Welcome to the Psyclone 2.0 README

Please note that this is a beta README only

Introduction

Psyclone is a general-purpose platform for deploying multiple processes with powerful message and stream communications. It employs a simple yet powerful publish-subscribe mechanism that allows you to quickly create and connect modules and make them interact.

Compiling Psyclone and the examples

On Windows you can use Visual Studio Express or Professional 2010, 2012 and 2013 to open the appropriate solution file in the root directory and compile the solution for either 32-bit or 64-bit for either Release or Debug.

On Linux you can use GCC-C++ and MAKE to compile the full solution for release ('make') or debug ('make debug'). Remember to do a 'make clear' when switching between release and debug.

This should result in the binary Psyclone executable (Psyclone.exe or PsycloneDebug.exe on Windows and Psyclone on Linux) to be created in the dir Bin/<arch> where <arch> is either win32, win64, linux32 or linux64.

In the same dir at least two DLLs/shared object will be, called Examples and PsySystem (with appropriate Debug, lib, .dll or .so). The Examples library contains all the Module Cranks from the Examples project (which you can extend or recreate) and the PsySystem library contains all the system components such as Whiteboards and Catalogs which you can copy to Examples and modify to create new types of system components.

Running Psyclone and the examples

Running the Psyclone executable with the command line 'psytest' will auto-generate a PsySpec with Modules, Whiteboards and Catalogs from the included Examples library and run the system on a single node. 'psytest=N' will spawn N nodes on the local computer and dynamically spread the test Modules across the system and run the tests.

You can create your own PsySpec configuration file and run Psyclone with this using the 'spec=<file>' parameter. To run Psyclone on a port different from port 10000 you can use 'port=<other port>'.

As long as the Examples library *(dll/so)* is located in the same dir as the Psyclone executable or in the current dir it will be found. The code for the examples are located in the*Examples* directory.

PsySpec

The *PsySpec* is a file that contains the setup specification for your system. Here is an example of two simple modules that ping/pong messages between them until they reach a certain number after which they request that the system is shutdown:

```
<psySpec>
     <library name="Examples" library="Examples" />
     <!-- auto path and name per OS (xx.dll or libxx.so) -->
     <module name="Ping">
          <trigger name="Ready" type="Psyclone.Ready" />
          <trigger name="Ball" type="ball.1" />
          <crank name="Ping" function="Examples::Ping" />
          <post name="Ball" type="ball.2" />
     </module>
     <module name="Pong">
          <trigger name="Ball" type="ball.2" />
          <crank name="Pong" function="Examples::Ping" />
          <post name="Ball" type="ball.1" />
          <post name="Done" type="Psyclone.Shutdown" />
     </module>
</psySpec>
```

Here you can see two modules, Ping and Pong, both loading their crank from the same function Ping inside the library Examples. Psyclone will search for the Examples (*dll* or *so*) in the same dir as the Psyclone executable and in the current dir, but a full path from the current dir can be entered in the library parameter.

Module Ping has two triggers. The first one is the *Psyclone.Ready* message type which is always posted once Psyclone has started up completely. We use this to get the system up and running. The second is the trigger for *ball.1* which comes back from Module Pong, after we post *ball.2* to trigger it.
This simple test system will measure the average time for a message to travel between the modules and on a relatively fast quad-core laptop we have seen times down to around 19us per message on Linux 64-bit, 23us on Windows 7 64-bit and 30us on Windows 7 32-bit.

Built-In Examples

This beta release comes with three examples.

The Ping/Pong example above which demonstrates simple messaging, subscriptions and shutting down the system. The PsySpec for this is located in *Examples/pingpong.x*. The full *Examples* header file looks like this:

```
#if !defined(_EXAMPLES_H_)
#define _EXAMPLES_H_

#include "PsyAPI.h"

namespace cmlabs {

extern "C" {
     DllExport int8 Simple(PsyAPI* api);
     DllExport int8 Ping(PsyAPI* api);
     DllExport int8 Shutdown(PsyAPI* api);
     DllExport int8 Print(PsyAPI* api);
     DllExport int8 Signal(PsyAPI* api);
}

} // namespace cmlabs

#endif //_EXAMPLES_H_
```

It includes the *PsyAPI.h* header file and declares extern the crank functions inside. A crank function returns an unsigned 8-bit integer, normally 0, when it completes. In the future we may add significance to the return value, but for now this is just ignored.

The PsyAPI object is the C++ object that the crank function uses to communicate with the rest of the system, such as waiting for new triggers, posting output messages, working with parameters, etc. This is very closely related to the old Messenger object in *Psyclone 1.x*. See the next section for more details.

The *Examples* source file includes the *Examples.h* file, declares the cmlabs namespace and then implements each of the crank functions.
A *crank* is the function that is run when a module is triggered. The simplest *crank* function is the default crank that posts all the module's output messages and then exits:

```
int8 Simple(PsyAPI* api) {
     DataMessage* inMsg;
     const char* triggerName;
     if (api->shouldContinue()) {
          if ( inMsg = api->waitForNewMessage(0, triggerName))
              api->postOutputMessage();
     }
     return 0;
}
```

When waiting for the input message you have to provide a *const char* pointer that will point to the name of the trigger given in the PsySpec. The time to wait is given in milliseconds. If you don't provide a post name for the posting you will post all your posts at the same time. If you do provide a name, all posts with this name will be posted. You can provide a message too, if you don't a default empty one will be used.

The crank doesn't have to exit and can continue running until it no longer should. The Ping crank does this:

```
int8 Ping(PsyAPI* api) {

     char* myName = new char[256];
     api->getModuleName(myName, 256);

     api->logPrint(0,"[%s] started running...", myName);

     DataMessage* inMsg, *outMsg;
     int64 counter = 0, start, end;
     const char* triggerName;
     bool print = false;

     while (api->shouldContinue()) {
          if ( inMsg = api->waitForNewMessage(100, triggerName)) {
              outMsg = new DataMessage();
              if (_stricmp(triggerName, "Ready") == 0) {
                   api->logPrint(0,"[%s] got start message type '%s'...", myName,
                        api->typeToText(inMsg->getType()).c_str());
                   start = GetTimeNow();
                   print = true;
              }
              else if (_stricmp(triggerName, "Ball") == 0) {
                   if (!counter++)
                        start = GetTimeNow();
                   else if (counter % 10000 == 0) {
                        end = GetTimeNow();
                        if (print)
                            api->logPrint(0,"[%s] got msg %llu, time per msg: %.3fus...",
                            myName, counter, (double)(end-start)/(2*10000.0));
                        start = GetTimeNow();
                   }
                   if (counter % 100000 == 0) {
                        api->postOutputMessage("Done", outMsg);
                        outMsg = NULL;
                   }
              }
              else {
                   api->logPrint(0,"[%s] got other message type '%s'...",
                        myName, api->typeToText(inMsg->getType()).c_str());
                   delete(outMsg);
                   outMsg = NULL;
              }
              if (outMsg)
                   api->postOutputMessage("Ball", outMsg);
          }
     }

     delete [] myName;
     return 0;
}
```

We start by reading our own name and use *LogPrint* to print to the main console. The format is similar to printf and the log values are for debug level filtering.

Here we actually look at the trigger name and post to a named post with a message that we create. Messages are now *DataMessage* objects and they have much the same capabilities as the old Messages, but are far more efficient.

The second example works with Signals:

```
<psySpec>
     <library name="Examples" library="Examples" />
     <!-- auto path and name per OS (xx.dll or libxx.so) -->
     <module name="Ping">
          <trigger name="Start" type="Psyclone.Ready" />
          <crank name="signal" function="Examples::Signal" />
          <signal name="Output" type="My.Signal.1" />
          <signal name="Input" type="My.Signal.2" />
          <post name="Done" type="Psyclone.Shutdown" />
     </module>
     <module name="Pong">
          <trigger name="Ready" type="Psyclone.Ready" />
          <crank name="signal" function="Examples::Signal" />
          <signal name="Input" type="My.Signal.1" />
          <signal name="Output" type="My.Signal.2" />
          <post name="Done" type="Psyclone.Shutdown" />
     </module>
</psySpec>
```

We still use the *Psyclone.Ready* message to wake up both modules, but now we communicate exclusively via global signals. If you emit a signal this is broadcast to every space in the whole system. Any module can then wait for the arrival of a signal and they will all be triggered at the same time with the data. No subscription is needed, you merely refer to signals by the name you give it in the PsySpec for the individual module. Context changes are also signals so you can use this to wait for contexts to change.

The third example demonstrates the use of contexts:

```
<psySpec>
     <library name="Examples" library="Examples" />
     <!-- auto path and name per OS (xx.dll or libxx.so) -->
     <module name="Startup">
          <trigger name="Ready" type="Psyclone.Ready" />
          <post name="done" context="My.Context.1" />
     </module>
     <module name="Print1">
          <context name="My.Context.1">
              <trigger name="Ball1" type="ball.1" interval="100" />
              <crank name="print" function="Examples::Print" />
              <post name="done" context="My.Context.2" />
          </context>
     </module>
     <module name="Print2">
          <context name="My.Context.2">
              <trigger name="Ball2" type="ball.2" interval="100" />
              <crank name="print" function="Examples::Print" />
              <post name="done" context="My.Context.1" />
          </context>
     </module>
</psySpec>
```

Here we see that instead of posting types we post contexts that effectively change the system contexts. Contexts still work the same in that we can have multiple root contexts (in this example Psyclone and My) and for each root context we can have one and only one active subcontext (either My.Context.1 or My.Context.2). In this example we also see the use of automatic timed triggers that will trigger the module automatically every 100ms.

PsyAPI

The functions currently implemented in the PsyAPI are:

```
class PsyAPI {
public:

    bool shouldContinue();
    bool getModuleName(char* name, uint32 maxSize);
    std::string typeToText(PsyType type);

    // ***************** Messaging *****************
    DataMessage* waitForNewMessage(uint32 ms, const char* &triggerName);
    int32 postOutputMessage(const char* postName = NULL, DataMessage* msg = NULL);

    // ***************** Signals *****************
    bool emitSignal(const char* name, DataMessage* msg = NULL);
    DataMessage* waitForSignal(const char* name, uint32 timeout, uint64 lastReceivedTime = 0);

    // ***************** Own Data *****************
    // Load, save or delete own persistent data
    bool setPrivateData(const char* name, const char* data, uint32 size);
    uint32 getPrivateDataSize(uint32 cid, const char* name);
    bool getPrivateDataCopy(uint32 cid, const char* name, char* data, uint32 maxSize);
    bool deletePrivateData(uint32 cid, const char* name);

    // ***************** Own Parameters *****************
    // Get, set, reset, create or destroy own parameters
    bool createParameter(uint32 cid, const char* name, const char* val, const char* defaultValue = NULL);
    bool createParameter(uint32 cid, const char* name, const char* val, uint32 count, uint32 defaultIndex);
    bool createParameter(uint32 cid, const char* name, std::vector<std::string> values, const char* defaultValue = NU
LL);
    bool createParameter(uint32 cid, const char* name, int64* val, uint32 count, uint32 defaultIndex);
    bool createParameter(uint32 cid, const char* name, std::vector<std::string> values, int64 defaultValue = 0);
    bool createParameter(uint32 cid, const char* name, float64* val, uint32 count, uint32 defaultIndex);
    bool createParameter(uint32 cid, const char* name, std::vector<std::string> values, float64 defaultValue = 0);
    bool createParameter(uint32 cid, const char* name, int64 val, int64 min = 0, int64 max = 0, int64 interval = 0);
    bool createParameter(uint32 cid, const char* name, float64 val, float64 min = 0, float64 max = 0, float64 interva
l = 0);
    bool hasParameter(uint32 cid, const char* name);
    bool deleteParameter(uint32 cid, const char* name);
    uint8 getParameterDataType(uint32 cid, const char* name);
    uint32 getParameterValueSize(uint32 cid, const char* name);
    bool getParameter(uint32 cid, const char* name, char* val, uint32 maxSize);
    bool getParameter(uint32 cid, const char* name, int64& val);
    bool getParameter(uint32 cid, const char* name, float64& val);
    bool setParameter(uint32 cid, const char* name, const char* val);
    bool setParameter(uint32 cid, const char* name, int64 val);
    bool setParameter(uint32 cid, const char* name, float64 val);
    bool resetParameter(uint32 cid, const char* name);
    bool tweakParameter(uint32 cid, const char* name, int32 tweak);

    uint8 retrieve(std::list<DataMessage*> &result, const char* name, uint32 maxcount = 0, uint32 maxage = 0, uint32
timeout = 5000);
    uint8 retrieveTimeParam(std::list<DataMessage*> &result, const char* name, uint64 startTime, uint64 endTime = 0,
uint32 maxcount = 0, uint32 maxage = 0, uint32 timeout = 5000);
    uint8 retrieveStringParam(std::list<DataMessage*> &result, const char* name, const char* startString, const char*
endString = NULL, uint32 maxcount = 0, uint32 maxage = 0, uint32 timeout = 5000);
    uint8 retrieveIntegerParam(std::list<DataMessage*> &result, const char* name, int64 startInteger, int64
endInteger = INT64_NOVALUE, uint32 maxcount = 0, uint32 maxage = 0, uint32 timeout = 5000);
    uint8 retrieveFloatParam(std::list<DataMessage*> &result, const char* name, float64 startFloat, float64 endFloat
= FLOAT64_NOVALUE, uint32 maxcount = 0, uint32 maxage = 0, uint32 timeout = 5000);
    uint8 retrieve(std::list<DataMessage*> &result, RetrieveSpec* spec, uint32 timeout = 5000);

    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);
    bool queryReply(uint32 id, uint8 status, char* data, uint32 size, uint32 count);
    bool queryReply(uint32 id, uint8 status, DataMessage* msg = NULL);

    bool logPrint(uint8 level, const char *formatstring, ... );
 };
```

Examples of module parameters in the PsySpec are:

```
<!-- standard integer value -->
<parameter name="MaxCount" type="Integer" value="0"/>
<!-- using PsySpec variables for global replace -->
<parameter name="Interval" type="Integer" value="%frameinterval%"/>
<!-- using PsySpec variables for global replace -->
<parameter name="BitmapFiles" type="String" value="%videofiles%" />
<!-- 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 integer in range, in steps of 2 -->
<parameter name="MaxCount" type="Integer" value="[0:110:2]"/>
<!-- any float in range -->
<parameter name="Interval" type="Float" value="[-2.5:2.5]"/>
<!-- 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']"/>
```

Whiteboards

Whiteboards are specified in the PsySpec

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

and any module can post to them by adding to=”WB1” in their post specification. However, Whiteboards can also subscribe to messages themselves

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

which means that modules don’t have to specify the to field and any message with the matching type will be routed to the Whiteboard automatically.

Any message entering a Whiteboard must have a TTL (time to live) higher than 0 and will be removed from the Whiteboard automatically when the TTL expires. If the message has a TTL of 0 it will not be indexed by the Whiteboard.

The default Whiteboard indexes its messages by time and a module can retrieve messages from it by specifying the retrieve in the PsySpec

```
<retrieve name="r1" source="WB1" maxcount="10" />
```

and then use the retrieve API to carry out a named retrieve and add additional parameters

```
uint8 retrieve(
         std::list<DataMessage*> &result,
         const char* name,
         uint32 maxcount = 0,
         uint32 maxage = 0,
         uint32 timeout = 5000);

uint8 retrieveTimeParam(
         std::list<DataMessage*> &result,
         const char* name,
         uint64 startTime,
         uint64 endTime = 0,
         uint32 maxcount = 0,
         uint32 maxage = 0,
         uint32 timeout = 5000);

uint8 retrieveStringParam(
         std::list<DataMessage*> &result,
         const char* name,
         const char* startString,
         const char* endString = NULL,
         uint32 maxcount = 0,
         uint32 maxage = 0,
         uint32 timeout = 5000);

uint8 retrieveIntegerParam(
         std::list<DataMessage*> &result,
         const char* name,
         int64 startInteger,
         int64 endInteger = INT64_NOVALUE,
         uint32 maxcount = 0,
         uint32 maxage = 0,
         uint32 timeout = 5000);
```

```
uint8 retrieveFloatParam(
         std::list<DataMessage*> &result,
         const char* name,
         float64 startFloat,
         float64 endFloat = FLOAT64_NOVALUE,
         uint32 maxcount = 0,
         uint32 maxage = 0,
         uint32 timeout = 5000);

uint8 retrieve(
         std::list<DataMessage*> &result,
         RetrieveSpec* spec,
         uint32 timeout = 5000);
```

The last call shows that one could create a RetrieveSpec manually in code only and use it to retrieve messages without having to specify it in the PsySpec.

As hinted above, Whiteboards can also index their messages by data other than time.

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

This whiteboard will add an index for an integer value added as custom data to a DataMessage called ‘count’. Custom data can be either time, string, integer or float and a single Whiteboard can have many simultaneous indexers. A module can then retrieve based on a custom index

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

and can specify additional filter parameters using the API above such as a start and an end value, maxcount and maxage.

Catalogs

Two standard Catalogs have been implemented in Psyclone 2.0 so far. One is the FileCatalog and the other is the DataCatalog. The former is used to offer modules access to files without each module having to be told the file locations and the latter is used as a persistent data store for <key> = <value> associations for string keys and binary data values.

The FileCatalog is specified like this

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

which means that any module can read and if allowed write files from the root directory with an optional custom subdir in the Module PsySpec entry

```
<query name="WB1" source="MyFiles" subdir="test" ext="txt" binary="yes" />
```

and a query API call

```
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);
```

An example read could be

```
status = api->queryCatalog(&data, datasize, "MyFiles", "test", "read");
if (status == QUERY_SUCCESS)
   // data now contains the result of the read query
else if (status == QUERY_TIMEOUT)
   // deal with the timeout
else
   // deal with the error
delete (data);
```

An example write could be

```
status = api->queryCatalog(&data, datasize, "MyFiles", "test", "write", filedata, size);
```

where the data to be written is provided at the end and no data is expected back, just the status.

Similarly for the DataCatalog which is specified in the PsySpec like this

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

a module can write data to it

```
status = api->queryCatalog(&data, datasize, "MyData", "test", "write", mydata, size);
```

and read data back

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

The content of the ‘database’ will be written to the file ./mydata.dat every 2 seconds and when the Catalog starts up it will load the content of the ‘database’ from the most recently written file, making the data storage persistent between system restarts. The writing of the database file is done by writing to a temporary file and atomically renaming this to overwrite the old file and the

existence of a temporary file will be checked on startup. Also, the file content is CRC checked to ensure data validity.

Whiteboards and Catalogs are physically implemented in the separate PsySystem shared library.

External Spaces

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" ... />
```

If the Space doesn't already exist it will be automatically started up as a separate system process and it is automatically destroyed when the Psyclone Node shuts down. A Node can have an unlimited number of Spaces attached and if one Space crashes because a component inside performs an illegal action only the components in this Space is affected and can be restarted (not currently done automatically).

Non-guaranteed messaging

Non-guaranteed messaging is implemented so that the output messages of any component can be dropped if message transfer queues are building up. This is done both at the local Node processing queues and at the Node network transmission queues which means that these messages might be delivered successfully locally, but will not be prioritised when delivered to subscribers on other Nodes. Any output message from any component can be marked as Transient by adding transient="yes" to the post specification in the PsySpec.