Module Messaging Extended Library, version 2.0

Source code: https://github.com/ADSWNJ/ModuleMessagingExt and
https://github.com/ADSWNJ/MMExt2

Intro

For a long time, we (i.e. Szymon and Andrew) have advocated for the ability for Orbiter modules (e.g. vessels, MFDs) to be able to communicate with each other in a way that makes the simulation experience more seamless. For example, we feel it is natural to allow TransX to talk to LaunchMFD to synchronize the launch parameters, or for Glideslope 2 to be able to look at BaseSync's determination of the best orbit to do a de-orbit burn? The Module Messaging series of library utilities makes this possible.

What's in this Library?

This library installs 5 features for Orbiter:

1. The core Module Messaging data interchange module (MMExt2.dll)
2. C++ Include files for developers to interface to Module Messaging if they choose to.
3. An updated Module Messaging Ext 1.1 interface (ModuleMessagingExt.dll) for existing module compatibility, connecting to the new core.
4. A new Module Messaging Ext MFD display (MMExtMFD) for you to see Module Messaging variables and activity (optional, not mandatory to run this).
5. This documentation, mainly for developers.

History

This library was originally released in 2014, as Module Messaging. This implemented the core capability to send and receive Booleans, Integers, Doubles, VECTOR3, MATRIX3, and MATRIX4 between independent orbiter modules (usually MFDs, but for example vessels could also use this). This release was enhanced in 2016 as Module Messaging Extended 1.0 and then 1.1, with added logic to support data interchange per vessel in a simulation (e.g. for different burn data for a fleet of vessels), and the ability to support the sending and receiving of arbitrary data structures by reference (e.g. to expose a larger set of parameters without having to constantly call dozens of put or get functions just to synchronize data). We were worried about causing unnecessary crashes to desktops by having modules referencing different versions of structures or different compiler implementations, or for a receiving module to be able to overwrite a sending module's data, so we added tons of safeguards for this. And so, we had a good library, and it has been utilized now for some of the core addons in the simulation, including TransX, LaunchMFD, BaseSyncMFD, Glideslope 2 and RV Orientation.

In 2018, we released the second version of Module Messaging Extended. This was driven by three goals: (1) stop Error 126 errors and the dozens of Orbiter Forum threads asking why a particular MFD cannot

load (due to a missing dependent DLL), (2) provide a simple and an advanced code experience to interact with Module Messaging according to what the developer prefers, and (3) provide backwards compatibility for any existing ModuleMessagingExt 1.1 module clients. We have achieved all these goals in this release.

As a Developer, How Do I Use This Utility?

There are two 'facades' or interfaces, to Module Messaging Extended – the Basic Interface and Advanced Interface. The Basic Interface supports Put, Get, and Delete functions for int, bool, double, VECTOR3, MATRIX3, MATRIX4, and std::string, for your current vessel. These are the core functions for basic interaction with Module Messaging Extended, and if this is sufficient, the Basic Interface is the simplest and cleanest solution for you.

The Advanced Interface implements six additional capabilities:

1. Getting and putting variables for vessels in the scenario other than your focus vessel (e.g. to synchronize a rendezvous with a target vessel).
2. Safe data interchange for two types of data structures, for more complex data transfer needs.
3. A generic find function to scan for variables or module:variable pairs from vessels you do not know about (e.g. to allow a burn MFD to pick up burn vectors and timing from a new MFD without additional coding).
4. An activity log trace, to see data interchange activity.
5. A get version call to inspect the version and compile date of the MMExt2.dll if present.
6. A method to update the sending module name, for compatibility with the old MMExt v1.

This interface is used by Module Messaging Extended v1.1 for compatibility with old MFDs linking directly to it, as well as for the new inspection MFD, MMExt2MFD.

You can use both interfaces interchangeably in your code (i.e. the data you can put or get in the Basic Interface can also be accessed from the Advanced Interface).

Coding to the Basic Interface

1. From the place where you need to get data from MMExt2 (e.g. in your class header file):

   ```cpp
   #include "MMExt2_Basic.hpp"
   ```

   Your default search path for Orbiter development (e.g. from the Orbiter plugin resource property sheets included in the Orbiter distribution), should include Orbiter\Orbitersdk\include, which is where this MMExt2_Basic.hpp will be found.

2. In your class header, declare the interface, e.g. mm, with your choice of module name. (Replace the red text with your preference).

```
MMExt2::Basic mm;
```

3. In your class constructor, you need to explicitly initialize the interface on the class definition – e.g.:

```
MyClass::MyClass() : mm("MyName") { … }
```

… where:
"MyName" is your module's name (i.e. replace with your choice)

The reason for this syntax is because the constructor for the MMExt2 Basic Interface requires your module name as a parameter, so it's a non-default constructor, and this is the C++ syntax to declare such an object. If you are declaring the interface in a block of code (i.e. not in a class), then the module name is put on the declaration as follows:

```
MMExt2::Basic mm("MyName");
```

4. In your code – add Put, Delete, and Get calls like this:

```
bool ret;
ret = mm.Put("var", val);
ret = mm.Delete("var");
ret = mm.Get("other_module", "var", &val);
```

… where:
"var" is your specific variable name (e.g. "Target", "Burn_Time", etc.)
&val is the address for the returned value for the variable name (as an int, bool, double, VECTOR3, MATRIX3, MATRIX4, or std::string), so long as the function returns true.
"other_module" is the name of the other MFD sending you the data,
ret is true if the function was successful

5. Usage notes for Put and Delete:
   a. Pick descriptive names for your variables, to make it clear what the data is.
   b. Do not duplicate names for different variable types (only one variable type per name is permitted).
   c. All data is passed by value – i.e. when you put the data, MMExt2 takes a copy, so you can destroy the original source if you want. If you want pass-by-reference semantics, then look at the structure-passing in the Advanced Interface.
   d. Delete is only called from the module that Puts the data. I.e. if you are just getting the data, you never call Delete, as you are not the owner of that data.
   e. The Delete function knows the data type, so it just needs the variable name.
   f. Delete will not remove any data in your code – it just removes the copy in MMExt2, such that all further calls to Get will return false.

6. Usage notes for Get:
   a. Look at the documentation or the source code for the other module to determine what data variables and data types it is sending, or use MMExtMFD to see the data inside the Orbiter simulation.
   b. When the Get completes, you will have a private copy of the data returned into your code. You need to repeat the Get call to get refreshed data – e.g. every simulation step, each fixed time interval, or manually on request from the user. Be sensitive to the amount of data you wish to collect if you are doing it on every simulation step – e.g. consider refreshing each 5 seconds instead. If you want a reference to a live variable in the other module, then have a look at the structure-passing capabilities in the advanced interface.

Coding to the Advanced Interface

This is an extension of the Basic Interface, so we will only point out the things that are different.

1. Include, declare, and initialize a MMExt2::Advanced interface with these commands in the appropriate places (replacing the red text with your choices):

```
#include "MMExt2_Advanced.hpp"
MMExt2::Basic mma;
MyClass::MyClass() : mma("MyName") { … }
```

2. Put, Delete, and Get calls:

```
ret = mma.Put("var", val, vessel);
ret = mma.Delete("var", vessel);
ret = mma.Get("other_module", "var", &val, vessel);
```

… where:
vessel is a *VESSEL pointer to the vessel definition. It is an optional parameter, and defaults to the current focus vessel if you leave it out.

3. Usage notes for Put, Delete, Get:
   a. The vessel parameter allows you to access data for other vessels in the scenario. For example – you may be sharing burn or targeting data across a fleet of vessels, or accessing information from a target vessel for a rendezvous. The vessel must exist in the scenario when you reference it, so be aware of the creation and destruction events on vessels if you want to use this functionality.

   b. In addition to the standard data types in the Basic Interface, the Advanced Interface can also pass pointers to data structures. As this is a complex topic, we present it below in a separate section.

4. GetVersion is used to inspect the version of the MMExt2.dll core, if installed. Note with all these calls, if the MMExt2.dll is not installed, you just get a false return on each call, but nothing breaks and you get no Error 126 on module load.

```
std::string ver;
ret = mma.GetVersion(&ver);
```

… where:

ret is true if the MMExt2.dll is found, else false, and if ret is true, then ver will be populated with version and compile date information for the MMExt2.dll.

5. Find is used to do wildcard searches for data in the MMExt2 core. For example, you can pick up all burn data for all vessels from all MFDs using this function. You supply a module name (or *), a variable name (or *), a VESSEL * pointer (or NULL for wildcard), and an index number initially set to 0. Find returns true, increments the index, and returns all the return strings, until there is no more data, when it will return false. Sample code:

```
const char findMod = "*"; // usually a literal on Find
const char findVar = "*"; // usually a literal on Find
int findIx; // Updated on each find call
const VESSEL *findVes = NULL // NULL is wildcard, else vessel*
const bool findOwn = false; // include our data in the find?
char   retTyp; //  return type: b, i, d, v, 3, 4, x, y
string retMod; // return module
string retVar; // return variable name
string retVes; // return vessel name (note: not VESSEL *)

findIx = 0; // Initialize the find
while (mma.Find(findMod, findVar, findIx,
                &retTyp, &retMod, &retVar, &retVes,
                findOurOwnData, findVes)) {
 // … do something with the return data
}
```

… notes:

findMod … is a char* literal for the module to find, or "*" for wildcard.

findVar … is a char* literal for the variable to find, or "*" for wildcard.

findIx … is an integer. Set to 0 for each find scan, and find will update as it needs. Note that the index will jump by more than one, when skipping over elements.

retTyp … is a single char: b for bool, i for int, d for double, v for VECTOR3, 3 for MATRIX3, 4 for MATRIX4, x for MMStruct* structures, and y for ModuleMessagingExtBase* structures. It's a char so you can do a simple switch block to parse the types.

retMod, retVar, retVes … are strings for the found module, variable, vessel. Why is the vessel a string? Because it may not be valid any more, if the vessel was deleted. You can tell immediately if it is still valid by doing an oapiGetVesselByName() call.

findOwn … is true if you want the search to find data from this module

findVes … is NULL for wildcard, else a valid VESSEL* pointer to search for.

6. GetLog is used to scan the activity log in the MMExt2 core. It's used in the MMExt2MFD, though you are welcome to call it in your own code if you choose. The default log order is a 'tail' – i.e. reverse log. Sample code:

```
int getIx;
char * rFunc;
bool * rSucc;
string *rCli, rMod, rVar, rVes, rAct;

getIx = 0;
while (mma.GetLog(getIx++, &rFunc, &rSucc, &rCli, &rMod,
                          &rVar, &rVes)) {
// … do something with log data
}
```

… notes:

getIx is manually incremented by the caller. (Why? Because you may want to implement a specific log sequence, so you have full flexibility to search for a specific element)

rFunc is a char indicating the action: P for Put, G for Get, D for Delete, L for this log function, V for GetVersion, F for Find.

rSucc is a bool indicating success or failure for the call.

rCli, rMod, rVar, rVes … are strings representing the requesting client, the module, variable and vessel name.

The activity log records one activity line for each unique request, rather than logging every request. For example – if you call get("M1","M2",&var) 10,000 times, it will record one log entry if the call ever succeeded, and one log entry if the call ever failed.

7. UpdMod is provided as a convenience for the ModuleMessagingExt 1.1 interface, to allow the module name to be changed if the sender ever changed on the calling module. There's no need to call this from your code – i.e. just make sure you initialize the ModuleMessaging interface with your module name, and leave it constant.

The Advanced Interface Structure Handling

Until now, we have worked with a set of simple Orbiter data types. We also handle std::string, with some careful work in the implementation to avoid compiler-specific dependencies with std::string binary formats. There are time, however, where you want to pass a data structure by reference, such that the receiving module can inspect the live data directly without having both modules to

have to explicitly update the data with an extended set of puts and gets every refresh.

When you pass data by reference in a structure, you get all the advantages of direct access to the internal data in the other module, but you are also exposed to many compiler-specific implementation details that can cause problems for data interchange (e.g. there is no guarantee of binary compatibility for standard library structures across compiler versions, so you may well cause crashes to desktop if you try to implement such things.) The structure handling code in MMExt2 tries as hard as possible to shield you from problems, but the key message is not to expose anything that uses the standard template library (e.g. no strings, vectors, maps). We want to avoid is a crash to desktop caused by modules accidentally referencing bad data structures, or making false assumptions about application binary implementations across versions of the compiler.

We implement five safety-checks to try to avoid these problems:

1. We insist that the class or struct you wish to pass is derived from one of two root structures we implement in Module Messaging. The old root structure was called EnjoLib::ModuleMessagingExtBase, and the new root structure is MMExt2::MMStruct. When you declare a structure for data interchange, you derive it from either of these structures, so we can implement the remaining checks. (Note: in C++, you can assign a child class pointer to the parent's class pointer, so this allows us to manipulate classes across the MMExt2 core).

2. We insist on a version number when you instantiate your class, and we check for that version number in the get module at runtime. This makes sure that the getting code is defining that same version of the export/import data structure. (The code putting the external structure should give you a header file defining the implementation.)

3. We implement run-time size checks for the structure, in case either the compiler implements the size differently, or the coder of the module modifies the structure without changing the version number. If the size is incorrect, we do not allow the pointer to be passed to the receiving module.

4. We insist on the receiving pointer being a const pointer. This prevents the receiving side from accidentally or intentionally trying to change the sending side's data structure. If you want to do such things, establish a two-way interchange and a message protocol of your own between the two modules, and have each module expose their sending side as separate structures.

5. The last check is to recommend that the structure packing is explicitly defined via #pragma pack directives. This removes yet another way for the various compiler versions to trip you up by putting miscellaneous whitespace in your structure for byte-alignment reasons.

OK – with all those caveats, here's the code you need to publish (put) an MMStruct-derived structure from your code :

1. Make a header file describing your external-facing structure – e.g. `MyModule_ExportStruct.hpp`, using this template:

```cpp
// Module Messaging Export Structure for MY MODULE
#pragma once
#include "MMExt2_Advanced.hpp"
#pragma pack(push)
#pragma pack(8)
#define MYSTRUCT_VER 1
struct ExportStruct : public MMExt2::MMStruct {
    ExportStruct() : MMExt2::MMStruct(MYSTRUCT_VER,
                                 sizeof(MyStruct)) {};

    // … add your elements here …
    // … e.g. bool valid;

};
#pragma pack(pop)
```

2. Make sure that this export structure derives from public MMExt2::MMStruct, and has no standard template library components inside.

3. In your class header, declare the structure in a place that will not go out of scope whilst the vessel is active. Sample code:

```cpp
struct ExportStruct mm_export;
```

4. In your code, publish the structure via your Module Messaging Extended Advanced Interface::

```cpp
ret = mm.PutMMStruct("var", &mm_export, vessel);
```

5. Note – once you have published a structure, there is no Delete function, as you may have other modules directly accessing your private memory location via your published pointer. Consider using a Boolean flag in the structure to indicate if the data is valid.

On the other side, to access (get) the structure, do this:

1. Include the header from the other module (e.g. they should put it in an accessible place suck as Orbitersdk\include):

```cpp
#include "MyModule_ExportStruct.hpp"
```

2. In your class header, define a constant pointer to the export structure (note: it must be constant, to guarantee that the client does not write to the structure):

```
const ExportStruct *pExportStruct;
```

3. In your code, establish a connection to the structure via your Module Messaging Extended Advanced Interface:

```
ret = mm.GetMMStruct("mod", "var", &pExportStruct, vessel);
```

4. Assuming you have a successful return code, you may now access the live data in the export structure without any further Module Messaging commands – e.g.

```
if (pExportStruct->valid) // do something…
```

Compatibility with previous ModuleMessagingExt modules

MMExt2 is backwards-compatible with ModuleMessagingExt 1.1 modules. By installing ModuleMessagingExt 2.0, any older client will access MMExt2.dll indirectly via an updated ModuleMessagingExt.dll. Clients of the older ModuleMessagingExt 1.1 will only be able to access the following data types: int, bool, double, VECTOR3, MATRIX3, MATRIX4, and EnjoLib::ModuleMessagingExtBase derived structures. The new MMExt2::MMStruct-derived structures and the std::string data type are only available via the new MMExt2 interface.

A note from ADSWNJ

I want to recognize the partnership I have had for many years now with Szymon "Enjo" Ender, on this module and on many other MFDs in the Orbiter simulation. This library is the result of many emails and many hours of coding and testing between us.

- Andrew Stokes, December 2017