library locations are normally specified globally for all Nodes, but can be specified individually per Node:

```
<node name="Node3" address="192.168.20.20" port="10000">
  <library name="otherlib" library="../path/mylib.dll" />
</node>
```



Process separation using Spaces

Normally when a Psyclone system starts up each Node in the system will automatically create one internal Process Space named 'Root'. This means that everything which happens on this Node will happen within the same OS process in the same memory space (using threads).

Spaces

A Psyclone Node can optionally create more Process Spaces. Each Space will run as a separate process in the local operating system and individual components can be asked to run inside any named Space within a Node. This means that if the code of a component does something bad and the process crashes this will not affect the main Psyclone Node which can then safely restart the crashed Space and recreate all the components inside. If these components have saved their data using the Persistent/Private data API this data will remain intact and accessible to the new instance of the component.

To create a new Space in a single computer Psyclone system just define the Space by name in the PsySpec:

```
<space name="YTTMSpace" />
```

Components can now be asked to run in this Space instead of the Root Space like this:

```
<module name="YTTM" space="YTTMSpace">
    ...

```

If running a distributed (multi-Node) Psyclone system a Space defined like this will be created in all Psyclone Nodes unless you specify which Node the space should be created in:

```
<space node="Node1" name="YTTMSpace" />
<module node="Node1" name="YTTM" space="YTTMSpace">
  ...

```

Any Catalog, Whiteboard or Module can be placed in a process separately from the Node process

```
<catalog space="MySpace" name="MyData" type="DataCatalog" interval="2000" ... />
```

External spaces

Spaces can also be used for allowing third-party software to connect to a running Psyclone system. For this purpose external Spaces can be defined in the PsySpec:

```
<space name="GUIspace" type="external" />
```

and external components can be defined as using this space and a Crank with just a name and no function:

```
<module name="CoCoMapsGUI" space="GUISpace">
  <crank name="CoCoMapsGUI" />
  ...
</module>
```

Process separation using Spaces -- External spaces

In this case neither the Space nor the module will be automatically created, are but rather placeholders for when the external process creates the Space and registers the module crank.

The external program would be compiled to link with the CMSDK library, include the PsyAPI.h header file and then do the following:

Create and connect the Space using the same name as specified in the PsySpec file:

```
PsyAPI* api = NULL;
PsySpace* space = new PsySpace("GUISpace");
if (!space->connect(sysid)) {
    // Local Psyclone not ready yet, delete space and wait and try again later...
    // utils::Sleep(1000);
}
else {
    if (!space->start()) {
        // An error occurred, delete space and wait and try again later...
    }
    else {
        if (!(api = space->getCrankAPI(crankName.c_str()))) {
            // Crank not found, report error, delete space and fail
        }
    }
}
}
```

From now on the newly returned PsyAPI object (api) can be used exactly like for internal modules:

```
while (api->shouldContinue()) {
    if (inMsg = api->waitForNewMessage(100, triggerName)) {
        api->logPrint(1, "Received trigger message: %s", triggerName);
        ...
    }
}
```

If multiple crank functions are defined the getCrankAPI() function can be used multiple times to return multiple PsyAPI objects, one for each function.

The external program can keep an eye on the Psyclone system it is connected to using

```
bool space->isConnected();
bool space->hasShutdown();
```

and if the local Psyclone Node shuts down or fails these will return false. The external program should then delete the space, but NOT the PsyAPI object as this is owned by the PsySpace object and will be destroyed as part of it:

```
delete(space);
space = NULL;
api = NULL;
```

The external process can then periodically retry the connection by creating the Space and trying to connect.

Like internal Spaces, if an external space is shutdown or the program crashes the user can just restart it and Psyclone will continue to run happily.

Communication between separate Psyclone systems

We have seen how components running anywhere in a single Psyclone system seamlessly can communicate with each other, regardless of whether they are in the same process, in separate process Spaces on the same computer or running inside Spaces in Nodes running on different computers in a distributed system.

In some cases, however, multiple independent Psyclone systems need to be able to communicate directly with each other. An example could be two or more robots, each running their own distributed Psyclone system across a couple of computers, needing to share data, either with each other directly or via a separate independent Psyclone system.

Remote requests

Two Psyclone systems can do this via remote queries. A component in one Psyclone system can register a query in their PsySpec configuration like normal, but instead of specifying a source component in its own Psyclone system:

```
<query name="Master" source="CCMMaster" />
```

it can specify a named component in a remote Psyclone system:

```
<query name="Master" source="CCMMaster" host="other.hostname" port="10010" />
```

The component in the remote Psyclone system will receive the query like normal and reply to the query like normal and the reply will be routed back to the original Psyclone system as if the query had been answered locally. There is no reason why any side of the communication should need to know that this query came from a different Psyclone system.

Recognising a remote query

The only way that the remote component can see that this query originated from another Psyclone system is by the fact that the query message contains a few additional entries such as:

```
const char* iden = inMsg->getString("INTERSYSTEM_IDENTIFICATION");
uint32 ipHost = (uint32)msg->getInt("INTERSYSTEM_ADDRESS");
uint16 ipPort = (uint16)msg->getInt("INTERSYSTEM_PORT");
const char* src = inMsg->getString("INTERSYSTEM_SOURCENAME");
```



The CMSDK library

The CMSDK core library is a C++ base library which contains much of the low-level functionality required by Psyclone. It is released under a BSD open source licence and is free for anyone to use for any purpose, except for those limited by the CADIA clause (see below).

It often abstracts the platform and operating system and provides an identical set of objects and utility functions for a very large number of functional areas such as threading and synchronisation, process handling, networking, shared memory handling, working with DLLs, timers, wait queues, binary message containers, HTML request handling incl. a fully functional HTTP server, maths and stats classes, 64-bit integer time functions with microsecond resolution, a request client/server/gateway system, a system-wide verbose and logging system and a very large library of utility functions for working with text and files.

Specifically, it provides the fundamental objects required for working with Psyclone and creating components to be loaded into Psyclone via DLLs or shared objects. The main two are PsyAPI and PsySpace, however, these underneath provide the foundation for the whole Psyclone system and multi-process/multi-computer/multi-platform infrastructure.

API documentation

The full CMSDK library is documented via Doxygen and the methods most commonly used objects by Psyclone users (PsyAPI, DataMessage, etc.) are documented in more detail.

To see the latest version of the documentation please browse to:

<https://cmsdk.cmlabs.com>

Compiling the CMSDK library

CMSDK is the core library providing OS dependent functionality and lots and lots of the base code.

The Python2 and Python3 projects are used to port important parts of CMSDK to Python 2.x and 3.x using SWIG. These libraries can then be used for accessing CMSDK functionality from within a Python program

You need to have Python 2.7 and/or 3.5 installed for either or both 32-bit and 64-bit versions to compile these.

Using Visual Studio 2015 (Windows)

In the Open Source distribution of CMSDK find the Visual Studio solutions file in the top directory called **CMSDK.vs2015.sln**. Open this in Visual Studio 2015 and compile the whole project either in Debug or Release mode. To build the SSL version of CMSDK select either Release SSL or Debug SSL. For this to work the following path must exist and contain the OpenSSL include files:

```
..\..\_libs\OpenSSL\openssl-1.0.2g-vs2015\include      for 32 bit
..\..\_libs\OpenSSL\openssl-1.0.2g-vs2015\include64  for 64 bit
```

and the library files

```
..\..\_libs\OpenSSL\openssl-1.0.2g-vs2015\lib        for 32 bit
..\..\_libs\OpenSSL\openssl-1.0.2g-vs2015\lib64      for 64 bit
```



The CMSDK library -- Linking with the CMSDK library

To use the CMSDK library from a Python program you will need to build either the CMSDKPY2 and/or the CMSDKPY3 DLLs. You do this by selecting them in the Solution Explorer, right click and select Build or Rebuild. For this to work you will need Python 2 and/or 3 installed in either 32 and/or 64-bit versions in the following directories:

C:\Python27	64-bit version of Python 2.7
C:\Python27-32	32-bit version of Python 2.7
C:\Python35	64-bit version of Python 3.5
C:\Python35-32	32-bit version of Python 3.5

Other versions and locations can be used by editing the Visual Studio CMSDKPY2 and CMSDKPY3 project files.

Using Make (Linux)

In the Open Source distribution of CMSDK find the main Makefile in the top directory called **Makefile**. On the command from this directory line issue the command

```
make          to build everything for release
make debug    to build everything for debug
```

To build the SSL version of Psyclone run either

```
make ssl      to build everything for release including SSL support
make ssldebug to build everything for debug including SSL support
```

For this to work the SSL development libraries must be installed on the computer (used via lssl and lcrypto).

To use modules written in the Python language you will need to build either the Python2Link and/or the Python3Link DLLs. You do this by running

```
make python2      to build all files needed for Python 2 integration in release
make python2 debug to build all files needed for Python 2 integration in debug
make python3      to build all files needed for Python 3 integration in release
make python3 debug to build all files needed for Python 3 integration in debug
```

For this to work you will need Python 2 and/or 3 packages installed in the following directories:

```
PYTHON2INCLUDE=-I/usr/include/python2.7
PYTHON3INCLUDE=-I/usr/include/python3.4m
PYTHON2LIB=-lpython2.7
PYTHON3LIB=-lpython3.4m
```

Other versions and locations can be used by editing the Makefile.sys file entries for these.

Linking with the CMSDK library

To link a third-party source project with the CMSDK library one needs to add the CMSDK binary library files to your project and include the appropriate header files needed. Depending on which additional libraries you may need (such as SSL) you may need to link to these as well.

Using Visual Studio 2015 (Windows)

The easiest way to set up a project to use CMSDK is to look at the Psyclone project as it does this already.

The project needs to link with the appropriate binary CMSDK library file. They are called CMSDK(Debug)<vsversion>.lib, such as CMSDKDebug2015.lib and are located in CMSDK/lib/<platform> such as CMSDK/lib/Win32. The SSL builds files have SSL in their names such as CMSDKSSLDebug2015.lib.

The CMSDK library -- Using the CMSDK library

The include header files are located in CMSDK/include.

Lastly, the project needs to be compiled with the code generation flag / runtime library Multi-threaded DLL.

Using Make (Linux)

The easiest way to set up a project to use CMSDK is to look at the Psyclone project as it does this already.

The project needs to link with the appropriate binary CMSDK library file. They are called CMSDK(Debug).a, such as CMSDKDebug.a and are located in CMSDK/lib/<platform> such as CMSDK/lib/32. The SSL builds files have SSL in their names such as CMSDKSSLDebug.a.

The include header files are located in CMSDK/include.

Using the CMSDK library

When the CMSDK library has been linked and the header files location have been set up any program can start including header files and using CMSDK objects and functions. They all live in the namespace called cmsdk which can either be used directly such as

```
cmllabs::PsyAPI* api = new cmllabs::PsyAPI();
```

or the namespace can be specified earlier:

```
using namespace cmllabs;
PsyAPI* api = new PsyAPI();
```

A lot of the utility functions are in a separate namespace called cmllabs::utils, such as

```
cmllabs::utils::StringFormat("(%.3f,%.3f)", x, y);
```

or

```
using namespace cmllabs;
utils::StringFormat("(%.3f,%.3f)", x, y);
```



CMSDK Licensing

CMSDK is released as open source under an EXTENDED BSD License.

Redistribution and use in source and binary forms, with or without modification, is permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright and collaboration notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of its copyright holders nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.
- **CADIA Clause:** The license granted in and to the software under this agreement is a limited-use license. The software may not be used in furtherance of: (i) intentionally causing bodily injury or severe emotional harm to any person; (ii) invading the personal privacy or violating the human rights of any person; or (iii) committing or preparing for any act of war.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Use Cases

The following sections provide some representative examples of the types of projects which the Psyclone platform has been in and how it uniquely benefitted the project in terms of speed of integration and by offering a production-ready, well tested commercial platform.

Integration of third-party applications

When two or more otherwise incompatible software applications need to communicate with each other developers face the prospect of either implementing their own networking protocol or integrating an already existing product. Linking with third-party libraries are not always easy and configuring these to allow any number of data types to be transferred either using publish/subscribe mechanisms or asynchronous queries can take a long time.

Using the Psyclone platform and the CMSDK it is easy for third-party software projects to act as external components, communicating with each other via a distributed Psyclone platform. This platform can either just ask as a publish/subscribe conduit to allow the components to communicate directly with minimal latency or it could use a combination of pub/sub messaging, signalling and asynchronous queries to allow a richer and more multi-modal communication framework.

In addition, the Psyclone platforms offers realtime communication data monitoring via the PsyProbe web interface where the developers can keep track of the dataflow and content. And they can even add standard Psyclone modules and Psyclone contexts to allow the dataflow and the processing to be dynamically updated in realtime depending on the situation and data driven context.

The DataMessage object offers the ability to attach and send any amount of data along including large binary chunks and even a hierarchy of other DataMessages. Everything is timestamped and synchronised to the nearest microsecond even across multiple computers.

Because Psyclone modules can be written in either C++ or Python it allows developers to mix these languages in a single system. More languages are being added such as Java.

Finally, the Psyclone platform allows applications running on a mix of operating systems and bitness architectures to communicate without the usual hurdles which normally comes with multi-platform interactions.

Grid data processing

Processing of data on a scalable grid architecture can be done as long as the application and algorithms fit with any one of the many architectures around such as Hadoop.

A single Psyclone system can be distributed onto any number of computers running a mixture of operating systems. It allows scaling by adding and removing nodes at runtime and manual and automatic load balancing by dynamically moving components to less busy computers.

It provides the PsyProbe web interface for remote non-interfering monitoring of data and custom component visualisation.

The algorithms can be written in any of the supported programming languages – currently C++ and Python 2 and 3 – and more languages are being added such as Java.

And the Psyclone contexts provide the ability to automatically and instantly update part of or the whole dataflow based on data, external dependencies and resource availability, and even allow modules to switch their processing algorithms entirely when in different contexts.



Agent-based simulation

Agent-based simulation traditionally deal with simulating the behaviour or responses of a very large number of actors in a time synchronised system. Each actor or agent have their own internal state, but are usually spawned from identical or groups of identical seed objects.

Psychone was designed to support large distributed simulations with both simple and complex agents. Specifically, Psychone supports:

- Variable time step size – Global time signalling – also called conductor’s stick – allowing all agents to complete their processing before progressing to the next time step
- Spawning of tens or hundreds of modules with identical cranks, but with variable or random initial variable values (via parameters)
- Complex agents comprising a group of modules as a single entity, allowing the developer to create richer agents which makes use of Psychone’s modular architecture and even using multiple programming languages
- Gradual development and integration with initial skeleton modules which over time gets implemented to provide richer and more complex functionality
- Non-interfering monitoring via the PsyProbe web interface with a view of the whole system and zooming into parts and even individual modules and dataflow
- Easy scalable platform to include any number of computers running any mix of supported OSes and bitness
- Mix agents programmed in different programming languages (C++ and Python for now, Java and others coming)

Interactive robots

Creating interactive robots involves a mix of a large number of areas of research such as vision, speech, robotics, navigation as well as working within a limited amount of resources.

The CoCoMaps project, partially funded by the EU via the Echord++ project, used one independent Psychone system per robot, distributed partly onboard the robot and partly on a larger server across a WiFi network. Each robot used the underlying ROS platform for low-level interaction with the robot and above that had a large number of modules and catalogs for state-of-the-art vision including face recognition, gaze and emotional response analysis, speech recognition and generation, role and task planning and execution, dialogue steering and management, virtual control panel interaction and joint search pattern navigation.

The cornerstone of the project was the Collaborative Cognitive Maps architecture, implemented by allowing the robots to communicate with each other in real time via a shared CCMCatalog object. Through this catalog each robot could publish, discuss and negotiate observations, roles and tasks with the other robots and the system specifically allows the robots to completely disagree or reach a partially-settled negotiated state.

To read more about the CoCoMaps project please visit: <http://cmlabs.com/cocomaps>



Tutorials

Creating your first Psyclone system

A simple pingpong PsySpec could look like this:

```
<psySpec>
  <module name="Ping">
    <trigger name="Ready" type="Psyclone.Ready" />
    <trigger name="Ball" type="ball.1" />
    <crank name="Ping" function="Ping" />
    <post name="Ball" type="ball.2" />
  </module>
  <module name="Pong">
    <trigger name="Ball" type="ball.2" />
    <crank name="Pong" function="Ping" />
    <post name="Ball" type="ball.1" />
    <post name="Done" type="Psyclone.Shutdown" />
  </module>
</psySpec>
```

This spec creates two simple modules, one called Ping and the other called Pong. They are both created the built-in crank function called Ping which means that no user code is required to run this system.

The dataflow between the modules goes as follows:

- 1) On system startup the Psyclone system itself posts the Psyclone.Ready message
- 2) Immediately the Ping module is triggered with this message as input
- 3) Once this message has been processed Ping posts an output message with type ball.2
- 4) Immediately the Pong module is triggered with the ball.2 message as input
- 5) Once this message has been processed Pong posts an output message with type ball.1
- 6) Immediately the Ping module is triggered with the ball.1 message as input
- 7) Once this message has been processed Ping posts an output message with type ball.2

And this back-and-forth flow continues until 100,000 messages have been received after which the Pong module posts a message with type Psyclone.Shutdown and the system shuts down. Every 10,000 messages the current throughput performance it logged to the console in lines like:

```
[1] 28/06/2018 08:44:11.323.934 [Ping] Got msg 40000, avg msg time: 89.4us, avg msg age: 89.0us (1.1us faster)...
```

To run this in Psyclone use the following command line:

```
Psyclone spec=pingpong.xml
```


Tutorials -- Creating your first Psyclone system

The system would run for a while and then exit, producing the following console output:

```
28/06/2018 08:44:03.947.650 Psyclone Node starting on port 10000...
[0] 28/06/2018 08:44:04.047.825 Parsing configuration and setting up system...
[0] 28/06/2018 08:44:04.051.347 *** Single Node ready, continuing system setup ***
[1] 28/06/2018 08:44:04.051.455 Success synchronising ID Manager...
[1] 28/06/2018 08:44:04.051.502 Configuring local node...
[1] 28/06/2018 08:44:04.053.195 Creating Module 'Ping'...
[1] 28/06/2018 08:44:04.053.286 SetupQ need more memory size: 5378344 use: 5377048 need: 2352
[1] 28/06/2018 08:44:04.058.505 PsySpace 'Root' (1) starting up...
[1] 28/06/2018 08:44:04.161.087 Configured Module 'Ping' (1) successfully
[1] 28/06/2018 08:44:04.161.204 Creating Module 'Pong'...
[1] 28/06/2018 08:44:04.162.020 Configured Module 'Pong' (2) successfully
[1] 28/06/2018 08:44:04.162.130 Sending config to all other nodes...
[1] 28/06/2018 08:44:04.162.186 *****
[1] 28/06/2018 08:44:04.162.232 ***** SYSTEM READY *****
[1] 28/06/2018 08:44:04.162.274 *****
[1] 28/06/2018 08:44:04.163.309 Space 'Root' connected
[1] 28/06/2018 08:44:04.163.233 [Ping] Started running (internal)...
[1] 28/06/2018 08:44:04.163.288 [Ping] Starting 10 cycles test...
[1] 28/06/2018 08:44:04.163.577 [Pong] Started running (internal)...
[1] 28/06/2018 08:44:05.937.545 [Ping] Got msg 10000, avg msg time: 88.7us, avg msg age: 88.4us...
[1] 28/06/2018 08:44:07.724.488 [Ping] Got msg 20000, avg msg time: 89.3us, avg msg age: 88.7us (0.7us slower)...
[1] 28/06/2018 08:44:09.535.502 [Ping] Got msg 30000, avg msg time: 90.5us, avg msg age: 90.4us (1.2us slower)...
[1] 28/06/2018 08:44:11.323.934 [Ping] Got msg 40000, avg msg time: 89.4us, avg msg age: 89.0us (1.1us faster)...
[1] 28/06/2018 08:44:13.051.856 [Ping] Got msg 50000, avg msg time: 86.4us, avg msg age: 86.1us (3.0us faster)...
[1] 28/06/2018 08:44:14.799.030 [Ping] Got msg 60000, avg msg time: 87.4us, avg msg age: 87.2us (1.0us slower)...
[1] 28/06/2018 08:44:16.566.469 [Ping] Got msg 70000, avg msg time: 88.4us, avg msg age: 88.1us (1.0us slower)...
[1] 28/06/2018 08:44:18.309.929 [Ping] Got msg 80000, avg msg time: 87.2us, avg msg age: 86.9us (1.2us faster)...
[1] 28/06/2018 08:44:20.100.120 [Ping] Got msg 90000, avg msg time: 89.5us, avg msg age: 89.2us (2.3us slower)...
[1] 28/06/2018 08:44:21.848.279 [Pong] Got msg 100000, avg msg time: 87.4us, avg msg age: 88.0us (2.1us faster)...
[1] 28/06/2018 08:44:21.848.336 *****
[1] 28/06/2018 08:44:21.848.526 ***** SYSTEM SHUTDOWN *****
[1] 28/06/2018 08:44:21.848.576 *****
[1] 28/06/2018 08:44:21.848.620 Module Pong requested system shutdown, shutting down...
[1] 28/06/2018 08:44:21.898.589 Local Node is preparing to shutdown...
[1] 28/06/2018 08:44:21.899.068 PsySpace 'Root' (1) shutting down...
[1] 28/06/2018 08:44:22.001.197 PsySpace 'Root' has shut down (1, 00000000057DBB0)
28/06/2018 08:44:22.959.571 Node Networking is shutting down...
28/06/2018 08:44:23.379.153 Psyclone Node on port 10000 has shutdown successfully...
```

The first part of the output up until SYSTEM READY shows the system starting up and configuring itself according to the PsySpec. After this the Psyclone.Ready message is posted and the system starts running, outputting the stats after each batch of 10,000 messages. Then the system receives the message to shut down and it goes through the process of doing just that.

Adding your own modules

Each component in the PsySpec will have at least one crank function to run. Module can specify one or more crank functions as part of their subscription, Whiteboards use a built-in crank function and Catalogs specify their crank function as the Type of the Catalog.

A crank function is the function name in the code library (a DLL or SO) which will be called when a trigger message arrives. Psyclone contains a number of built-in cranks such as the Ping crank above and these do not need to specify the library. For all other components a library needs to be created using the CMSDK, compiled and specified to inform Psyclone about which file on disk contains the compiled code for the crank.

Libraries containing crank functions are specified using one or more library entries in the spec:

```
<library name="MyExamples" library="Examples" />
```

The library name is how subsequent cranks refer to the library and the library entry tells Psyclone which actual file to load. On Windows Psyclone would look for the file Examples.dll in the current directory and on UNIX Psyclone would look for the file libExamples.so. (In addition, when running a debug build of Psyclone it will first look for the debug version of the library by appending Debug to the filename such as ExamplesDebug.dll or libExamplesDebug.so).

Once the library has been defined cranks can refer to the library names:

```
<crank name="Ping" function="MyExamples::Ping" />
```

A custom library can be created using just the CMSDK Open Source library, i.e. Psyclone does not need to be installed on the computer to compile a library, nor does it need the Open Source version of Psyclone to compile or run.

A crank function is normally defined in a C++ header file:

```
#include "PsyAPI.h"
namespace cmlabs {
extern "C" {
    DllExport int8 MyCrankFunction(PsyAPI* api);
    ... more crank function definitions ...
}
} // namespace cmlabs
```

and implemented in a C++ source file:

```
#include "MyCranks.h"

namespace cmlabs {

int8 MyCrankFunction(PsyAPI* api) {
    DataMessage* inMsg, *outMsg;
    const char* triggerName;
    if (api->shouldContinue()) {
        if (inMsg = api->waitForNewMessage(100, triggerName)) {
            api->logPrint(1, "Received trigger message: %s", triggerName);
            outMsg = new DataMessage();
            ... add custom data entries to outMsg ...
            outMsg->setInt("mycount", 3);
            outMsg->setString("mytext", "my data");
        }
    }
}
```

Tutorials -- Adding your own modules

```
api->postOutputMessage("MyPost", outMsg);  
    }  
    }  
    return 0;  
}  
} // namespace cmlabs
```

(For Python-based modules please refer to the Python sections.)

A crank function takes one input parameter which is a pointer to a PsyAPI object. This object can be used by the function to communicate with the Psyclone system including waiting for new messages and posting output messages, but also for reading and changing parameters, retrieving or querying messages from other components, etc.

The definitions of crank functions need to have the `DllExport` classifier which makes them visible to Psyclone when loading the DLL/SO.



Creating Python modules

In addition to modules written in C++ Psyclone also supports running modules written in the language of Python, both version 2.x and 3.x. This is done by allowing the crank function to be written in and loaded from Python.

Crank functions for C++ modules use the PsyAPI object from the CMSDK library to communicate with Psyclone and this in turn makes use of a large proportion of the other objects and functions in the CMSDK library. Full support for the CMSDK library has been added to Python by adding a SWIG interface – which means that any Python code can use the PsyAPI and other objects and functions just like a C++ module would.

Where a C++ module's crank function is specified in the PsySpec like this:

```
<module name="Ping">
  <trigger name="Ready" type="Psyclone.Ready" />
  <trigger name="Ball" type="ball.1" />
  <crank name="Ping" function="Ping" />
  <post name="Ball" type="ball.2" />
</module>
```

a Python module is almost the same except for the crank entry now referring to a Python code file:

```
<module name="Ping">
  <trigger name="Ready" type="Psyclone.Ready" />
  <trigger name="Ball" type="ball.1" />
  <crank name="Ping" language="python2" script="path/to/ping.py" />
  <post name="Ball" type="ball.2" />
</module>
```

The language can be either python2 or python3.

Inline Python modules

For simple Python modules it is even possible to write the crank code directly in the PsySpec file:

```
<crank name="p1" language="python2">
  <![CDATA[
    name = api.getModuleName()
    print("Module name: %s" % name)
    key = api.getParameterInt("Key")
    while (api.shouldContinue()):
      msg = api.waitForNewMessage(20)
      if msg != None:
        mykey = msg.getInt("MyKey")
        if mykey == 0:
          mykey = key
        triggerName = api.getCurrentTriggerName()
        api.logPrint(1, "Got Key: " + str(mykey))
        outMsg = cmsdk.DataMessage()
        outMsg.setInt("MyKey", mykey + 1)
        api.postOutputMessage("Output", outMsg)
  ]]>
</crank>
```

The inline Python code will automatically load all the libraries needed and create the PsyAPI object called api.

Tutorials -- Python code file modules

Please note: Because Python is very sensitive about indentation, please take care that the inline Python code keeps these intact. Psyclone will when loading the module attempt to adjust the indentation.

Python code file modules

The equivalent simple Python module written in external files looks like this:

```
def PsyCrank(apilink):
    api = cmsdk.PsyAPI.fromPython(apilink);
    name = api.getModuleName();
    print("Module name: %s" % name)
    key = api.getParameterInt("Key");
    while (api.shouldContinue()):
        msg = api.waitForNewMessage(20)
        if msg != None:
            mykey = msg.getInt("MyKey")
            if mykey == 0:
                mykey = key
            triggerName = api.getCurrentTriggerName()
            api.logPrint(1, "Got Key: " + str(mykey))
            outMsg = cmsdk.DataMessage()
            outMsg.setInt("MyKey", mykey + 1)
            api.postOutputMessage("Output", outMsg)
```

The first two lines are needed in this case (they are added automatically for inline modules) and there is no need to Import the CMSDK library as Psyclone will do this automatically.

Python import libraries and paths

When Psyclone runs any Python, both inline and in a code file, module it needs to have access to

- 4) The correct version of Python installed correctly
 - a. Correct major version, either 2.x or 3.x
 - b. Correct bitness, either 32 or 64-bit
 - c. The python DLLs in the search path
- 5) The CMSDK binary library and its Python interface file
- 6) Any libraries which the Python module may import in addition

The CMSDK binary library is a file called _cmsdk.pyd on Windows and _cmsdk.so on Linux. This file is accompanied by the Python interface file called cmsdk2.py for Python2 and cmsdk3.py for Python3. In addition, for Debug builds of Psyclone debug versions of these file will be used (_cmsdkdebug.pyd/so and cmsdk2/3debug.py).

These files will either need to be

- d) in the same directory as the Python program as specified in the crank:

```
<crank name="Ping" language="python2" script="path/to/ping.py" />
```

- e) in the same directory as the Psyclone binary used to run the system
- f) and if not, an additional parameter can be specified on the module to tell Psyclone where to find these:

```
<parameter name="libpath" type="String" value="../CMSDK/bin/Win32" />
<crank name="Ping" language="python2" script="path/to/ping.py" />
```



Creating external modules

External modules use Spaces to allow third-party software to connect to a running Psychone system. For this purpose external Spaces can be defined in the PsySpec:

```
<space name="GUISpace" type="external" />
```

and external components can be defined as using this space and a Crank with just a name and no function:

```
<module name="CoCoMapsGUI" space="GUISpace">
  <crank name="CoCoMapsGUI" />
  ...
```

In this case neither the Space nor the module will be automatically created, are but rather placeholders for when the external process creates the Space and registers the module crank.

The external program would be compiled to link with the CMSDK library, include the PsyAPI.h header file and then do the following:

Create and connect the Space using the same name as specified in the PsySpec file:

```
PsyAPI* api = NULL;
PsySpace* space = new PsySpace("GUISpace");
if (!space->connect(sysid)) {
    // Local Psychone not ready yet, delete space and wait and try again later...
    // utils::Sleep(1000);
}
else {
    if (!space->start()) {
        // An error occurred, delete space and wait and try again later...
    }
    else {
        if (!(api = space->getCrankAPI(crankName.c_str()))) {
            // Crank not found, report error, delete space and fail
        }
    }
}
}
```

From now on the newly returned PsyAPI object (api) can be used exactly like for internal modules:

```
while (api->shouldContinue()) {
    if (inMsg = api->waitForNewMessage(100, triggerName)) {
        api->logPrint(1, "Received trigger message: %s", triggerName);
        ...
    }
}
```

If multiple crank functions are defined the getCrankAPI() function can be used multiple times to return multiple PsyAPI objects, one for each function.

The external program can keep an eye on the Psychone system it is connected to using

```
bool space->isConnected();
bool space->hasShutdown();
```

and if the local Psychone Node shuts down or fails these will return false. The external program should then delete the space, but NOT the PsyAPI object as this is owned by the PsySpace object and will be destroyed as part of it:



```
delete(space);  
space = NULL;  
api = NULL;
```

The external process can then periodically retry the connection by creating the Space and trying to connect.

Like internal Spaces, if an external space is shutdown or the program crashes the user can just restart it and Psyclone will continue to run happily.

Creating your own catalogs

In addition to the built-in Catalogs users can create custom Catalogs which can be queried from any other component using either a string-based or message-based interface.

This section first describes how another component can query a Catalog and after that how a custom Catalog can be created to handle such queries.

The string-based query interface is:

```
uint8 queryCatalog(char** result, uint32 &resultsize, const char* name,
    const char* query, const char* operation = NULL, const char* data = NULL,
    uint32 datasize = 0, uint32 timeout = 5000);
```

and can be called like this:

```
status = api->queryCatalog(&result, datasize, "MyFiles", "test", "read");
```

The message-based interface allows more flexibility in that a custom DataMessage can be used to hold the query (including any number of user entries and data types such as images, etc.) and the reply is returned as an equally flexible DataMessage:

```
uint8 queryCatalog(DataMessage** resultMsg, const char* name,
    DataMessage* msg, uint32 timeout = 5000);
```

This can be used like this:

```
status = api->queryCatalog(&resultMsg, "MyQueryName", queryMsg, 5000);
```

Status can be one of

```
#define QUERY_FAILED          1
#define QUERY_TIMEOUT         2
#define QUERY_NAME_UNKNOWN    3
#define QUERY_COMPONENT_UNKNOWN 4
#define QUERY_QUERYFAILED     5
#define QUERY_SUCCESS         6
#define QUERY_NOT_AVAILABLE   7
#define QUERY_NOT_REACHABLE    8
```

Custom Catalogs can be created by writing a custom Catalog crank function. This crank function works exactly like the module crank function and a Catalog can receive and process subscription trigger messages and post output messages. In addition it can also receive synchronous query messages to which is should reply.

```
while (api->shouldContinue()) {
    if (inMsg = api->waitForNewMessage(100, triggerName)) {
        if (triggerName && strcmp(triggerName, "SomeTrigger") == 0) {
            ... process normal trigger message ...
        }
        else if (inMsg->getType() == PsyAPI::CTRL_QUERY) {
            ... process query message ...
            ... and then reply to the query ...
        }
    }
}
```


Tutorials -- Creating your own catalogs

To reply to a query not requiring return data the query reference number is required and a status flag providing information about whether the query succeeded or not.

```
api->queryReply((uint32)inMsg->getReference(), status);
```

The incoming query message for string based queries will contain the textual parameters inside and the Catalog crank can read them like this:

```
if (str = inMsg->getString("Subdir"))
    subdir = str;
if (str = inMsg->getString("Ext"))
    reqext = str;
if (str = inMsg->getString("Binary"))
    reqbinary = (strcmp(str, "yes") == 0);
if ((str = inMsg->getString("Operation")))
    reqwrite = (strcmp(str, "write") == 0);
```

and the reply needs to be done like this:

```
bool queryReply(uint32 id, uint8 status, char* data, uint32 size, uint32 count);
```

such as

```
api->queryReply((uint32)inMsg->getReference(), QUERY_SUCCESS, data, size, 0);
```

For message-based queries the incoming query message can contain any number of custom data entries such as text, numbers, raw data and attached messages. Replies to these should be a Data Message too which similarly can contain any number of data entries:

```
bool queryReply(uint32 id, uint8 status, DataMessage* msg = NULL);
```

such as

```
api->queryReply((uint32)inMsg->getReference(), status, replyMsg);
```

Tutorials -- Add custom data in PsyProbe

Add custom data in PsyProbe

If any component saves private data in the crank using the PsyAPI

```
bool setPrivateData(const char* name, const char* data, uint64 size, const char* mimetype = NULL)
```

API, this data will show up in the component's Data section in the PsyProbe web interface if a mimetype has been specified. The mimetypes are the standard browser types and could be textual

```
api->setPrivateData("JSON", json.c_str(), json.length(), "application/json");  
api->setPrivateData("LastSentence", utterance.c_str(), utterance.length(), "text/plain");  
api->setPrivateData("LastSentenceTime", time.c_str(), time.length(), "text/plain");
```

or binary

```
api->setPrivateData("Face_2", bmpData, size, "image/bmp");
```

This data can also be retrieved either from a custom tab template page (see above) or by any browser page by the following HTTP GET request:

```
GET /api/getcomponentdata?compid=15&name=MyDataName&format=text
```

You just need to know the component ID of the module to query, the name of the private data entry and the format that the crank stored the data in ('text', 'xml', 'json', 'html' or 'binary').

You can enter this URL manually in a browser and see the result directly or you can retrieve this information via AJAX in any HTML page you create.

Tutorials -- Add a custom tab in PsyProbe

Add a custom tab in PsyProbe

PsyProbe allows the user to add custom tabs to the component entry on the main PsyProbe page. It involves creating a new template html file, putting this file somewhere within the HTML directory structure and then telling Psyclone about it from within your crank via the PsyAPI:

```
api->addPsyProbeCustomView("Stored Messages",
    "elements/whiteboardmessages.html");
```

The content of the template html file needs to be as a minimum

```
<script>
    var TemplateCompID = 0;
    var TemplateCompName = "";

    function loadTemplate($target, compID) {
        // called once when the template is loaded
        var comp = componentMap[compID];
        TemplateCompID = compID;
        TemplateCompName = comp.name;
        updateTemplate($target, compID);
        ... your custom code
    }

    function updateTemplate($target, compID) {
        // called regularly for every update cycle
        ... your custom code
    }
</script>

... HTML, styles, more scripts, etc.
```

The custom code can then dynamically retrieve information from Psyclone retrieving either private data from the module itself (see below) or querying a RequestStore Catalog (see below).

An example of the template for the whiteboard custom tab can be seen in the template file

```
elements/whiteboardmessages.html
```

Tutorials -- Example of a PsySpec for a large system

Example of a PsySpec for a large system

The following PsySpec is from the CoCoMaps project and consists of a complete independent robot system. For more information about the CoCoMaps project please see <http://cmlabs.com/cocomaps>.

<psySpec>

```

<!-- ***** -->
<!-- System configuration -->
<!-- ***** -->

<include file="system2.inc" />
<library name="SpeechAnalyser" library="SpeechAnalyser" />
<library name="Nuance" library="Nuance" />
<library name="Perception" library="Perception" />
<library name="ROSInterface" library="ROSInterface" />
<library name="InteractionManager" library="InteractionManager" />

<module name="DefaultRoleDetector">
  <trigger name="NowDefaultRole" type="Self.Role.Searcher" />
  <post name="DialogOff" type="dialog.off"/>
  <post name="AutoNavigationOn" type="navigation.auto.on" />
  <post name="StopMicrophone" type="cmd.input.audio.off" />
</module>

<module name="PrimaryRoleDetector">
  <trigger name="NowPrimaryRole" type="Self.Role.Communicator" />
  <post name="DialogOn" type="dialog.on"/>
  <post name="AutoNavigationOff" type="navigation.auto.off" />
  <post name="StartMicrophone" type="cmd.input.audio.on" />
  <post name="NavigateCancel" type="Robot.Navigate.Cancel" />
</module>

<module name="SecondaryRoleDetector">
  <trigger name="NowSecondaryRole" type="Self.Role.Controller" />
  <post name="DialogOff" type="dialog.off"/>
  <post name="AutoNavigationOff" type="navigation.auto.off" />
  <post name="StopMicrophone" type="cmd.input.audio.off" />
  <post name="NavigateCancel" type="Robot.Navigate.Cancel" />
</module>

<!-- ***** -->
<!-- PsyProbe configuration -->
<!-- ***** -->

<psyprobe location="%PsyDir%/html">
  <!-- http://localhost:10000/robot/file.html -->
  <alias name="robot" location="%HTMLDir%" default="index.html" />
  <post name="CommandMoveForward" type="Robot.Command.Move.Forward" />
  <post name="CommandMoveBackward" type="Robot.Command.Move.Backward" />
  <post name="CommandTurnLeft" type="Robot.Command.Turn.Left" />
  <post name="CommandTurnRight" type="Robot.Command.Turn.Right" />
  <post name="CommandMoveForwardLeft" type="Robot.Command.Move.ForwardLeft" />
  <post name="CommandMoveForwardRight" type="Robot.Command.Move.ForwardRight" />
  <post name="CommandMoveBackwardLeft" type="Robot.Command.Move.BackwardLeft" />
  <post name="CommandMoveBackwardRight" type="Robot.Command.Move.BackwardRight" />
  <post name="CommandReturnToDock" type="Robot.Command.ReturnToDock" />
  <post name="ReportRobotPosError" type="Report.Robot.PositionError" />
  <post name="ReportHumanExit" type="Report.Human.Exit" />
  <post name="ReportHumanPosError" type="Report.Human.PositionError" />
  <post name="ReportHumanIDError" type="Report.Human.IDError" />
  <post name="ReportHumanEntry" type="Report.Human.Entry" />
  <post name="ReportHumanLeft" type="Report.Human.Left" />
</psyprobe>

<catalog name="MessageDataCatalog" type="MessageDataCatalog">
  <trigger name="RobotStatus" type="Robot.Status" />
  <trigger name="Blob" type="Robot.Vision.Blob" />
  <trigger name="USB" type="Robot.Camera.USB" />
  <trigger name="Color" type="Robot.Camera.Color" />
  <trigger name="Depth" type="Robot.Camera.Depth" />
  <trigger name="Registered" type="Robot.Camera.Registered" />
  <trigger name="IR" type="Robot.Camera.IR" />
  <trigger name="Map" type="Robot.Map" />
  <setup>
    <store name="USBImage" trigger="USB" key="Image" keytype="data"
      datatype="raw" mimetype="image/bmp" maxkeep="1" maxfrequency="1" />
    <store name="USBImageRaw" trigger="USB" key="Image" keytype="data"
      mimetype="application/binary" maxkeep="1" maxfrequency="1" />
    <store name="ColorImage" trigger="Color" key="Image" keytype="data"
      datatype="raw" mimetype="image/bmp" maxkeep="1" maxfrequency="1" />
    <store name="ColorImageRaw" trigger="Color" key="Image" keytype="data"
      mimetype="application/binary" maxkeep="1" maxfrequency="1" />
  </setup>
</catalog>

```

Tutorials -- Example of a PsySpec for a large system

```

<store name="DepthImage" trigger="Depth" key="Image" keytype="data"
  datatype="raw" mimetype="image/bmp" maxkeep="1" maxfrequency="1" />
<store name="DepthImageRaw" trigger="Depth" key="Image" keytype="data"
  mimetype="application/binary" maxkeep="1" maxfrequency="1" />
<store name="RegisteredImage" trigger="Registered" key="Image" keytype="data"
  datatype="raw" mimetype="image/bmp" maxkeep="1" maxfrequency="1" />
<store name="RegisteredImageRaw" trigger="Registered" key="Image" keytype="data"
  mimetype="application/binary" maxkeep="1" maxfrequency="1" />
<store name="IRImage" trigger="IR" key="Image" keytype="data"
  datatype="raw" mimetype="image/bmp" maxkeep="1" maxfrequency="1" />
<store name="IRImageRaw" trigger="IR" key="Image" keytype="data"
  mimetype="application/binary" maxkeep="1" maxfrequency="1" />
<store name="Map" trigger="Map" key="Image" keytype="data"
  datatype="raw" mimetype="image/bmp" maxkeep="1" maxfrequency="1" />
<store name="MapRaw" trigger="Map" key="Image" keytype="data"
  mimetype="application/binary" maxkeep="1" maxfrequency="1" />
<store name="StatusJSON" trigger="RobotStatus" mimetype="json" maxkeep="1" maxfrequency="1" />
<store name="StatusText" trigger="RobotStatus" mimetype="text" maxkeep="1" maxfrequency="1" />
<store name="StatusXML" trigger="RobotStatus" mimetype="xml" maxkeep="1" maxfrequency="1" />
</setup>
</catalog>

<!-- ***** CCMCatalog configuration ***** -->
<!-- CCMCatalog configuration -->
<!-- ***** -->

<catalog name="CCMMaster" type="CCMProxyCatalog">
  <parameter name="SystemID" type="Integer" value="%SystemID%" />
  <query name="Master" source="CCMMaster" host="%MasterAddress%" port="%MasterPort%" />
  <trigger name="ObjectInfo" type="Object.Information" />
  <post name="CommandMoveForward" type="Robot.Command.Move.Forward" />
  <post name="CommandMoveBackward" type="Robot.Command.Move.Backward" />
  <post name="CommandTurnLeft" type="Robot.Command.Turn.Left" />
  <post name="CommandTurnRight" type="Robot.Command.Turn.Right" />
  <post name="CommandMoveForwardLeft" type="Robot.Command.Move.ForwardLeft" />
  <post name="CommandMoveForwardRight" type="Robot.Command.Move.ForwardRight" />
  <post name="CommandMoveBackwardLeft" type="Robot.Command.Move.BackwardLeft" />
  <post name="CommandMoveBackwardRight" type="Robot.Command.Move.BackwardRight" />
  <post name="CommandStop" type="Robot.Command.Stop" />
  <post name="CommandNavigateTo" type="Robot.Request.Navigate" />
  <post name="CommandReturnToDock" type="Robot.Command.ReturnToDock" />
  <post name="ReportRobotPosError" type="Report.Robot.PositionError" />
  <post name="ReportHumanExit" type="Report.Human.Exit" />
  <post name="ReportHumanPosError" type="Report.Human.PositionError" />
  <post name="ReportHumanIDError" type="Report.Human.IDError" />
  <post name="ReportHumanEntry" type="Report.Human.Entry" />
  <post name="ReportHumanLeft" type="Report.Human.Left" />
  <post name="RoleAssigned" type="Role.Assigned" />
  <post name="RoleLeft" type="Role.Left" />
  <post name="RoleGiven" type="Role.Given" />
  <post name="TaskCreated" type="Task.Created" />
  <post name="TaskAssigned" type="Task.Assigned" />
  <post name="TaskAccepted" type="Task.Accepted" />
  <post name="TaskUpdated" type="Task.Updated" />
  <post name="TaskCompleted" type="Task.Completed" />
  <post name="TaskTimeout" type="Task.Timeout" />
  <post name="TaskCancelled" type="Task.Cancelled" />
</catalog>

<catalog name="PositionCollector1" type="CCMCollector">
  <query name="Master" source="CCMMaster" />
  <trigger name="RobotStatus" type="Self.Position.Data" />
  <trigger name="RobotDetection" type="Robot.Self.Detected" />
  <trigger name="HumanDetection" type="Human.Self.Detected" />
  <trigger name="IncomingSpeech" type="input.speech.detected" />
  <trigger name="OutgoingSpeech" type="output.speech.detected" />
  <setup>
    <obs name="Location" obstype="Location" trigger="RobotStatus">
      </obs>
    <obs name="Location" obstype="Location" trigger="RobotDetection">
      <mapping entry="x" key="PosX" />
      <mapping entry="y" key="PosY" />
    </obs>
    <obs name="Location" obstype="Location" trigger="HumanDetection">
      <mapping entry="x" key="PosX" />
      <mapping entry="y" key="PosY" />
    </obs>
    <obs name="Utterance" obstype="String" trigger="IncomingSpeech">
      <mapping entry="val" key="Utterance" />
    </obs>
    <obs name="Utterance" obstype="String" trigger="OutgoingSpeech">
      <mapping entry="val" key="Utterance" />
    </obs>
  </setup>

```



```

</setup>
</catalog>

<!-- ***** -->
<!-- Whiteboards -->
<!-- ***** -->

<whiteboard name="MessagesOfInterest">
  <trigger name="Dialog" type="dialog.*" />
  <trigger name="Navigation" type="navigation.*" />
  <trigger name="Roles" type="Role.*" />
  <trigger name="Tasks" type="Task.*" />
  <trigger name="RobotNavigation" type="Robot.Status.Navigate.*" />
  <trigger name="NavigationRequests" type="Robot.Request.*" />
  <trigger name="InputFace" type="Input.Face.*" />
  <trigger name="FaceRecog" type="Face.Detection.*" />
  <trigger name="Human" type="Human.*" />
  <trigger name="InputSpeech" type="input.speech.*" />
  <trigger name="OutputSpeech" type="output.speech.*" />
  <trigger name="AudioStarted" type="output.audio.started" />
  <trigger name="AudioFinished" type="output.audio.ended" />
  <trigger name="Panel1" type="Panel1.Command.*" />
  <trigger name="Panel2" type="Panel2.Command.*" />
</whiteboard>

<!-- ***** -->
<!-- Demo data recording configuration -->
<!-- ***** -->

<catalog name="DemoRecording" type="ReplayCatalog" root="%RecordingDir%/Demo3SlaveRecording" maxsize="100000000" -->
  <trigger name="ReportRobotPosError" type="Report.Robot.PositionError" />
  <trigger name="ReportHumanExit" type="Report.Human.Exit" />
  <trigger name="ReportHumanPosError" type="Report.Human.PositionError" />
  <trigger name="ReportHumanIDError" type="Report.Human.IDError" />
  <trigger name="ReportHumanEntry" type="Report.Human.Entry" />
  <trigger name="ReportHumanLeft" type="Report.Human.Left" />
  <trigger name="IdentityData" type="Self.Identity.Data" />
  <trigger name="RobotStatus" type="Self.Position.Data" />
  <trigger name="RobotDetection" type="Robot.Self.Detected" />

  <trigger name="HumanAppearedSelf" type="Human.Self.Appeared" />-->
  <trigger name="HumanAppearedOther" type="Human.Other.Appeared" />-->
  <trigger name="HumanLeft" type="Human.Left" />

  <trigger name="NavigationActive" type="Robot.Status.Navigate.Active" />
  <trigger name="NavigationComplete" type="Robot.Status.Navigate.Complete" />
  <trigger name="NavigationFailed" type="Robot.Status.Navigate.Failed" />
  <trigger name="NavigationTimeout" type="Robot.Status.Navigate.Timeout" />
  <trigger name="NavigationBusy" type="Robot.Status.Navigate.Busy" />
  <trigger name="NavigateCommand" type="Robot.Command.Navigate" />
  <trigger name="UnknownFace" type="Face.Detection.Unknown" />
  <trigger name="BadMatchFace" type="Face.Detection.Bad" />
  <trigger name="NoFace" type="Face.Detection.NoFace" />
  <trigger name="Face" type="Face.Detection.Person" />
  <trigger name="FaceTooBusy" type="Face.Detection.Server.Skip" />
  <trigger name="FaceQueueSkip" type="Face.Detection.Queue.Skip" />
  <trigger name="FaceProcFailed" type="Face.Detection.Failed" />
  <trigger name="FaceProcTimeout" type="Face.Detection.Timeout" />
</catalog>

<!-- ***** -->
<!-- Robot control -->
<!-- ***** -->

<module name="RobotStatus">
  <parameter name="MapMessage" type="String" value="%DataDir%/map.msg" />
  <parameter name="ColorMessage" type="String" value="%DataDir%/color.msg" />
  <parameter name="DepthMessage" type="String" value="%DataDir%/registered.msg" />
  <parameter name="simulateMoving" type="String" value="%SimulateSystem%" />
  <parameter name="interval" type="Integer" value="5000" />
  <parameter name="libpath" type="String" value="%PylibDir%" />
  <parameter name="initpose_x" type="Float" value="%initpose_x%" />
  <parameter name="initpose_y" type="Float" value="%initpose_y%" />
  <parameter name="initpose_oz" type="Float" value="%initpose_oz%" />
  <parameter name="initpose_ow" type="Float" value="%initpose_ow%" />

  <trigger name="Ready" type="Psychone.Ready" />
  <trigger name="CommandMoveForward" type="Robot.Command.Move.Forward" />
  <trigger name="CommandMoveBackward" type="Robot.Command.Move.Backward" />
  <trigger name="CommandTurnLeft" type="Robot.Command.Turn.Left" />
  <trigger name="CommandTurnRight" type="Robot.Command.Turn.Right" />

```

Tutorials -- Example of a PsySpec for a large system

```

<trigger name="CommandMoveForwardLeft" type="Robot.Command.Move.ForwardLeft" />
<trigger name="CommandMoveForwardRight" type="Robot.Command.Move.ForwardRight" />
<trigger name="CommandMoveBackwardLeft" type="Robot.Command.Move.BackwardLeft" />
<trigger name="CommandMoveBackwardRight" type="Robot.Command.Move.BackwardRight" />
<trigger name="NavigateCommand" type="Robot.Command.Navigate" />
<trigger name="NavigateCancel" type="Robot.Command.Navigate.Cancel" />
<post name="NavigateConfirm" type="Robot.Navigate.Confirm" />
<post name="NavigateSuccess" type="Robot.Navigate.Success" />
<post name="NavigateFailed" type="Robot.Navigate.Failed" />
<post name="NavigateTimeout" type="Robot.Navigate.Timeout" />
<post name="NavigateBusy" type="Robot.Navigate.Busy" />
<post name="NavigateCancelled" type="Robot.Navigate.Cancelled" />
<crank name="RobotStatus" language="python2" script="%AppDir%/robotstatus.py" />
<xxxxcrank name="RobotStatus" function="ROSInterface::RobotStatusVirtual" />
<post name="Status" type="Robot.Status" />
<post name="Blob" type="Robot.Vision.Blob" />
<post name="USB" type="Robot.Camera.USB" />
<post name="Color" type="Robot.Camera.Color" />
<post name="Depth" type="Robot.Camera.Depth" />
<post name="Registered" type="Robot.Camera.Registered" />
<post name="IR" type="Robot.Camera.IR" />
<post name="Map" type="Robot.Map" />
<post name="Button0Pressed" type="Robot.Button.0.Pressed" />
<post name="Button0Released" type="Robot.Button.0.Released" />
<post name="Button1Pressed" type="Robot.Button.1.Pressed" />
<post name="Button1Released" type="Robot.Button.1.Released" />
<post name="Button2Pressed" type="Robot.Button.2.Pressed" />
<post name="Button2Released" type="Robot.Button.2.Released" />
</module>

<module name="RobotSelf">
  <parameter name="InvertPosX" type="String" value="Yes" />
  <parameter name="Image" type="String" value="%DataDir%/turtlebot2.bmp" />
  <parameter name="IdentityID" type="Integer" value="%SystemID%" />
  <trigger name="Map" type="Robot.Map" />
  <trigger name="RobotStatus" type="Robot.Status" />

  <trigger name="OutgoingSpeech" type="output.speech.utterance" />
  <post name="DetectedSpeech" type="output.speech.detected" />

  <query name="Master" source="CCMMaster" />
  <crank name="RobotSelf" function="Perception::RobotSelf" />
  <post name="IdentityData" type="Self.Identity.Data" />
  <post name="PositionData" type="Self.Position.Data" />
</module>

<module name="RobotNavigation">
  <parameter name="HumanTimeout" type="Integer" value="30000" />
  <parameter name="PointTimeout" type="Integer" value="10000" />
  <parameter name="NavigationTimeout" type="Integer" value="30000" />
  <trigger name="AutoNavigationOn" type="navigation.auto.on" />
  <trigger name="AutoNavigationOff" type="navigation.auto.off" />
  <trigger name="NavigateToNamedPoint" type="Robot.Request.Navigate.NamedPoint" />
  <trigger name="NavigateRequest" type="Robot.Request.Navigate" />
  <trigger name="NavigateConfirm" type="Robot.Navigate.Confirm" />
  <trigger name="NavigateSuccess" type="Robot.Navigate.Success" />
  <trigger name="NavigateFailed" type="Robot.Navigate.Failed" />
  <trigger name="NavigateTimeout" type="Robot.Navigate.Timeout" />
  <trigger name="NavigateBusy" type="Robot.Navigate.Busy" />
  <trigger name="NavigateCancel" type="Robot.Navigate.Cancel" />
  <trigger name="NavigateCancelled" type="Robot.Navigate.Cancelled" />
  <query name="Master" source="CCMMaster" />
  <crank name="RobotNavigation" function="Perception::RobotNavigation" />
  <post name="NavigationActive" type="Robot.Status.Navigate.Active" />
  <post name="NavigationComplete" type="Robot.Status.Navigate.Complete" />
  <post name="NavigationFailed" type="Robot.Status.Navigate.Failed" />
  <post name="NavigationTimeout" type="Robot.Status.Navigate.Timeout" />
  <post name="NavigationBusy" type="Robot.Status.Navigate.Busy" />
  <post name="NavigateCommand" type="Robot.Command.Navigate" />
  <post name="NavigateCancel" type="Robot.Command.Navigate.Cancel" />
</module>

<!-- ***** -->
<!-- Human detection -->
<!-- ***** -->

<module name="FaceFinder">
  <parameter type="String" name="ClassifierA" value="haarcascade_frontalface_alt.xml" />
  <parameter type="String" name="ClassifierB" value="haarcascade_frontalface_alt2.xml" />
  <parameter type="String" name="ClassifierC" value="haarcascade_eye.xml" />
  <trigger name="Start" type="Pyclone.Ready" after="5000" />
  <crank name="FaceFinder" function="Perception::FaceFinder" />
  <post name="Face" type="Input.Face.Found" />

```

Tutorials -- Example of a PsySpec for a large system

```

<post name="ImageFaces" type="Robot.Camera.USB" />
</module>

<module name="FaceRecognition">
  <parameter name="Username" type="String" value="%FaceServerUsername%" />
  <parameter name="Password" type="String" value="%FaceServerPassword%" />
  <parameter name="ServerAddr" type="String" value="%FaceServerAddress%" />
  <parameter name="ServerPort" type="Integer" value="%FaceServerPort%" />
  <parameter name="PrintSuccess" type="String" value="Yes" />
  <parameter name="PrintNoFace" type="String" value="Yes" />
  <parameter name="MatchThreshold" type="Integer" value="50" />
  <query name="Master" source="CCMMaster" />
  <trigger name="Start" type="Psychone.Ready" />
  <trigger name="Face" type="Input.Face.Found" />
  <xxxtrigger name="VideoFrame" type="Robot.Camera.USB" />
  <crank name="FaceRecognition" function="Perception::FaceRecognition" />
  <post name="Ready" type="Face.Detection.Ready" />
  <post name="NoFace" type="Face.Detection.NoFace" />
  <post name="UnknownFace" type="Face.Detection.Unknown" />
  <post name="BadMatchFace" type="Face.Detection.Bad" />
  <post name="Face" type="Face.Detection.Person" />
  <post name="TooBusy" type="Face.Detection.Server.Skip" />
  <post name="InputQueueSkip" type="Face.Detection.Queue.Skip" />
  <post name="FaceProcFailed" type="Face.Detection.Failed" />
  <post name="FaceProcTimeout" type="Face.Detection.Timeout" />
</module>

<module name="HumanDetection">
  <parameter name="FaceToWidthScale" type="Float" value="3" />
  <parameter name="AngleFromPixelsOffCentre" type="Float" value="3" />
  <parameter name="VisualToDepthOffsetX" type="Integer" value="300" />
  <parameter name="VisualToDepthOffsetY" type="Integer" value="200" />
  <parameter name="DepthTopShade" type="Integer" value="200" />
  <parameter name="VisualToDepthScaleX" type="Float" value="0.38" />
  <parameter name="VisualToDepthScaleY" type="Float" value="0.60" />
  <parameter name="HumanLeftTimeout" type="Integer" value="30000" />
  <trigger name="Face" type="Face.Detection.Person" />
  <trigger name="Depth" type="Robot.Camera.Registered" />
  <trigger name="Status" type="Robot.Status" />
  <trigger name="PositionData" type="Self.Position.Data" />
  <query name="Master" source="CCMMaster" />
  <crank name="HumanDetection" function="Perception::HumanDetection" />
  <post name="Detection" type="Human.Self.Detected" />
  <post name="HumanAppearedSelf" type="Human.Self.Appeared" />
  <post name="HumanAppearedOther" type="Human.Other.Appeared" />
  <post name="HumanLeft" type="Human.Left" />
  <post name="OtherSystemDetection" type="Human.Other.Detected" />
  <trigger name="IncomingSpeech" type="input.speech.info.semantic" />
  <post name="DetectedSpeech" type="input.speech.detected" />
</module>

<!-- ***** -->
<!-- Speech input -->
<!-- ***** -->

<!-- *** Audio input and output *** -->
<module name="RTAudioIn">
  <parameter name="SampleFrequency" type="Integer" value="16000" />
  <parameter name="BufferSize" type="Integer" value="768" />
  <parameter name="List" type="Integer" value="0" />
  <parameter name="DeviceName" type="String" value="%InputAudioDevice%" />
  <trigger name="AudioStart" type="cmd.input.audio.on" />
  <trigger name="AudioStop" type="cmd.input.audio.off" />
  <trigger name="Guidance" type="cmd.input.audio.guidance" />
  <post name="AudioFrame" type="input.audio.frame" />
  <crank name="RTAudioInDirectional" function="Nuance::RTAudioInDirectional" />
</module>

<!-- *** Pitch input *** -->
<module name="PitchTrackerInput" >
  <description>
    Detects pitch, pitch derivative and slope, silences and hums in a continuous speech.
    Typical latency: 15ms. subject to scheduling latencies (ex: add Psychone latencies when
    the scheduler yields a sequential run)
    Typical Psychone routing latency: random, in {0,15} ms (scheduling sensitive)
  </description>
  <parameter name="ModuleID" type="Integer" value="1" />
  <parameter name="BufferSize" type="Integer" value="768" />
  <parameter name="SamplingFrequency" type="Integer" value="16000" />
  <parameter name="Debug" type="Integer" value="0" />
  <parameter name="StoreResultsInFile" type="Integer" value="0" />
  <parameter name="AmplitudeThreshold" type="Float" value="0.65" />
  <parameter name="NoiseThreshold" type="Float" value="6.0" />

```


Tutorials -- Example of a PsySpec for a large system

```
<parameter name="PitchHighPass" type="Integer" value="0" />
<parameter name="PitchLowPass" type="Integer" value="200" />
<parameter name="PitchDeltaConstraint" type="Float" value="5.0" />
<parameter name="PitchMonitorWindow" type="Integer" value="1000" />
<parameter name="PauseThreshold" type="Integer" value="1200" />
<parameter name="SegmentIntervalThreshold" type="Integer" value="70" />
<parameter name="SpeechOnDelay" type="Integer" value="2" />
<parameter name="HumDurationThreshold" type="Integer" value="200" />
<parameter name="HumPitchThreshold" type="Integer" value="10" />
<parameter name="PrintPitch" type="String" value="No" />
<trigger name="Start" type="input.audio.on"/>
<trigger name="Stop" type="input.audio.off" />
<trigger name="AudioFrame" type="input.audio.frame" />
<post name="SpeechOff" type="input.speech.raw.off"/>
<post name="SpeechOn" type="input.speech.raw.on" />
<post name="SpeechPause" type="input.speech.raw.pause" />
<crank name="PitchRuntimeInput" function="SpeechAnalyser::Runtime" />
</module>

<module name="PitchTrackerDetector">
  <parameter name="Threshold" type="Float" value="1.5" />
  <trigger name="AudioFrame" type="input.audio.frame" />
  <trigger name="SpeechOff" type="input.speech.raw.off"/>
  <trigger name="SpeechOn" type="input.speech.raw.on" />
  <trigger name="SpeechPause" type="input.speech.raw.pause" />
  <crank name="PitchTrackerDetector" function="SpeechAnalyser::PitchTrackerDetector" />
  <post name="SpeakerChange" type="input.speaker.change" />
  <post name="SpeechOff" type="input.speech.off" />
  <post name="SpeechOn" type="input.speech.on" />
  <post name="SpeechPause" type="input.speech.pause" />
</module>

<module name="OverlapAnalyzer" >
  <description>
    Alerts to temporal overlaps between two or more voices
  </description>
  <trigger name="SpeechInputOff" type="input.speech.off" />
  <trigger name="SpeechInputOn" type="input.speech.on" />
  <trigger name="SpeechSelfOn" type="output.audio.started" />
  <trigger name="SpeechSelfOff" type="output.audio.ended" />
  <post name="OverlapStart" type="audio.overlap.on"/>
  <post name="OverlapStop" type="audio.overlap.off"/>
  <post name="PitchSilenceStart" type="audio.silence.started"/>
  <post name="PitchSilenceStop" type="audio.silence.ended"/>
  <crank name="PitchAnalyzeSlope" function="SpeechAnalyser::AnalyzeTwo" />
</module>

<module name="InterruptionDetector" >
  <parameter name="Print" type="String" value="Yes" />
  <parameter name="PossibleOverlap" type="Integer" value="600" />
  <parameter name="DefiniteOverlap" type="Integer" value="1200" />
  <trigger name="SpeechInputStart" type="input.speech.on" />
  <trigger name="SpeechInputEnd" type="input.speech.off" />
  <trigger name="SpeechOutputStart" type="output.audio.started" />
  <trigger name="SpeechOutputEnd" type="output.audio.ended" />
  <post name="InputInterruptPossible" type="input.speech.interrupt.possible"/>
  <post name="InputInterruptDefinite" type="input.speech.interrupt.definite"/>
  <post name="InputInterruptCancel" type="input.speech.interrupt.cancel"/>
  <crank name="InterruptionDetector" function="SpeechAnalyser::InterruptionDetector" />
</module>

<!--module name="YTMM" space="YTMMspace" -->
<module name="YTMM">
  <description>
    Turn-taking module which controls all reactive (low-level) turn decisions.
  </description>
  <parameter name="libpath" type="String" value="Psyclone/CMSDK/bin/Win32" />
  <trigger name="SpeechOffInput" type="input.speech.off"/>
  <trigger name="SpeechOnInput" type="input.speech.on" />
  <trigger name="SpeechOffOutput" type="output.audio.ended"/>
  <trigger name="SpeechOnOutput" type="output.audio.started" />
  <trigger name="OverlapStart" type="audio.overlap.on"/>
  <trigger name="OverlapStop" type="audio.overlap.off"/>
  <trigger name="DialogOn" type="dialog.on"/>
  <trigger name="DialogOff" type="dialog.off"/>
  <trigger name="SpeakCommandSent" type="dialog.on.utterance.started"/>
  <trigger name="StoppedSpeaking" type="dialog.on.utterance.ended"/>
  <post name="AudioPause" type="cmd.output.audio.pause" />
  <post name="AudioResume" type="cmd.output.audio.resume" />
  <post name="IHaveTurn" type="dialog.on.i-have-turn"/>
  <post name="OtherHasTurn" type="dialog.on.other-has-turn"/>
  <post name="IGiveTurn" type="dialog.on.i-give-turn"/>
  <post name="OtherGivesTurn" type="dialog.on.other-gives-turn"/>
  <post name="IAcceptTurn" type="dialog.on.i-accept-turn"/>
</module>
```

Tutorials -- Example of a PsySpec for a large system

```
<post name="OtherAcceptsTurn" type="dialog.on.other-accepts-turn"/>
<post name="IWantTurn" type="dialog.on.i-want-turn"/>
<post name="OtherWantsTurn" type="dialog.on.other-wants-turn"/>
<crank name="YTTM" language="python2" script="yttm.py" />
</module>

<module name="SpeechRecogniser">
  <description>
    Analyses input audio frames detecting speech events and utterances
  </description>
  <parameter name="AcmodFile" type="String" value="%NuanceVoconModels%/acmod6_6000_enu_gen_car_f16_v1_0_0.dat" />
  <parameter name="CTXFile" type="String" value="%NuanceVoconModels%/ctx_enu_messaging_v6_1_0_1_0.fcf" />
  <parameter name="ITNCTXFile" type="String" value="%NuanceVoconModels%/itn_enu_messaging_v1_0.s3c" />
  <parameter name="AudioInputType" type="String" value="Raw" />
  <parameter name="Verbose" type="Integer" value="1" />
  <trigger name="Start" type="Psychone.Ready" />
  <trigger name="Start2" type="input.audio.start" />
  <trigger name="AudioFrame" type="input.audio.frame" />
  <trigger name="Stop" type="input.audio.stop" />
  <post name="Started" type="xxx.input.speech.on" />
  <post name="Stop" type="xxx.input.speech.off" />
  <post name="Begin" type="input.speech.begin" />
  <post name="MaybeSpeech" type="input.speech.maybe" />
  <post name="NoSpeech" type="input.speech.none" />
  <post name="Recognition" type="input.speech.reco" />
  <post name="PassEnd" type="input.speech.passReport" />
  <post name="Result" type="input.speech.hypothesis" />
  <post name="Semantic" type="input.speech.info.semantic" />
  <post name="Timeout" type="input.speech.timeout" />
  <post name="BadSNR" type="input.speech.badsnr" />
  <post name="Overload" type="input.speech.overload" />
  <post name="TooQuiet" type="input.speech.tooquiet" />
  <post name="NoSignal" type="input.speech.signal.off" />
  <post name="PoorMic" type="input.speech.poormic" />
  <crank name="SpeechRecogniser" function="Nuance::NuanceRecogniser" />
</module>

<!-- ***** -->
<!-- Speech output -->
<!-- ***** -->

<module name="NuanceTTS">
  <description>
    Takes textual utterances and turns them into audio frames to be played back
  </description>
  <parameter name="BaseDir" type="String" value="%NuanceTTSengine%" />
  <parameter name="VoiceName" type="String" value="kate" />
  <parameter name="VoiceQuality" type="String" value="premium-high" />
  <parameter name="SampleFrequency" type="Integer" value="22000" />
  <parameter name="BufferSize" type="Integer" value="768" />
  <trigger name="GenerateSpeechAudio" type="output.speech.utterance" />
  <post name="AudioStart" type="output.speech.on" />
  <post name="AudioFrame" type="output.audio.frame" />
  <post name="AudioEnd" type="output.speech.off" />
  <post name="GenerationError" type="output.audio.error" />
  <crank name="NuanceTTS" function="Nuance::NuanceTTS" />
</module>

<module name="SpeakerOutput">
  <description>
    Plays audio frames to the default speaker
  </description>
  <parameter name="InputSampleRate" type="Integer" value="22000" />
  <parameter name="OutputSampleRate" type="Integer" value="48000" />
  <parameter name="BufferSize" type="Integer" value="1675" />
  <parameter name="DeviceName" type="String" value="%OutputAudioDevice%" />
  <trigger name="DefaultTrigger" type="Psychone.Ready" />
  <trigger name="AudioStart" type="output.speech.on" />
  <trigger name="AudioFrame" type="output.audio.frame" />
  <trigger name="AudioEnd" type="output.speech.off" />
  <trigger name="AudioFlush" type="cmd.output.audio.flush" />
  <trigger name="AudioPause" type="cmd.output.audio.pause" />
  <trigger name="AudioResume" type="cmd.output.audio.resume" />
  <post name="AudioOn" type="output.audio.on" />
  <post name="AudioOff" type="output.audio.off" />
  <post name="AudioStarted" type="output.audio.started" />
  <post name="AudioFinished" type="output.audio.ended" />
  <post name="AudioPaused" type="output.audio.paused" />
  <post name="AudioResumed" type="output.audio.resumed" />
  <post name="AudioFlushed" type="output.audio.flushed" />
  <crank name="RTAudioOut" function="Nuance::RTAudioOut" />
</module>
```



```

<module name="StopSpeakingDetector">
  <description>
    Detects when the output audio device has finished playing the TTS audio
  </description>
  <trigger name="AudioOutStopped" type="output.audio.ended"/>
  <post name="StoppedSpeaking" type="dialog.on.utterance.ended"/>
</module>

<module name="RobotSpeechMonitor">
  <parameter name="SpeakingTaskName" type="String" value="OutputSpeech" />
  <trigger name="TaskAccepted" type="Task.Accepted" />
  <trigger name="TaskUpdated" type="Task.Updated" />
  <trigger name="TaskCompleted" type="Task.Completed" />
  <trigger name="TaskCancelled" type="Task.Cancelled" />
  <trigger name="TaskTimeout" type="Task.Timeout" />
  <crank name="RobotSpeechMonitor" function="Perception::RobotSpeechMonitor" />
  <post name="RobotStartedSpeaking" type="Robot.Speaking.Started" />
  <post name="RobotFinishedSpeaking" type="Robot.Speaking.Finished" />
  <post name="SpeakerChange" type="input.speaker.change" />
</module>

<module name="RobotNavigationWithSpeech">
  <trigger name="NavigateToNamedPointSpeech" type="Robot.Task.NavigateSpeech" />
  <post name="NavigateToNamedPointSpeechStatus" type="Robot.Task.NavigateSpeech.Active" />
  <post name="NavigateToNamedPointSpeechFailed" type="Robot.Task.NavigateSpeech.Failed" />
  <post name="NavigateToNamedPointSpeechSuccess" type="Robot.Task.NavigateSpeech.Complete" />
  <post name="OutputSpeech" type="output.speech.utterance" />
  <crank name="RobotNavigationWithSpeech" function="Perception::RobotNavigationWithSpeech" />

  <post name="PerformTask" type="Task.Perform" />
  <post name="CancelTask" type="Task.Cancel" />
  <trigger name="TaskAccepted" type="Task.Accepted" />
  <trigger name="TaskUpdated" type="Task.Updated" />
  <trigger name="TaskCompleted" type="Task.Completed" />
  <trigger name="TaskTimeout" type="Task.Timeout" />
  <trigger name="TaskCancelled" type="Task.Cancelled" />
</module>

<module name="RobotPINCodeWithSpeech">
  <trigger name="EnterScreenPINSpeech" type="Robot.Task.EnterPINSpeech" />
  <post name="EnterScreenPINFailed" type="Robot.Task.EnterPINSpeech.Failed" />
  <post name="EnterScreenPINSuccess" type="Robot.Task.EnterPINSpeech.Success" />
  <crank name="RobotPINCodeWithSpeech" function="Perception::RobotPINCodeWithSpeech" />

  <post name="PerformTask" type="Task.Perform" />
  <post name="CancelTask" type="Task.Cancel" />
  <trigger name="TaskAccepted" type="Task.Accepted" />
  <trigger name="TaskUpdated" type="Task.Updated" />
  <trigger name="TaskCompleted" type="Task.Completed" />
  <trigger name="TaskTimeout" type="Task.Timeout" />
  <trigger name="TaskCancelled" type="Task.Cancelled" />
</module>

<!-- ***** -->
<!-- Control Panels -->
<!-- ***** -->

<module name="PanelController1">
  <query name="ControlPanel" source="Panel1" host="%MasterAddress%" port="%MasterPort%" />

  <trigger name="Reset" type="Panel1.Command.Reset" />
  <post name="ResetSuccess" type="Panel1.Command.Reset.Success" />
  <post name="ResetFailed" type="Panel1.Command.Reset.Failed" />

  <trigger name="OptionSelect" type="Panel1.Command.Select" />
  <post name="OptionSelectSuccess" type="Panel1.Command.Select.Success" />
  <post name="OptionSelectFailed" type="Panel1.Command.Select.Failed" />

  <trigger name="ReadScreen" type="Panel1.Command.ReadScreen" />
  <post name="ReadScreenSuccess" type="Panel1.Command.ReadScreen.Success" />
  <post name="ReadScreenFailed" type="Panel1.Command.ReadScreen.Failed" />

  <trigger name="BackNavigation" type="Panel1.Command.Back" />
  <post name="BackNavigationSuccess" type="Panel1.Command.Back.Success" />
  <post name="BackNavigationFailed" type="Panel1.Command.Back.Failed" />

  <trigger name="EnterPIN" type="Panel1.Command.EnterPIN" />
  <post name="EnterPINSuccess" type="Panel1.Command.EnterPIN.Success" />
  <post name="EnterPINFailed" type="Panel1.Command.EnterPIN.Failed" />

  <crank name="PanelController1" function="PsySystem::ControlPanelController" />
  <post name="ViewUpdated" type="Panel1.View.Update" />
  <post name="ErrorView" type="Panel1.View.Error" />

```



```

</module>

<module name="PanelController2">
  <query name="ControlPanel" source="Panel2" host="%MasterAddress%" port="%MasterPort%" />

  <trigger name="Reset" type="Panel2.Command.Reset" />
  <post name="ResetSuccess" type="Panel2.Command.Reset.Success" />
  <post name="ResetFailed" type="Panel2.Command.Reset.Failed" />

  <trigger name="OptionSelect" type="Panel2.Command.Select" />
  <post name="OptionSelectSuccess" type="Panel2.Command.Select.Success" />
  <post name="OptionSelectFailed" type="Panel2.Command.Select.Failed" />

  <trigger name="ReadScreen" type="Panel2.Command.ReadScreen" />
  <post name="ReadScreenSuccess" type="Panel2.Command.ReadScreen.Success" />
  <post name="ReadScreenFailed" type="Panel2.Command.ReadScreen.Failed" />

  <trigger name="BackNavigation" type="Panel2.Command.Back" />
  <post name="BackNavigationSuccess" type="Panel2.Command.Back.Success" />
  <post name="BackNavigationFailed" type="Panel2.Command.Back.Failed" />

  <trigger name="EnterPIN" type="Panel2.Command.EnterPIN" />
  <post name="EnterPINSuccess" type="Panel2.Command.EnterPIN.Success" />
  <post name="EnterPINFailed" type="Panel2.Command.EnterPIN.Failed" />

  <crank name="PanelController2" function="PsySystem::ControlPanelController" />
  <post name="ViewUpdated" type="Panel2.View.Update" />
  <post name="ErrorView" type="Panel2.View.Error" />
</module>

<!-- ***** -->
<!-- Role management -->
<!-- ***** -->

<module name="RoleNegotiator">
  <query name="Master" source="CCMMaster" />
  <parameter type="String" name="DefaultRole" value="Searcher" />
  <parameter type="String" name="PrimaryRole" value="Communicator" />
  <parameter type="String" name="SecondaryRole" value="Controller" />
  <trigger name="Start" type="Psyclone.Ready" />
  <trigger name="UpgradeRole" type="Human.Self.Appeared" />
  <trigger name="DowngradeRole" type="Human.Left" />
  <trigger name="RoleAssigned" type="Role.Assigned" />
  <trigger name="RoleLeft" type="Role.Left" />
  <trigger name="RoleGiven" type="Role.Given" />
  <crank name="RoleNegotiator" function="PsySystem::RoleNegotiator" />
  <post name="NowDefaultRole" type="Self.Role.Searcher" />
  <post name="NowPrimaryRole" type="Self.Role.Communicator" />
  <post name="NowSecondaryRole" type="Self.Role.Controller" />
</module>

<!-- ***** -->
<!-- Task management -->
<!-- ***** -->

<module name="TaskNegotiator">
  <query name="Master" source="CCMMaster" />
  <trigger name="PerformTask" type="Task.Perform" />
  <trigger name="CancelTask" type="Task.Cancel" />
  <trigger name="TaskAccepted" type="Task.Accepted" />
  <trigger name="TaskUpdated" type="Task.Updated" />
  <trigger name="TaskCompleted" type="Task.Completed" />
  <trigger name="TaskTimeout" type="Task.Timeout" />
  <trigger name="TaskCancelled" type="Task.Cancelled" />
  <crank name="TaskNegotiator" function="PsySystem::TaskNegotiator" />
</module>

<module name="TaskExecutor">
  <query name="Master" source="CCMMaster" />
  <trigger name="TaskAssigned" type="Task.Assigned" />
  <trigger name="TaskTimeout" type="Task.Timeout" />
  <trigger name="TaskCancelled" type="Task.Cancelled" />
  <crank name="TaskExecutor" function="PsySystem::TaskExecutor" />

  <post name="NavigateToNamedPoint" type="Robot.Request.Navigate.NamedPoint" />
  <trigger name="NavigateToNamedPointStatus" type="Robot.Status.Navigate.Active" />
  <trigger name="NavigateToNamedPointFailed" type="Robot.Status.Navigate.Failed" />
  <trigger name="NavigateToNamedPointSuccess" type="Robot.Status.Navigate.Complete" />

  <post name="ResetScreen" type="Panel.Command.Reset" />
  <trigger name="ResetScreenFailed" type="*.Command.Reset.Failed" />
  <trigger name="ResetScreenSuccess" type="*.Command.Reset.Success" />

```

Tutorials -- Example of a PsySpec for a large system

```

<post name="ReadScreen" type="Panel.Command.ReadScreen" />
<trigger name="ReadScreenSuccess" type="*.Command.ReadScreen.Success" />
<trigger name="ReadScreenFailed" type="*.Command.ReadScreen.Failed" />

<post name="NavigateBack" type="Panel.Command.Back" />
<trigger name="NavigateBackFailed" type="*.Command.Back.Failed" />
<trigger name="NavigateBackSuccess" type="*.Command.Back.Success" />

<post name="SelectScreenOption" type="Panel.Command.Select" />
<trigger name="SelectScreenOptionFailed" type="*.Command.Select.Failed" />
<trigger name="SelectScreenOptionSuccess" type="*.Command.Select.Success" />

<post name="EnterScreenPIN" type="Panel.Command.EnterPIN" />
<trigger name="EnterScreenPINFailed" type="*.Command.EnterPIN.Failed" />
<trigger name="EnterScreenPINSuccess" type="*.Command.EnterPIN.Success" />

<post name="NavigateToNamedPointSpeech" type="Robot.Task.NavigateSpeech" />
<trigger name="NavigateToNamedPointSpeechStatus" type="Robot.Task.NavigateSpeech.Active" />
<trigger name="NavigateToNamedPointSpeechFailed" type="Robot.Task.NavigateSpeech.Failed" />
<trigger name="NavigateToNamedPointSpeechSuccess" type="Robot.Task.NavigateSpeech.Complete" />

<post name="EnterScreenPINSpeech" type="Robot.Task.EnterPINSpeech" />
<trigger name="EnterScreenPINSpeechFailed" type="Robot.Task.EnterPINSpeech.Failed" />
<trigger name="EnterScreenPINSpeechSuccess" type="Robot.Task.EnterPINSpeech.Success" />

<post name="TestTask" type="Task.Test.Start" />
<trigger name="TestTaskFailed" type="Task.Test.Failed" />
<trigger name="TestTaskSuccess" type="Task.Test.Success" />
</module>

<module name="AutoPanelSelectController">
  <description>
    This module translates Control Panel tasks such as Panel.Command.Reset to go to
    the correct panel, i.e. either to Panel1.Command.Reset or Panel2.Command.Reset
    depending on the current state which is either ControlPanel1 or ControlPanel2
  </description>

  <parameter type="String" name="DefaultState" value="ControlPanel1" />
  <parameter type="String" name="SwitchKey" value="PointName" />

  <!-- Main control messages -->
  <trigger name="Ready" type="Psychone.Ready" />
  <trigger name="SwitchMessage" type="Robot.Status.Navigate.Complete" />
  <post name="ControlPanel1" type="Robot.Location.Panel1" />
  <post name="ControlPanel2" type="Robot.Location.Panel2" />
  <post name="NoState" type="Robot.Location.None" />

  <!-- Convert Reset message -->
  <trigger name="Reset" type="Panel.Command.Reset" />
  <post name="Reset_ControlPanel1" type="Panel1.Command.Reset" />
  <post name="Reset_ControlPanel2" type="Panel2.Command.Reset" />

  <!-- Convert OptionSelect message -->
  <trigger name="OptionSelect" type="Panel.Command.Select" />
  <post name="OptionSelect_ControlPanel1" type="Panel1.Command.Select" />
  <post name="OptionSelect_ControlPanel2" type="Panel2.Command.Select" />

  <!-- Convert ReadScreen message -->
  <trigger name="ReadScreen" type="Panel.Command.ReadScreen" />
  <post name="ReadScreen_ControlPanel1" type="Panel1.Command.ReadScreen" />
  <post name="ReadScreen_ControlPanel2" type="Panel2.Command.ReadScreen" />

  <!-- Convert BackNavigation message -->
  <trigger name="BackNavigation" type="Panel.Command.Back" />
  <post name="BackNavigation_ControlPanel1" type="Panel1.Command.Back" />
  <post name="BackNavigation_ControlPanel2" type="Panel2.Command.Back" />

  <!-- Convert EnterPIN message -->
  <trigger name="EnterPIN" type="Panel.Command.EnterPIN" />
  <post name="EnterPIN_ControlPanel1" type="Panel1.Command.EnterPIN" />
  <post name="EnterPIN_ControlPanel2" type="Panel2.Command.EnterPIN" />

  <crank name="AutoPanelSelectController" function="MessageToggler" />
</module>

<!-- A test module for the TaksExecutor -->
<module name="TestTask">
  <trigger name="TestTask" type="Task.Test.Start" after="2000" />
  <post name="TestTaskFailed" type="Task.Test.Failed" />
</module>

```



```
<!-- ***** -->
<!-- Dialog management -->
<!-- ***** -->

<module name="TDM">
  <description>
    Task Dialog Manager spec file. It tries to input new words each
    time they are available and tries to respond when YTTM gives turn
  </description>

  <parameter name="SystemID" type="Integer" value="%SystemID%" />

  <post name="RobotStartedSpeaking" type="Robot.Speaking.Started" />
  <post name="RobotFinishedSpeaking" type="Robot.Speaking.Finished" />

  <trigger name="Ready" type="Psychone.Ready"/>
  <trigger name="DialogOn" type="dialog.on"/>
  <trigger name="DialogOff" type="dialog.off"/>
  <trigger name="NewWords" type="input.speech.info.semantic" />

  <trigger name="Speak1" type="dialog.on.i-have-turn"/>
  <trigger name="Speak2" type="dialog.on.i-accept-turn"/>
  <trigger name="Speak3" type="dialog.on.other-gives-turn"/>

  <trigger name="InputInterrupt" type="input.speech.interrupt.definite"/>

  <post name="IWantTurn" type="dialog.on.i-want-turn"/>
  <post name="Talk" type="output.speech.utterance"/>
  <post name="MoveExecutor" type="move.executor.point" />
  <post name="DialogOff" type="dialog.off" />

  <!-- Modular testing functions, using TaskExecutor -->
  <trigger name="TaskCompleted" type="Task.Completed" />
  <post name="PerformTask" type="Task.Perform" />
  <post name="CancelTask" type="Task.Cancel" />
  <trigger name="TaskAccepted" type="Task.Accepted" />
  <trigger name="TaskUpdated" type="Task.Updated" />
  <trigger name="TaskTimeout" type="Task.Timeout" />
  <trigger name="TaskCancelled" type="Task.Cancelled" />

  <crank name="PsyCrank" language="python2" script="TDM_psychone.py" />
</module>

</psySpec>
```



ACTIVITY	45	ONE-SHOT	15
ADVANCED FILTERING	23	PARAMETERS	16
AFTER	21	PASSTHROUGH MODULES	20
AGENT-BASED SIMULATION	63	PORT	24
API DOCUMENTATION	58	PREREQUISITES	6
CATALOGS	34	PRIVATE DATA	49
CMSDK LIBRARY	58	PROCESS SEPARATION	55
CoCoMAPS	63	PSYPROBE	42
COMPONENT CRANKS	14	PSYPROBE PORT	26
CONTEXTS	27	PSYPROBE SUBSITE	51
CONTINUOUS COMPONENTS	15	PSYSPEC VARIABLES	18
CUSTOM CATALOGS	38	PSYSPEC XML FILE	11
CUSTOM CONFIGURATION DATA	18	PYTHON IMPORT LIBRARIES	31
CUSTOM TABS	48	PYTHON MODULES	30
CUSTOM VIEWS	47	REMOTE QUERY	57
DATA CATALOG	35	REMOTE REQUESTS	57
DATAFLOW	21	REPLAY CATALOG	36
DEBUG	24	REQUEST STORE CATALOG	37
DISTRIBUTED PSYCLONE SYSTEMS	54	ROBOTS	63
EXTERNAL SPACES	55	RUNNING PSYCLONE ON UNIX	8
FILE CATALOG	34	RUNNING PSYCLONE ON WINDOWS	8
FILTERS	22	SEPARATE PSYCLONE SYSTEMS	57
FROM	22	SERVICES	52
GRID DATA	62	SIGNALS	21
INCLUDING OTHER FILES	19	SIMULATION	63
INLINE PYTHON MODULES	30	SPACES	55
INTERACTIVE ROBOTS	63	STATISTICS	47
INTERFACES	52	SYSTEM PARAMETERS	24
INTERVAL	21	TAGGING	23
INTRODUCTION	5	THIRD-PARTY APPLICATIONS	62
LIBRARIES	14	TO	22
LICENSING	6	TUTORIALS	64
LINKING WITH THE CMSDK LIBRARY	59	TYPES	41
LOCAL PERSISTENT (PRIVATE) DATA	16	USE CASES	62
MAKE	10	USER CONTENTS	41
MAXAGE	22	VERBOSE	24
MESSAGE TYPES	12	VISUAL STUDIO 2015	9
MODULES	32	WHITEBOARDS	32
NODES	54		