Writing Modules For MUBBS

11/19/91

By N. Hawthorn for V.5

Here I try to cover some important things you need to know when creating a module for MUBBS.

I need to make a section to go over every global and routine, I will try to cover some here.

Check on the Support BBS or AOL and see if there are any updates to this file.

WRITING MODULES FOR MUBBS IS EASY !!

I tried to make writing a module as easy as possible. There will be lots of examples out there to look at that will help you out. The EXAMPLES are your BEST SOURCE on how to write a module.

Some of this is confusing now, but as you write and test your module this will all come in handy. Just start writing your "application" for MUBBS, and it will all fall together.

Remember this: YOU ALWAYS HAVE HELP AVAILABLE VIA E-MAIL FROM ANY MUBBS PROGRAMMER ! Programmers are always willing to help !

I hate writing code that has already been written. Please release your source code so that others can use it in their modules and can BUILD ON TOP OF IT ! Don't worry about how perfect it is, just put it out. If everyone does this we will be able to start writing modules "above" the basic stuff and do some really insanely great stuff with MUBBS !

Are you waiting for a "SOLID" MUBBS ? Forget it ! There will be changes always, and some modules will cause

problems with other modules YOU BET! So write that module NOW ! I tried to make it so that any changes or

additions will not affect most modules. We have a standard "struct" that has lots of EXTRAS in it so that it won't

change much! But there will be times that you have to recompile your code. I will always supply the latest

"Module Headers" files to you via AOL or my BBS, and most of the time you will only have to replace the current

files, and recompile. Most of the additions will be for things I couldn't possibly imagine at this time, like a global

to tell whether or not a UFO passed over your BBS's location last night :)

I really think I covered things pretty good, and there are "extra" variables and data space that can be used so

that changes don't affect YOUR module. So don't worry too much about this.

I have commented the code in the examples very well, and I hope that you can figure it out without me writing a

whole book on the subject. I spent many hours figuring out how to get all the basics working, looking at machine

language code, and crashing over and over. Now that the basics are in, the rest is down hill! You now have a

BASE to stand on. I tried to make programming a module as easy as possible. MOST EVERYTHING IS

HANDLED FOR YOU ! You just simply call routines that have already been written, and access BBS globals

using "G->variable". It should make your life much easier. I want to make it so you can write GREAT

MODULES, and not spend your time messing with the BBS code.

This version is pretty solid, and we have tested several new modules on it.

SET UP & COMPILING:

The examples are the best source for you to learn how to use the MUBBS routines, and how to make a module.

USE THE EXAMPLES TO FIGURE OUT WHAT TO DO ! There's no way I can put everything here.

They were written using Think C 4.0, this allows 4.0 and 5.0 to use the examples.

You need to put the file "MUBBS module.h" into your Think C folder so that the compiler can find it.

In 5.0 you need to set up the project type: FIle type "rsrc", creator "RSED" and set it as a code resource. You

also need to turn off "use prototypes" in the "Options" menu. Why Think didn't make a "converter" program I

don't know. The names are in the source code, a module is always a type of "MOD1" for now (you can have up

to ID#999).

MAKE SURE YOU HAVE "USE MACHEADERS" & "MACTRAPS" OFF !

You can compile these examples yourself, and mess around with the code. If you change the name of the modules, you might have to change the source code too.

You can "merge" (and "smart link") your module to the "modules" resource file, or make them a seperate

resource file. The "modules" file is opened at the startup of MUBBS and MUST remain in the same folder as

MUBBS.

Your resources have to be NAMED, because they are called by name. You will notice that they are of type

'MOD1' and "maincaller" is ID 128. You have to change the ID number otherwise it WILL write over one with the

same ID number when it merges it.

If you want you can chose *NOT* to merge and use resedit to move the resource over.

Other than that, everything else is easy. Just call the routines and make the BBS do what you want.

I haven't built a multi segment resource yet, but it should work OK.

GLOBALS:

Globals, you can use them as normal. They are stored in your DATA space within your code resource.

When using Globals, you must use the [u] to access your globals if you want them to be different for each node

because they can be accessed by other nodes. Globals that you do not want or care if another node reads them

or modifies them can remain normal. See the "MUBBS Module.h" file to get an idea of how to do this.

It's IMPORTANT to understand that EACH USER can access your SAME CODE !

This means that GLOBALS are GLOBALS for ANY USER using your code.

An example is if you have a string as a global with "hello" in it. If you make it so that a user on node 1 can

modify that global to "hoho", if the user on node 2 reads the global, it will say "hoho" NOT "hello" (give you any

ideas for a CHATTING between nodes ?).

If you used string[u], each node would have its OWN INDEPENDENT string, and each node would get a

different string and the other lines would leave your node's data alone.

You MUST therefore use either [u], OR define your variables inside of your function (they are then pushed on

the stack). YOU DON'T HAVE TO USE [u] IF YOU DEFINE VARIABLES WITHIN A FUNCTION (a "sub routine",

"procedure", etc...).

You define your global like this "string [maxport][1000]". "maxport" is defined for you.

Thinking about this is not EASY, but it's part of the overall programming of a module in a multi user environment like MUBBS. You have lots of freedoms to do what you want, there's just a few rules you can't just ignore without having multi line problems.

STACK SPACE:

When you define a variable WITHIN your function/routine, the space for that variable is made on the STACK.

So if you make "string[1000]", you have taken up 1K of stack space until you return from that routine.

MUBBS gives each user 20K or so of stack space each standard. MUBBS allows that to be adjustable, but 20K is a good amount.

MUBBS minimum configuration will always be a Mac Plus with 1 meg (at least for the next few years). So you need to be concerned with saving precious stack space.

Globals use memory space within your module. The problem with using the stack is that your module and others can be CALLED FROM OTHER MODULES. The stack is used each time, and gets smaller with each call.

Try not to use much stack space wherever possible, but don't get ridiculous.

A good example of this is the Menu Module. It reloads it's strings each time it's called with the new menu items so the globals are effectively "reused" for each new menu. But this really only takes up about 1K. Read the modules calling modules section below.

Globals are NOT pushed on the stack. They are things you define outside of a "function" or "routine". They take up space in your "DATA & JUMP" area of your resource so they only take up memory there.

Variables you put INSIDE of a "function" or "routine" ARE pushed on the stack and take up stack space. They are made for each node that calls the routine, thus they are "local" to that user.

Remember your routine is called over and over (the same actual code) for each user that calls them.

LONG TIME:

If you are doing something that takes allot of time (like a sort) you must call "otheruser(FALSE)". This switches to another user and gives him time.

otheruser(TRUE) means SWITCH NOW! even if it's not time to switch to another user. See, MUBBS doesn't just switch all the time, it gives each user some time. If you don't need any time (like sitting in a loop waiting for the user to type something) then call it "TRUE". Otherwise call it "FALSE".

If you are printing to the user, you are switching. When you "send" or "in" or "out" the system automatically switches. Don't call this all the time, you will waste CPU time calling it. Just call it when you are stuck doing something.

YOU DON'T HAVE TO CALL IT IF YOU ARE SENDING OR USING in() ! It's done for you. You don't want to slow down the system by calling it all the time when you don't have to.

It does all the system tasks for you too.

ERRORS:

REPORT ALL ERRORS !

You should always tell the sysop what's going on. The "G->debug" level is good to use for displaying technical info, but sysops can turn it off so that they don't get sick of it all.

MODULES CALLING MODULES:

What happens when your module is called by another module?

You are passed everything just like any module is, and you are called the same. You even get the name of the module that called you ! Other than that, you may or may not need to take action depending on what your module does.

An example of this is when you are called by a MENU module. The menu module is a module and is calling YOU !

Menu modules call menu modules till they get to some sort of function, like "Files". Then "Files" would call "Xmodem" or "Zmodem" to transfer a file, passing it a filename and a possibly a mode.

If you know how to call "Xmodem", your module could transfer files too !

You can write your module as SEVERAL modules, and call each one when it is needed. This would allow others to call your modules in their modules (it seems endless doesn't it ?).

What happens when your module is called AGAIN by another module, by the same node ?

You could have your module call itself, I don't know why. Also, your module could get called by another module that YOUR module called !

It's kinda like figuring out Space-Time and the speed of light !

Anyway, its not a good idea for this to happen, so try not to let this happen. Figure it this way, if your code

doesn't use any globals it will probably be OK. If it does, your globals may get changed.

I don't worry about this because one rule is to not do this. I'm sure someone will, but when it comes down to it, it will mess up and be seen on a multi-line system and be reported.

If this happens, your module stays in memory, it doesn't get "reloaded" with a new set of code, or take up any more memory. It is effectively "reused" on that call.

There may be applications for this, but I can't think of one right now.

PASSING VALUES TO A MODULE:

If your module needs values (like a filename or value other than the standard "mode" or something other than the "main globals" have supplied), pass them as a struct.

You MUST tell other programmers what your struct is, and what the values are.

To make things simple, if you're not into structs and don't need to pass a value or receive a value for your module (most don't need this), then just pass "0L" (zero long) as the "P" value when calling a module. When receiving a call, just ignore the "P" value. See the example modules and the files "your globals.h" included with them.

If you need to pass values via a struct just make a "user struct" as in the examples and pass a pointer as the "P" value to the module you are calling. Again, see the examples.

The reason the "P" is a pointer to a "user struct" is that you can then pass ALL KINDS of information to a module (as long as you know what the module's "user struct" looks like). You can also CHANGE this information and the calling module can read your changes !

It's really not that hard to use, the examples show the use in a small struct.

As a rule, if you use a struct CHECK TO SEE IF THE POINTER IS "0L" (zero). If it is 0L, exit or handle the call but don't use the struct ! Also when calling a module with a struct, you call it in mode 3.

Most calls like a initialize call or a general call will pass a 0L. You must handle this properly so that other modules can call yours without problems when you use "user structs". In other words, always handle mode 2, even if it's to send out a error condition.

In the example, choosing "Module" calls the module WITH a "user struct" and passes some values. When the called module gets the struct, it modifies one of the values by setting it to 100. Then when you return to the original module, the value has been changed. If you call it with 0L it reacts by NOT using the struct. See the example files.

HEADER FILES:

"MUBBS Module.h" is mostly the struct that contains all the globals you can read/modify within the BBS. It gets passed to all modules.

"MUBBS misc.c" is a file containing routines you can call to do things. There are some routines you shouldn't call unless you know EXACTLY what you are doing. I tried to mark them as such.
Look at the "Module routine list.txt" file to see the descriptions on these routines.

You should become VERY FAMILIAR with these files. They wont change too much, just additions mostly.
You need to use both of these files to make a module work.

ANSI-MUBBS:

Use "ANSI-MUBBS" in your project. It's a special version of "ANSI-A4" made for MUBBS. In the original version

of ANSI-A4 there were files included like "printf", "fwrite" and "scanf" that are not needed in a MUBBS module

(you are supplied with call backs to the main module for this).

"ANSI-MUBBS" will save you precious code space. If you need a routine that is not supplied by MUBBS Main,

include the file supplied by Think C in your project. You should look at this code and make sure that it does what

you want. You may have to modify it to call the routines that are in MUBBS Main.

"ANSI-MUBBS" is a "project" and can be looked at as such if you want to see how it was made and what was included.

Don't just include "MacHeaders" in your project. Check the "Options" selection under the "edit" menu to make

sure Think C isn't including it for you automatically. If you need a particular header, include it yourself. You

should look at it and make sure it's doing what you want it to.

"MacTypes.h" is already included for you, it should cover most of the types of variables you will need.

You don't need to use "MacTraps" in your module. If you do include it you will take up 9K of code space. You

may be able to find a way to call ONLY the trap routines you need somehow, so it will take up less code space

to do so. I find out what the code is for calling a TRAP, and just include that only.

Do this, if you need to use MacTraps, ask yourself "WHY" first. MUBBS supplies you with most calls you need to perform BBS stuff.

PRINTING and SENDING:

"print" sends to the sysops local screen. "send" sends to the user.

Example: print("C> FILE ERROR, cannot open %s\n",file);

"send" and "print" use the standard "printf" type of stuff.

When you use send, the "]" character means "send a CRLF", don't use "\n" in send !

Example: send("hahaha]");

DON'T PRINT TO THE MAC SCREEN UNLESS IT'S IMPORTANT ! The screen is real slow, and the whole system gets bogged down.

OPEN FILES:

Keep those files CLOSED ! There's only so many files that can be opened at a time. Besides it's not good to leave files open.

Open the file, get what you want, and then close it !

SHORTCUTS:

When looking at my example code (the ones I personally release) don't think that I'm checking too much, or

"that line's too long" because you BET it's there for a REASON ! I like writing tight code, but sometimes you

have to check for things like CARRIER LOSS, or people that TIMEOUT and things like this.

Please follow the things I do and your routines wont get into trouble when someone uses your code in a MULTI line system.

ALWAYS CHECK FOR "G->online[u]" !! You should check it BEFORE your module does anything. If you're not

online, it's kinda stupid to "send" something. You should check it after a "sendfile" or after you call another

module too.

TESTING:

I HIGHLY RECOMMEND USING A TERMINAL and your Mac's screen to test your code. You MUST be able to log in under at least TWO nodes in order to test your code in a "multi line" configuration.

I find stuff all the time that RUNS FINE under one line, but get two "users" on and it writes out all kinds of neat outer space stuff !

When I connect my APPLE II (yes I still have it) as a terminal, it leaves the carrier detect input OPEN (that pin is not connected) and the Mac thinks it has a carrier detect so I don't get logged out. You need to know what condition the carrier detect is in when using a terminal. Later there may be a way to disable carrier detect.

When you use "me(CR)" to log in from the port, MUBBS disables carrier detect till you log out.
If you select "Ports to Ring" from the "sysop" pull down menu, the code jumps to wait for ring.

Turn your terminal's "auto CR LF" feature off. You will catch all kinds of stuff like when you forgot to send LF after a CR !
Set the debug level to 99 so that you will get ALL levels of error reporting from ALL modules.

You need a way to log into your BBS from ELSEWHERE. The monitor screen is just that A MONITOR SCREEN. It's made so that you can locally log in because the Mac can do it, but you really need to see what your user's are seeing.

Old Terminals are about $10 each. No one wants them anymore, I see people at swap meets using them as seats and most of them would rather GIVE you one then have to LUG it home !

You will find many problems when you access it this way, one's you may never have expected (like globals writing over globals).

The easy way to test is to start MUBBS, goto sysop menu select "local logon" say "Y" (modems to wait for ring)

and do whatever. If you're on the port via a terminal, just type "ME (CR)" at the modems MAX baudrate setting.

You may have to try a few times. Read more in the "MUBBS Instructions" about this.

The problem with testing a code resource is that you can't run Think C's debugger in "run" mode. You should write most of your code in a stand alone application so that you can do a "run" and see what it does. When you get it mostly working, then try it as a module.

You can always install your module directly to the "modules" resource and save time.

It's a real good idea to operate the BBS and your compiler on a seperate hard drive if possible. This will limit the dammage if it decides to write over your disk. I have never had this happen yet, and I still operate on the same disk. I always look over my code twice before running it, especially if I am accessing files with it (open, close). I ALWAYS quit out of Think C BEFORE running MUBBS, and I press reset if it crashes. I almost ALWAYS

"restart" the system after running MUBBS just to make sure everything is OK before I go to the project again.

Every night I back up my whole project to a floppy just incase. You can do what ever you feel comfortable with

(and back up a lot !)

TEST EVERYTHING ! Remove files, rename them or whatever so that you get your own error messages.

MAKE SURE YOU COVER ALL ERROR CONDITIONS ! Hang up on your module in different places. Type in

false commands.

Some people will write modules and say "it works fine", till a user does something stupid, then it crashes !

TEST, TEST, and RE-TEST !!!

Call a few friends and have them call in and test out your program. Make sure they have a pencil and a piece of paper and WRITE down problems they find. If you don't INSIST on this, you won't get much good feedback.

Keep a small pad of paper and pencil ready to make a "check list" of problems. When you test your program, you can test it all (if it doesn't crash) and just jot down the problems to fix later. I can guarantee you that when you quit the program you won't remember more than one or two errors that occurred without writing them down !

RESTART YOUR COMPUTER IF MUBBS CRASHES !

Never ASSUME that people have the same SETUP or computer that you do !

Look at the examples, you may see something you didn't before.

LOCKED MODULE:

You need to lock your own module (code is supplied in the example).

Your module should remain locked while you are in it. After you leave your module, your globals are not valid anymore, so don't count on them.

If you want globals to remain permanent throughout the entire time MUBBS is up, when the BBS gets the programmer's name, lock it then. Then unlock it when you get a bye bye call. LOOK AT THE MODEM MODULE SOURCE TO SEE HOW THIS IS DONE! Most of the time you don't need to do this. There's almost always a

work around (like a file) to keep this from happening.

MUBBS Main module is locked at all times. It does not move around.

QUITTING:

You should ALWAYS quit out of MUBBS. If it crashes back to multifinder when testing, you should hit RESET

because serial buffers are still open and other things may still be open. Always LOG OUT before quitting from

the menu.

You should keep a original of the MUBBS I distribute incase you mess it up somehow. IF YOUR MODULE

CRASHES, RESTART YOUR COMPUTER USING THE RESET BUTTON ! This is important since under

multifinder your project file may get trashed by a bad memory write or something. The RESET BUTTON does

not shut down any running applications, it just restarts from scratch, so your project file is not closed and saved

to disk. I always save all files, compile and then quit out of Think C before starting MUBBS. Always back up

everything you have !! and save, save, save.... DEVELOPING ANY CODE IS ALWAYS DANGEROUS TO

YOUR DATA !

UPDATES:

When MUBBS is updated, you will get a new set of files for "MUBBS Module.h" and "MUBBS MIsc.c". They are found in the " Module std routines" folder. Check into the support BBS or AOL once and a while.

I won't have a section on AOL, but I will run one on the BBS and if a "net echo" starts up I will try to get on that too.

Any addition of variables will be made to the end of the list WHEN POSSIBLE. This will allow module's that already exist to still function with MUBBS. HOWEVER, THERE IS NO GUARANTEE THAT A OLD MODULE WILL FUNCTION WITH A NEW VERSION OF MUBBS.

Hopefully there won't be any major changes in the "main" MUBBS module after about 6 months of it's release. It should be pretty solid if everyone gives feedback during this period.

Again, if changes are needed they will be made in a way as to not make all modules already written useless.

If there is a major change, you will most likely just have to replace the "header" files with the new one's and recompile your code and then release it.

Each module is called and the version of MUBBS is checked against it. You should set it to the value of the

MUBBS version you are developing for. This prevents people from installing modules that just crash. If you can

"back" check your module against older versions of MUBBS, you could lower this value.

AUTHORS:

You MUST support your own module !

The idea of MUBBS is that it's easier for lots of people to support small modules than it is for one person to support a whole BBS.

I'm not talking about quitting your job to support your code. You just need to be available via E-Mail or U.S.

"Snail" Mail to answer questions. Your support shouldn't take more than a hour a week MAX ! If it ever gets to

be more than that, you need to spend some time writing a better instruction file to go with your module.

Most modules for MUBBS will be free, and authors will include their source code so that others can use their

code to develop better modules, and YOU can use their module on YOUR BBS ! Make sure you include a

disclaimer and any notices on how your code can be used. Include something telling sysops that they CAN

charge for access to their BBS. Most Sysops want to charge for access and if you don't specifically say they

can, they may not use your module.

If you are building a LARGE GAME, or a LARGE BUSINESS DATABASE SEARCHER or something like that, it

would be OK to charge a "shareware" fee or something like that. If you do release a LARGE module, it wouldn't

be a good idea to release source since it is your money. Since you are "selling" your module, you are

responsible for making your code work properly and be "hacker" proof.

If you're module isn't over 32K, it's really just a "hack". You shouldn't charge for simple "hacks".

Don't send out new updates to your modules every week. You should test your module as much as you can,

and leave it out there a while. People get real frustrated having to "re setup" their system all the time.

YOUR NAME AND FAME:

Place your name in the switch in the main() routine, and it will appear after every log out!

Please keep your name the same for all modules you have ever developed. If your name shows up three or more times people won't think you're so hot but more of a bother !

NO ADVERTISING HERE ! I don't care if your module is shareware or not, just the programmer's name please !

NO COMPANY NAMES !

You have 20 characters MAX to put your name in.

THE DETAILS:

modes are as follows;

0= Init call, return the programmers name and do any init stuff if needed.
1= Bye bye call, if you returned 99 in the init calls, you get called when the program quits.
  You should do a "unlock" to release your module's memory space.
2= Normal call, no pointer is passed to a struct.
3= Normal call, pointer is passed to a user struct, YOU HAVE DATA WAITING !
98= Version check, you check the version you want against MUBBS.
99= Reserved.

Other modes are not defined, you can use 10 and above for your own stuff.

What are the ProcPtr's ?

If you want to call "in" you call "G->in()" because the serial I/O routines are pointed to by the global. This makes

access the serial routines MODULAR and also fast.

Serial open, close, in, out and flush are all called this way. You will probably only call "in" and "out" most of the

time.

The event loop and idle loop are provided, and you place pointers to your routines in these pointers if you want

to be called then. There isn't much need for this, but I provided it anyway. Don't use the idle and event loops

unless you really need to. Talk to me before you do! You have to write REAL fast code here, because it will slow

down the system ALOT! If you use them you need to check to make sure a pointer isn't being used. They are

called from 1 to 5, if 2 isn't being used and 3 is, 3 IS NOT called. They must be in sequence and remain

installed all the time MUBBS is running (you have to be called at init time).

Event loop supplies a standard Mac "event record" pointer to you when you are called. You are called after

MUBBS main determines that the call isn't for MUBBS.

The idle loop is when "switch" is called (actually when "handle event" is called).

Most of the globals are commented in the "MUBBS Module.h" file.

"u" starts at 0. "localuser" is the "u" number that the local user would be if "local[u]" is true.
"noprint" stops all "print()" to the Mac screen, for when you are local logged on.
"cont[u]" indicates that the user has selected continuous output (no pause).
"nocheck[u]" is used in the serial routines, TRUE means don't check for cntl S or C.
"monitor[u]" if this is TRUE, any users output is shown to the Mac's screen.
"nottransfer[u]" when FALSE means to allow control characters to pass (all 8 bits).
"userfilepos[u]" this is so that it's quick to look up this users info.
"CR[u]" use this to send out a CR LF string, this is so that maybe someone wants just CR.
"userlocation[u]" is where the user is now, if you are on as a sysop on another port
from remote you could read this and see where a user is in the system.

routines;

"G->out(x)" wants a char as x to send out to the user.
"G->in()" waits for a character and returns TRUE if one is ready. It's in "input[u]".

"showline()" Don't call this, its for the Mac Monitor screen update.

"loguser("here")" This updates the "userlocation" var and also logs to a file if the
log mode is set on.

see "Module Routine List.txt" and other source code for more stuff.

READING:

I hope you read this WHOLE FILE. If you didn't you will have problems for sure.

UPDATES:

Updates are via the support BBS. All my contact info is in the MUBBS instruction manual.

CHECKLIST:

Here's a short checklist to use before "shipping" your module (as much as I can think of right now).

[ ] I close all my files properly.
[ ] I checked my pointers (if any) against the "example" files (MAKE SURE YOU ARE
POINTING TO THE CORRECT PLACE!)
[ ] I checked ALL possible selections and/or data entries the user could make.
[ ] Test with terminal AND local log on at the same time.
[ ] Checked on a friends computer (a different model, minimum mac plus).
[ ] Took out files that my module needs to see if it reports errors properly
or creates them properly when they are missing.
[ ] Connected a modem and HUNG UP on my module in several places.
[ ] Put it online as my personal BBS and allowed some people to call and use it.
[ ] Spent FOUR hours MORE than I planned (went the extra mile) writing instructions.
[ ] I don't assume that someone has the same computer or accessories or set up as I do.
[ ] My "hacker" friend tried to crash my module.
[ ] My source is Think 4.x compatible so most people can use it.