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 an already compiled GalaxC file (e.g., `xoebase.gi`), though it may include source code instead (e.g., `xoedrinc.gal`).

All GalaxC `include` statements that link `.gi` files must appear at the beginning of the file, before any definitions or declarations. *This restriction may be relaxed.* 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. When you include a `.gi` file, XXICC executes the file's main program after loading it.

4. Compiling Multiple Files

Compiling multiple files is a little ugly in the current implementation. It is fine for compiling XOE, which is

the first large program written in GalaxC, but needs a lot of refinement for general use. A careful programmer can use it now, but it will be a lot easier in the future.

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 Incremental Recompilation

XXICC has an incremental recompilation mechanism for rapidly testing changes to XOE code. This capability needs some work to be of general use, but is quite nice for XOE development. Basically, instead of pressing `F6` to recompile a module, you press `sh-F6` which causes XXICC to write a `.gi` file for the XOE module *with the specified module number*. XXICC also checks whether the new `.gi` file has (a) no changes, (b) just changes to code inside functions, or (c) general changes to patterns such as added or deleted global variables or macros. XXICC then marks the currently-included modules to flag those that need to be reloaded because (b) has moved functions around and those that need to be recompiled because (c) has made general changes. Then the user closes all windows except the Output Window and enters `F12` in the Output Window. This causes XXICC to reload and/or recompile XOE according to the flags, in essence performing an automatic `make`.

Most changes are to the bodies of functions, in which case reloading is very fast. This allows one to use `printf` statements for debugging without getting tired of waiting for recompiles. Even when there are general changes it's still pretty fast depending on how many modules are affected by the change.

XXICC also allows you to to press `sh-F12` in the Output Window, which causes XXICC to recompile all of XOE from scratch. This is the same as running "`xxicw xoe`" or similar (see *Compiling and Installing XXICC*). `F12` in the Output Window only works in Windows versions at the present time. X11 versions need to use "`xxicx xoe`" or similar.

At some point this capability will be expanded to a more complete `make` functionality, comparing the dates of `.gi`, `.gal`, and `.xoe` files to recompile automatically.

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.

  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.

6. 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.