Compiling GalaxC Programs

This document describes how to compile GalaxC programs in the XXICC environment. This is a preliminary document as some aspects are changing, particularly compiling multi-file programs. The author has mostly used XXICC to compile and run the XXICC Object Editor (XOE), as well as numerous small test programs. As such, it is optimized for those tasks.

The XXICC programming environment is based on how the author likes to develop software. He was exposed to LISP implementations at an impressionable age. LISP had an extremely interactive environment, which encouraged the programmer to develop by creating small functions and immediately test them. Once a function was tested, it could be treated as a built-in function and became a building block for constructing more complex functions. This was in total contrast to the batch-oriented systems of the time, where you submitted a deck of cards and got your results hours later. The latter approach took away the luxury of rapid modular test, encouraging you to build the entire program in one shot since there were limited runs in a given day.

While LISP was a nice environment and great for symbolic programming, the language was not well suited to most other tasks. A strongly-typed language like C or Pascal was much better for those, but they required a long compile and link process for testing changes.

XXICC tries to get the best of both worlds, with a clean interactive environment to encourage rapid modular testing along with a more expressive language. The environment is based on what the author and his graduate students developed circa 1990 for the Galaxy Programming Language [JFB 91] and Galaxy CAD System [JFB 92]. Meanwhile, some powerful programming environments with similar goals have been developed such as the Borland C environment and Eclipse. The author has little experience with the former and essentially none with the latter, but first impressions of Eclipse suggest that it is a complex environment akin to flying a jet airliner with 100,000 switches. This would appear to violate the author's Reduced Software Complexity philosophy. XXICC tries to keep things as simple as possible.

At the present time, XXICC only supports running GalaxC programs within the GalaxC environment. This is good enough for developing XOE, for small test programs, and for learning GalaxC. At some point we'll add a capability for generating stand-alone GalaxC programs.

1. GalaxC File Names

A GalaxC source code file name has extension `.gal` or `.xoe`. A `.gal` file contains ASCII characters and perhaps format control characters as described in *Programming in the GalaxC Language*, section "Format Control". You may be able to edit `.gal` files using your favorite text editor. Otherwise, use XOE. A `.xoe` file may also have graphics, as described in *The XXICC Anthology*, chapter "XXICC Objects". Edit these files using XOE.

2. Compiling Single Files

Compiling a single file GalaxC program is easy. Just edit the file in a XOE window and press `F6` to compile the program. If there are any errors, XOE scrolls to and highlights the error and prints a message in the Output Window (on Win32 versions) or the `xterm` window that launched XXICC (on X11 versions).

For example, if your program is:

```
printf("Hello, world!\n);
```

GalaxC will complain that the string constant does not have a closing quote. Correcting the program will remove the error:

```
printf("Hello, world!\n");
```

You should always save your source code before compiling since XXICC is experimental and the compiler may crash. XOE invites you to save when you press `F6` with a "dirty" file (a file that needs to be saved).

To run a program, press `ctl-F6`. This runs the most recently compiled program and displays calls to `printf` in the Output Window or `xterm`:

```
Hello, world!
```

You have to be careful with `ctl-F6`, because at the present time there is nothing to protect `ctl-F6` from executing bad code. In particular, if the compile fails there may be some code generated, and `ctl-F6` will happily execute the partial code and most likely crash XXICC.

Another consideration is that there is a shared code space for compiling single files. If you have multiple XOE windows, `F6` overwrites the shared code space and `ctl-F6` executes whatever is in the code space no matter which window created it. This usually isn't a problem, because you're usually only working on one program at a time. However, if your program opens a window using G-SWIM, the open window points to functions in the shared code space. If `F6` overwrites the shared code space while the old window is open, the next window event for that window will most likely crash XXICC. Thus you should close test windows before pressing `F6`. As always, save frequently.

3. Program Structure

This is as good a place as any to describe the structure of a GalaxC program. A GalaxC program is a sequence of statements separated by '`;`', as follows:

`include` statements`;`
function and other definitions`;`
main program

3.1 `include` *statements*

A GalaxC `include` statement is similar in purpose to a C `#include`, but has some important differences. `#include` is a C pre-processor command and simply inserts a file as text into the source code being compiled, without checking the included file for validity. On the other hand, a GalaxC `include` usually links in a GalaxC object code file (e.g., `xoebase.gi`), though it may include source code instead (e.g., `xoedrinc.gal`).

An `include` statement has the form:

`include` *file*

Where *file* is a string literal or a string-valued expression that is constant at compile time. Here are some examples:

```
include "gswimlib.gi"          Include object code.
include "xoedrinc.gal"         Include source code.
```

All GalaxC `include` statements that link `.gi` files must appear at the beginning of the file, before any definitions or declarations. GalaxC automatically assumes that you include the GalaxC library `gxclib.gi` which defines most operations and statements. If your program uses G-SWIM, you need:

```
include GswimAPI
```

`GswimAPI` is a compile-time string constant equal to the G-SWIM API for your computer, e.g., `win32api.gi` or `x11api.gi`. See the G-SWIM chapters in *The XXICC Anthology*.

When you include a `.gi` file, XXICC automatically includes all `.gi` files included by that file. For example, if you include `xoedxo.gi`, XXICC includes `GswimAPI` and `xoebase.gi`.

Including a `.gi` file means that you can access all the functions, macros, and variables (etc.) defined in that file. The usual GalaxC search order applies: a pattern in a later `.gi` file may hide a pattern in an earlier one. However, GalaxC does not re-include a `.gi` file if it has already been included, perhaps by an earlier `.gi` file.

If you are building GUIs, feel free to call functions in XOE. For example, `xoedial.gi` includes functions needed to create dialogs and `xoemenu.gi` includes functions for menus. Currently they are somewhat specific to XOE -- we hope to make them more general-purpose as XXICC evolves.

GalaxC `include` statements that include `.gal` and `.xoe` files may appear anywhere in a file. These are closer in behavior to C `#includes` except that GalaxC scans and parses them before including them in the code. Currently this capability has only been tested for `xoedrinc.gal`, which is included in both `xoedraw.gal` and `xoeprint.gal` to provide a primitive generic function capability. Error handling for included `.gal` and `.xoe` is incomplete: XOE does not show you where the error is if it occurs in an included `.gal` or `.xoe`.

3.2 Main Programs

C requires that all executable code be inside functions, with one function called `main` which is called by the operating system. GalaxC allows main program code after the last function in a source file. This is the code that is executed when you press `ctl-F6`. Any top-level code between functions is unexecutable.

Consider the following program:

```
fn main = printf("Hello, world\n");
```

If you compile and run this program, you will get absolutely nothing in the Output Window. This is because it only defines function `main`: nobody told `ctl-F6` to call anything. What you need is:

```
fn main = printf("Hello, world\n");
main
```

The statement "`main`" generates code to call function `main`.

Each file in a multi-file program can have a main program. They are often used to initialize data structures declared in the file. Whenever XXICC loads a `.gi` file, it always executes the file's main program.

4. Compiling Multiple Files: Make

Revision 0.0a added Make capability based on file time stamps. It has not had long-term testing and has mostly been used for compiling XOE, which is the first large program written in GalaxC. A careful programmer can use it now, but should be alert for bugs.

4.1 Module Numbers

A GalaxC **module** is a separately-compiled program unit. It is usually a single source code file, but may be a source file that includes other source files like `xoedraw` and `xoeprint`. Each module has a **module number**, which is a unique 16-bit number currently from 1-100. At some point these will be transparent to the programmer, but for now they must be given explicitly for each module using a `module` statement, e.g.,

```
module 25;
```

Module 1 is the GalaxC primitive library built into the compiler (`primlib.c`). Module 2 is `gxclib`. G-SWIM uses modules 3-5. XOE uses 6-27. Single file compilation uses module 100. For now, use modules 50-99 for your own work.

Module numbers are used for dynamic linking in `.gi` files. When modules are in memory, all pointers are 32-bit absolute addresses. In a `.gi` file, XXICC uses *base-displacement* addressing for pointers so that XXICC can load the `.gi` file anywhere. Most of the base-displacement addresses are into a module's *pattern table*, where *base* is a module number to get the beginning of the module's pattern table, and *displacement* is a 16-bit index into that table. For global variables, *base* is a module number to get the beginning of the module's global data area and *displacement* is a 16-bit offset into that area. The code for reading and writing `.gi` files and doing the transformations is in `files.c`.

When you compile a module using `F6`, XXICC uses module number 100 no matter what `module` says. This is needed for compiling XOE in itself, since we cannot be overwrite a XOE module while recompiling it or XOE will crash.

4.2 Make

Starting with release 0.0a, XXICC has a Make capability built into the GalaxC compiler. There is no Make file: the dependencies are automatically inferred from `include` statements. For the most part, its function is the same as Make, i.e., it looks at file time stamps and recompiles an object code file *X* if the source and object files upon which *X* depends are newer than *X*. However, GalaxC's built-in Make is a bit more sophisticated because it considers three kinds of changes:

1. No change: if all you do is change a few comments, or maybe add or delete unessential parentheses from an expression, the object code generated for a file *Y* does not change. However, if all Make does is look at time stamps, it will recompile all the files that depend on *Y* even though they won't change either. If *Y* is a header file that is used by lots of programs, this can result in a lot of unnecessary recompilation, discouraging improving comments.

GalaxC automatically detects this situation and avoids recompiling the files that depend on *Y*.

2. Code-only changes: if the changes are isolated to function bodies, there is also no need to recompile files that depend on *Y*. However, it is necessary to relink them since their function bodies may call functions in *Y* that have moved to new addresses.

   GalaxC automatically links functions as it loads object code files, so it handles code-only changes by reloading the modules that depend on *Y*. This is usually very fast. For example, you can add `printf` statements for debugging and recompile and retest almost instantly.

3. General changes: if the change involves adding or deleting global variables, changing macro bodies, changing function arguments, or any change that is not code-only, then GalaxC must recompile all files that depend on *Y*. We call these **general modifications**.

   GalaxC's Make automatically recompiles files that depend on *Y*'s general modifications. This can be slow or fast depending on how many modules are affected by the change.

GalaxC has two time stamps associated with an object file. **Ftime** is when the file was last modified and is managed by the operating system. **GMtime** is when the latest general modification occurred, either within the object file itself or in an object file it includes, perhaps indirectly. When deciding whether to recompile or reuse an object module *X*, GalaxC considers the following:

- If *X*'s source file is newer than *X*'s Ftime, then recompile *X* since the source code could have arbitrary changes.

- If *X* includes any source files newer than *X*'s Ftime, then recompile *X*.

- If *X* includes any object files with GMtime newer than *X*'s Ftime, then recompile *X*.

- If *X* does not need to be recompiled, it needs to be reloaded if it includes an object file that was reloaded since the last time *X* was reloaded. This is handled using status flags rather than time stamps.

GMtime is stored in the `.gi` file. This may be problematic in some versions of XXICC. Both Ftime and GMtime use the Unix `time_t` type which is the number of seconds since 0:00:00 UTC January 1, 1970. All XXICC versions have Unix functions to get `time_t` values such as `time()` and `stat()`. This ought to be consistent across all implementations, but experiments have proven otherwise particularly for Borland C.

I think it's safe to assume that Ftime and GMtime will be consistent on a single platform, so long as you always recompile `.gal` and `.xoe` files instead of copying `.gi` files from another platform you should be safe. Also, if you change the date of a `.gi` file by copying or touching it, GMtime does not change.

The Borland C run-time library used when developing XXICC has inconsistent values for `time()` and `stat()`, each off by a fixed number of hours. GalaxC takes care of this automatically by "calibrating" the time functions to work out how many hours to add to `time()` and `stat()` so that Ftime and GMtime behave correctly. However, this was derived by experiment and because of the vagueness of the documentation, we fear that other versions of the run-time library may behave differently. So we have added some `xxicc` command line options to use if Make is not working correctly:

`-nocal`
: Do not calibrate the `time()` and `stat()` functions in the Borland C version `xxibor`. Use this if the Borland C library already computes Ftime and GMtime correctly. This option only affects `xxibor`:

the Cygwin and GNU/Linux versions do not require calibration.

`-nogm`
: Do not use GMtime at all. This causes GalaxC's Make to only consider Ftime and not take advantage of GMtime to avoid unnecessary recompilation. This option applies to all versions of XXICC.

[See *Installing and Running XXICC* §4 for general information on running XXICC from the command line.]

So how do you tell if you need either of these options? Well, first see if GalaxC's Make is behaving correctly by examining the trace in the Output Window. Does GalaxC automatically recompile and/or reload files as it should? Are there extra or missing recompliations?

Another is to press `F6` in the Output Window. This causes GalaxC to print a list of all modules currently in memory, along with their includes. Each line has two hexadecimal numbers representing Ftime and GMtime. Ftime should always be the same or later than GMtime. If you make a general modifcation to an object file, Ftime and GMtime should be nearly identical. If you make a code-only modification to an object file, Ftime should update but GMtime should stay the same.

4.3 XOE Compile Commands

XOE uses `F6` in combination with shift keys to compile and run GalaxC programs. `F6` behavior depends on whether the file being compiled is part of XOE or a non-XOE file. `F6` also behaves differently in the Output Window.

XXICC manages two module lists to keep track of which object modules are loaded in memory. **MDBlist** contains modules that have been loaded and linked to each other and are ready to run. It normally includes all XOE modules plus non-XOE modules that are ready to run. **MDBpending** contains modules that are ready to be recompiled, and reused if recompiling makes no changes.

Here what `F6` does in a XOE window:

`F6`: *ad-hoc* compile
: Compile the GalaxC source code in the XOE window, but ignore its module number and do not save the resulting object code in a `.gi`file. This is an *ad-hoc* compilation, used primarily single file programs or for making sure a program compiles before continuing edits.

`sh-F6`: make
: Compile the GalaxC source code in the XOE window and consider its module number. This is for compiling one file of a multi-file GalaxC program. If the compilation is successful, store the resulting object code in *xxx*`.gi` and see if it has changed. If so, see whether the changes are code-only or general modifications. Set *xxx*`.gi`'s GMtime to the current time if it has general modifications, otherwise set it to the latest GMtime of all object modules included in *xxx*`.gi`.

  GalaxC always compiles source code using module number 100, and changes the module number to *N* when it stores *xxx*`.gi`. *N* is either specified in a `module` statement or is the next unused module number. Compiling under a different module number is particularly important when recompiling a XOE file, since we don't want to change XOE modules while XOE is running.

  If *xxx* is a XOE module, do not reload *xxx*`.gi` at this time: you must enter `sh-F6` in the Output Window as described below. This is because if you are changing XOE you may want to make changes to several XOE files and then reload them together. Recompiling XOE module *xxx* does not

affect MDBlist. GalaxC does not re-make any modules nested in *xxx*: you must enter `sh-F6` in the Output Window.

If *xxx* is not a XOE module, `sh-F6` first moves all non-XOE files to MDBpending. Then it performs a Make of *xxx*, recompiling included object modules if their time stamps require it. When GalaxC recompiles a nested module *yyy*, it moves it from MDBpending to MDBlist if *yyy* is unchanged. Otherwise GalaxC deletes the MDBpending copy, and reloads and reruns the *yyy* object file. As with XOE modules, *xxx* itself is not reloaded at this time.

`ctl-F6`: run

Run the code resulting from the last `F6` or `sh-F6` compilation, displaying results in the Output Window. Code in memory does not have any module numbers, so `ctl-F6` doesn't care if you used `F6` or `sh-F6`.

`ctl-sh-F6`: toggle debug level

Toggle the compiler's debug level, so that it generates increasingly verbose debug text. This is most useful for GalaxC compiler writers rather than users.

Here what `F6` does in the Output Window. GNU/Linux versions use `xterm` instead of an Output Window and cannot currently use these commands directly.

`F6`: display module lists

Display the contents of MDBlist and MDBpending. For each module, show its module number, its Ftime and GMtime as hexadecimal numbers, and a list of which object modules it includes.

If there is no Output Window, you can display these lists by calling C function `PrintAllIncludes()` from `files.c` by running the following GalaxC program:

```
void inline PrintAllIncludes = cdecl;
PrintAllIncludes
```

`sh-F6`: re-make all

Re-make all the modules MDBlist and MDBpending. Move all modules in MDBlist to MDBpending, and then make all the files in MDBpending. `sh-F6` is the best way to re-make XOE after recompiling files using `F6` in XOE windows.

Close all XOE windows before executing `sh-F6`, since if any XOE files are reloaded open windows may have invalid function pointers.

If a compilation fails, uncompiled files are left in MDBpending. If XOE is intact, you can use it to edit and recompile the failed file(s). However, if it's a XOE file that failed to recompile, you cannot use this XXICC environment to correct it. You should always keep a working version of XXICC around in another directory so you can start up a rescue copy of XOE in another set of windows.

If MDBlist only contains XOE files, `sh-F6` is equivalent to executing "`xxicc xoe`" from the command line.

`clt-F6`: delete pending modules

Delete all the modules in MDBpending.

`ctl-sh-F6`: re-make XOE (forced)

Peform forced make of XOE. This is equivalent to executing "`xxicc -force xoe`" from the command line. `ctl-sh-F6` deletes all modules in MDBpending and MDBlist and makes the XOE root file.

Close all XOE windows before executing `ctl-sh-F6`, since if any XOE files are reloaded open windows may have invalid function pointers.

5. Hints

This section included some (we hope) helpful hints on compiling XXICC programs.

- "Cannot find a match for this formula": this is the most common error message when compiling a GalaxC program. It means that GalaxC has successfully scanned and parsed your program, but the GalaxC analyzer was unable to find a match for an expression. The failed expression is highlighted.

  Sometimes it's easy to see what's wrong, for example you left out a semicolon between two statements or an identifier is capitalized wrong. Here are some other possible causes:

  1. Missing or extra space between identifier and '`(`' or '`[`'. Unlike C, GalaxC considers "`f(x)`" to be different from "`f (x)`", and considers "`A[i]`" to be different from "`A [i]`".

  2. Function definition may have argument erroneously declared to be a variable. For example:

     ```
     int var x,
     fn f(x) = ...
     ```

     may look fine as a function declaration, but since `x` is a `var` and not an `arg`, GalaxC considers `x` to be literal text in `f(x)`. When you try to call `f(x)` using `f(42)`, GalaxC cannot match the latter expression.

     The correct form is:

     ```
     int arg x,
     fn f(x) = ...
     ```

  3. Make sure identifiers have the correct font. Version 0.0 of GalaxC considers font when matching identifiers, so `cat` is different from *cat*. This will become optional in future versions.

  4. If a function `f` has an argument τ, a call to `f` accepts expressions that are subtypes of τ. However, it does not accept an expression that is a supertype of τ. For example, the above function `f(x)` is happy to be called with `f(2)` but not `f(y)` where `y` is of type `ulong`. You may need to write `f(y: int)` or `f(long(y))`.

  It can be challenging to find what's wrong with a complex expression. Try copying the expression to a separate line and then break up its sub-expressions into separate statements. Then you can often find which expression GalaxC cannot match.

  At some point we plan to have the pattern matcher list the closest matches, which should help.

- Starting in 0.0c, GalaxC checks variable attributes to make sure you don't assign to a `const` or similar

errors. For example, if you write:

```
var s = "cat";
*s = 'b'
```

you'll get an error like:

```
ACG error: Cannot assign to const &con char
```

This error means you cannot assign to `const` type τ, where τ = `&(const char)`. The error message abbreviates `const` as `con` and omits the parentheses. In this example, `"cat"` is a read-only `string`, so `s` is a `string` as well. `*s` converts pointer type `string` = `@(const char)` to reference type `&(const char)`, which is a variable of type `const char`. Since you cannot assign to a `const` variable, GalaxC's ACG (Analyzer / Code Generator) produces the error.

6. Issues

- Currently XXICC works best if the `xxicc` executable and all program files are in the same directory. This needs to be relaxed.

7. References

[JFB 91] John F. Beetem, *The Galaxy Programming Language*, A. B. Creative Consulting, 1991.

[JFB 92] John F. Beetem, *Galaxy CAD User Manual*, A. B. Creative Consulting, 1992.