Module Messaging Extended Library, version 2.0a

All source code in: https://github.com/ADSWNJ/

Intro

For a long time, the authors of this library 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 Extended Library 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 to use the features.
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 many safeguards to prevent 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 this 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 an End User, How Do I Use This Utility?

As an end user, you do not have the ability to change the data being sent or received from an MFD, but you can at least monitor them to see what is going on. To do this, enable ModuleMessagingExtMFD in the Modules tab of your Orbiter Launchpad, and select this MFD. It has two modes (selected by the MOD button): Data and Activity. In Data mode, you can see what variables have been exported by each of the MFDs. In Activity mode, you can see unique activities between the various modules. You can page through the lists with PRV and NXT, and you can reset the activity log with RST.

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, OBJHANDLE, and std::string, for your current vessel. It also has a function to validate object handle types for you. 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 function 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. #### Include

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

```
#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. Declaration

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

```
MMExt2::Basic modMsg;
```

3. Initialization

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

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

… where:
MyName is your module's name. 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 put the module name on the declaration as follows:

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

4. Accessing variables

In your code – add simple Put(), Get(), Delete() functions like this:

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

… 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, OBJHANDLE,** 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 MMExt2 is installed, and the function put, got, r deleted some data.

5. Determining Object Type

Use this to find the type of your OBJHANDLE data (e.g. pointing to vessels, bodies, or bases), and to re-validate the object still exists. In your code – add an ObjType() function like this:

```
int obj_type;
bool ret;
OBJHANDLE hObj;
ret = modMsg.Get("other_module", "var", &hObj);
obj_type = modMsg.ObjType(val);
```

… obj_type will return one of these values (literal values defined in OrbiterAPI.h):

`OBJTP_INVALID` (= 0) if the object is invalid. Stop using this OBJHANDLE.
`OBJTP_STAR` (= 3)
`OBJTP_PLANET` (= 4)
`OBJTP_VESSEL` (= 10)
`OBJTP_SURFBASE` (= 20)

If Module Messaging Extended is not installed, this function will return -1.

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, Declaration, Initialization

Include, declare, and initialize a MMExt2::Advanced interface with this code in the appropriate places (replacing the red text with your choices):

```
#include "MMExt2_Advanced.hpp"
class MyClass {
…
private:
  MMExt2::Advanced modMsgAdv;
…
}
MyClass::MyClass() : modMsgAdv("MyName") { … }
```

2. Accessing data for other vessels

Put(), Get(), and Delete() functions in the Advanced Interface all take an optional vessel parameter, to fetch for a vessel different to the active focus vessel:

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

… where:

`obj_vessel` is a `OBJHANDLE` to a vessel. It is an optional parameter, and defaults to the active focus vessel's object handle if you leave it out.

3. More data types

Put(), Get(), and Delete() additionally support specific struct pointers. The struct pointer handling is a complex topic, so we present it below in a dedicated section.

4. Checking the installed version of MMExt2 (if any)

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

```
std::string ver;
ret = modMsgAdv.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. Wildcard find functionality

Use the Find() function to do wildcard searches for data in the MMExt2 core. For example, you can find all burn data for all vessels from all MFDs using this function (assuming we curate well-known names for developers to use). You supply a module name (or *), a variable name (or *), an OBJECTHANDLE to a vessel (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 function
const OBJHANDLE findOVH = 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, o, x, y
string retMod; // return module
string retVar; // return variable name
OBJHANDLE retOVH; // return vessel object handle

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

6.

… where:
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 **int**. 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 **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() function.
`retOVH` … is an `OBJHANDLE` for the vessel. The MMExt code will automatically purge any invalid `OBJHANDLEs` for you, you are assured that this is a valid handle immediately on return from Find.
`findOwn` … is true if you want the search to find data from this module
`findOVH` … is `NULL` for wildcard, else a valid `OBJHANDLE` pointer to a vessel to search for.

7. Reviewing the Activity Log

The GetLog() function 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 want. 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 (modMsgAdv.GetLog(&rFunc, &rCli, &rMod, &rVar, &rVes,
                        &rSucc, &getIx, skipSelf)) {
// … do something with log data
}
```

… where:
`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 function.
`rCli`, `rMod`, `rVar`, `rVes` … are strings representing the requesting client, the module, variable and vessel name. Why is the vessel a string? Because it may still be in the activity log, despite now being an invalid object.
`skipSelf` … if true tells the GetLog to skip our own activity.

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 function ever succeeded, and one log entry if the function ever failed.

8. Clearing the Activity Log

The RstLog() function is used to reset activity log in the MMExt2 core, so you remove old

entries if you wish. There are no arguments.

```
ret = modMsgAdv.RstLog();
```

9. Updating the Module Name

The UpdMod() function is provided to the ModuleMessagingExt 1.1 interface, to allow the module name to be changed if the sender ever changes on the calling module. There's no need to call this from your native MMExt2 code – i.e. just make sure you initialize the ModuleMessaging interface with your module name, and then leave it alone.

Developer Usage Notes, Tips and Tricks

1. Why should I use the Basic Interface?

If you can get everything you need from the Basic Interface, then we prefer you to use it. Why? It's minimalistic and clean, resulting in clear and simple code in your application. It is also guaranteed to be the most consistent of interfaces (i.e. future code enhancement will mostly be adding features just to the Advanced Interface). You can use both interfaces interchangeably to access all the Basic Interface functions – i.e. a sender can use the basic Interface, and the client can receive via the Advanced Interface, or you can use both interfaces in the same code if needed.

2. What is the best way to Put or Get vessels, celestial bodies, or bases?

Use OBJHANDLE handles to pass Orbiter objects. On the Get side, either repeat the Get() function for the OBJHANDLE on each simulation pre-step, or call ObjType(hObj) to re-validate the object exists. ModuleMessaging guarantees to return valid OBJHANDLEs on every call, but if the simulation has advanced and your object is deleted by another module, then the additional call to ObjType() protects you from unforeseen crashes to desktop.

Internally, ModuleMessaging calls the Orbiter API oapiGetObjectType() function to validate OBJHANDLEs, with careful wrapping and compiler flags to prevent an Access Violation that can be thrown by the Orbiter API.

Whilst you could use object names (e.g. passing a string), this could be problematic if the object is renamed. Alternatively, you could pass objects by index, but this causes issues if an earlier-created object is deleted (as the index will now point to another object, or it will be past the end of the object index list). For these reasons, we prefer OBJHANDLEs, and we recommend you to re-validate the OBJHANDLE on each call as discussed above, for the safest code practice.

3. What should I call my variables?

Make them descriptive so other developers can easily understand what they are receiving. E.g. `TargetObjectHandle` or `BurnTime`. Don't try to use the same variable name for multiple types of variables, as it will simply overwrite the last Put. Make your names generic (e.g.

`TargetObjectHandle`) rather than specific (e.g. `LaunchMFDTargetObjectHandle`), so that developers can implement wildcard Find if preferred, to automatically pick up your object - e.g. with `modMsgAdv.Find("*", "TargetObjectIndex",&ix, &typ, &mod, &var, &ves);`

If this Module Messaging system becomes more widespread across other modules, then we would be happy to consider more formal naming standards for variables for common data.

4. How is data passed (e.g. by reference, by value)?

All data is passed by value. Each Put() command transfers a copy of your data into the MMExt2.dll master store, and each Get() will receive a copy of that data. This means that you can use modMsg.Put() on a locally scoped variable, without worrying about leaving a reference to an invalid heap location when your function returns.

The one exception to this is for structure handling in the Advanced Interface, where the structure is indeed passed by reference, and therefore you need to be very careful not to destroy the source location after the structure has been put using modMsgAdv.mmPutStruct() or modMsgAdv.mmPutBase().

5. How do I know what the variable is called, or what it represents?

Ideally, the developer putting the data into ModuleMsssaging has provided you documentation, or at least provided you their source code so you can see what the variable represents. If not, then you can use MMExt2MFD to look at the variables directly in a running Orbiter simulation, and then decide how you want to handle it. Here is some sample code from MMExt2MFD to show how easy this is:

```
int ix = 0;
char typ;
string mod, var, vesName, msg;
VESSEL *ves;
list<string> mmListVesTyp, mmListModVar;
while (mm.Find("*", "*", &ix, &typ, &mod, &var, &ves, false, NULL)) {
  msg = string() + ves->GetName() + " (" + typ + ")";
  mmListVesTyp.push_back(msg);
  mmListModVar.push_back(mod + ":" + var);
}
```

6. How often should I scan for updated data?

If you are responding to user-input (e.g. key press to synchronize data), then of course just access the data once. If you are monitoring status on another module, then consider putting in a timer to trigger only after a certain about of time (e.g. 2 seconds). Here is some sample code from MMExt2MFD:

```
void ModuleMessagingExtMFD_GCore::corePreStep(double simT,
                                              double simDT,
```

```
                                                    double mjd) {
  if (simT - coreSimT > 2.0) {
    coreSimT = simT;
    // ... refresh data here
  }
}
```

If you have two very-closely coordinated MFDs that you want to synchronize a lot of data in real time, then executing dozens of Put() and Get() functions on every simulation pre-step can get quite tedious. Consider using the structure-passing capability in the Advanced Interface for this use-case. The downside of this is that you are now very code-specific between the modules, and any change to the data layout or usage in one MFD will require changes to the other MFD, and your end-users will need to remember to download both MFD updates, or things will not work as expected. You have been warned!

The Advanced Interface Structure Handling

Until now, we have worked with a set of simple Orbiter data types. There are times, 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:

```
// 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)
```

   Note – the constructor is needed in this format to initialize the version and size data in the MMStruct, which is used for checking on the GetMMStruct side.

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:

```
struct ExportStruct mm_export;
```

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

```
ret = modMsg.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):

```
#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 = modMsg.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, and MATRIX4. 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, January 2018