The XXICC Anthology

John F. Beetem

Chapters 2-4 are derived from *The Galaxy Programming Language* by John F. Beetem, with changes and new material. Section 8.7 is derived from *Galaxy CAD User Manual* © 1992 John F. Beetem with changes and new material. All other chapters are new.

Revision 0.0h, October 2013.
For the latest XXICC release and documentation, please visit `xxicc.org`.

Note to the Reader

Thank you for your interest in the XXICC and GalaxC research projects. This book is a collection of miscellaneous XXICC topics. It supplements *Programming in the GalaxC Language* [JFB 11], which describes the textual core of the GalaxC programming language.

Contents

Some of the chapters are in preliminary form and will be expanded in future editions. XXICC is a research work in progress, and there are plenty of rough edges at this time. These are being addressed by the author, with priority given to those things he needs most. Users must realize that at the present time XXICC is still in the experimental stages and while most things work, it has never been stress-tested or even tested by anyone other than the author. Thus XXICC comes "as is" with no warranty whatsover.

Acknowledgements

The author would like to renew thanks to the talented individuals who helped make the original version of the Galaxy environment circa 1988. Foremost he wishes to thank Anne Beetem for her ideas, inspiration, support, and scholarly collaborations. He would also like to thank Jim Rose for his contributions to the original implementation the Galaxy environment, especially G-SWIM and the Galaxy Editor for Text (GET). The author thanks Monty Denneau for asking The Question which led to the original conception of Galaxy.

The author would also like to thank his family and friends for their support and encouragement over the years which led to GalaxC and XXICC reaching this point, with special thanks to Joni Beetem. He would also like to acknowledge the literary inspirations of Miguel de Cervantes and Edmond Rostand towards putting Quixotic idealism ahead of practicality, and Kurt Vonnegut for *Cat's Cradle.*

Revision History

8 August 2011: Revision 0.0a made minor changes to Chapters 2, 3, and 5.

29 January 2012: Revision 0.0d made minor changes to Chapters 2, 3, and 4.

18 June 2012: Revision 0.0e made major changes to Chapter 7 and minor changes to §2.12.

18 February 2013: Revision 0.0f added horizontal scrolling to §2.4, zooming to §2.4 and §6.6, changed `ctl-T` (Times) to `ctl-D` (Document font) in §2.7, added Angles using BAMs in §3.1.2, added additional arc drawing functions to §3.4, and added new Chapters 8 and 9.

2 June 2013: Revision 0.0g added rotated fonts to §3.5, implementation of rotated text (§6.7), rotation and mirror commands (§8.5), and rotating/mirroring components to §8.6.3. Rev 0.0g also adds invisible text and unattached component text (§8.6.2). These changes are also reflected in Chapter 9. Chapter 8 has many other improvements such as §8.6.1 showing an example of drawing a schematic and removal of many issues from §8.9.

25 October 2013: Revision 0.0h added Chapter 10: "Using GalaxC for Hardware Design", which describes the GCHD extensions to the GalaxC language. Chapter 8 has numerous improvements such as hierarchical schematics (new Section 8.8), attaching `FIGtext` to nets (§8.6.4), and updates to Hints (now §8.9) and Issues (now §8.10). Chapter 2 has minor updates to the `F6` commands (§2.12).

Chapter 1
Reducing Software Complexity

> Everything should be made as simple as possible, but not simpler.
> *Albert Einstein*

> Simplicity, when it removes encumbering details, makes for beauty in music, in art, and in living. It clears the springs of life and permits wholesome mirth and gladness to bubble up; it cleans the windows of life and lets joy radiate.
> *Quaker Philadelphia Yearly Meeting, Faith & Practice, 1961*

1.1. Prologue

In the early 1980's a controversial idea called RISC (Reduced Instruction Set Computer) started to take root in computer architecture. The basis behind RISC is that even though it had become possible and inexpensive to build computers with an extremely rich collection of opcodes and addressing modes, it was not actually a good idea to do so. Before RISC, computer architectures proudly proclaimed the number of instructions, native data types, and addressing modes, asserting that "more is better". The RISC movement noted several problems with this assertion:

1. Writing compilers that can take advantage of a huge number of opcodes and addressing modes is very difficult, especially if the instruction set is irregular. Complex compilers are very hard to debug and update. In fact, most compilers only used a subset of the available opcodes and addressing modes.

2. The presence of rarely-used opcodes and addressing modes slows down the execution of frequently-used opcodes and addressing modes. Simple decoding logic is faster than complex decoding logic.

3. Any irregularity in the instruction set makes pipelining difficult.

With RISC, each opcode and addressing mode must justify its existence. Adding an opcode or addressing mode always slows down the CPU cycle time, but if adding it results in fewer instructions so that the overall software performance improves, then doing so is worth it. On the other hand, if emulating the opcode or addression mode using the existing instruction set has better overall performance, then do not add the new feature no matter how tempting it is to do so.

RISC was treated with scepticism in the 1980's [B&M 86] but it has mostly triumphed over CISC (Complex Instruction Set Computer) architectures. The principal holdout is the x86 Intel Architecture, which achieves high performance by using huge numbers of high-power transistors cooled by increasingly sophisticated schemes. In contrast, the most successful RISC architecture is ARM (Advanced RISC Machines), which is almost universally used in cell phones because of superior performance per watt.

So what does this have to do with software? Isn't software free, or essentially free because the cost per MB of Flash and DRAM has become so ridiculously low that people have Flash cards that programmers would have killed for back in 1980 filled with inane snapshots? Well, that's true in the same sense that in 1980 transistors became so cheap that adding new opcodes, addressing formats, native data types, and instruction formats was a no-brainer... until you calculated the impact on compiler complexity and decode logic performance, at which point it became clear that RISC was better. So let's see if adding that new software feature is really free after all.

1.2. Introduction

The last few decades has seen an explosion in software complexity, fueled by massive increases in computer performance coupled with a staggering drop in cost of memory. Modern desktop computers have gigabytes of DRAM and hundreds of gigabytes of disk. Because computers are so fast (to be revisited later) and memory is so cheap, software complexity has practically no direct cost. However, indirect costs are present and increasing.

Let's first see where we've come from.

One of the most popular computers of the 1970s was the DEC PDP-11. It had a small but extremely well thought-out instruction set which was more pleasant to program in that most of the high-level languages available at the time. One of the key reference books was the *PDP-11 Peripherals Handbook* [DEC 73], which provided everything a programmer needed to know to interface to disk drives, printers, and every other DEC product that could interface to a PDP-11.

The *PDP-11 Peripherals Handbook* has a very nice chapter called "Programming", which describes -- with assembly-language examples -- how to write device-driver code for a typical PDP-11 peripheral, using both wait loops and interrupts. This chapter is *eight pages long*. Its simplicity invites the programmer to write some code and try it out. Compare this to device driver reference books with hundreds of pages for Windows and Unix, which seem to have the goal of preventing programmers from getting anywhere near their computers.

Well, nowadays the PDP-11 is considered a relic from the past, a 16-bit computer that could not directly access more than 64KB of memory. The argument today goes that we need multi-GB to run things like Windows and Linux with their snazzy GUIs. Of course, that argument forgets that the original Unix was developed on a PDP-11, as was the C programming language. That argument also forgets that GUIs worked very nicely on the original Apple Macintosh, which had 128KB of RAM. Now scaling up screen sizes does require a few extra megabytes, but gigabytes? Is a 1GB Windows machine today really 16,000 times better than a PDP-11 with 64K? Or is it only 100 times better, with rest pure waste?

Now that we've seen where we came from, let's look at the indirect costs of software complexity.

- Learning to use the software: As more features get added to software, getting started as a new user gets increasingly difficult. It used to be that software came with a user's guide, but nowdays software is supposed to be "intuitive" and there is no user's guide. There's on-line help, but that is usually far from adequate. An amusing example of this is how Microsoft guru Charles Simonyi couldn't figure out how to switch off Microsoft Office's infernal "Clippy" even though he managed Office development [SR 07, p. 38].

- Developing and maintaining the software: As software increases in size, the number of possible interactions rises even faster. This makes adding features more expensive, since they may affect more code. The effect on bugs gets even worse: more interactions results in more possible bugs and they get more subtle and difficult to find and reproduce.

  Here is an even more damaging concern: Once software hits a certain level of complexity, no one person understands it any more. Programmers then become reluctant to make large changes that may improve program structure and maintainability because they fear they may break code they do not understand. Instead, they add patches upon patches, until the orginal software structure vanishes within a tangled mess of patched-together features. According to Alan Perlis, "Every program eventually becomes rococo, and then rubble."

An excellent example of this is Macintosh OS version 7. Prior to version 7, an individual programmer could read *Inside Machintosh* volumes 1-5 and learn everything anyone needed to know to program the Macintosh. With version 7, everything became much more complex (in this author's opinion) and you needed a team to do any significant programming. No one person understood the entire program any more, and the path to rococo (and then rubble) became assured.

- Loading and running the software: One could argue that computers are so fast today that software complexity does not have a significant effect on performance as far as users are concerned. This argument fails to hold water as each faster computer comes with a new version of the operating system that takes even longer to boot. It is patently absurd that a Radio Shack TRS-80 with a Z80 microprocessor can boot instantantly while it takes a minute for a 1GHz 64-bit processor to do so. This is clearly an example of overly complex software. Yes, it is possible for users to switch off features to make their computers boot faster, but most cannot because the system is so complex and fragile that they do not know which ones can safely be turned off.

  While CPU performance and memory size and performance have increased, the I/O bandwidth between them has not grown anywhere as quickly. It certainly has not kept up with the increase in program size, which is why it takes longer to load and run so many programs. Installing large programs takes much longer as well. Yes, CD-ROMs are much faster than floppy disks, but not when you have to install hundreds of megabytes.

  The effect is particularly dramatic with embedded computers like cell phones, which have less memory and run processors slower to conserve power. There is a reason cell phones do not run full implementations of Windows or Linux.

- Power consumption: Intel has shown us that it is possible to get high performance from bloated software by using lots of hot transistors. Similarly, memory bandwidth has increased significantly using DDR schemes. However, all this takes considerable electrical power, which adds up over the millions of computers in operation. How much of this power is actually being put to good use?

CAR Hoare has an excellent paper called "The Emperor's Old Clothes" [CH 81] which discusses software complexity at length. The title comes from a story an emperor who acquired a fine new set of clothes each year, but insisted on draping the new set of clothes on top of the existing ones. Eventually there was a huge, immovable mound of beautiful clothes but nobody could see the emperor any more.

This is often the effect of attempting to maintain legacy compatibility by adding patches and layers to emulate previous versions. At some point it becomes far better to discard the old code and start over.

1.3. How to Reduce Software Complexity

1.3.1 Omit Unnecessary Features

Following Einstein's and Occam's advice, we must avoid the temptation continually to add unnecessary features to software. The author's own experience is that new versions usually do not add any new features he really needs, and often screw up existing features that he does need. In many cases, it seems that the new features are there solely so there is a reason "on paper" to upgrade to the new version. Technology magazines are complicit in this, since they tend to celebrate the new features without considering whether they are useful or merely "newsworthy".

Each software feature inherently reduces usability and productivity by making the software more complex

to the user. As with RISC opcodes and addressing modes, each new software feature should have to justify its existence by proving that its presence improves usability and productivity more than the inherent losses. If users can accomplish the same function with existing features, is it really necessary to add the new feature? If it's used often enough to improve productivity significantly, it's worth it. Otherwise, it's probably not.

A fine example is Microsoft Office's infernal "Clippy", mentioned earlier. While the author has met people who find Clippy entertaining and impressive animation, he has yet to meet anyone who actually finds Clippy useful. Most have to either put up with Clippy's nonsense or have taken the time to figure out how to turn Clippy off. In any case, the large amount of development time that went into Clippy seems to be a total waste from the author's point of view.

In general, one should minimize the number of ways of doing something. For example, having keyboard equivalents of menu selections is good, but tabbing between check boxes and groups of check boxes compilicates the code for dialog boxes and makes it harder to get right. While one can argue that users can choose to ignore the feature if desired, the reality is that a misplaced keystroke can trigger the feature unexpectedly, resulting in user confusion.

A good way to control proliferation of features is to require authors to document all features before implementing them. Since many programmers hate to document anything, this limits new features to ones that those programmers want really badly. It also improves the quality of software documentation.

1.3.2 Encapsulate Complexity

Once a software feature has justified its existence, adding it inevitably increases software complexity both for users and developers. Modular design can reduce this effect significantly by encapsulating complexity. For example, putting "advanced options" is a separate dialog from "basic options" presents a simplified view to new users while allowing sophisticated users to have full control. Similarly, APIs benefit from having basic and extended versions of system calls.

Developers can reduce complexity by encapsulating advanced features in separate program modules so that basic functionality is simplified. The key to making this work is well-defined and well-documented interfaces between modules. Documenting what functions do is far more important than writing the code of a function, particularly as the software is maintained and improved.

Abstraction layers can also be a powerful tool to encapsulate complexity. One of the most successful examples of this is networking, which uses the Open Systems Interconnection (OSI) Model to layer the functionality at the physical, link, network, transport, and other layers. All physical layer considerations such as mechanical connection and electronic components are encapsulated so that the link layer and above do not need to consider them. Similarly, the phyical layer does not care what data or control is being set over it. The link layer only considers immediate connections between nodes, so it does not have to deal with alternate routing paths and other higher level considerations. The networking layer such as IP does not care what underlying technology (Ethernet, USB) is used to transfer data between adjacent nodes, so it can concentrate on routing. In IP, the networking layer makes a "best effort" to transfer data reliably, but could lose packets. However, it does not need to recover from this, which keeps IP simpler. The transport layer deals with reliable data transfer with a protocol such as TCP.

As long as layers have well-defined and well-documented interfaces, it is relatively easy to replace a layer with a different underlying technology, or to provide alternate technologies for a given layer.

Operating systems should also be designed in layers, so that one can easily substitute lower layers to run on

different machines or higher layers to provide the infrastructure for applications. This has only been moderately successful, as evident by the fact that so many applications are Windows-only or Linux-only. There is no operating-system equivalent to the OSI model. POSIX, X Windows, and TrollTech's Qt are notable steps in the right direction as they provide an adaptation layer to permit applications to run on multiple systems.

Abstraction layers are attractive, but programmers and users must beware of the *Law of Leaky Abstractions* [SR 07, SR 08]. The problem is that no matter how carefully one defines abstraction layers, there is always the possibility that bugs will cause a lower-level effect to show up at a higher level. A classic example of this is the buffer overflow problem in C. C arrays have defined sizes, but the object code does not enforce array limits so it is possible for an erroneous program to read or write data past the end (or before the beginning) of an array, clobbering other data or even clobbering return addresses on the stack. This is a well-known way of introducing worms into unsuspecting computers. C is often called a "high level" language, so naïve C programmers may expect that writing to an array cannot affect anything other than the array. In fact, C should be thought of as a "portable assembly language", which uses high-level notations to write portable machine language. Once C object code is running, its abstraction level is machine language. A correct program behaves as if it is a high-level abstraction. But erroneous programs can cause fail in ways that can only be understood at the machine language level.

1.3.3 Reuse Modules

An important way to reduce complexity is to reuse software modules whenever possible. This reduces program size and improves test coverage (and therefore software quality) since modules are executed more frequently. A single bug fix can fix many programs at once. Dynamically linked sharable libraries make this easy to do, so there is really no excuse.

One of the most glaring examples of where this is *not* done is the Microsoft Office suite. Microsoft Office began as a set of independent tools acquired from various companies with minimal interaction between them. After decades, Microsoft Office is *still* a set of independent tools, even though there is no technical need for them to be independent tools. So while you can create a table in Word, you cannot fill the table with dynamically calculated expressions like Excel. While you can create text in PowerPoint, you do not have the same capabilities as Word for adding special characters. You would think that after decades the user interfaces would have become completely common between the tools, or better yet, Microsoft would have developed a generalized tool that covers all the functionality. The author can only assume that the internal complexity of the tools is so high that no one person knows how they all work, so everyone is afraid to do more than maintain the code, adding rococo features until the software eventually becomes rubble.

Reusing software modules may require adding parameters for different contexts. This adds complexity, and at a certain point it no longer makes sense to reuse a module since doing so would complicate its use elsewhere. When this occurs, consider that:

- It may not be appropriate to reuse the module in the given context. However, it may be possible to reuse components of the module.

- It may be time to rethink the module interface entirely. Perhaps there is a generalization of the module that would simplify its use everywhere.

1.3.4 Generalize

Generalizing software features and modules is probably the most difficult of all software development tasks.

It is relatively easy to take a well-written set of specifications and implement it as code. Taking several bodies of code that were created for different purposes and finding the common elements so as to create a general form is challenging, particularly with deadlines looming. Yet, it is usually the best way to achieve large reductions in complexity and if successful will save huge amounts of effort in the long run.

A useful analogy here is Plato's *Allegory of the Cave* [Wiki 1] in which prisoners are held from birth in a cave in which their only life experience is shadows projected onto the walls. The shadows become their reality, and they are unable to conceive of the three-dimensional objects that cast those shadows. Similarly, particular implementations of software are projections of abstract objects into a specific programming language. A generalized object, if it exists, may project into many different implementations. If a programmer can conceive of the generalized object, then each of the specific implementations are simply special cases of the generalized object with general parameters given fixed values.

The challenge is that the generalized object is currently abstract, which makes it very difficult to think about and write about. Object-oriented programming styles have tried to realize this goal with limited results. Some day there will be programming languages that achieve this, making C and C++ look like assembly language in comparison.

An excellent example of generalization and reuse is the PDP-11 register set. Previous DEC computers had an accumulator. The PDP-11 has a set of eight "general-purpose" registers, which could be used for all opcodes and addressing modes, with a few minor restrictions. Two of these registers are not quite general-purpose: R6 is the stack pointer and R7 is the program counter. However, by using the standard opcodes and addressing modes with R6 and R7 the PDP-11 gets stack operations, immediate operands, and PC-relative addressing for free.

Similarly, some RISC machines designate register 0 to always have the value 0. This allows the RISC machine to omit instructions like "negate", since it can be done by subtracting from 0. It also allows the machine to implement direct addressing using base-displacement addressing with register 0 as the base register. This simplifies the set of opcodes and speeds up decoding.

1.3.5 Start Over

In the hurry to ship software there is a tendency to program incrementally by making small changes to existing programs. While this makes sense for bug fixes and minor tweaks, at some point adding one feature after another makes the software so complex that it collapses under its complexity: "rococo, then rubble". It then makes sense to "throw everything away" and take what one has learned to build an entirely new system.

The PDP-11 is once again a good example. It was created as a totally new architecture, not at all based on the popular PDP-8 or PDP-10. Best of all, DEC formed a team of both computer designers and system programmers who worked together to create an extremly elegant architecture. On the other hand, one could argue that the PDP-11's beauty and simplicity was largely because of economic considerations and the available gate density available at the time. This is true to some extent, but the reality is that the PDP-11 was much nicer than competing minicomputers of the time with the same number of gates.

Whenever creating a new system, one makes many decisions based on the constraints at that time. As time progresses, many of those constraints vanish yet the system continues as if they were still there. For example, most programming languages use monospaced ASCII characters, typically 80 column lines. This is a throwback to 80-column punch cards, which most programmers have not seen in decades, if at all. Yet the monospaced ASCII practice continues, even though proper mathematical notations have long been supported by displays and printers.

The great supercomputer designer Seymour Cray was a big proponent of throwing things away and starting over from scratch. He felt that he could not develop the next world's fastest computer by just reworking current machines. According to legend, he also applied this same philosophy to his hobby of building sailboats. Each winter he would build a new sailboat in his basement, bring it out in the spring, use it all summer, and at the end of the season he would burn it in a kind of Viking funeral. Next year's sailboat would always be better since he had learned from the previous one, and it did not have to carry along the mistakes of the previous years.

1.4. General References

Here are some general references on the subject of excessive program size and its implications.

1. Ed Perratore, Tom Thompson, Jon Udell, Rich Malloy, "Fighting Fatware", *Byte*, April 1993.

2. Niklaus Wirth, "A Plea for Lean Software", *IEEE Computer*, February 1995.

3. Robert Capps, "The Good Enough Revolution: When Cheap and Simple is Just Fine", *Wired*, August 2009.

Chapter 2
XOE User's Guide

The XXICC Object Editor (XOE) combines the necessary capabilities of a program editor, word processor, spreadsheet, and drawing editor into a single tool with a uniform user interface. While any ASCII text editor may be used to edit ASCII-only GalaxC programs, the true power of GalaxC comes out when using formatted text and graphics, which requires using XOE to edit GalaxC programs. XOE is also a simple but powerful word processor, suitable for writing documents such as this one. Users who are very fussy about getting formatting exactly right will most likely find XOE too limiting.

XOE also includes table, figure, and dialog box editing, though these are somewhat limited at the present time. These graphical structures are built from XXICC Objects (XOs). Some XOs are automatically formatted and you can edit them using text editing commands, as described in this chapter. Figure XOs have their own editing commands described in Chapter 8. XOs are described in detail in Chapters 5 and 9.

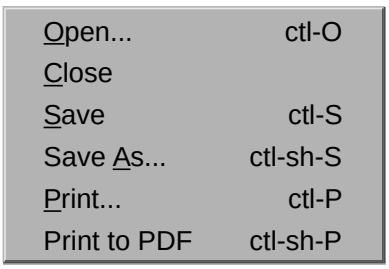
Basic XOE editing should be very familiar and intuitive to people used to modern GUI-based editors like LibreOffice and Microsoft Word. Whenever possible, XOE uses standard cursor keys and mouse operations. However, its initial appearance may be a bit startling to the new user in that menus and scroll bars are not normally displayed: all you see in a XOE window is the document being edited. Menus and scroll bars appear automatically when you move the mouse to the edge of the screen. The main purpose of this is so that XOE makes the best use of a small screen such as a 10" tablet or notebook computer. The second purpose is because XOE's author dislikes visual clutter and would prefer that valuable screen area not be wasted with lots of silly icons.

Caution: XOE is still an experimental program and may crash at any time, causing you to lose changes since the last save. Also, bugs may cause it to lose data without any warning. Please save frequently to avoid the first problem and make extra copies of the saved file so that you can get back to something useful if a serious error occurs. Basic text editing and document formatting are quite stable and were used to write this book. Editing dialogs and tables is less stable since they are more recent additions. Figure editing was released for the first time in 0.0f, so it's the least stable. Be particulaly careful when pasting, as it is one of the more complex parts of XOE and is most likely to have bugs.

2.1. Menus

As with most GUI-based editors, XOE has a number of menus. Menu items can be selected with the mouse, or you can use keyboard equivalents for most of them. The author finds that the menus can be very useful to see which commands are available, but keyboard equivalents are much more convenient once learned. *Chacun a son goût* (YMMV). To view the menus, move the mouse to the top of a XOE window and wait a moment. A **menu bar** appears showing which menus are available, e.g.,

You then click one of the menu bar buttons, which causes that menu to drop down, e.g.,



You can then click on one of the items in the drop-down menu.

If you prefer keyboard equivalents, you can also get the menu bar by pressing the `Menu` key or on Microsoft Windows machines by pressing and releasing an `Alt` key. Or, to get the File menu directly you can press `alt-F` (note the underlined letter in "File"). Once the menu has dropped down, you can select an item by entering the underlined letter, e.g., press 'o' or 'O' to select Open. XOE ignores case for underlined letters. Alternatively, you can select Open at any time (whether the menu is exposed or not) by pressing `ctl-O` (hold down a `control` key and press the **O** key without `shift`). For the keyboard equivalents on the right side of the menu, letters are only shifted if prefixed with "`sh-`": `ctl-sh-O` means something different.

Which menus are available depends on what is selected or where the text cursor is when you open the menu bar. For example, if you are editing a table or figure, special table or figure editing commands appear. This is an advantage to having the menu bar appear only as needed: it can be different each time.

2.2. Opening and Closing Documents: The File Menu

To open document, select the Open command from the File menu. This brings up a file selection dialog:

> *Insert sample file selection dialog here. For now, assume it looks like a typical GUI file selection dialog with a directory path, a list of files to select, etc.*

The "Open in:" field shows which directory to search for files, i.e., the *current directory*. Below this is the list of files for that directory, possibly with a horizontal scroll bar at the bottom. Names that end with '/' or '\' are subdirectories: if you click on one of them, XOE switches to that subdirectory. "../" or "..\" is the parent directory of the current directory.

The "File name:" field is the name of the file to be opened. You can type in or edit this field, or select a file from the file list which updates the "File name:" field. The "File types:" field filters which files are shown in the file list. It contains a sequence of file extensions such as ".gal.xoe.c.h" which means to list all files with extensions ".gal", ".xoe", ".c", and ".h". If the "File types:" field is null, XOE lists all files in the current directory.

Select the file you want either by double-clicking one of the files in file list, or by entering the file name and clicking the Open button. Since Open has an extra black border, it is the **default button** and you can press the `ENTER` or `RET` (`RETURN`) key instead of clicking it. If you decide that you don't want to open a file after all, click Cancel or press the `ESC` key. As is usual with dialogs, you can use the `TAB` key to move between editable fields (`sh-TAB` to go in the opposite order).

If you want to edit a new file, just type the name of the new file in the "File name:" field. XOE will ask if you are sure you want to create a new file, and if you click Yes it will open a blank file for editing.

The "Open..." menu item opens the file in the existing window, giving you a chance to save changes to an existing file before doing so. If you want to open a new window, use keyboard equivalent `ctl-sh-O`. The X11 version of XXICC brings up a file selection dialog automatically when you get started. The Win32 version brings up the Output Window, which does not have any menus. Press `ctl-O` in the Output Window to get the file selecton dialog.

XOE can open XOE documents with the ".xoe" extension or ASCII text files with any other extension, e.g., ".gal" for textual GalaxC files or ".c" for C source code. *At some point we will change this to look for a*

magic number at the beginning of ".xoe" files and ignore the file extension. XOE supports full document editing for document files, including paragraph formatting, tables, dialog editing, and figures. Text files have limited formatting capabilities, including text styles and page breaks. XOE can read text files with new line represented as `LF` (standard Unix), `CR` (Macintosh), or `CR+LF` (MS-DOS), converting all of them to `LF`. It accepts `TAB` characters, which it current treats as every 4 characters. It also accepts `FF` (form feed) characters as page breaks.

Here are the other commands available in the File menu:

Close

Close the XOE window. If the file has been changed since it was last opened or saved, XOE asks you if you want to save the changed file first.

Save ctl-S

Save the file using the same file name as when you opened it. XOE always writes `LF` as new line characters.

Save As... ctl-sh-S

Save the file under a new name, which you select using a file selection dialog similar to the one used for opening a file. The XOE window changes to the new file name.

Print... ctl-P

Print all or part of the file. On Win32, this brings up the standard printer dialog which allows you to select whether to print the entire file, the current selection, or a page range. This is not used on X11, since the author was unable to figure out how GNU/Linux likes to do this sort of thing.

Print to PDF ctl-sh-P

Print the current selection (or the entire file is the current selection is null) to an Adobe PDF file (version 1.3). The PDF file has the same name as the document, with extension replaced with ".pdf".

2.3. Cursor Positioning

When you first open a document, you will see a blinking vertical line at the beginning of the document. This is called the **cursor** and it indicates where the next character is to be inserted. We will sometimes call it the **text cursor** because it is usually used to enter text, but it may also be used to enter non-textual XOs.

You can move the cursor to a different position in the document using the mouse and cursor control keys. XOE tries to use the standard meanings for these keys whenever possible. In the following description, assume `shift` and `control` keys are released unless specified.

Click mouse button 1

Clicking mouse button 1 (usually the left one) without changing the mouse position moves the cursor to that position.

`LEFT`, `RIGHT`, `UP`, `DOWN`

Move the cursor left or right one character, or up or down one line. If the cursor is at an XO, move the cursor into, out of, or over the XO as appropriate.

`HOME`, `END`
: Move the cursor to the beginning or end of a text line.

`ctl-HOME`, `ctl-END`
: Move the cursor to the beginning or end of a document. If inside an XO, move the cursor to the beginning or end of that XO.

If the cursor is blinking, no characters or XOs are selected. You can select multiple characters and/or XOs using the mouse and cursor keys as follows:

Drag mouse button 1
: Press mouse button 1 at one end of the selection and release it at the other end.

Click or drag mouse button 1 with `shift`
: Clicking mouse button 1 while holding a `shift` key extends or reduces the selection to the mouse position. You can also drag the mouse while pressing button 1 and `shift` to extend or reduce the selection.

`sh-LEFT`, `sh-RIGHT`, `sh-UP`, `sh-DOWN`
: Extend or reduce the selection left or right one character, or up or down one line. Do not extend the selection into or out of an XO. However, you can extend or reduce the selection by one or more entire XOs.

`sh-HOME`, `sh-END`
: Extend or reduce the selection to the beginning or end of a text line.

`ctl-sh-HOME`, `ctl-sh-END`
: Extend or reduce the selection to the beginning or end of a document. If inside an XO, extend or reduce the selection to the beginning or end of that XO.

If anything is selected, the unshifted cursor control keys unselect the current selection before moving the cursor. `LEFT` and `UP` move the cursor to the beginning of the former selection. `RIGHT` and `DOWN` move the cursor to the end of the former selection.

2.4. Scrolling and Zooming

There are three ways to scroll a XOE document.

1. If you move the mouse to the right or bottom of the window, a scroll bar appears after a short delay. You can then drag the middle part of the scroll bar (called the **thumb**) to scroll the document up and down or left and right. You can also click in the regions above and below the thumb (to the left and right for the horizontal slider) to scroll by a page.

2. You can use the mouse wheel -- if available -- to scroll up and down. Hold down `shift` to scroll left and right. The mouse wheel is associated with the keyboard focus window, so you may need to click in the window or its heading before scrolling.

3. XOE implements the standard scrolling keys:

   `PRIOR`, `NEXT` (Page Up, Page Down): Scroll vertically to previous or next page. Unselect the

current selection and set cursor to approximately the same visible position in the window.

`sh-PRIOR`, `sh-NEXT`: Scroll vertically to previous or next page, and extend selection.

`ctl-PRIOR`, `ctl-NEXT`: Scroll horizontally by window width.

`ctl-UP`, `ctl-DOWN`: Scroll up or down one line. Keep cursor at approximately the same visible position the the window.

`ctl-LEFT`, `ctl-RIGHT`: Scroll left or right by 1/32 of the window.

Note: XOE uses `ctl-sh-UP` and `ctl-sh-DOWN` for superscripts and subscripts. This may cause a few surprises until you get used to it.

2.4.1 Zooming

The Zoom commands enlarge or reduce the image of a XOE document or text file on the screen without changing the document itself. To zoom in, press `ctl-+` which enlarges the image by 20%. To zoom out, press `ctl--` (hold down `control` and press the '`-`' key) which reduces the image by 20%. *At the present time, do not edit figures when zoomed out: XOE may round coördinates incorrectly if zoomed out.*

2.5. Inserting and Deleting Text and XOs

XOE uses the text insertion/deletion convention pioneered by Smalltalk and popularized by the Apple Macintosh. There is no such thing as "insert mode" or "replace mode". Rather, entered characters or XOs always replace the current selection. If no characters are selected, this has the effect of inserting characters or XOs at the text cursor. If one or more characters are selected, this has the effect of replacing these characters. The `BACKSPACE` key deletes the current selection or the character/XO *before* the cursor if nothing is selected. The `DELETE` key deletes the currently selection or the character/XO *after* the cursor if nothing is selected.

The `ENTER` or `RET` (`RETURN`) key inserts a new line (`LF`) character.

The `TAB` key inserts a `TAB` character. Tab stops are currently fixed at every 4 spaces. If you are editing in an XO, `TAB` inserts a `SPACER` XO.

Standard PC keyboards have an `INSERT` key which is normally used to toggle between insert and replace modes. Since XOE does not have replace mode, pressing `INSERT` brings up the Insert menu for inserting XOs into a document. Here is a sample Insert menu:

Each of these inserts an XO with default properties and contents which can be edited into something else. XOs are described in detail in Chapter 5. XOE tries to make sure that you can only insert XOs which are appropriate to whatever the cursor is in now, but this has not been full developed and it is possible to insert the wrong kind of XO and crash XOE.

Break is used to insert page breaks in documents, as well as for some formatting options. Page breaks may be inserted in ASCII text files; they are stored as form feed (FF) characters.

2.6. The Edit Menu

The Edit menu contains standard editing operations like Copy, Paste, Undo, Find, and Replace. Most have the usual keyboard equivalents.

Undo ctl-Z

Undo the last change made. XOE can undo any number of changes back to the last Save. XOE does not currently have Redo.

Cut ctl-X

Delete the current selection and save it to the clipboard. Ignored if no selection. The clipboard has two versions of the copied data, one with formatting and one with plain text.

Copy ctl-C

Copy the current selection to the clipboard. Ignored if no selection.

Paste ctl-V

Replace the current selection with the contents of the clipboard. Retain clipboard's formatting if possible, otherwise just paste text using the current format.

Paste Text ctl-sh-V

Replace the currrent selection with the plain text contents of the clipboard.

Find... ctl-F

Open the Find/Replace dialog so that you can enter a search string, and search for that text.

Find Again F3

Repeat search for the text last entered in the Find/Replace dialog.

Replace... ctl-R

Open the Find/Replace dialog so that you can enter a search string and a replacement string, and then search and replace that text with various options.

Select All ctl-A

Select the entire document. If given inside an XO, select the entire XO.

Edit props ctl-E

Bring up dialog to edit properties of the XO that contains the cursor. *At some point expand to edit properties of multiple XOs.*

2.7. The Style Menu

The Style menu sets the font style for character formatting.

Bold ctl-B
: Toggle **bold** style.

Italic ctl-I
: Toggle *italic* or *oblique* style.

Underline ctl-U
: Toggle underline style.

Normal ctl-N
: Turn off bold, italic, and underline styles.

Document font ctl-D
: Switch to default document font, usually Times Roman or similar serif font. Earlier versions of XOE used `ctl-T` to select Times Roman. `ctl-T` is now reserved for Transpose.

Helvetic ctl-H
: Switch to Helvetic sans-serif font such as Helvetica or Arial.

Monospaced ctl-M
: Switch to `Monospaced` font such as `Courier`.

Symbol ctl-G
: Switch to Symbol font with Greek letters.

Enlarge fonts ctl-<
: Enlarge all fonts in selection. '<' is the music notation for *crescendo* (increase).

Reduce fonts ctl->
: Reduce all fonts in selection. '>' is the music notation for *decrescendo* (decrease).

Superscript ctl-sh-UP
: Convert selection to superscript. If already a superscript, convert to base text.

Subscript ctl-sh-DOWN
: Convert selection to subscript. If already a subscript, convert to base text.

If there is a selection, the selection is changed as per the command. Otherwise, XOE remembers the new setting as the *desired format* for any new text that gets entered. Whenever the cursor is moved, the desired format changes to what it is at the new cursor position. This is usually the format of the character before the cursor, but is the character after the cursor at the beginning of a line or if you just moved the cursor left.

The Style menu has check marks to indicate the current style settings.

2.8. The Parag Menu

XOE paragraph formatting is loosely based on Microsoft Word and LaTeX. It is considerably simpler, providing capability adequate to most users' needs rather than trying to be the world's most feature-laden product so that even the most picky user is satisfied (*as if*).

XOE paragraphs are sequences of characters and/or in-line XOs, separated by `LF` characters created using `ENTER` or `RET`. Words within a paragraph automatically wrap to the paragraph's margins. Each paragraph has a paragraph style (**Pstyle**) which specifies formatting properties such as default font, justification, left and right margins, indentation of the first line, and list or heading numbering. Like LaTeX, the Pstyles of a document define the overall appearance of the document so that it is consistent and can be easily changed at a single location. There can be up to 256 base Pstyles, plus 5 levels of indentation.

Parag menu items change the properties of all paragraphs within the selection, which is temporarily expanded to paragraph boundaries. The first two Parag menu items change paragraph indentation level:

Indent	ctl-TAB
Outdent	ctl-sh-TAB

Changing indentation level only works if all selected paragraphs can be indented or outdented by that amount. For example, headings cannot be indented.

Next, we have commands to select Pstyle:

Left	Left-justified paragraph.
Center	Centered paragraph.
Right	Right-justified paragraph.
Quote	Multi-line quote indented on both sides.
Describe	Description list item. First line is outdented for a label.
Itemized	Bulleted (itemized) list item, with different bullet styles at different indent levels.
Numbered	Numbered list item, with Arabic, Roman, or alphabetical item labels.

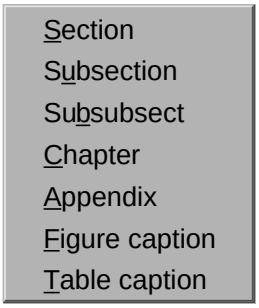
Finally, we have two commands for working with list items. If you press `ENTER` (`RET`) after a list item, XOE generates a *list item successor*, which is a continuation paragraph that does *not* have a list label like a bullet or item number. To get a new list item, you must enter `sh-ENTER` (`sh-RET`). (This is backwards from Microsoft Word.) In any case, you can toggle between list item and list item successor by pressing `ctl-L`.

Insert list item	sh-RET
Toggle list item	ctl-L

A numbered list can start with a new sequence number or continue the previous sequence. Normally XOE automatically does the right thing, but if you want to tell it to do the opposite insert a `BREAK` XO and change its properties using `ctl-E`.

2.9. The Heading Menu

The Heading menu contains additional Pstyles using for headings and captions, based on LaTeX:



2.10. Editing Tables

You can edit the contents of table cells as normal document text. You can also select an entire cell or a rectangular array of cells, callled a *sub-table*. XOE shows selected cells in inverse video.

- If you click an unselected cell in a table with mouse button 1, you select the entire cell. If you shift-click or shift-drag, you select a rectangular table.

- If you click in a selected cell, you start editing the text within that cell. `ESC` gets you back up a level and selects the entire cell.

- If cells are selected, `DELETE` deletes the contents of those cells. If an entire row is selected, `sh-DELETE` deletes that entire row. If an entire column is selected, `sh-DELETE` deletes the entire column. This also works for multiple selected rows and/or columns.

- Some table commands are single characters without `shift` or `ctl`. Press **r** to extend the current selection to one or more rows. Press **c** to extend the current selection to one or more columns.

  Need to add Table Menu.

- The `INSERT` key inserts a row or column before the currently selected row(s) or column(s). `sh-INSERT` inserts a row or column after the currently selected row(s) or column(s). If your keyboard does not have an `INSERT` key, press **i** for `INSERT` or **a** (append) for `sh-INSERT`.

- The commands in the Edit and Style menus generally do the proper things when table cells are selected. For example, a Style command changes the character style for all the selected cells, Copy copies the selected sub-table to the clipboard, and Paste copies the clipboard one or more times to the selected sub-table.

2.11. Editing Figures

Chapter 8 describes how to edit figures.

2.12. Miscellaneous Commands

The following commands are mostly for debugging and for compiling GalaxC programs. They will be replaced by or supplemented with menu commands some day. The term “file” refers to the file in the window that receives the command.

`ctl-W` Toggle the width of the document between wide format suitable for programming and narrow

format suitable for documents. *This will be replaced by a Page Layout dialog some day.*

F5 Redraw window without updating underlying data structures.

sh-F5 Rebuild a window's internal data structures and then redraw it. This is sometimes useful for recovering from internal errors.

The F6 options are described in detail in the document *Compiling and Running GalaxC Programs.*

F6 Compile file using GalaxC compiler. Show results in the Output window. It is generally a good idea to save the file before compiling since the GalaxC compiler could crash.

sh-F6 Compile and run file that is part of a multi-file program using the GalaxC compiler's Make capability.

ctl-F6 Run the code just generated by F6 or sh-F6. There is no "sandbox" or other protection, so running the code could easily crash XXICC. printf output goes to the Output window.

If the file contains tables (Chapter 7), XOE enters "show cell values" mode and shows the initial values of table cells. If you change a cell's value, XXICC recalculates the entire table like a spreadsheet. Press ESC to leave "show cell values" mode.

ctl-sh-F6 Toggle compiler debug options. This is primarily for debugging the GalaxC compiler itself. At some point we'll replace it with a dialog box.

F8 Print the contents of the window as PSI code to the Output window. This is for debugging XOE.

ctl-F8 Print the PSI code generated by F6 to the Output window.

sh-F8 Print the contents of the window as DXOs to the Output window. This is for debugging XOE.

ctl-sh-F8 Same as sh-F8 except that it also print DXO memory addresses.

F12 Toggle between displaying XREF symbols or their values.

sh-F12 Print the file's XREF symbols and their values to the Output window.

ctl-F12 Print all XREF symbols and their values to the Output window. This is for a multi-file document with XREFs, like this book. *Needs documentation.*

The Output window may have a limited number of lines so the first part of a long listing may scroll off.

2.13. Hints

- Ubuntu GNU/Linux 11.10 numeric keypad behavior: The usual behavior is to have unshifted numeric keypad keys behave as number keys if Num Lock is on, or as cursor keys if Num Lock is off. In the latter case, shifted numeric keypad keys behave as number keys. If you select Windows compatibility mode in System Settings: Keyboard Layout, shifted numeric keypad keys with Num Lock off behave as shifted cursor keys. Other versions of Ubuntu or other GNU/Linux distributions

may behave differently.

Chapter 3
The GalaxC Simplified Window Manager (G-SWIM)

> Our life is frittered away by detail... Simplify, simplify.
> *Henry David Thoreau*
>
> *Entia non sunt multiplicanda praeter necessitatem.*
> *William Occam*

XXICC recognizes the ubiquitous presence of window-based Graphical User Interfaces (GUIs) in modern interactive computing. It therefore tries to make window management as simple as possible for application programs by providing a simple, portable window manager as part of XXICC. Since its purpose is to provide window functions to GalaxC programs, it is called the *GalaxC Simplified Window Manager*, or **G-SWIM**.

Here are the principal features of G-SWIM:

1. G-SWIM is very simple and easy to use, much more so than standard window managers like Microsoft Win32 and MIT X Windows, yet serves most applications quite well. Function calls follow GalaxC's philosophy of matching notations to the problem domain, with function calls like "`draw line from z1 to z2`" instead of C calls like "`XDrawLine(display, drawable, gx, x1, y1, x2, y2)`".

2. G-SWIM runs on top of the machine's *underlying windows manager* (UWM). Galaxy applications run on the different hardware platforms without modification and automatically maintain the UWM's familiar characteristics. Currently G-SWIM has implementations for Microsoft Win32 and MIT X11, as well as G-SWIM functions for generating PDF.

3. Most windows managers require an application to be a stand-alone program with its own command loop that waits for and dispatches user input or system events. G-SWIM treats each application as a set of subroutines that are called by a common command loop that is part of G-SWIM.

   G-SWIM allows an arbitrary mix of application programs to be in memory at the same time, each occupying its own windows but otherwise sharing memory. G-SWIM directs user inputs to the appropriate application. In addition, applications can send messages to each other anonymously or directly. G-SWIM handles all these **events** by calling the appropriate application, which processes the event and returns to G-SWIM. While not true multitasking, the effect is the same provided that applications handle events quickly enough. We call this feature **quasi-multitasking**.

The remainder of this chapter (and the next) discuss how to use G-SWIM to create interactive graphical GalaxC applications. *[fn:* A **G-SWIM application** is a GalaxC program that uses G-SWIM to provide a GUI. The G-SWIM data structures, constants, and function calls available to the application programmer is called the **Application Programmer's Interface** (API). As long as you only use the API, your G-SWIM programs should port easily between machines.*]* It presumes no GUI programming experience, though you should be familiar with using GUIs and remember your coördinate geometry. G-SWIM programming can seem a bit daunting at first, but is actually quite simple in practice. The careful reader will find these chapters sufficient background to write complete interactive window-based applications with minimal effort.

G-SWIM is a work in progress and many features have yet to be implemented, such as images. We have concentrated on the subset needed to implement XXICC, especially XOE. Other features will be added when appropriate, with keeping the API as simple as possible a primary requirement.

3.1. Points and Rectangles

As with most window managers, the fundamental data types are 2-dimensional points and rectangles with integer coördinates. G-SWIM `Point` and `Rect` structures are based on Win32. They are defined in `gswimlib.gal`, which we will refer to simply as "`gswimlib`".

A `Point z` has two `int` coördinates `z.x` and `z.y` and is defined as:

```
typedef Point = struct(z) {int z.x, int z.y}
```

A `Point` value can be pushed on the stack as two `int`s or can be referenced as a `&Point` or `@Point` address. `z.x` and `z.y` are normally in pixel units. *[fn:* A **pixel** (picture element) is the smallest drawable unit on the screen. Each pixel has a single color or shade of gray. A screen is an array of pixels, e.g., 1400 x 900.*]*

To construct a `Point` value on the stack, use the notation "`[x, y]`", where `x` and `y` are `int`:

```
int arg {x, y},
def [x, y] = (x, y): Point
```

`gswimlib` also defines `Point` arithmetic operations such as `Point` addition and subtraction:

```
Point arg {u, v},
inline {-u}     = [-u.x, -u.y],
inline {u + v}  = [u.x + v.x, u.y + v.y],
inline {u - v}  = [u.x - v.x, u.y - v.y],
inline {u == v} = u.x == v.x and u.y == v.y,
inline {u != v} = u.x != v.x or  u.y != v.y,

inline min(u, v) = [min(u.x, v.x), min(u.y, v.y)],
inline max(u, v) = [max(u.x, v.x), max(u.y, v.y)],

fn |u| = u.x == 0? |u.y|: u.y == 0? |u.x|: sqrt (u.x*u.x + u.y*u.y),

int arg {&x, &y},
fn {[x, y] = u} = {x = u.x; y = u.y}
```

G-SWIM defines `min` and `max` of two `Points` to be diagonally opposite corners of the smallest rectangle that contains both `Points`. `|u|` is the Euclidian length of a vector from [0, 0] to `u`. The notation "`[x, y] = u`" provides a way to get both coördinates of a complicated expression `u` without assigning it to an intermediate variable. Most of these are defined as `inline` rather than `fn` or macro. `inline` saves function call overhead at the expense generating a bit more code, while preserving the property that each argument (which could be a complicated expression) is evaluated exactly once.

The only relational operators defined for `Points` are "`==`" and "`!=`".

A `Rect r` has two `Point` coördinates `r.z1` and `r.z2` and is defined as:

```
typedef Rect = struct(&r) {Point {r.z1, r.z2}}
```

It is drawn parallel to the *x* and *y* axes. To construct a `Rect` value on the stack, use the notation "`Rect[z1, z2]`", where `z1` and `z2` are `Point`s:

```
Point arg {z1, z2},
def Rect[z1, z2] = (z1, z2): Rect
```

You can access the individual coördinates of `r` using `r.z1.x`, `r.z1.y`, `r.z2.x`, and `r.z2.y`. In addition, `gswimlib` provides a set of macros to access these coördinates using alternate notations:

```
Rect arg *r,
def r.x1     = r.z1.x,
def r.y1     = r.z1.y,
def r.x2     = r.z2.x,
def r.y2     = r.z2.y,
def r.left   = r.z1.x,
def r.top    = r.z1.y,
def r.right  = r.z2.x,
def r.bottom = r.z2.y,
inline r.w   = r.z2.x - r.z1.x,      // Rect width.
inline r.h   = r.z2.y - r.z1.y       // Rect height.
```

These are only defined for a `Rect` pointer or variable, not for a `Rect` value on the stack.

In GalaxC, as in most window managers, `x` coördinates increase from left to right and `y` coördinates increase from top to bottom, which is flipped from standard mathematical coördinates. By convention, `r.z1` is the top left (upper left) coördinate and `r.z2` is the bottom right (lower right) coördinate which means `r.x1` ≤ `r.x2` and `r.y1` ≤ `r.y2`. This is reflected in the alternate coördinate names and results in non-negative `r.w` and `r.h`. However, this is not automatically checked so it is possible to create `Rects` that violate this convention, causing unpredictable behavior as different UWMs may handle unconventional `Rects` differently.

Another convention is that the left and top edges of the `Rect` are considered to be part of the `Rect`, but the right and bottom edges are not. That is, the `Rect` goes from `r.x1` up to but not including `r.x2`, so that the width is indeed `r.x2 - r.x1`. This becomes important when we want to determine if `Point z` is contained in `Rect r`, for example to select it with the mouse. Here is the `gswimlib` function "`z in r`" which computes this, assuming a conventional `Rect`:

```
Point arg *z, Rect arg *r,
fn z in r = z.x >= r.x1 and z.x < r.x2 and z.y >= r.y1 and z.y < r.y2
```

Note that the comparisons to `r.x2` and `r.y2` are strict inequalities, while the others allow equality. There is an alternate version of this function "`z on r`" which allows `z` to match the right and bottom edges:

```
Point arg *z, Rect arg *r,
fn z on r = z.x >= r.x1 and z.x <= r.x2 and z.y >= r.y1 and z.y <= r.y2
```

The "`z in r`" and "`z on r`" functions shown assume both `z` and `r` are variables or pointers. `gswimlib` includes alternate versions of these functions that allow `z` and/or `r` to be `Point` or `Rect` values. They have

the same notation and are transparent to the programmer. See `gswimlib` for details.

We also have functions to see if two `Rects` overlap (by at least one pixel) or touch (overlap or abut):

```
Rect arg {*r, *s},
fn r overlaps ref s = r.left < s.right and r.right > s.left and
                      r.top < s.bottom and r.bottom > s.top,
fn r touches ref s = r.left <= s.right and r.right >= s.left and
                     r.top <= s.bottom and r.bottom >= s.top
```

The only difference is whether comparisons include equality. There are alternate versions for `Rect` values.

We can also compute the union or intersection of two `Rects`. In this case, the first argument is always a `Rect` pointer or variable and is modified by the second argument which can be a pointer, variable, or value:

```
Rect arg {*r, *s},
fn {r = union s} =
{    // Enlarge Rect r with Rect s.
     r.left = min(r.left, s.left); r.right  = max(r.right,  s.right);
     r.top  = min(r.top,  s.top);  r.bottom = max(r.bottom, s.bottom);
},

// Compute intersection of Rect r with Rect s.
fn {r = intersect ref s} =
{    // Shrink Rect r with Rect s.
     // If they do not touch, r.left > r.right and/or r.top > r.bottom.
     r.left = max(r.left, s.left); r.right  = min(r.right,  s.right);
     r.top  = max(r.top,  s.top);  r.bottom = min(r.bottom, s.bottom);
}
```

Intersection is only well-defined if `Rects` `r` and `s` touch, or we end up with an unconventional `Rect`.

Finally, we have functions to compare `Rects` and to sort `Rect` coördinates:

```
Rect arg {*r, *s},
fn  {r == s} = r.z1 == s.z1 and r.z2 == s.z2,
def {r != s} = !(r == s)
fn sort r = Rect[[min(r.x1, r.x2), min(r.y1, r.y2)],
                 [max(r.x1, r.x2), max(r.y1, r.y2)]]
```

A sorted `Rect` is always conventional.

3.1.1 Pixels and Coördinates

This section discusses some low-level considerations that may be skipped on first reading.

Each UWM has its own conventions regarding coördinates and where pixels are. G-SWIM tries to hide these idiosyncrasies from the application programmer as much as possible.

All window managers treat the screen or a window on the screen as a two-dimensional array of pixels. The

(x, y) coördinates of a pixel typically refer to either the center of the pixel (X11) or to an invisible grid of zero-width lines between pixels (Win32, I think, and Apple QuickDraw). In the latter case, (x, y) refer to the upper-left corner of the pixel.

For drawing lines and outlines of rectangles, the center-of-pixel convention is more natural: you can think of drawing a line as stroking a one-pixel (or larger) pen from (x_1, y_1) to (x_2, y_2) along the line connecting the endpoints' centers. On the other hand, for filling rectangles and drawing text, it is more natural fill the region between invisible grid lines with a solid color, a fill pattern, or a character pixel map. *[fn:* A **pixel map** or **pixmap** is a rectangular array of pixels and may be on the screen, on a piece of paper, or in memory. To draw text, a window manager first converts a text font into an internal pixel map representing each character of the font. Then it copies the pixel map for each character in a string to the screen or to another internal pixmap. To print text or graphics, the window manager draws to an internal pixmap that represents (part of) a page. Then it transfers that pixmap to the actual device. A **bitmap** is a single-bit pixmap: each pixel can be either black or white.*]*

With G-SWIM, it doesn't matter which convention the UWM uses since the graphics functions are equivalent. For example, a rectangle from (x, y) to ($x+w$, $y+h$) represents a $w \times h$ pixel region whether it is filled or drawn as an outline. The filled region includes all pixels from (x, y) through ($x+w-1$, $y+h-1$), but does not include ($x+w$, $y+h$) since that would make it $w+1 \times h+1$. On the other hand, the outline does include ($x+w$, $y+h$) since we measure the dimensions of the rectangle from the centerline of the pen used to draw it.

Similarly, when drawing text with upper-left corner at (x, y) it doesn't matter whether (x, y) is the upper-left pixel or the invisible grid lines above and to the left of it: the text pixel map ends up at the same location.

If a line drawing pen is multiple pixels thick, it is centered around the invisible grid (if even thickness) or around pixel centers (if odd thickness).

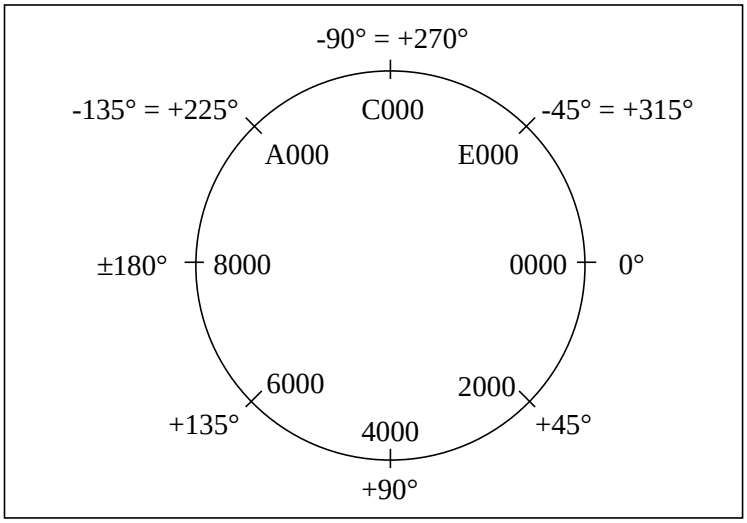
Both Win32 and X11 permit 0-thickness lines which are rendered with 1-pixel thickness. X11 draws 0-thinkness lines as quickly as possible and they may not be as nicely rendered as a 1-pixel line. G-SWIM normally uses 1 pixel as the default line thinkness.

Win32 has an unusual feature in that when you draw a line it does not draw the last pixel of the line. This is not a problem if you are drawing a closed polygon as a series of lines, since the beginning of each line segment takes care of the end of the last one. On the other hand, if the last line segment is dangling then it may be a pixel shorter than it should be. Win32 does this to avoid the situation where drawing a pixel twice may look strange. G-SWIM line drawing functions compensate for this.

3.1.2 Angles: Binary Angular Measurement

G-SWIM represents angles using **Binary Angular Measurement** (BAM, [Wiki 2]). BAM uses an n-bit unsigned or 2's complement number to represent an angle as a n-bit binary fraction of a circle. Angle 0° is 000..., +90° is 010..., ±180° is 100..., and +270° and -90° are both represented as 110... There is no way to represent ±360° or larger angles: they are always stored modulo 360°. All n-bit binary adds and subtracts automatically perform modulo 360° arithmetic, so there is no need to do it explicity as is the case if you represent angles in degrees or radians.

This diagram shows the 16-bit hexadecimal representations of various angles. Angles increase in the clockwise direction, which is more natural for a flipped coördinate system with x increasing to the right and y increasing down.



You can interpret BAMs as unsigned or 2's complement signed integers. If unsigned, BAMs represent angles from 0° up to but not including 360°. If signed, BAMs represent angles from -180° up to but not including 180°.

In most cases, G-SWIM uses 16-bit BAMs, called BAM-16. This gives you resolution of 1/65,536 of a circle, or 0.0055°. XXICC makes most BAM calculations using BAM-24 and represents angles in XO lists using BAM-15.

3.2. Colors

G-SWIM has two color representations. An `RGBcolor` is a 32-bit value of the form 00BB.GGRR where RR, GG, and BB are 8-bit red, green, and blue intensities. This is the same representation as a Win32 `COLORREF`. The high 8 bits are reserved. `gswimlib` defines `RGBcolor` as a subtype of `ulong` as follows:

```
typedef RGBcolor = subtype(ulong);

RGBcolor arg color,
def red(color)   = ubyte(color),
def green(color) = ubyte(color >> 8),
def blue(color)  = ubyte(color >> 16);

int arg {red, green, blue},
def RGBcolor(red, green, blue) = red | green << 8 | blue << 16: RGBcolor;
```

The `red`, `green`, and `blue` macros extract individual red, green, and blue components from an `RGBcolor`. The `RGBcolor` macro creates an `RGBcolor` from `red`, `green`, and `blue` `ints` which must have values between 0 and 255.

The actual display hardware may not be able to represent 24-bit colors. It may only support 8- or 16-bit color, with fewer bits per color. It could even be an e-reader with only 8 levels of gray and no color at all, or a monochrome printer. The display hardware may use a color look-up table, where a small index looks up an 18- or 24-bit color value in a hardware table.

G-SWIM applications should not have to worry about this, so G-SWIM provides a second color representation called `Icolor` (internal color). `Icolor` is a 32-bit *amorphous pointer*, described in [JFB

11: Programmer-Defined Types]. This means an application program does not know the representation of the data pointed to by `Icolor`, or the `Icolor` pointer could itself represent the color as an index into a color table (X11) or as an `RGBcolor` value (Win32). The representation of `Icolor` is only known to the G-SWIM layer and varies from one window manager to the next.

G-SWIM applications must allocate colors before they use them by calling:

```
create color rgb
```

where `rgb` is an `RGBcolor` value. "`create color`" returns an `Icolor` value which can then be used as a text color or for creating a pen or brush. The Win32 version of "`create color`" just returns its `RGBcolor` argument as an `Icolor` value. The X11 version calls a X11 function to allocate a new color and returns it as an index or pointer. If "`create color`" cannot allocate a color, it returns the nearest color or a default color.

When you no longer needed `Icolor i`, it is good manners to deallocate it by calling:

```
free i
```

so that the `Icolor` can be reallocated. All colors are deallocated when G-SWIM exits.

3.3. Pens and Brushes

When drawing a line or filling a shape, there are many options. A line has a color, a thickness, and a style (solid, dotted, dashed, etc.) A fill has a color, a pattern, or perhaps even an image. Including all possible options for each graphic function call would be very inefficient and would make GUI programs look awful.

So instead, window managers have a *device context* (Win32) or *graphics context* (X11) which is simply a data structure that contains the current values of all these properties. We will use "**GC**" to refer to both kinds of context. Functions that draw shapes use the current values in the GC, and the API provides additional functions to modify the GC. Most drawings and text reuse the same colors, styles, and fonts so the approach is generally more efficient than passing every possible argument.

X11 takes a very simple approach where each property -- e.g., foreground color or line style -- is changed individually. Win32 adds a second layer by defining **pens** and **brushes** which group together line drawing and area fill properties so that a whole group of properties can be changed simultaneously by selecting a pen or brush. *[fn:* TrollTech's Qt also uses this approach.*]* G-SWIM uses the latter approach, and automatically makes the appropriate X11 calls when selecting a different pen or brush.

To create a new pen, call the function:

```
create pen (style, th, color)
```

where `color` is the pen's `Icolor`, `th` is the pen thickness in pixels, and `style` is `SolidLine`, `DashedLine`, `DottedLine`, `DashDotLine`, or `DashDotDotLine`. "`create pen`" returns type `Pen`, which is an amorphous pointer. The Win32 G-SWIM calls Win32 function `CreatePen` which returns an `HPEN` object handle. The X11 G-SWIM allocates its own pen structure and returns a pointer to it. If G-SWIM cannot allocate a new pen, it returns the standard solid black pen (1 pixel thick).

When G-SWIM draws a dotted or dashed line, it does not draw anything in the gaps between dots and

dashes. Whatever pixels were there before (usually the background color or pattern) are unaltered.

There are also a set of pre-allocated standard pens which can be accessed using:

```
standard pen (id)
```

where `int id` is `StdBlackPen` or `StdWhitePen`.

To select a pen, call:

```
select pen
```

where `pen` is of type `Pen`. G-SWIM uses the selected pen until you select a different one.

When a non-standard `pen` is no longer needed it is good manners to deallocate it by calling:

```
free pen
```

so that the `Pen` can be reallocated. All pens are deallocated when G-SWIM exits. In general, do not deallocate standard pens. However, it is OK to attempt to deallocate the standard black pen that is returned by a failed call to `create pen`.

To create a new brush, call one of these functions [*to be implemented*]:

```
create brush (color)
create brush (color, pat)
```

where `color` is the brush's `Icolor` and `pat` is stipple pattern. If `pat` is missing G-SWIM assumes a solid `color` brush. "`create brush`" returns type `Brush`, which is an amorphous pointer. If G-SWIM cannot allocate a new brush, it returns the standard solid black brush.

As with pens, there is a set of pre-allocated standard brushes which can be accessed using:

```
standard brush (id)
```

where `int id` is `StdBlackBrush` or `StdWhiteBrush`, both of which are all solid.

To select or deallocate a brush, call:

```
select brush
free brush
```

In general, do not deallocate standard brushes. However, it is OK to attempt to deallocate the standard black brush that is returned by a failed call to `create brush`.

3.4. Drawing Functions

This section describes the functions for drawing shapes. If you are familiar with other window managers, you'll notice that the drawing functions do not explicitly identify which window is being drawn. Instead, G-SWIM has a global variable `SelWin` which points to the **selected window**. We will describe windows in

more detail in a later section.

All coördinates for drawing functions are relative to the upper-left corner of the *client* or *user* area of `SelWin`, i.e., the area enclosed by `SelWin`'s frame and below its header. The window coördinates of the upper-left corner are `[0, 0]`.

Some of the drawing functions use the *current position*, which is where to start drawing if the initial coördinates are not specified explicitly. The current position is internal to G-SWIM. Functions "`move to z`", "`line to z`", and "`draw line from z1 to z2`" update the current position to a well-defined value. Other drawing functions may change it to an unspecified value.

`move to z`

Move the **current position** to `Point z`. Do not draw anything. '`z`' may be a variable or a `Point` expression such as:

```
move to [100, 50]
move to (z + [1, 1])
```

These notations use the fact that GalaxC syntax considers the space before '`[`' or '`(`' to be significant.

`line to z`
`line thru z`

Draw a line from the current position to `Point z` using the selected pen, and then perhaps update the current position to `z`. As described earlier, Win32 does not draw the endpoint of a line whereas X11 and most other window managers do. To cover both cases efficiently, we have two ways to draw a line. "`line to z`" may or may not draw `z`: use it if you don't care whether `z` is drawn or not since it will be followed by another "`line to`" or will touch other graphics. Use "`line thru z`" if you want to make sure `z` is drawn: the Win32 G-SWIM draws a an extra stub at `z` to make sure `z` appears. "`line to z`" updates the current position to `z`. "`line thru z`" may change the current position to an unspecified value.

`draw line from z1 to z2`
`draw line from z1 thru z2`

Draw a line from `Point z1` to (or through) `Point z2` using the selected pen. "`draw line ... to z2`" updates the current position to `z2`. "`draw line ... thru z2`" may update the current position to an unspecified value. The `draw line` functions are equivalent to:

```
Point arg {z1, z2},
fn {draw line from z1 to z2}   = {move to z1; line to z2},
fn {draw line from z1 thru z2} = {move to z1; line thru z2}
```

As with "`line to`", `z1` and `z2` can be any `Point` expression, e.g., "`draw line from [50, 100] to (z - [20, 30])`".

`draw Hline from z to x2`
`draw Hline from z thru x2`
`draw Vline from z to y2`
`draw Vline from z thru y2`

Draw horizontal or vertical line from `[z.x, z.y]` to (or through) `[x2, z.y]` (if horizontal) or

`[z.x, y2]` (if vertical) using the selected pen, where `z` is a `Point` and both `x2` and `y2` are `int`. These functions may take advantage of faster routines for drawing horizontal or vertical lines if the window manager has them. It also handles "`to`" versus "`thru`" more explicitly: "`to`" never includes the endpoint, while "`thru`" always includes it. X11 adds or subtracts 1 from the endpoint coördinate to implement the "`to`" form -- this is easy for a horizontal or vertical line, but difficult for a diagonal which is why the more general "`draw line from z1 to z2`" makes drawing `z2` optional. The current position becomes undefined.

`draw point at z`
Draw a single point at `z` using the selected pen's color.

`draw r`
`draw rect from z1 to z2`
Draw a rectangle outline using the selected pen given diagonally opposite points `r.z1` and `r.z2` (or `Points z1` and `z2`) . '`r`' is a `Rect` value, variable, or pointer. The functions are equivalent to:

```
move to r.z1; line to [r.x2, r.y1]; line to r.z2;
line to [r.x1, r.y2]; line to z1;
```

'`r`' may be an unconventional `Rect`.

`fill r`
`fill rect from z1 to z2`
Fill a rectangular region using the selected brush given diagonally opposite points `r.z1` and `r.z2` (or `Points z1` and `z2`). '`r`' is a `Rect` value, variable, or pointer. Do not fill the right and bottom edges at `r.x2` and `r.y2`. The behavior of "`fill rect`" is only predictable if `r` is a conventional `Rect`, i.e., its coördinates must be sorted.

`draw circle (z, rad)`
Draw a circle outline using the selected pen given a center at `Point z` and `int` radius `rad`.

`fill circle (z, rad)`
Fill a circle outline using the selected brush given a center at `Point z` and `int` radius `rad`.

```
draw arc r (init, len) from z1 to z2
draw arc r (init, len) from z1 thru z2
draw arc r (init, len)
draw arc (z, rad) from z1 to z2
draw arc (z, rad) from z1 thru z2
```
Draw a circular arc from `Point z1` to (or through) `Point z2`, using the selected pen. The arc is part of the circle defined by bounding square `r`, which may be a `Rect` value, variable, or pointer. '`r`' must sorted and have equal width and height. "`init`" is the initial arc direction at `z1`, represented as a BAM-24 `int` (§3.1.2). "`len`" is the arc length, also a BAM-24 `int` but with additional sign bits. If `len` is positive, the arc is clockwise. If `len` is negative, the arc is counter-clockwise and has length = `-len`.

`Points z1` and `z2` are optional since the arc is defined without them. If present, the implementation should line the arc up exactly with `z1` and `z2` rather than have exact values for `init` and `len`.

The last two "`draw arc`" functions draw the arc using `Point` center `z` and `int` radius `rad`. The arc direction is clockwise. The arc should line up exactly with `z1` and `z2`: it's OK for the center `z` to move along the line from `z` to `z1` so the arc matches. Radius `rad` is always `|z1 - z|`, but that can be expensive to calculate so G-SWIM lets the application calculate it ahead of time and store it.

Some window managers may choose to implement arcs (and circles too, for that matter) as Bézier curves.

3.5. Fonts and Drawing Text

G-SWIM assumes the UWM supports scalable fonts, such as TrueType. These are built into Win32 and are added to X11 using the Xft interface to the FreeType rasterizer. Currently, G-SWIM only supports fonts that read from left to right.

To create a font, call:

```
create font (faceName, size, options)
```

where `string faceName` is the name of the font, `int size` is the desired font height in pixels not including internal leading (defined below), and `int options` are font options such as `BoldFont` and `ItalicFont` which can be `OR`ed together as "`BoldFont | ItalicFont`". `faceName` depends on which fonts are available on your system, but G-SWIM always recognizes these fonts:

1. `"Monospaced"`, a fixed-width font usually implemented as `Courier` or `Courier New`.
2. `"Times"`, a proportional serif font usually implemented as **Times Roman** or **Times New Roman**.
3. `"Helvetic"`, a proportional sans-serif "Swiss" font usually implemented as Helvetica or Arial.
4. `"Symbol"`, a standard font with Greek and mathematical characters.

G-SWIM maps the quoted names to implementation-specific names. It is reasonable to expect that the first three fonts are available on every system in normal, **bold**, *italic*, and ***bold italic*** styles since Microsoft at one time made them available for free as "net fonts". Symbol was not part of this set of free fonts, but is a standard Windows font and is often present in GNU/Linux distributions, or can be purchased for $20 or so. PDF considers all four fonts including their alternate styles to be "standard fonts" which can be used by all PDF documents, so all the fonts should be available on any system that has a PDF reader.

You can create a rotated font by `OR`ing a clockwise BAM-16 angle (§3.1.2) into `options`. Only the top 14 bits of the angle are used -- the LSbs are `BoldFont` and `ItalicFont`. If you don't know if the LSbs of your angle are 00, `AND` the angle with `RotateFontMask`. `gswimlib` defines constants for ±90° and 180° angles: `RotateFontCW`, `RotateFont180`, and `RotateFontCCW`.

"`create font`" returns type `IntFont` (internal font), which is an amorphous pointer. If G-SWIM cannot allocate a new font, it returns the default *system font*. The system font can also be accessed using:

```
standard font (id)
```

where `int id` is `StdSystemFont`.

As with pens and brushes, to select or deallocate `font` call:

```
select font
```

```
free font
```

Attempts to deallocate the system font are ignored.

3.5.1 Font Dimensions

A font has a number of important dimensions which you many need for an advanced GUI, e.g., a WYSIWYG document editor or a graphics editor that includes text objects. Here are the most important dimensions:

- The *ascent* of a character is how much it rises above the baseline. Many lower case letters -- e.g., ‘a’, ‘c’, and ‘o’ -- have a small ascent while other lower-case and all capital letters have a large ascent: ‘d’, ‘h’, and ‘M’. Capital letters with accents may have an even larger ascent: ‘Á’ and ‘Ñ’. Some special characters have a small or zero ascent: ‘.’ and ‘_’. The ascent of a font is the maximum ascent of any character in the font.

- The *descent* of a character is how much it falls below the baseline. The characters with the most descent are usually lower-case *descender* characters like ‘j’, ‘y’, but also include some accented characters like ‘ç’ and some punctuation marks like ‘;’. The descent of a font is the maximum descent of any character in the font.

- *Leading* (prounounced “ledding”) is extra vertical space inserted between lines to improve readability. The name refers to thin strips of lead from days of movable type. *Internal leading* refers to the leading immediately above a line and is often used for capital letter accents -- ‘Á’ and ‘Ñ’ -- so that lines with accented capitals have the same baseline-to-baseline distance (*vertical pitch*) as lines without them. G-SWIM includes internal leading in a font’s ascent.

  External leading is extra leading beyond the internal leading and is optional. It’s a recommendation from the font designer which can be followed or ignored by a program using the font.

- The *height* of a character is the sum of its ascent and its descent. The height of a font is the sum of maximum ascent plus the maximum descent and may or may not include internal leading. When you specify a font in G-SWIM -- e.g., “create an 18 pixel high font” -- do not include internal leading. On the other hand, when you ask G-SWIM for the actual height of a font it does include internal leading.

- If you draw a character at coördinates *z*, the *origin* of that character is the location in its pixmap that corresponds to *z*. For example, *z* could be the upper left corner of the pixmap. However, the most useful location (and the one chosen by G-SWIM) is the left edge of the character at its baseline. This is because if you are drawing a sequence of characters from different fonts you generally want them to align at the baseline.

- The *width* of a character is usually width of its visible pixels plus any additional space between it and the previous and following characters. However, this becomes tricky when a character is designed to overlay adjacent characters, like the ‘*f* ’ in “*difficult*”. This is called *kerning.* Since we normally draw a sequence of adjacent characters, this definition is much more useful:

  > The width of a character is the number of pixels to add to the character’s origin in the horizontal direction to get to the next character’s origin.

  A character like ‘*f* ’ may have pixels to the left of its origin and pixels to the right of the next

character's origin as well.

High quality document formatters may perform advanced kerning to squeeze together pairs of letters in text like "WAY" so it looks less like "W A Y". G-SWIM does not support this feature at this time.

Once you have created `font`, you can obtain its dimensions by calling:

```
get font metrics (width, height, ascent, descent, iLead, xLead)
```

"`get font metrics`", which only works for unrotated fonts, sets `&ushort` reference arguments `width` - `xLead` to `font`'s dimensions as follows:

- `width`: Average character width in pixels. If a monospaced font, `width` is the width of all characters.
- `height`: Font height in pixels, including internal leading. Recall that "`create font`" requests font height without internal leading. *(This may change.)*
- `ascent`: Maximum font ascent above the base line in pixels, including internal leading.
- `descent`: Maximum font descent below the base line in pixels. In all cases `height` = `ascent` + `descent`.
- `iLead`: Internal leading in pixels, included in `height` and `ascent`.
- `xLead`: External leading in pixels, not included in `height`.

You may pass `NULL` for any of these arguments if you don't need the value.

3.5.2 Text Dimensions

If the selected font is unrotated, you can get the size of a text string using that font by calling:

```
size of text(n)
size of text
width of text(n)
width of text
height of text(n)
height of text
```

where `text` is a `string` and `int n` is the length of the string in bytes. If `n` is missing, G-SWIM calculates `|text|`. "`size of`" returns a `Point` equal to `[width, height]`. The others return an `int` width or height.

If the selected font is rotated ±90°, you can gets its width by calling:

```
rotwidth of text(n)
rotwidth of text
```

At some point we will generalize this to work with any angle font. For now, these constraints work fine with XOE since it does formatting using unrotated fonts, and rotates text later by ±90° if needed.

The width of `text` is the sum of its character widths, where the width of a character is the number of pixels to add to the character's origin in the horizontal direction to get to the next character's origin. The width

returned by "`size of text`" or "`width of text`" where `text` is a multiple character string should be the same as the sum of the widths obtained by calling "`size of text`" or "`width of text`" for each individual string. However, this does depend on the UWM. We have found empirically that this is true for Win32 and X11 (with and without Xft) provided G-SWIM uses screen coördinates and avoids character-by-character kerning. Future UWMs should be careful to maintain this property.

The height of `text` equals the height of its font and G-SWIM ignores the characters of `text`.

If you want to get the dimensions of `text` for the characters actually drawn, call:

```
glyph box of text(n)
glyph box of text
```

Both return a `Rect` with the dimensions of the smallest box containing the text *glyphs* (their visual representations) with [0, 0] at the first character's origin. The glyph box may be wider than "`width of text`" if a character extends past its origin or the following character's origin. The glyph box is generally shorter than "`height of text`", which uses the selected font's maximum ascent and descent. At the present time, "`glyph box`" only works for unrotated fonts.

G-SWIM has a specialized text width function to format unrotated text from left to right with automatic word wrap. Its definition looks like:

```
string arg text, int arg {n, margin, limit, &m, @psw},
fn {width of text (n, margin, limit, m, psw)} = ...
```

This function calculates the width of `text`, with length `n`. "`margin`" is a soft right margin where `text` should break and `limit` is a hard right margin which `text` must not exceed (`margin` ≤ `limit`). Both `margin` and `limit` are relative to the initial *x* coördinate of `text`. '`m`' returns the number of characters of `text` that can be used. If the entire `text` fits within `margin`, return `m` = `n`. If `text` width exceeds `margin`, then you may need to break text: `m` returns the number of characters that fit within `limit` and `psw[0:m]` is set to an `int` array of *partial string widths* where `psw[i]` = width of first `i` characters of `text`. "psw" must have at least `n+1` elements. In all cases return the width of the first `m` characters of `text`.

3.5.3 Text Color

To select text color, call the function:

```
text color = i
```

where `i` is an `Icolor` value.

3.5.4 Drawing Text

To draw text at `Point z` using the selected font and text color, call:

```
draw text(n) at z
draw text at z
```

where `text` is a `string` and `int n` is the length of the string in bytes. If `n` is missing, G-SWIM calculates `|text| = strlen(text)`. '`z`' is the left end of `text`'s base line. "`text`" should consist entirely of printable characters defined by the font. In most cases this means ASCII, ISO-8859-1 Latin-1, or Windows-1252 and does not include any control characters.

G-SWIM draws text using the solid `Icolor` selected by the last call to "`text color = ...`", by default black. As with dotted and dashed lines, G-SWIM only draws the forcground pixels and leaves the background pixels alone. The UWM generally uses anti-aliasing and/or TrueType to create high-quality characters from a scalable font.

At some point we may to add a function to draw outlined text, where the outline uses the current pen, and filled text where the text uses the current brush.

3.6. Window Terminology

G-SWIM follows the conventional window terminology popularized by Microsoft Windows, X Windows, Apple Macintosh, and others. G-SWIM manages a collection of rectangular **windows** on a bit-mapped display screen also called the **desktop**, since it resembles the surface of a desk covered with papers. Windows may overlap, just as papers may overlap on a desk. A window is either *fully exposed* (no other windows cover it), *fully obscured* (it is completely covered by one or more windows), or *partially obscured* (part is visible, other parts are obscured by other windows).

The main part of a window is its rectangular *client area*, which displays graphical and text generated by the application. The *title bar* is an optional rectangle above the client area containing the window's title -- e.g., the name of a document being edited in the window -- and perhaps some platform-specific controls for resizing and/or closing the window. Typically you can move a window by pressing the mouse in the title bar and dragging. If the window does not have a title bar, e.g., a pop-up dialog or menu, you may be able to move the window by pressing an inactive location and dragging, or it may not be possible to move the window. A window usually has a *border* enclosing the client are and title bar. On some platforms, you can drag the border with the mouse to resize the window.

Some UWMs allow additional window parts, such as scroll bars and menus. G-SWIM does not support these itself, but applications may do so in the client area.

The UWM determines when windows need to be redrawn, e.g., if:

1. A new window is created.
2. The window is resized to make it taller or wider.
3. A (partially) obscured window becomes exposed because a window on top of it is moved or deleted.
4. The title of the window changes.
5. The application has changed the data contained within the window.

The UWM takes care of redrawing the title bar and border, and automatically calls the application program to redraw the client area if changed, passing an *update rectangle* which indicates what subset of the client area to redraw. The application program stores its data in some internal form, e.g., an array of ASCII characters for a text editor . When called, the application *refreshes* (part of) the client area by calling G-SWIM drawing functions described earlier. For example, a text editor makes multiple calls to "`draw text`" to refresh the changed part of the window.

The UWM automatically clips graphics operations so that an application can simply refresh all its internal data and not worry about whether it is partially obscured. However, if there is a lot of internal data it is

generally much more efficient to consider the update rectangle and limit drawing to data that could intersect that rectangle.

Most user input to a window is in the form of *mouse* and *keyboard events*. A mouse event occurs whenever the user moves the mouse, or presses or release a mouse button in the window. A keyboard event occurs whenever the user presses a key in the window. The UWM processes some of these events itself (such as mouse events in the title bar or border of the window) and calls the application program to handle the rest of them.

Mouse events are usually sent to the window that the mouse is in. However, it is possible to *capture* the mouse temporarily so that all mouse events go to a particular window.

Keyboard events go to the window that has the *keyboard focus*, sometimes called the *top window* or *selected window* in UWM documentation. However, G-SWIM uses "selected window" to mean the window pointed to by global variable `SelWin`. Most UWMs require you to click the mouse in a window to give it the keyboard focus. Others may automatically assign keyboard focus to the window that contains the mouse.

A modern mouse often has a *mouse wheel*, which is used for scrolling windows up and down. While it would make sense that the mouse wheel should scroll the window it is in, some UWMs (notatbly Win32) always send mouse wheel events to the keyboard focus window. This can cause unexpected inputs, but unfortunately is the convention.

A *child window* is a window that is created as part of another window, called the *parent window*. Some examples of child windows include:

1. *Pop-up dialogs* requesting user input before the parent's application can continue.
2. Menus, either *drop-down menus* at the top of a window or *pop-up menus* that appear at the mouse.
3. Scroll bars.

A child window may be attached to its parent so that it moves with its parent, or it may be free to move independently. Scroll bars and drop-down menus are usually attached and pop-up menus and dialogs are usually free. An attached child window may be clipped to its parent's client area.

Child windows may be nested to any desired depth. The parents, grandparents, great-grandparents, etc. of a window are referred to collectively as its *ancestors*. Each window can have at most one parent window. A window with no parents is called a *primary window* and usually has a title bar and border.

In G-SWIM, a child window captures all the input events that would otherwise go to its parent or other ancestor, e.g.,

- If an application brings up a pop-up dialog, all mouse clicks in its parent window go to the pop-up window instead.
- To edit a text field, a dialog may bring up a text editing child window that exactly covers the text field. All keytrokes and mouse clicks go to that text editor until text editing is complete.

Some UWMs use windows for almost any rectangular screen object, such as menu and dialog buttons, and all parts of a scroll bar. G-SWIM, on the other hand, typically uses windows only for large objects with many subcomponents such as a complete menu with many buttons or a complete dialog, and uses XXICC Objects for small objects.

G-SWIM has two coördinate systems, both in pixels. *Screen coördinates* are for the whole screen, and go from `[0, 0]` at the upper-left corner down to but not including `[ScreenWidth, ScreenHeight]` at the lower right, where `ScreenWidth` and `ScreenHeight` are global variables containing the dimensions of the screen. *Window coördinates* are for the client area of a window, and go from `[0, 0]` at the upper-left corner of the window's client area down to but not including `[win.width, win.height]` at the bottom right of the client area.

3.7. G-SWIM Window Structure

G-SWIM stores the properties of a window in a `GwinStruct`, which is defined in `gswimlib`:

```
typedef GwinStruct = struct(&win)
{
    @GwinStruct win.next,        // Next allocated window.
    ushort  win.options,         // Window options.
    ushort  win.state,           // Window state.
    @GwinStruct win.parent,      // Parent of this window.
    @GwinStruct win.captor,      // Send input events to this window.
    fnptr   win.RefreshEvent,    // Refresh Event handler.
    fnptr   win.KeyboardEvent,   // Keyboard Event handler.
    fnptr   win.MouseEvent,      // Mouse Event handler.
    fnptr   win.WindowEvent,     // Multi-purpose window event handler.
    short   win.screenX,         // Window client area screen X coördinate.
    short   win.screenY,         // Window client area screen Y coördinate.
    ushort  win.width,           // Window width in pixels.
    ushort  win.height,          // Window height in pixels.
    ushort  win.minW,            // Minimum window width in pixels.
    ushort  win.minH,            // Minimum window height in pixels.
    ushort  win.maxW,            // Maximum window width in pixels, or 0.
    ushort  win.maxH,            // Maximum window height in pixels, or 0.
    string  win.title,           // Window title (path name).
    ushort  win.request,         // G-SWIM event requested by window.
    ushort  win.spare1,          // Spare field.
    ulong   win.spare2,          // Spare field.
    Brush   win.BGbrush,         // Background brush pointer or handle.
    ubyte   win.priv[0x20]       // Reserved for underlying window manager.
};
def Gwindow = @GwinStruct;
```

Here are the detailed descriptions of each field:

`win.next`

All allocated windows are linked by their `win.next` fields. This field is generally only used by G-SWIM and should never be changed by application programs. Child windows always precede their parents in the list of allocated windows.

`win.options`

Options for this window expressed as bitwise `OR`ed symbolic constants. The MSB is reserved for

application programs. The LSB is for G-SWIM and is encoded as follows:

`0x07`	`WindowTypeMask`	G-SWIM window type bits, with values:
`0x00`	`PrimaryWindow`	Primary window with title bar and sizing frame.
`0x01`	`PopupWindow`	Pop-up window, preferably without a border.
`0x02`	`AttChildWindow`	Child window attached to and enclosed in parent.
`0x03`	`AttToPrimaryWin`	Child window attached to and enclosed in primary window.

G-SWIM moves an `AttChildWindow` with its parent window, and clips the child to its parent's boundary. An `AttToPrimaryWin` is similar, except that it is attached to and clipped by its primary window ancestor instead of its immediate parent. `AttToPrimaryWin` is used by cascading drop-down menus.

These options are subject to change. Specifically, I'm planning to distinguish between dialog and menu pop-up windows, and need to decide how to deal with drop-down menus that are too long for the client area.

`win.state`

Current state of this window expressed as bitwise `OR`ed symbolic constants. The MSB is reserved for application programs. The LSB is for G-SWIM. State is not normally used by G-SWIM applications -- look at `gswimlib` to find out more.

`win.parent`

Parent window of `win`, or `NULL` if `win` is a primary window. Application programs should not modify this field.

`win.captor`

Points to the child window of `win` that should receive all input events to `win`, or `NULL` if `win` has no children. Application programs should not modify this field.

`win.RefreshEvent`

Points to a GalaxC function which G-SWIM calls to to redraw (part of) `win`'s client area. If none is specified, G-SWIM uses a dummy function that does nothing. The refresh event function has the form:

```
Rect arg @updateRect,
void fn refresh event (updateRect) = ...
```

where `updateRect` points to a `Rect` that indicates what part of the client to redraw, viz., from window coördinates `updateRect.z1` down to but not including `updateRect.z2`. If `win.BGbrush` is not `NULL`, G-SWIM automatically erases `updateRect` by filling it with `win.BGbrush` before calling .

`win.KeyboardEvent`

Points to a GalaxC function which G-SWIM calls when the user presses a key if `win` is the keyboard focus window. If none is specified, G-SWIM uses a dummy function that does nothing. The keyboard event function has the form:

```
char arg key, ulong arg shifts,
```

```
void fn keystroke event (key, shifts) = ...
```

where `key` is the (usually) ASCII code of the key just pressed, and `shifts` is a combination of bitwise `OR`ed symbolic constants that encode keyboard modifiers:

`0x80`	`VKEY`	"`key`" is a *virtual key* encoding a special character.
`0x40`	`MWHEEL`	"`key`" encodes mouse wheel movement.
`0x20`	`ALT_KEY`	An `Alt` key was pressed with a virtual key.
`0x08`	`CTRL`	A `Ctrl` key was pressed with a virtual key.
`0x04`	`SHIFT`	A `Shift` key was pressed with a virtual key.
`0x03`	*`kk`*	Encode `key` using character set *`kk`*.

You can think of `shifts` as a binary number `VWA0.CSkk` where `V` = `VKEY`, `W` = `WHEEL`, `A` = `ALT`, `C` = `CTRL`, and `S` = `SHIFT`.

Virtual keys are special characters that usually do not have ASCII codes, such a cursor movement keys (`UP`, `DOWN`, `LEFT`, `RIGHT`, `HOME`, `END`), page scroll keys (`PRIOR`, `NEXT`), function keys `F0` through `F24`, and other special characters (`INSERT`, `DELETE`, `BS`, `TAB`, `CR`). They may be modifed by an combination of `CTRL` and `SHIFT`. Control characters are also treated as virtual keys with `key` equal to '`A`' through '`Z`', and `shifts` equal to `VKEY | CTRL` or `VKEY | CTRL | SHIFT`. G-SWIM encodes virtual keys using Win32 conventions: see `gswimlib`.

G-SWIM encodes a mouse wheel event with `key` equal to an `sbyte` value equal to number number of lines to scroll, usually either +3 or –3.

If the `VKEY` bit is 0, `key` encodes a normal (printing) character. If *`kk`* is 0, `key` is encoded in ASCII, Latin-1, or Microsoft-1252. If *`kk`* is 1, `key` is encoded using the Symbol font's character set. An application program may ignore *`kk`*, or use it to switch to Symbol automatically when the user enters Symbol character like '≠'.

G-SWIM automatically handles accented characters using the standard method of preceding them with the appropriate accent while holding down `Ctrl`. Valid combinations are shown in Section 4.7.

`win.MouseEvent`

Points to a GalaxC function which G-SWIM calls when there is a mouse event in `win`. If none is specified, G-SWIM uses a dummy function that does nothing. Mouse events include mouse movement and changes to the mouse buttons. The mouse event function has the form:

```
int arg {x, y}, ulong arg shifts,
void fn mouse event(x, y, shifts) = ...
```

where `x` and `y` are the window coördinates of the mouse and `shifts` is a combination of bitwise `OR`ed symbolic constants that encode mouse button changes, the current state of the mouse buttons, and shift keys:

`0x8000`	`MOUSEWIN_EXIT`	Released button outside mouse window (*may omit*).
`0x2000`	`LBUTTONUP`	Left button up event: `LBUTTON` is now 0 (released).
`0x1000`	`LBUTTONDN`	Left button down event: `LBUTTON` is now 1 (pressed).
`0x0800`	`MBUTTONUP`	Middle button up event: `MBUTTON` is now 0 (released).

0x0400	MBUTTONDN	Middle button down event: MBUTTON is now 1 (pressed).
0x0200	RBUTTONUP	Right button up event: RBUTTON is now 0 (released).
0x0100	RBUTTONDN	Right button down event: RBUTTON is now 1 (pressed).
0x0080	HIT_TEST	Win32 "hit test" for moving a window with no title bar.
0x0020	ALT_KEY	An Alt key is currently pressed.
0x0010	MBUTTON	The middle mouse button is currently pressed.
0x0008	CTRL	A Ctrl key is currently pressed.
0x0004	SHIFT	A Shift key is currently pressed.
0x0002	RBUTTON	The right mouse button is currently pressed.
0x0001	LBUTTON	The left mouse button is currently pressed.

LBUTTON, MBUTTON, and RBUTTON are the *current state* of the left, middle, and right mouse buttons. *x*BUTTONDN is a *button event*: it means that button *x* has changed from up to down (has just been pressed) and the *x*BUTTON bit is now 1. Similarly, *x*BUTTONUP means that button *x* has changed from down to up (has just been released) and *x*BUTTON is now 0. *[fn:* G-SWIM currently treats double-clicks as two pairs of normal clicks. However, we may in the future set both *x*BUTTONDN and *x*BUTTONUP to mean the second *x*BUTTONDN of a double-click, so give priority to *x*BUTTONDN in mouse event handlers.*]*

SHIFT, CTRL, and ALT_KEY are simply the combination of shift keys currently pressed.

HIT_TEST is a special Win32 feature that G-SWIM uses for moving a dialogs that doesn't have a title bar. HIT_TEST provides a way for a Win32 application to tell Win32 explicitly what the mouse is currently over. In our case, if the mouse is in a dialog and it is not over a button or other clickable object, the "`dialog mouse event`" function returns the value MOVE_EVENT to tell G-SWIM to tell Win32 that the mouse is over the title bar so that you can drag the title bar with the mouse. HIT_TEST always appears by itself in shifts; most mouse event handlers discard HIT_TEST events.

When you press a mouse button in a window, G-SWIM makes that window the *mouse window* and automatically captures the mouse, sending all mouse events to the mouse window until you release the button. This is the same as "automatic grab" in X11. If the mouse is captured, window coördinates x and y may be negative (to the left of or above the client area) or may exceed the dimensions of the client area (to the right of or below the client area).

MOUSEWIN_EXIT tells the mouse window that the mouse button has been released outside the window. This is used when dragging a shape within a G-SWIM window: if you release the mouse outside the window, you usually want to abort the dragging operation and either return the shape to its original position or leave it at its last position. If your application does not drag shapes, you can ignore MOUSEWIN_EXIT. MOUSEWIN_EXIT always appears by itself in shifts and does not provide useful values for x and y.

We may discard MOUSEWIN_EXIT *at some point and replace it with another mechanism. Also, automatic capture and* MOUSEWIN_EXIT *are only well-defined for a single button. We need to decide the right way to ignore multiple buttons.*

win.WindowEvent

Points to a GalaxC function which G-SWIM calls to report various changes in the state of win, such as creating a window and changing its size. If none is specified, G-SWIM uses a dummy function

that does nothing. The window event function has the form:

```
int arg action, ulong arg {arg1, arg2, arg3},
void fn window event (action, arg1, arg2, arg3) = ...
```

where `action` is a symbolic constant indicating which window event and `arg1`-`arg3` are `action`-specific arguments. Simple applications do not need window events, so we defer details to Section 4.2.

`win.screenX`, `win.screenY`
: Screen coördinates of the upper left corner of `win`'s client area. These can be used to convert the window coördinates `x` and `y` of a mouse event to screen coördinates by adding `[win.screenX, win.screenY]`. They may also be used to convert coördinates in one child window to those of another child of the same primary window.

`win.width`, `win.height`
: Width and height of `win`'s client area in pixels. This can be used to determine what part of an application's data is currently visible on the screen. If the user resizes `win`, G-SWIM automatically updates `win.width` and `win.height`, and reports the change by calling `win.WindowEvent(RESIZE_EVENT)`.

`win.minW`, `win.minH`, `win.maxW`, `win.maxH`
: Minimum and maximum width and height of `win`'s client area in pixels. The UWM uses this data to control how much a window can be resized. For example, a text editor window may have a minimum size, or a dialog may be forced to a fixed size so the user cannot resize it.

  If `minW` or `minH` is 0, then `win` has no minimum dimension and can be shrunk to whatever minimum size the UWM allows. If `maxW` or `maxH` is 0, then `win` has no maximum dimension and can be enlarged to whatever maximum size the UWM allows.

`win.title`
: Current title of `win`, normally displayed in the title bar. Always use the "`window title = ...`" to set the window title rather settting `win.title` directly. Note that `win.title` is a `string` pointer: the application is reponsible for allocating storage containing the actual text of the title.

`win.request`
: This field allows a G-SWIM application to respond to a keyboard, mouse, or window event by requesting futher action from G-SWIM. For details see Section 4.3.

`win.BGbrush`
: This field points to the `Brush` which G-SWIM uses to erase the background before calling a G-SWIM application's `win.RefreshEvent` function. It is normally set to a solid white brush or a gray brush for dialogs. If `NULL`, G-SWIM does not erase the background. This is appropriate if the application is painting all pixels, e.g., it is displaying an image.

`win.priv`
: This field contains private data for a specific UWM, usually pointers to the UWM's own window and GC data structures. Application programs should never change these values.

G-SWIM applications may append additional fields to `GwinStruct` by creating a `substruct` of it.

3.7.1 Calling Event Functions

Before calling `win.RefreshEvent`, `win.KeyboardEvent`, `win.MouseEvent`, or `win.WindowEvent`, G-SWIM saves the current value of `SelWin` and then sets `SelWin` to `win` so that application code knows which window is receiving the event and so that drawing functions already have `SelWin` set to the correct window. In addition, except for `win.WindowEvent` and `HIT_TEST`, G-SWIM selects the system font, black text color, and the standard solid black pen and brush. This way the application can draw something useful (and consistent) without having to select a font, text color, pen, and brush explicitly.

G-SWIM restores `SelWin` to its previous value after the event function returns.

G-SWIM does not support multi-threaded applications at this time: each event function must complete before the next one is called for consistent operation.

3.8. Creating a G-SWIM Window

To create a G-SWIM window call `create window`, which is defined as:

```
Gwindow arg {win, parent}, string arg title,
int arg {options, x, y, w, h}, Brush arg BGbrush,
fn create window (win, title, options, x, y, w, h, BGbrush, parent)
```

It has the following arguments:

`win`	Pointer to a pre-allocated initialized `GwinStruct`.
`title`	Window title, assigned to `win.title`.
`options`	Window options, assigned to `win.options`.
`x, y`	Initial coördinates of window. If negative, use defaults. If primary or pop-up window, `x` and `y` are screen coördinates. If attached child window, `x` and `y` are relative to parent.
`w, h`	Initial window dimensions. If negative, use defaults.
`BGbrush`	Background brush or NULL for transparent background..
`parent`	Pointer to parent window, or `NULL` if primary window.

Before calling "`create window`", allocate a `GwinStruct` `win` and initalize all its fields, setting `win.RefreshEvent` to the application's refresh function and most other fields to 0 or `NULL`. "`create window`" sets `NULL` event functions like `win.KeyboardEvent` to a dummy function that does nothing. "`create window`" overwrites `win.title` and `win.options` with the `title` and `options` arguments. If you want G-SWIM to set window position and size automatically, pass negative numbers in `x`, `y`, `w`, and/or `h`.

There is also a simplified version of "`create window`" if you want the usual defaults:

```
Gwindow arg {win, parent}, string arg title,
fn create window (win, title)
```

This is equivalent to calling "`create window (win, title, PrimaryWindow, -1, -1, -1, -1,`

`standard brush (StdWhiteBrush), NULL)`".

As part of the window creation process, "`create window`" calls `win.WindowEvent` with `CREATE_EVENT` to perform any creation-time initialization, `MOVE_EVENT` after computing initial position, and `RESIZE_EVENT` after computing initial window dimensions. "`create window`" also calls `win.RefreshEvent` to draw the new window's contents.

If `create window` is successful, it returns `TRUE` with `SelWin` set to `win`. If anything went wrong, it returns `FALSE`. The caller must deallocate any storage created for `win` and `title`.

This is the only window function we need to get started with some examples. Other window functions are described in Chapter 4.

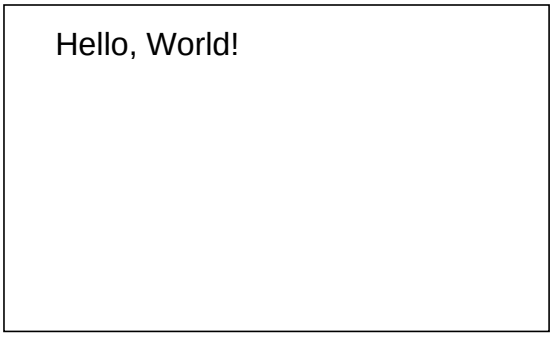
3.9. Sample Window Applications

This section preents some sample G-SWIM application programs. The first one creates a window with the familiar text "Hello, World":

```
include GswimAPI;

// G-SWIM calls this function whenever it wishes to redraw the window.
Rect arg @updateRect,
fn sample refresh (updateRect) =
{
    draw "Hello, world!" at [30, 30];
};

// Allocate gwin as a global variable.
GwinStruct var gwin;
memset(&gwin, 0, |GwinStruct|);       // Initialize gwin fields to 0/NULL.
// Tell G-SWIM to call "sample refresh" to redraw the window.
gwin.RefreshEvent = fnptr {sample refresh (NULL)};
// Create window for gwin.
create window (&gwin, "Sample Application");
```

First we define the refresh function "`sample refresh`", which simply draws the `string` "`Hello, world!`" at window coördinates `[30, 30]`. Then we need to allocate space for a `GwinStruct`. Since this example has only one window, we just allocate `gwin` as a global `GwinStruct` variable. We then clear `gwin` to 0 and `NULL` by calling `memset`, and then set `gwin.RefreshEvent` to the `fnptr` address of the "`sample refresh`" function.

The last step calls "`create window`", which creates a nice window on the screen containing "`Hello, World!`" using the system font, looking something like:

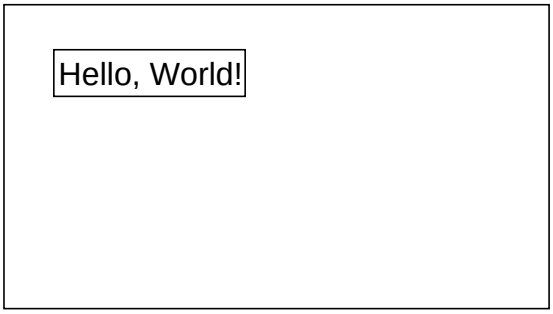


Note that there is no explicit GalaxC code to close the window. This means that G-SWIM will close the window in the default way when the user clicks the `CLOSE` button or whatever the UWM provides.

We have drawn "`Hello, World!`" at `[30, 30]`, but it looks much closer to the top of the window than to the left edge of the window. This is because `[30, 30]` is the baseline of "`Hello, World!`", not its upper corner. If we want to be more careful how we place "`Hello, World!`", we can get its ascent and descent by calling `get font metrics`. Here is a more sophisticated version of `sample refresh` which also draws a rectangle around "`Hello, World!`":

```
Rect arg @updateRect,
fn sample refresh2 (updateRect) =
{
    ushort var {ascent, descent};
    get font metrics (NULL, NULL, ascent, descent, NULL, NULL);
    var w = width of "Hello, world!";
    var h = ascent+descent;
    var z = [30, 30];
    draw "Hello, world!" at (z + [0, ascent]);
    draw rect from (z - [3, 3]) to (z + [w + 3, h + 3]);
};
```

Here we call `get font metrics` to get the `ascent` and `descent` values for the selected font, and also call `width of "Hello, world!"` to get its width. Given these values, we can draw "`Hello, World!`" relative to its upper left corner by drawing it at `z + [0, ascent]`, and then draw a box around it with an extra 3 pixels on each side.



Centering "`Hello, World!`" in the window turns out to be quite simple, as shown by this modified example:

```
Rect arg @updateRect,
```

```
fn sample refresh3 (updateRect) =
{
    ushort var {ascent, descent};
    get font metrics (NULL, NULL, ascent, descent, NULL, NULL);
    var h = ascent+descent;
    var w = width of "Hello, world!";
    var z = [(SelWin.width - w)/2, (SelWin.height - h)/2];
    draw "Hello, world!" at (z + [0, ascent]);
    draw rect from (z - [3, 3]) to (z + [w + 3, h + 3]);
};
```

The only difference (shown in **bold**) is that we calculate the upper left corner `z` of "`Hello, World!`" using window dimensions `[SelWin.width, SelWin.height]`.

Here's a fun example. In this case, instead of using the system font we create a giant **Times** font, making it 1/3 the height of the window.

```
IntFont var BigFont;

// Create a big font with height h, freeing the previous version.
int arg h,
fn create BigFont (h) =
{
    if BigFont != NULL then free BigFont;
    BigFont = create font ("Times", h, 0);
    select BigFont;
};

Rect arg @updateRect,
fn sample refresh4 (updateRect) =
{
    create BigFont (SelWin.height/3);
    ushort var {ascent, descent};
    get font metrics (NULL, NULL, ascent, descent, NULL, NULL);
    var h = ascent+descent;
    var w = width of "Hello, world!";
    var z = [(SelWin.width - w)/2, (SelWin.height - h)/2];
    draw "Hello, world!" at (z + [0, ascent]);
    draw rect from (z - [3, 3]) to (z + [w + 3, h + 3]);
};

BigFont = NULL;
```

In this example we re-create `BigFont` each time we call `sample refresh4`, freeing the old version. This is not a good way to do this, since reallocating a font can involve quite a bit of computation. A more sophisticated version would use `RESIZE_EVENT` to detect when a window has changed size and only recreate `BigFont` when that happens. Also, since we have not told G-SWIM how to close the window explicitly using `WindowEvent`, G-SWIM will not deallocate `BigFont`.

Now let's add some mouse input, specifically free-hand drawing whenever the left mouse button is pressed. To do this we add a mouse event function:

```
Point var prev;
int arg {x, y}, ulong arg shifts,
fn sample mouse (shifts, x, y) =            // Change order.
{
    if (shifts & LBUTTONDN) != 0 then prev = [x, y] else
    if (shifts & LBUTTON) != 0
    then {draw line from prev to [x, y]; prev = [x, y]};
};

BigFont = NULL;
GwinStruct var gwin;
memset(&gwin, 0, |GwinStruct|);     // Initialize gwin fields.
gwin.RefreshEvent = fnptr {sample refresh4 (NULL)};
gwin.MouseEvent   = fnptr {sample mouse (0, 0, 0)};
create window (&gwin, "Sample Application");
```

Global variable `prev` stores the mouse `[x, y]` coördinates from the previous call to `sample mouse`. When the user first presses the left mouse button (`LBUTTONDN`), we initialize `prev` to the current coördinates. Then for each mouse movement event with the left button pressed (`LBUTTON`), we draw a line from `prev` to the current coördinates and update `prev`.

This is a simple example. It does not store the free-hand drawing anywhere, so if G-SWIM refreshes the window the free-hand drawing is erased. Also, drawing graphics inside a mouse event function is generally a bad idea: in almost all cases you should use the refresh function to update graphics from the application's data structures.

Here's another simple program which displays the characters of a font.

```
include GswimAPI;

IntFont var SampleFont;      // Font to display.

def dx = 30;                 // Horizontal and vertical character pitch.
def dy = 30;

Rect arg @updateRect,
fn showfont refresh (updateRect) =
{   // Print all characters of font starting with ASCII space.  Do not print control characters.
    select SampleFont;
    var ch = ' '; int var {x, y};
    // Print 14 rows with 16 characters per row.
    for y = dy thru (dy*14) by dy do
    {
        for x = dx thru (dx*16) by dx do {draw (&ch)(1) at [x, y]; ch++};
    };
};

// Mouse handler: quit if left button is pressed.
int arg {x, y}, ulong arg shifts,
```

```
fn showfont mouse (x, y, shifts) =
{
    if (shifts & LBUTTONDN) != 0 then SelWin.request = DESTROY_EVENT
};

GwinStruct var gwin;
SampleFont = create font ("Symbol", 25, 0);      // Display Symbol font.
memset(&gwin, 0, |GwinStruct|);
gwin.RefreshEvent = fnptr {showfont refresh (NULL)};
gwin.MouseEvent = fnptr {showfont mouse (0, 0, 0)};
create window (&gwin, "Show Font");
```

3.10. Summary

This chapter has introduced the fundamentals of GUI programming using G-SWIM. This will be enough for many useful applications. More sophisticated GUIs will need the advanced G-SWIM functions and events described in the next chapter.

Chapter 4
Advanced G-SWIM

This chapter contains additional G-SWIM information needed for advanced applications.

4.1. Invalidating Regions

In §3.7, we saw that each window has a refresh function `win.RefreshEvent` which G-SWIM calls whenever it needs to refresh (part of) a window. Sometimes the UWM tells G-SWIM that a window needs refreshing, e.g., when part of the window is exposed because an overlapping window is moved or deleted. On the other hand, if the application data changes and should be redrawn, the UWM has no idea that this should happen unless the application tells it.

In general, a GUI application should always make changes to a data structure and then invalidate the changed part of the window so that G-SWIM calls `win.RefreshEvent` to draw the changes. This ensures that the window is consistent with the application's data structures, which is not guaranteed if the application tries to be clever and perform drawing functions without using `RefreshEvent`. It's also usually less work, since each application needs a `RefreshEvent` function anyway.

G-SWIM provides the following functions for invalidating and redrawing `SelWin`:

`invalidate r`
`invalidate rect from z1 to z2`

Invalidate a rectangular regions given diagonally opposite points `r.z1` and `r.z2` (or `Points z1` and `z2`). '`r`' is a `Rect` value, variable, or pointer. Do not invalidate the right and bottom edges at `r.x2` and `r.y2`. The behavior of "`invalidate rect`" is only predictable if `r` is a conventional `Rect`, i.e., its coördinates must be sorted. All coördinates are relative to the upper left corner of the window, i.e., they are *client* or *user* coördinates.

The UWM usually does not redraw the region immediately. Instead, it keeps track of multiple invalidations and then make one or more calls to `SelWin.RefreshEvent` to show the changes.

`invalidate window`

Invalidate the entire client area of `SelWin`. This is equivalent to calling "`invalidate rect from [0, 0] to [SelWin.width, SelWin.height]`".

`clip to r`

G-SWIM sets the clipping rectangle before calling `RefreshEvent`, which is usually enough for most applications. You can restruct drawing to a smaller region by calling `clip to r`, where `r` is a `Rect` variable or pointer in client coördinates.

`scroll window (dx, dy)`

Scroll the contents of `SelWin` by horizontal offset `dx` and vertical offset `dy`, both in pixels. Negative `dx` scrolls the contents left and positive `dx` scrolls right. Negative `dy` scrolls the contents up and positive `dy` scrolls down. The UWM copies pixels if it can and generates `RefreshEvent` calls to fill in the rest. If there are invalidated regions that have not been refreshed yet, the UWM modifies their coördinates by `dx` and `dy`. "`scroll window`" is used for word processors, figure editors, and other applications where only part of the data is visible in the window.

`update window`

Immediately refresh any invalidated regions of `SelWin` instead of waiting until the UWM has nothing better to do. "`update window`" is rarely needed since the automatic UWM redrawing is usually good enough. "`update window`" should be used with care, since it can result in a lot of redundant processing.

4.2. Window Events

As we saw in §3.7, each window has a `win.WindowEvent` function to report various state changes. Each `WindowEvent` function has the form:

```
int arg action, ulong arg {arg1, arg2, arg3},
void fn window event(action, arg1, arg2, arg3) = ...
```

Here are the possible values of `action`, called **window events**:

`CREATE_EVENT`

The "`create window`" function calls `WindowEvent(CREATE_EVENT)` to allow the application to allocate additional structures. `CREATE_EVENT` occurs before `RESIZE_EVENT` and `MOVE_EVENT`.

`CLOSE_EVENT`

G-SWIM has received a request to close this window, e.g., the user has clicked the window's `CLOSE` button. However, the application can check with the user before closing a window that has unsaved changes (for example). Before calling `WindowEvent(CLOSE_EVENT)`, G-SWIM sets `SelWin.request` to `DESTROY_EVENT`. If the application wants to prevent closing the window, it sets `SelWin.request` to 0. If it leaves `SelWin.request` equal to `DESTROY_EVENT`, G-SWIM will finish closing the window.

`DESTROY_EVENT`

G-SWIM is closing `SelWin` and the application cannot stop this from happening. `DESTROY_EVENT` tells the application to deallocate any storage it has allocated for `SelWin`, including `SelWin` itself.

`RESIZE_EVENT`

The UWM has changed the size of `SelWin`, with updated values in `SelWin.width` and `SelWin.height`. It will be calling `RefreshEvent` soon to draw newly-exposed regions. The application may need to update its data structures if they are affected by window size.

`MOVE_EVENT`

The UWM has moved `SelWin`, with updated values in `SelWin.screenX` and `SelWin.screenY`.

`BLINK_EVENT`

This is used to control a blinking cursor in `SelWin`. If `arg1` = `0`, erase the cursor. If `arg1` = `1`, draw it. If `arg1` = `2`, toggle it.

`OPEN_EVENT`, `SAVE_EVENT`

These events tell the application to create an Open File or Save File dialog box. They are not originated by the UWM, but an application like XOE may pass an `OPEN_EVENT` or `SAVE_EVENT` through G-SWIM.

`FIND_EVENT`, `REPLACE_EVENT`, `REPLALL_EVENT`

These events tell the application to find text `arg1`, replace text `arg1` with `arg2`, or replace all instances of text `arg1` with `arg2`. They are not originated by the UWM, but an application like XOE may pass them through G-SWIM to implement Find and Replace dialogs.

`SCROLL_EVENT`

This event tell the application to scroll to a origin `arg1` = `x0` or `y0` according to options in `arg2` which are interpreted like the `shifts` argument in a `KeyboardEvent`. `SCROLL_EVENT` is not originated by the UWM, but an application like XOE may pass it through G-SWIM to implement scroll bars.

`PRINT_EVENT`, `PRINTSEL_EVENT`

The Win32 version of G-SWIM uses these events to communicate with the Win32 Printer dialog. `PRINT_EVENT` tells the application to print with `arg1` = printer device context, `arg2` = first page (0 to print selected text), and `arg3` = last page (32,000) to print all pages or selected text.

`HOTWIN_EVENT`

XOE uses `HOTWIN_EVENT` to detect when the mouse has been near the edge of a window long enough to expose the menu bar or scroll bars. See the "`HotWin delay`" function in §4.5.

`GET_CB_EVENT`, `PUT_CB_EVENT`, `EMPTY_CB_EVENT`, `PASTE_EVENT`

These events are for implementing the Clipboard. See §4.6.

`APPL_SPEC_EVENT`

This is the first `EVENT` code available for application-specific use. Each application has its own events and can number them independently.

Very simple applications -- such as the ones in §3.9 -- can use the default `WindowEvent` function, which ignores all window events. In this case `CLOSE_EVENT` always results in a `DESTROY_EVENT`, and the application should not have any dynamically-allocated structures since it won't be able to use `DESTROY_EVENT` to deallocate them.

4.3. Window Requests

Applications may also use the window events listed in §4.2 to ask G-SWIM to take some action after a keyboard or mouse event. The application does this by setting `SelWin.request` (which is normally 0) to one of the window event `action` values. This is called passing a **window request** back to G-SWIM.

For example, if a XOE user enters `ctl-P`, G-SWIM calls `win.KeyboardEvent` which points to XOE's keyboard event handler. The handler determines that `ctl-P` is the keyboard equivalent of the Print command from the File menu, and needs to tell G-SWIM to open a UWM Print dialog box to select a printer and which pages to print. `KeyboardEvent` returns this window request by setting `SelWin.request` to either `PRINT_EVENT` or `PRINTSEL_EVENT`. `PRINTSEL_EVENT` is used if there is a non-null selection and affects the default Print dialog settings.

Window requests are handled by C function `EventRequest` in `gswim.h`. Here the most common actions:

`OPEN_EVENT`, `SAVE_EVENT`

The application has received the keyboard equivalent for the Open or Save As commands from the

File menu. `KeyboardEvent` sets `SelWin.request` to `OPEN_EVENT` or `SAVE_EVENT`. G-SWIM calls `SelWin.WindowRequest(OPEN_EVENT` or `SAVE_EVENT)` to create an Open File or Save File dialog box.

`DESTROY_EVENT`
: The application wants G-SWIM to close and deallocate `SelWin` immediately, along with all of `SelWin`'s captor windows (§3.7). G-SWIM calls `SelWin.WindowRequest(DESTROY_EVENT)` so the application can deallocate data associated with `SelWin`, and does this for each captor as well.

`CLOSE_EVENT`
: The application wants G-SWIM to close `SelWin`, but only if the user agrees. G-SWIM calls `SelWin.WindowRequest(CLOSE_EVENT)` with `SelWin.request` = `DESTROY_EVENT`. If `SelWin.request` is still `DESTROY_EVENT` when `WindowRequest` returns, G-SWIM finishes closing the window.

G-SWIM looks at `SelWin.request` immediately after calling `SelWin.KeyboardEvent` or `SelWin.MouseEvent` and processes non-zero requests in `EventRequest`. A request may result in one or more subsequent requests, e.g., `CLOSE_REQUEST` may produce `DESTROY_REQUEST`. `EventRequest` loops until it has handled all requests for `SelWin`.

On the other hand, G-SWIM does *not* usually check for requests after calling `SelWin.WindowEvent` except in `EventRequest`.

The curious reader may wonder why G-SWIM goes to so much trouble when closing windows. The problem is that there may be quite a few data structures allocated for each window, at the very least a `GwinStruct` and a UWM-specific window structure. As always with dynamically-allocated structures, it is crucial that you do not deallocate until you are completely finished with the structure, and that can be tricky with windows particularly when there is one or more levels of captor windows, which must be closed in the correct order.

G-SWIM solves this problem having applications ask G-SWIM to close all windows, so it can close them in the correct order and call the application with `DESTROY_EVENTs` to tell it when to deallocate structures.

4.4. Miscellaneous Window Functions

Here are some advanced window functions that do not fit any particular category. In all cases, `win` is a `Gwindow`.

`win title = text`
: This function call -- which has the syntax of an assignment -- changes `win`'s title to `string` argument `text` and also sets `win.title` = `text`. The caller is responsible for providing storage for the characters of `text`: this function only copies the pointer.

```
char arg key, ulong arg shifts,
int arg {x, y, action}, ulong arg {arg1, arg2, arg3},
keyboard event (win, key, shifts)
mouse event (win, x, y, shifts)
window event (win, action, arg1, arg2, arg3)
```
: Call another window's `win.KeyboardEvent`, `win.MouseEvent`, or `win.WindowEvent`, saving

and restoring the current value of `SelWin`. Call `EventRequest` after calling the event function to process `win.request` if non-zero. These functions are used to send messages to other windows in the form of events. For example, a child window may want to send certain keystrokes to its parent window.

`request win destroy (defer)`

Request destroying `win` by setting `win.request` = `DESTROY_EVENT`, and call `EventRequest` unless `Boolean defer` is set. Set `defer` if `win` needs to stay around temporarily because `win` has a child window, but we want to destroy `win` as soon as the child is closed.

`request win close`

Request closing `win` by sending window event `CLOSE_EVENT` with `win.request` = `DESTROY_EVENT` so that G-SWIM destroys `win` if the user agrees.

`begin text sizing`
`end text sizing`

These rarely-used functions are needed when calling "`size of text`" and other text dimension functions if `SelWin` is not associated with an open window. For example, XOE uses "`begin text sizing`" when initializing default document margins to a multiple of a standard font's character width. Call "`begin text sizing`" before calling the text dimension functions and "`end text sizing`" when you are finished. The token "`text`" in the function patterns is a literal, not a `string` pointer.

`enlarge window frame (options)`

Enlarge window frame by one or more pixels depending on frame options, passed as `int options`. With some UWMs, a dialog or other window looks better with an extra pixel on the right and bottom edges, while others are better without it. Return the extra width as an `int`.

4.5. Mouse Functions

`create mouse (w, h, z, shape, mask)`

Create a mouse symbol (bit map) given `int` dimensions `w` (width) and `h` (height), which are normally 16 x 16, 24 x 24, or at most 32 x 32 pixels. '`w`' must be 8, 16, 24, or 32. `Point z` is the mouse's "hot spot" relative to the upper left corner of the pixel map. The (x, y) coördinates passed to a mouse event are the client coördinates of the mouse symbol's hot spot. For example, an arrow symbol's hot spot is at the tip of the arrow, whereas the hot spot of cross-hairs is where they intersect.

`shape` and `mask` are two `char` arrays containing the mouse bitmap in row major order, with `w`/8 `chars` per row. Bits are left-to-right in MSb-to-LSb order; bytes are in left-to-right order by increasing array index. "`shape`" is the mouse symbol with bit values 1 = black and 0 = white, assuming black on white graphics. "`mask`" indicates which bits of `shape` should be drawn, where 1 = opaque and 0 = transparent. "`mask`" is usually the same as `shape`, but expanded by one pixel in each direction.

`create mouse` returns an amorphous pointer of type `IntMouse`.

`mouse = m`

This function call -- which looks like an assignment -- sets `SelWin`'s mouse to `IntMouse m`.

`free m`

Deallocate `IntMouse m`. It should not be assigned to any window. Typically, an application creates all the mouse symbols it needs at initialization time and does not deallocate any until the application ends.

`capture mouse`
`release mouse`

"`capture mouse`" temporarily captures the mouse so that all mouse events go to `SelWin` instead of the window the mouse is currently in. An application should capture the mouse for as short a time as possible and then call `release mouse` to restore normal operation. XOE uses `capture mouse` for scroll bars and menus.

In most cases, `capture mouse` is not needed because G-SWIM automatically captures the mouse while a mouse button is pressed.

`HotWin delay = x`

This function call -- which looks like an assignment -- is used for detecting when the mouse has been moved to a particular area of the window (called a *hot zone*) and has stayed there longer than `x` milliseconds. XOE uses this to detect when the mouse is at the edge of the window to expose the menu bar or a scroll bar.

If `x` > 0, G-SWIM will call `SelXOE.WindowEvent` with `action` = `HOTWIN_EVENT` in approximately `x` milliseconds. (The resolution of `x` may be much coarser than 1 msec, but should be better than 0.1 seconds.) If `x` = 0, cancel any pending `HOTWIN_EVENT`.

4.6. Clipboard Functions

G-SWIM provides support for the UWM clipboard for transferring data between applications. The Win32 clipboard is quite different from X11, so G-SWIM tries to provide a simple way to handle both.

The clipboard is designed to handle many different kinds of data, including text and graphics. However, applications may only support a subset of standard data formats, e.g., some applications can only transfer plain ASCII text whereas others can transfer formatted text and various kinds of graphics. At the present time, XOE can only transfer formatted text and graphics between XOE windows, but it can transfer plain text between any windows.

Here is a simplified view of how the clipboard works, using XOE as an example. When the user issues a Copy or Cut command from the File menu, XOE copies the selected data to the clipboard. When the user issues a Paste command, XOE copies the clipboard data into the document being edited.

There are two complications: first, XOE can copy either formatted data or plain text, but does not know which clipboard format is going to be needed. The safest thing is to copy both formats, which takes extra time. Second, the user may issues any number of Copy or Cut commands before any application issues a Paste command, so the effort made to extract the selected data and copy it to the clipboard (perhaps in multiple formats) may be wasted.

G-SWIM supports *delayed rendering* where XOE does not copy data to the clipboard until it is actually requested by XOE or another application. Instead, XOE *advertises* that it has data ready to copy and which formats are available. When another (or the same) application needs the data, it sends a message to XOE asking it to copy the data to the clipboard, telling it which format. This prevents unnecessary copying and

formatting. See [CP 99] for details.

Delayed rendering adds an interesting complication: if the XOE window containing the advertised data is about to close or if the data is about to be deleted, XOE must immediately render the data in all formats. Copying selected objects in a figure also requires XOE to render immediately, since it doesn't have a way to preserve which objects are selected.

Here are the G-SWIM variables, constants, functions, and window events needed for the clipboard. The X11 G-SWIM needs work -- the author was unable to find clear documentation of how X11 handles the clipboard so some features may not work properly.

`TEXTclipFormat`

> This `ulong` variable contains a unique 32-bit code to identify the plain ASCII text format. For Win32, it is a numeric constant `CF_TEXT = 1`. For X11, it is the atom `XA_STRING`.

`ClipboardCRLF`

> This `Boolean` constant indicates whether application programs should convert `LF` to `CR+LF` when copying plain ASCII text to the clipboard. It is `TRUE` for Win32 and `FALSE` for X11.

`create clipboard format (s)`

> Create new clipboard format given `string s`, and return a unique `ulong` code for it. The clipboard format must be unique across all applications. XXICC defines the clipboard format `"XXICC_OBJECT_LIST"`.

`acquire clipboard (format1)`
`acquire clipboard (format1, format2)`
`acquire clipboard (formats)`

> Acquire the clipboard for `SelWin`, and advertise which clipboard formats we can generate. If there are just one or two formats, pass them as `ulong` format codes `format1` and `format2`. If there are more codes, pass `ulong` array `formats`, which is terminated with `0`. If the clipboard is currently held by another application, the UWM makes it release the clipboard. Return `TRUE` if the clipboard was acquired successfully (should always happen) or `FALSE` if there is an unexpected problem.
>
> If there is delayed rendering, the application must remember what data it should copy to the clipboard when requested to do so. If there is no delayed rendering, the application copies the data immediately.

`begin clipboard write (format1, requestor, n)`

> Prepare to copy at most `int n` bytes of data to the clipboard using `ulong format1`. Allocate a global memory buffer with at least `n` bytes. "`requestor`" is an `@unknown` pointer to a UWM-specific control variable, and should point to an array with at least one `ulong`.
>
> "`begin clipboard write`" returns a `vstring` pointer to the data area of a global buffer, or `NULL` if it cannot allocate the buffer. If successful, the caller copies data to the the buffer and calls "`end clipboard write`".

`end clipboard write (buf, format1, requestor, n, dsize, openClose)`

> Finish writing `int n` bytes to the clipboard. Arguments should have the same values as the "`begin clipboard write`" call: `buf` should be the value returned by the call, '`n`' must be less than or

equal to the original value, and `format1` and `requestor` should be the same. If `buf` is `NULL`, do nothing.

"`dsize`" is the clipboard's data size in bits, and is either 8, 16, or 32. X11 uses this for changing byte order when transferring data. Set "`dsize`" to 8 for most data formats.

"`openClose`" is for Win32 and controls opening and closing the clipboard before or after writing to it. See `GET_CB_EVENT` and `PUT_CB_EVENT`.

`GET_CB_EVENT`

This window event is for delayed rendering. It tells the application program that someone wants a copy of the advertised data, with `arg1` = format and `arg2` pointing to the `requestor` control variable. The application program should call "`begin clipboard write`", copy the advertised data, and call "`end clipboard write`" with `openClose` = `0`, since Win32 automatically opens and closes the clipboard for `GET_CB_EVENT`. The application which generated `GET_CB_EVENT` can then copy the data from the clipboard.

`PUT_CB_EVENT`

This window event is for delayed rendering. It tells the application program to copy all advertised formats to the clipboard. The application program should call "`begin clipboard write`", copy the advertised data, and call "`end clipboard write`" for each advertised format.

Win32 requires that "`end clipboard write`" open and close the clipboard when responding to `PUT_CB_EVENT`. If you are copying multiple formats, set `openClose` = `1` for the first call, `openClose` = `2` for the last call, and `openClose` = `0` for any others. If there is just one call, set `openClose` = `3`.

`EMPTY_CB_EVENT`

This window event is for delayed rendering. It tells the application program that clipboard data is no longer needed, so you no longer need to remember what data you advertised. It is mostly used when another application acquires the clipboard.

`PASTE_EVENT`

When an application, e.g., XOE, wishes to paste the current clipboard -- say, in response to a `ctl-V` or `ctl-sh-V` command -- it sets `SelWin.request` = `PASTE_EVENT` and G-SWIM variable `RequestClipFormat` equal to the desired format, and returns to G-SWIM. G-SWIM then gets the current clipboard data in a UWM-specific way and calls `SelXOE.WindowEvent(PASTE_EVENT)` with `arg1` = format, `arg2` = buffer containing clipboard data, `arg3` = length of data in clipboard, and `SelXOE.request` = 0.

`WindowEvent` should attempt to paste the data into the application's data structures. If this is not successful, `WindowEvent` can try again with a different format by setting `SelWin.request` = `PASTE_EVENT` and `RequestClipFormat` to the new format. If the paste was successful or the application doesn't want to try again, it leaves `SelWin.request` = 0.

At the present time, clipboard handling is tricky, especially since Win32 and X11 are so different. See the G-SWIM and XOE source code for details.

4.7. Typing Accented Characters

G-SWIM keyboard input supports the Latin-1 and Microcosft-1252 codings for accented letters and other special characters using the standard method of preceding them with the appropriate accent while holding down a `Control` key. By including this capability in G-SWIM, application programs do not have to repeat this functionality and G-SWIM provides a consistent user interface for all applications.

Here are the combinations that are currently defined, including some for the Symbol character set:

	ctl-'	ctl-`	ctl-^	ctl-:	ctl-~
a	á	à	â	ä	ã
e	é	è	ê	ë	
i	í	ì	î	ï	
o	ó	ò	ô	ö	õ
u	ú	ù	û	ü	
n, y	ý			ÿ	ñ
A	Á	À	Â	Ä	Ã
E	É	È	Ê	Ë	
I	Í	Ì	Î	Ï	
O	Ó	Ò	Ô	Ö	Õ
U	Ú	Ù	Û	Ü	
N, Y	Ý			Ÿ	Ñ

	a	A	C	d	D	P	R	S	T	@
ctl-@	å	Å	©	†	‡	¶	®	§	™	°

	a	o	A	O	s
ctl-&	æ	œ	Æ	Œ	ß

	s	S	z	Z
ctl-^	š	Š	ž	Ž

	c	C	,	"
ctl-,	ç	Ç	¸	„

	`	'	"	d	D
ctl-`	‘		“		
ctl-'		’	”	ð	Ð

	o	p	O	P	=
ctl-/	ø	þ	Ø	Þ	≠

	<	>	=	+
ctl-_	≤	≥	≡	±

4.8. G-SWIM File Structure

In its present form, G-SWIM is partly written in C and partly in GalaxC. Each UWM has its own version of G-SWIM, but they present a common API to GalaxC programs. Here are the C components:

`gswim.h`
: This is a header file containing declarations common to all G-SWIM versions. Unlike most `.h` files, it also includes several functions which are the same for all versions. Some `gswim.h` structures have variants for different UWMs: these are defined in *xxx*`gswim.c` before it includes `gswim.h`.

xxx`gswim.c`
: This is the main part of G-SWIM which implements most API functionality such as creating windows and translating UWM events into G-SWIM events. There is a version of *xxx*`gswim.c` for each UWM, e.g., `w32gswim.c` for Win32 and `x11gswim.c` for X11.

Here are the GalaxC components:

`gswimlib.gal`
: This defines G-SWIM types and constants which are the same for all UWMs. It also defines `Point` and `Rect` operations from §3.1.

xxx`int.gal`
: This file contains UWM-specfic internals such as `inline` `cdecl` or `stdcall` declarations for calling UWM library functions and *xxx*`gswim.c`. There is a version of *xxx*`int.gal` for each UWM, e.g., `win32int.gal` for Win32 and `x11int.gal` for X11. G-SWIM applications should not call *xxx*`int.gal` or else they will not be portable between UWMs.

xxx`api.gal`
: This file defines most of the G-SWIM API functions. The function patterns are the same in all versions of *xxx*`api.gal`, but they are implemented by different calls to *xxx*`int.gal`. There is a version of *xxx*`api.gal` for each UWM, e.g., `win32api.gal` for Win32 and `x11api.gal` for X11. G-SWIM defines `GswimAPI` to be the `string` "*xxx*`api.gal`" for the UWM you are using.

Chapter 5
XXICC Objects

So far we have only considered textual GalaxC programs, which look more or less like C programs augmented with some document formatting features such as subscripts, superscripts, flexible fonts, and mathematical symbols. You can also add graphics to GalaxC programs if that is a better way to specify functionality. Here are some examples:

- Dialogs: You can include a dialog in graphical form instead of coding it textually or using a separate "resource editor" and awkward symbolic linkages. This can make GUI programs easier to understand and maintain.

- Tables: You can specify functional behavior in spreadsheet-like tabular form. While most spreadsheet tools require you to specify all behavior in tabular form, GalaxsC allows a program to mix tables and conventional program text. *This is an experimental feature in the current implementation.*

- Figures: GalaxC allows you to incorporate figures in program text. A simple use of this is to illustrate comments with explanatory figures, eliminating the need for a separate document or character-based figures. Figures can also have meaning, such as logic or flow diagrams. *This is also experimental. It is intended primarily to support future hardware/software codesign using XXICC.*

XXICC's goal is to let a programmer select whatever representation is best suited to each task, rather than requiring programmers to encode everything into one textual representation for the convenience of the compiler [JFB 92b]. This should reduce program complexity, making programs easier to write, debug, and maintain.

But isn't is maddeningly complex to analyze graphics? Isn't scanning and parsing text hard enough? Paradoxically, analyzing graphical XXICC constructs is actually *simpler* than text because they are stored and edited in forms that are already parsed.

Specifically, graphics is represented as a tree of XXICC objects (*XOs* or *XOBJs*), where an XO is either a *leaf* or a *subtree*. A leaf is a primitive such as a line, rectangle, circle, arc or text. A subtree (*XOST*) is a collection of XOs such as a table row, entire table, menu, dialog, or figure.

Internally, XOs are represented using PSI instructions. A leaf consists of instructions to specify leaf properties such as a rectangle's dimensions, followed the leaf's operation such as `RECT` or `TEXT`. An XOST consists of instructions to specify XOST properties such as cell width, followed by the XOST's operation (`CELL`, `TABLE`, `FIGURE`, etc.), followed by the child XOs of the XOST, followed by a closing `ENDOBJ` instruction. The child XOs, also called *components*, may be leaf XOs or subtrees. Except for `FIGUREs`, XOs do not usually specify placement information (x/y coordinates) explicitly: XXICC automatically formats them as needed.

You normally edit XOs using the XXICC Object Editor (XOE), in which case the internal representation is invisible. For example, when editing a table you can insert or delete rows or columns without considering the `CELL` and `HSTACK` instructions that XOE creates and deletes automatically.

A XOE document is usually a single `PARABOX` XOST, which contains `TEXT` XOs and perhaps other XOs such as `DIALOGs` and `TABLEs`. We often call this top-level XOST an *XO list* since XOE usually processes it as a list instead of as a tree, as we will see in Section 5.8.

You can also create XOs by including them as textual `inline` calls in GalaxC programs. This is generally preferable when the contents of an XOST need to be created dynamically from run-time variables.

In both cases we are including XOs as data within a textual program, and executing (interpreting) that data to create run-time versions of XOs. This mixture of code and data is unusual for a pure-text language like C, but has been used extensively in LISP and Galaxy, particularly in the Galaxy CAD System [JFB 92a]. We will see examples of this later in the chapter after we define the standard leaf XOs and XOSTs.

5.1. Leaf XOs

Here are the leaf XOs currently defined. We show them as they appear as `inline` calls in GalaxC program text, with operands following the XO opcode. In PSI, the operands are evaluated and pushed in right-to-left order before executing the XO operation. If an operand is not fully described here or in 5.3, see the XOE source code for more information.

`TEXT` *string*

Textual string, which may have multiple lines separated by `LF`s. Strings may contain (horizontal) `TAB` characters. In addition, the string may contain non-printing format control to indicate text style, font, and color. These are described in JB 11: see Format Control]. Characters are 8-bit bytes and include Latin-1 and Windows-1252 characters.

`SPACER(`*width*, *height*`)`
`SPACER` *width*

Invisible *width* by *height* box for adding spacing inside a dialog. If *height* is missing, assume it is the same as *width.*

`TBUTTON(`*string*, *options*`)`

Button with a simple unformated text string (perhaps `NULL`), generally used for selectors (*not yet implemented*) and sliders. For more formatting options, e.g. a string with an underlined character such as "Press Me", use the `BUTTON` XOST.

As with most GUIs, the user presses and releases a `TBUTTON` using a mouse or touchpad. Doing this may call an *action function*. *Options* selects one of the following button types:

- PushButton: pressing changes appearance, release calls action function.
- TouchButton: pressing calls action function.
- ☐ CheckButton: pressing toggles square check box to left (*or right someday*).
- ◯ CircleButton: pressing toggles check circle to left (*or right someday*).
- MenuButton: pressing selects menu item, which may have check mark to left (*or right someday*).

Options also indicates whether a PushButton or TouchButton is the default button (can be clicked by pressing `ENTER` or `RET`), whether the check mark is to the left or right (*not yet implemented*), and whether or not a PushButton closes a dialog. A `TBUTTON` may have an associated variable to implement check boxes and radio button. This and action functions are described in §6.5 and §6.4.

XOE computes the width and height of a TBUTTON automatically from the text string and current font.

BREAK *options*

Insert page break (if selected by *options*) in a XOE document. This corresponds to an ASCII form feed (FF) character in a text file. BREAK is also used for controlling enumerations and other purposes.

PSTYLE *j*

Change paragraph formatting style for the following XOs to *j*, which encodes both base style (left justify, right justify, itemized list, etc.) and indentation level. Paragraph style is described in detail in Section 5.6. PSTYLE is a *pseudo-XO* rather than a real object: it just changes how the following objects are formatted. PSTYLE may only occur between paragraphs.

ENDOBJ

ENDOBJ is a special leaf XO which ends an XOST.

The following XOs are used in FIGUREs. For details, see §9.1.

MOVETO(*x*, *y*)

Pseudo-XO which sets the **current position** z_0 to [*x*, *y*].

LINETO(Δ*x*, Δ*y*)

Line segment from current position z_0 to z_0 + [Δ*x*, Δ*y*].

RECT(*width*, *height*)

Width by *height* rectangle with upper-left corner at z_0.

CIRCLE(*r*)

Circle with center at z_0 and radius *r*.

ARC(*dir*, Δ*z*)

Circular arc from z_0 to z_0 + Δ*z*, with initial direction *dir*.

INST(*submod*, *rotmir*)

Instance of submodule *submod* with origin at z_0, rotated and mirrored according to *rotmir*.

PIN *net*

Indicates that a pin of an INST is connected to NET *net*.

DOT

Connection dots in a NET, drawn as a filled circle at z_0.

5.2. XO Subtrees

Here are the subtree XOs currently defined. Some XOSTs -- called *LFSTs* -- cause paragraph breaks inside a PARABOX. That is, the LFST breaks the previous paragraph as if there were an LF character, and the first

character or XO after the LFST starts a new paragraph as if the LFST had an LF at the end of it.

PARABOX(*width, options*)

Paragraph box containing text and other XOs, including subtrees. The contents are placed like formatted text, i.e., from left to right with objects past the right margin wrapping to the next line. If positive, *width* defines a fixed width: XOE formats PARABOX contents to *width*, less margins. If *width* is negative, it represents the ideal aspect ratio of a variable-width PARABOX, expressed as *-width*1024/height*. -1536 is 3:2, -2048 is 2:1. (*May change 1024 to a different value to increase range.*) A PARABOX in a DIALOG usually has variable width so that it is automatically sized to fit the DIALOG. A *width* of 0 means to make the aspect ratio as wide as possible so that the PARABOX usually contains a single line.

XOE computes PARABOX height automatically from its contents.

A PARABOX provides the most general paragraph formatting capabilties, including multiple levels of indentation, various kinds of lists, and headings. A XOE document is usually a PARABOX at the top (root) level. A PARABOX is an LFST.

CELL(*width, options*)
CELL *width*

A CELL is similar to a PARABOX, but usually only contains a single line of text. Paragraph formatting is limited to left, centered, and right justification and CELLs cannot contain as many kinds of XOSTs. CELLs are primarily used within DIALOGs for editable fields, within FIGUREs for placing text, and for constructing TABLEs.

A positive *width* is the width of the CELL. In a TABLE, the width of a column is set by the CELLs of the first row. XOE formats CELL contents to that width, less a fixed CellBorder. Zero *width* places the contents relative to the *x* coordinate of the CELL and inhibits word wrap. Zero *width* is primarily used for FIGURE TEXT, where the fact that TEXT is contained in a CELL is transparent to the user. These CELLs are called FIGtext.

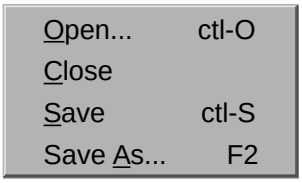
XOE sets a TABLE CELL's height to its TABROW height -- i.e., the height of the parent HSTACK -- less separator line thickness. XOE sets a FIGURE CELL's height to its contents plus CellBorder, and other CELLs' height to the current font height plus CellBorder.

Options includes text justification, by default centered. If *options* is missing, assume 0.

VSTACK(*width, options*)
VSTACK *options*
VSTACK

A *Vertical Stack* is a group of zero or more components placed from top to bottom. VSTACKs are mostly used for constructing DIALOGs and menus. XOE computes the height (and width if *width* = 0) of the VSTACK automatically, using *options* to set horizontal and vertical justification and matching. SPACERs can be used to put vertical space between VSTACK components.

Here is a sample VSTACK, in this case a menu containing four MenuButtons:



If *width* and/or *options* is missing, assume 0.

`HSTACK(`*height, options*`)`
`HSTACK` *options*
`HSTACK`

An `HSTACK` is the same as a `VSTACK` except that components are places horizontally from left to right. XOE computes the width (and height if *height* = 0) of the `HSTACK` automatically. An `HSTACK` contained within a `TABLE` is called a `TABROW` and sets the height for all `CELLs` in the `TABROW`.

Here is a sample `HSTACK`, in this case part of a `DIALOG` with two PushButtons and some `SPACERs`:

If *height* and/or *options* is missing, assume 0.

`BUTTON(`*height, options*`)`
`BUTTON` *options*

A `BUTTON` is an `HSTACK` with the same button functionality as a `TBUTTON`. `BUTTONs` can have arbitrary formatted text with underlined characters, such as "Press me", as opposed to `TBUTTONs` which can only have an unformatted `string` argument. `BUTTONs` can also have non-text labels such as images (*some day*). XOE computes the width (and height if *height* = 0) of the `BUTTON` automatically, setting *width* to match the contents plus a fixed ButtonBorder.

`XREF(`*height, options*`)`
`XREF` *options*

An `XREF` is an `HSTACK` used to define and use cross references. An `XREF` contains one or more `TEXT` XOs to define a symbol or use a symbol's value. An `XREF` is handled like an in-line `BUTTON` except that it does not have a frame or margin: just a gray background which is displayed but usually not printed. An `XREF` that defines a symbol begins with '`:`'. For example, `:xxx` in a heading defines symbol `xxx` to be the heading's label, e.g., "3.5.2". To get to a symbol's value, omit '`:`'. For example, the `XREF` `xxx` references the current value of `xxx`. *[fn:* An `XREF` also has an option bit that causes it to be treated as a literal so it can be used as an example in a document such as this one.*]*

A XOE user may choose to display `XREF` symbols or values. This is similar to how `CELLs` may show expressions or their computed values in a spreadsheet. If a symbol is undefined, XOE shows the symbol symbol name. The value of an `XREF` that defines a symbol is shown as a space and like a space does not wrap lines. When first entering text, it usually makes sense to display `XREF` symbols since you may be editing lots of them. Later on it makes sense to display `XREF` values so you can see how the document will be printed.

`XREFs` are also used for predefined symbols such as dates. `XREFs` are based on LaTeX `\label` and `\ref` tags [LL 86].

`TABLE(`*Tstyle, options*`)`
`TABLE` *options*

A `TABLE` is a special kind of `VSTACK` consisting of a vertical stack of `TABROWs` (`HSTACKs`), each

containing one or more CELLs. Each TABROW has the same number of CELLs, and the CELLs in each column have the same width. *Tstyle* defines the TABLE style, currently 0. If *Tstyle* is missing, assume 0.

A TABLE is an LFST. Tables are described in detail in Chapter 7.

DIALOG(*width*, *options*)
DIALOG *options*

A DIALOG is just like a VSTACK, except that it is an LFST. There are also minor differences in how the border is rendered on some window managers. If *width* is missing, assume 0.

FIGURE(*width*, *height*, *options*)

A FIGURE is a line drawing made of LINETOs, RECTs, FIGtext, CIRCLEs, ARCs, etc. Unlike most XOSTs which automatically place their components using a formatting algorithm, FIGURE components have explicit coordinates set by MOVETO. FIGUREs are LFSTs.

The following XOSTs are used in FIGUREs, especially in schematic diagrams. For details, see §9.2.

SUBMOD(*name*, *options*)

A SUBMOD is a schematic symbol such as a logic gate, a circuit element, or a port.

COMP

A COMP is an INST plus optional connectivity information and instance-specific text.

NET

A NET contains LINETOs and DOTs for wires that interconnect instance pins in a schematic.

5.3. Options

Many XOs have an *options* operand, which is a bit vector of additional properties. In the current implementation, *options* is a 16-bit number, of which only the low 14 bits are used so that they fit into a PSI small integer. The low 8 bits are for generic XO options:

- Horizontal justification: specify how XOE should place a horizontal XOST's components. The four options are LeftJustify, Centered, RightJustify, and HorizJustify which means to spread out non-SPACER components uniformly.

LeftJustify
Centered
RightJustify
HorizJustify (Spread out)

If there is only one non-SPACER HorizJustify looks the same as LeftJustify. Menu buttons normally use HorizJustify -- see the VSTACK example in Section 5.2.

- Vertical justification: specify how XOE should place a vertical XOST's components. The four options are TopJustify, Centered, BottomJustify, and VertJustify which means to spread out non-SPACER components uniformly.

TopJustify	Centered	BottomJustify	VertJustify (Spread out)

If there is only one non-SPACER VertJustify looks the same as TopJustify.

- Frame style: An XO may have one of four frame styles: None, Thin, Motif, or Thick. The width and height of an XO include the frame thickness, which is added to CellBorder or ButtonBorder when creating CELLs and BUTTONs. Push buttons usually have Motif frames and the others have None, though this is entirely up to the programmer.

The high bits are for specific XO classes. Here are the HSTACK, VSTACK, TABLE, and DIALOG options:

- MatchWidths: make all non-SPACER components the same width by widening the narrow ones. This is used for both the VSTACK and HSTACK examples in Section 5.2.

- MatchHeights: make all non-SPACER components the same height by making the short ones taller.

- SepLines: put horizontal or vertical separator lines between components, as in the above justification examples. Currently, the thickness of a SepLine is always 1.

BUTTONs and TBUTTONs have these options:

- ButtonKind: PushButton, TouchButton, CheckButton, CircleButton, MenuButton. See TBUTTON in Section 5.1.
- DefaultPB: PushButton or TouchButton is the default button associated with pressing ENTER or RET. The button is shown with an extra black border: DefaultPB
- CheckSide: which side the check mark is on, viz., CheckLeft or CheckRight. *Not yet implemented.*
- DoesNotClose: pressing this PushButton, TouchButton, or MenuButton does not close the dialog as it normally would.

Here are the CELL and PARAG options, normally only used in DIALOGs:

- Editable: XOST is an editable field, normally for CELLs that are used for entering text.

- RETisTAB: treat ENTER or RET as if it were a TAB character, i.e., go to the next field instead of closing the dialog. XOE uses this in its Open/Save File dialog.

- StretchWidth: Stretch width of XO to match an enclosing HorizJustify HSTACK.

Here are the BREAK options, which affect paragraph formatting (§5.6):

- ResetListNum: reset the following Numbered paragraph's counter, overriding its standard behavior.

- ContListNum: do not reset the following Numbered paragraph's counter.

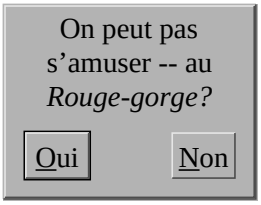
- ExcluClass: set the following paragraphs' *exclusion class* to 0 (none) through 3. Text and XOs in an

exclusion class may be omitted when displaying and/or printing a document. XXICC documentation uses exclusion class 1 to identify copyright and license notices that should be included if a `.xoe` file is edited or printed on its own, but omitted if the file is printed as part of a book where the notices are at the beginning of the book.

The ClipSubtree option is available for all XOSTs. It tells XOE to clip the contents to the limits of the XOST when redrawing. XOE uses ClipSubtree in the Open/Save File dialog to prevent drawing file names that are to the left or right of the scrollable window.

5.4. DIALOG Example

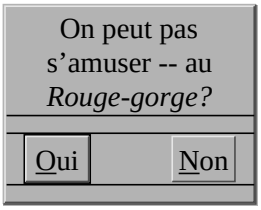
Here is a sample `DIALOG` box to illustrate the concepts of leaf XOs and subtrees:



Here are the equivalent GalaxC `inline` calls:

```
DIALOG ThickFrame;
  PARABOX(-3072, 0);      // Variable-width PARABOX with 3:2 aspect ratio: XOE reformats
    TEXT "On peut pas s'amuser -- au Rouge-gorge?";          // TEXT to fit automatically.
  ENDOBJ;
  SPACER 10;              // Vertical space between text and buttons.
  HSTACK MatchWidths;     // All buttons have the same width.
    SPACER 10;            // Horizontal space to left of "Oui" button.
    BUTTON (PushButton | DefaultPB | MotifFrame);
      TEXT "Oui";
    ENDOBJ;
    SPACER(50, 10);       // Horizontal space between buttons.
    BUTTON (PushButton | MotifFrame);
      TEXT "Non";
    ENDOBJ;
    SPACER 10;            // Horizontal space to right of "Non" button.
  ENDOBJ;
  SPACER 10;              // Vertical space below buttons.
ENDOBJ;
```

Here is the same dialog with separation lines between `DIALOG`'s components to illustrate structure:



5.5. Character Formatting

Character formatting for changing typeface, font style, font size, subscripts, and superscripts are specified within TEXT XOs using control characters. For details, see [JFB 11: Format Control].

5.6. Paragraph Formatting

Paragraph formatting defines how to break a sequence of words and/or characters into lines subject to margins. It also includes formatting headings and lists. XOE paragraph formatting is loosely based on Microsoft Word and LaTeX. It is considerably simpler, providing capability adequate to most users' needs rather than trying to be the world's most feature-laden product so that even the most picky user is satisfied (*as if*).

Each paragraph has a paragraph style (*Pstyle*) which specifies formatting properties such as default font, justification, left and right margins, indentation of the first line, and list or heading numbering. Like LaTeX, the Pstyles of a document define the overall appearance of the document so that it is consistent and can be easily changed at a single location. There can be up to 256 base Pstyles, plus 5 levels of indentation.

Here are the standard paragraph Pstyles:

Left	Left-justified paragraph.
Center	Centered paragraph.
Right	Right-justified paragraph.
Quote	Multi-line quote indented on both sides.
Describe	Description list item. First line is outdented for a label.
Itemized	Bulleted list item, with a different bullet style at each indent level.
Numbered	Numbered list item, with Arabic, Roman, or alphabetical item labels.

Here are the standard heading Pstyles, based on LaTeX: Section, Subsection, Subsubsection, Chapter, Appendix, Figure caption, Table caption.

Each Pstyle has an indentation (nesting) level from 0-4. Quote and list items have a nesting level from 1-4. Headings have nesting level 0. Numbered paragraphs have associated counters, one for each nesting level. Each time XOE processes a Numbered paragraph it increments the counter for the item's nesting level. A paragraph with nesting level *n* resets the counters for all levels > *n*. In addition, switching Pstyle between Itemized or Describe and Numbered resets the level *n* counter. BREAK(ResetListNum) and BREAK(ContListNum) override the standard behavior for an immediately-following Numbered paragraph, either resetting *n* or continuing *n*.

In an XO list, LF characters break text into a sequence of paragraphs. An LF is part of the previous paragraph. Pstyle for a paragraph *P* is specified by a PSTYLE(*j*) instruction before *P*'s text, where *j* encodes the base Pstyle (> 0) and nesting level (0-4). [fn: PSTYLE instructions are only allowed between paragraphs, i.e., immediately after an LF.] If an LF is not followed by PSTYLE, the following paragraph uses the default Pstyle determined by the previous paragraph. In most cases the successor Pstyle is the same as its predecessor and the successor's nesting level is the same as the predecessor's. However, headings are normally followed by Left, and list items are handled differently. Each paragraph in a list is either a *list item* (LI) or a *list item successor* (LIS), usually Left. The default is LIS. Each LI must be preceded by a *list item flag* (LIF = ctl-L = ASCII FF), which is created using sh-RET in XOE. The LIF is stored in a TEXT XO.

(*At some point we plan to include* TABs *as part of Pstyle, perhaps supplemented with "set tab" and "clear tab" instructions.*)

As discussed earlier, some XOSTs are treated as paragraphs. An LFST ends the previous paragraph as if there were an LF, and the character or XO that follows the LFST begins a new paragraph. LFSTs have the same Pstyle options as a text paragraph, including numbered and itemized lists, and may be preceded or followed by PSTYLE instructions.

To change a Pstyle, the user selects one or more paragraphs. A paragraph is considered selected if any part of it is in the selection region between cursor and mark. If cursor and mark are the same, select the single paragraph containing cursor/mark or immediately following them. The user then chooses a new paragraph style from the **Pstyle** menu. All the selected paragraphs adopt the new style unless the first paragraph is switching from one list Pstyle to another list Pstyle in which case the LIS paragraphs remain LIS paragraphs. The nesting levels usually stay the same, but if the new style is a heading the new nesting level is 0 and if the new style is Quote (or a list), the new nesting level is at least 1.

XOE also has Pstyle menu commands to increment or decrement nesting level to indent or outdent paragraphs. These have keyboard equivalents ctl-TAB and ctl-sh-TAB.

5.7. XXICC Files

An XXICC document is stored in a file as an XO list, with the following structure:

1. A VERSION instruction which specifies document version so documents can be read by future versions of XOE.
2. Zero or more SUBMODs. SUBMODs are only used by documents which have figures, such as a collection of logic diagrams.
3. One XOST, usually a PARABOX, which contains the body of the document.

In memory, an XXICC document has the same form with the addition of a RETNF instruction to mark the end of the document. RETNF (return w/o popping frame pointer) ends execution of the document when it is being interpreted as a program (§5.8). The memory version also has a single SKIP instruction which identifies unused memory available for insertions. The SKIP instruction can be anywhere before the final RETNF.

XOE can also edit text files. A simple text file contains just ASCII printable characters plus control characters LF, CR, TAB, and FF (form feed). When reading text files, XOE converts CR and CR+LF into LF so it can read Macintosh and Windows files. It also converts FF into BREAK XOs. XOE always writes files using LF as the end of line character, and converts BREAK into FF.

XOE can also read and write text files with Latin-1 and Windows-1252 characters, as well as the format control characters described in [JFB 11: Format Control].

To edit a text file, XOE converts it into XO list form with a single PARABOX XOST containing TEXT XOs. TEXT XOs use PSI strings as argument and each is limited to 4K characters. XOE automatically fragments a long text file into valid PSI strings.

5.8. Interpreting XO Lists

One of the most powerful features of using PSI as the underlying representation of XO lists is that they can be executed (interpreted) as program code. This technique of executing data is rare in C programs, but has a long history of use in LISP and was used extensively in the Galaxy CAD System [JFB 92a]. Another great example is the DEC GT40 Graphic Display System [DEC 72] which mixes instructions with characters and line graphics at the machine language level.

The same data can be interpreted in different ways for different purposes. This is implemented by the `XOIstruct` data structure which contains an array of function pointers indexed by XO opcodes. When intepreting an XO list, PSI variable `XOIP` points to an `XOIstruct` and calls the indexed function to interpret each kind of XO.

For example, the "`reformat XOlist`" function in `xoeform.gal` formats an XO list for display in a window or for printing. It does this by running PSI with `XOIP` set to `XOIformat`, an `XOIstruct` with pointers to formatting instructions for each kind of XO. The most complex of these is "`format TEXT`", which calculates where characters should be placed in a window, wraps lines at word boundaries, processes format control characters, and does paragraph formatting.

Another important `XOIstruct` is `XOIcopy`, which copies all or part of an XO list. This is used for copy and paste, as well as for cleaning up a modified XO list before storing it to a file.

Representing an XO list as executable code could make it trivial to pass malware. Thus XOE does not just execute any document it opens. Instead, it makes sure that all instructions either push data or execute XOs before allowing a document to be opened. It does not allow `LOAD` and `STORE` instructions, or `CDECL` calls which may modify system files. Removing this protection is *not* a good idea.

5.9. Incremental Changes and UNDOs

The memory representation of an XO list can keep track of changes made by the user and can undo changes made since the last save. XO lists represent changes using `INSERT` and `DELETE` tags, which are PSI opcodex instructions each with a 32-bit operand *X*. In this case, *X* is a *transaction number k* which identifies an atomic group of inserts and deletes which were performed atomically and must be undone atomically as well. `INSERT` and `DELETE` tags are analogous to a human proofreader's insert and delete marks.

An XO list consists of `TEXT` and other XOs punctuated by `INSERT` and `DELETE` tags. An `INSERT<`*k*`>` tag sets insert level *insK* to *k* until the next `INSERT` tag sets it to a higher or lower level. Similarly, a `DELETE<`*k*`>` tag sets delete level *delK* to *k* until the next `DELETE` tag. *insK* and *delK* are initially 0, which means that we are not within a transaction.

When XOE processes an XO list, it skips XOs if *delK* > 0, i.e., they are marked for deletion.

Insertions can be nested within insertions. XOE simply tags the beginning of the insertion with `INSERT<`*new*`>` and the end with `INSERT<`*old*`>`, where *new* and *old* are the new transaction number and the previous *insK* at the insertion point. If there is already an `INSERT` instruction at the end, XOE does not add the second `INSERT<`*old*`>`.

To undo an insertion, XOE removes the initial `INSERT<new>` and the inserted objects. It removes the ending `INSERT<`*old*`>` provided that the insert level before `INSERT<`*new*`>` is the same as *old*.

Deletions are not nested, though they may abut. XOE never deletes a group of objects that contain deletions. Instead, it deletes the subgroups that are not currently deleted, i.e., their deletion level is 0. This has the same effect as a nested deletion. XOE may need to add `DELETE<`*new*`>` at the beginning of the group and `DELETE<0>` at the end. It also changes `DELETE<0>` tags within the group to `DELETE<`*new*`>`.

To undo a deletion, XOE changes `DELETE<`*new*`>` tags to `DELETE<0>` and removes redundant ones.

When XOE saves a file, it removes the `INSERT` and `DELETE` tags, and discards deleted text. It also resets the transaction number to 0.

In addition to insert and delete, XOE has Format Change operations where a selected group of objects has its font, style, color, or Pstyle changed to a new value. Instead of going through the group changing every format control, creating an Undo nightmare, XOE inserts `FORMAT` instructions at the beginning and end of the group that override the format control within the group. `FORMAT` instructions are converted to format control prior to Save and are easily removed by Undo. For details, see `xoebase.gal` and `xoedxo.gal`.

While not yet implemented, this approach also supports Redo. Basically, an Undo operation is not completed until the user saves the file or performs an edit. However, objects with `INSERT` level above *k* are not drawn and objects with `DELETE` level above *k* are not skipped.

The approach also supports the future ability to track changes, e.g., underlining insertions and striking out deletions. In this case, low values of *k* represent revision numbers. *revK* is the last revision number.

When processing an XO list, skip XOs if *delK* > *insK* and *insK* > *revK*, i.e., do not show deletions made to insertions since *revK*. If *delK* > *insK* and *insK* ≤ *revK*, we have deleted XOs from the last or earlier revision: either do not show the deletion or strike it out. If *delK* < *insK* and *insK* > *revK*, we have inserted since *revK*: show the insertion underlined if that option is selected.

To save a file while tracking changes, compress `INSERT` and `DELETE` and discard deletions made to insertions since *revK*. Tag undeleted insertions with `INSERT<`*revK*+1`>` and tag deletions of *revK* and earlier with `DELETE<`*revK*+1`>`. New transactions start at *revK*+2.

Chapter 6
Decoded XXICC Objects

In the last chapter we introduced XXICC Objects (XOs), which are the fundamental building blocks for documents, dialogs, and other GUI structures. While XOE could use an XO list as its only internal structure, formatting a large document from scratch takes considerable computing time. It's not a good idea to reformat each time part of a document needs to be refreshed on the screen. So XOE transforms an XO list into an intermediate structure called a *Decoded XO list* (*DXO list*).

- An XO list is a highly-encoded structure suitable for storing in a file. It also includes incremental update tags to support Undo and for tracking changes.

- A DXO list is a temporary structure which is easy to render on the screen. For example, a `TEXT` XO may contain many lines of text with lots of format control. Decoding this results in multiple `TEXT` DXOs, each with placement cöordinates, uniform character format, and no control characters so they can be drawn with a single graphics API call.

  The DXO list does not contain any incremental update tags like `INSERT` and `DELETE`. XOE considers them when formatting an XO list to a DXO list, but only the visible XOs become DXOs.

An XO list is stored in contiguous storage so that it can be executed as in-line code. In contrast, a DXO list is a linked list to simplify incremental updates. An XO list is actually a linearlized form of an *XO tree*, where the tree structure is defined by XOST and `ENDOBJ` instructions. Similarly, a DXO list is a linearized *DXO tree*, with XOST and `ENDOBJ` DXOs that correspond to the XOs.

A large document can have a huge number of DXOs. There is no point in saving all of them, since only a fraction will be visible in a window or on a single printed page. XOE therefore *abridges* the DXO list, omitting DXOs that are not currently visible. However, it retains enough information so that if scrolling exposes DXOs they can be regenerated quickly instead of reformatting the entire document from scratch. Abridgement follows the tree structure of the DXO list, i.e., XOE can omit the contents of a subtree if that entire subtree is not currently visible. This approach is loosely based on outline editors with elision capability and also on incremental processing algorithms in general.

DXOs are also loosely based on the CRISP Microprocessor's *Decoded Instruction Cache*. In CRISP [BDM 87], CPU instructions are highly encoded in memory to conserve memory size and bandwidth requirements, but are expanded into a decoded form when they are fetched into cache so that the CPU can execute them more efficiently. This is particularly effective for tight loops where the same instructions are executed many times. Similarly, DXOs are a form that is easily redrawn on the screen, which may occur many times before other XOs need to be decoded.

DXOs are also used for pop-up dialogs and menus created on the fly and deallocated when the user is finished with them. In this case DXOs are like X windows widgets or Win32 child windows. The DXOs are generated by GalaxC programs with embedded graphics or `inline` XO calls instead of from a XOE document file.

6.1. DXO Internal Structure

Each DXO node has the following fields:

next Next DXO in DXO list, or NULL if end of DXO list.

kind Kind of XO, e.g., `TEXT`, `RECT`, `HSTACK`, `ENDOBJ`. *Kind* is equal to the XO's PSI opcode.

state Various state bits, including:

- *SelectedObject*: selected DXOs are shown differently from unselected DXOs. For example, XOE highlights a `BUTTON` DXO to show it is being pressed. `FIGUREs` show selected DXOs by displaying vertices that the user may drag to move or copy the DXO.
- *MarkedObject*: DXO is marked for special processing.
- *HiddenObject*: do not display or select this DXO.
- *AbridgedObject*: part or all of a DXO subtree's contents are absent to save storage and processing.
- *ShadowedObject*: display this DXO with reduced contrast and do not allow it to be selected. XOE uses this to disable dialog and menu buttons temporarily, while leaving them partially visible to provide a more consistent user interface.
- *ZoomedObject*: DXO has been zoomed by the formatting algorithm.

options Usually the same as the *options* operand of the DXO's XO, but may include additional bits. All DXOs have an *options* field, but some XOs do not have options. For example, an `ENDOBJ` XO does not have options, but its DXO has a number of option bits which encode properties of the `ENDOBJ` or the XOST it ends:

- *NTbreak*: `ENDOBJ` ends a `LINE` or `PARAG` (see below) that is broken by a following non-text (NT) XO, i.e., an LFST or `ENDOBJ`. NTbreak affects cursor behavior.
- *MTline*: `ENDOBJ` is an empty line at the end of a `CELL` or `PARABOX`. An empty line can have its own Pstyle: this is necessary so that XOE knows how to format any text inserted into the empty line.
- *LFSTend*: `ENDOBJ` ends an LFST. Since DXOlists are singly-linked, finding the XOST corresponding to a given `ENDOBJ` requires searching. This bit avoids the need for such a search if just checking to see if `ENDOBJ` ends an LFST.
- *HorizCursor*: XOE usually uses a vertical blinking cursor to show where text and other XOs will be inserted. When inserting at the end of a `VSTACK` or `DIALOG`, XOE prefers to have a horizontal cursor.

XOE provides special treatment for zero-width text `CELLs` in a `FIGURE`, so that the `CELL` structure is transparent to the user. XOE sets the FIGtext option to identify `FIGtext CELLs`.

When inserting or editing `FIGURE` lines, rectangles, and other shapes XOE must tell the drawing routines to stretch the DXO instead of moving it without stretching. XOE sets the RubberDXO option bit for these DXOs.

z Upper left (usually) corner of DXO, in document cöordinates. By using document cöordinates, XOE does not need to recompute the cöordinates of DXOs within `FIGUREs`, `DIALOGs`, or other nested XOSTs. This fits the philosophy of XO lists versus DXO lists: XO lists contain highly-encoded data which requires formatting, while DXO lists contain completely formatted data that can be drawn and searched quickly and easily.

x, y Individual cöordinate: $x = z.x$, $y = z.y$.

size DXO width and height expressed as a `Point`. In almost all cases width and height are non-negative, simplifying construction of clipping rectangles and testing whether mouse cöordinates are within the

DXO. One case where they are not is the `LINETO` DXO, where *z* is the start or the line and *z*+*size* is the end of the line.

DXO width and/or height may be specified by XO operands, but are usually computed dynamically. In some cases, like a variable-width `PARABOX` inside a `DIALOG`, the computation is quite involved so *size* is only recomputed when needed.

If a DXO is unzoomed, DXO.*size* is its unzoomed size. If DXO is zoomed, DXO.*size* is it zoomed size and its unzoomed dimensions are generally stored in DXO.*uaw* and DXO.*uh*: see §6.6 for details.

dz DXO displacement, same as *size.z*. `LINETO`, `ARC`, and other graphics use DXO.*dz* instead of DXO.*size* for clarity.

w, *h* DXO width and height, same as *size.x* (*dz.x*) and *size.y* (*dz.y*).

pc DXO.*pc* usually points to the XO instruction that was decoded to form DXO, in which case the XO's opcode is the same as DXO.*kind*. However, in a `TEXT` DXO, *.pc* usually points to an address within a `TEXT` XO's string operand. If the DXO list is derived from an XO list, the DXO.*pc* values should be monotonically non-decreasing. This is required for incrementally updating the DXO list.

text Same as DXO.*pc*, except that *text* is a character pointer (i.e., a `string`) whereas *pc* is a PSI pointer.

format 32-bit character format encoding typeface, font style, and color. For details, see `xoedxo.gal`. Unlike a `TEXT` XO, a `TEXT` DXO has a single, uniform format and contains no control characters so it can usually be drawn with a single API call. Other DXOs use *format* in various ways. For example, an MTline `ENDOBJ` uses *format* to set the format of any text entered at that `ENDOBJ`, and XOSTs use *format* to set the initial character format if the XOST's contents are regenerated.

len Normally used for the length of a `TEXT` string and passed to the API call that displays a DXO. XOE breaks a `TEXT` XO into multiple DXOs, using *text* to point to substrings and *len* to set the length of each one.

XOE uses *len* to distinguish between an `ENDOBJ` used for word wrap (*len* = 0 or 1) and an XOST `ENDOBJ` (*len* = 2). See `PARAG` and `LINE`, below.

index DXO object index, or 0. XOE uses object indices to find `BUTTON`, `CELL`, or other kinds of DXO within a `DIALOG`. XOE also uses *index* to number paragraphs within numbered lists, as well as headings.

action Action function for a `BUTTON` or other DXO, or NULL if not used.

v Pointer to a `ubyte` value, with variants for `Boolean` (*.bv*), `sbyte` (*.sbv*), `short` (*.sv*), `ushort` (*.usv*), `long` (*.lv*), `ulong` (*.ulv*), `string` (*.sv*), and `DXOptr` (*.dv*). DXO.*bv* and DXO.*usv* is used for check boxes and radio buttons in §6.5.

BGbrush Background brush for this DXO, normally the same as its parent DXO.

a Font ascent of `LINE`, `TEXT`, `ENDOBJ`, and any DXO that may be placed in a line. DXO.*a* is the

offset of the DXO's baseline from DXO.*z*, which is usually the DXO's upper left corner. When XOE places DXOs in a line it matches their baselines so that fonts with different heights look right.

DXO.*a* has other uses for other DXOs. For example, a PARABOX uses DXO.*a* to store the PARABOX's aspect ratio.

If a DXO is unzoomed, DXO.*a* is its unzoomed ascent. If DXO is zoomed, DXO.*a* is it zoomed ascent and its unzoomed ascent is generally stored in DXO.*uaw*: see §6.6 for details.

Pstyle Paragraph style for a PARAG, ENDOBJ, or XOST. The field is used for other purposes by other DXOs. For example, BUTTONs use it for .*Vsize* which is the size in bytes of the variable pointed to by .*v* (0 for Boolean).

uaw Unzoomed DXO ascent and width if DXO is zoomed.

cd General-purpose 32-bit field used for various purposes by different DXOs. DXO.*cd* is used for unzoomed height (DXO.*uh*) by zoomed leaf DXOs that may be placed in a LINE. ENDOBJ.*uh* is the unzoomed height of a zoomed NT subtree.

Figure editing reuses some of these fields for drawing and for connectivity analysis: see §9.6.

6.2. PARAG and LINE

For the most part, each XO maps into a single DXO. However, a TEXT XO may map into zero or many DXOs, of the following kinds:

TEXT

A TEXT DXO cannot contain control characters and has uniform format, so if a TEXT XO contains LF, TAB, or other format control XOE maps the TEXT XO into multiple DXOs. In addition, if a TEXT XO is too long to fit on a line XOE wraps it at word boundaries, producing multiple TEXT DXOs.

XOE converts each TAB character into a TEXT DXO with .*len* = 1, .*w* = width of tab stop, and .*index* = number of 'n' characters in tab stop if exact match. It also sets the TABtext bit in TEXT.*options.*

LINE

When formatting a TEXT XO, XOE automatically creates a LINE subtree and its closing ENDOBJ to contain the TEXT and other DXOs that appear on that text line. LINEs are instrumental to managing cursor movement and highlighting selected text and/or XOs. LINEs are also important for abridging DXOs: if a LINE is not visible in a window, XOE can omit the contents between the LINE and its closing ENDOBJ to save storage and search time.

Each LINE contains of zero or more TEXT and/or non-text DXOs. LINE.*h* is the maximum line height, including external leading. LINE.*a* is the maximum font ascent from any DXO.*y* in LINE to LINE's common baseline. LINE.*tex*t points to first visible character of the line after any initial format control characters, including LIF.

LINEs never appear in XO lists or XOE document files: they are only present in DXO lists and vanish when the DXO list is deleted from memory. It is easy to confuse the term LINE, which is

generally a line of text, and LINETO which is a graphical line segment in a FIGURE.

ENDOBJ

Each automatically-generated LINE has a closing ENDOBJ. XOE closes a LINE when it reaches an *end of line* (EOL) character, which can be LF, a wrapping space or TAB, or (*some day*) an explicit or implicit hyphen. XOE also closes a line when XOE reaches an LFST or ENDOBJ XO. ENDOBJ.*text* points to line's end of line (EOL) character or to the breaking LFST. There may be format control characters between the last visible character in a LINE and the EOL character, so the format of the last object in a LINE may be different from ENDOBJ.

Here is a summary of the possible ends of LINE:

- Explicit LF: ENDOBJ.*text* points to LF, ENDOBJ.*len* = 1, ENDOBJ.*w* = 0. An explicit LF ends a document paragraph.
- EOL space or TAB: ENDOBJ.*text* points to EOL character, .*len* = 1, .*w* = 0.
- Explicit hyphen: ENDOBJ.*text* points to hyphen char, .*len* = 1, .*w* > 0. *To be implemented.*
- Implicit hyphen: ENDOBJ.*text* points to ???, .*len* = 0, .*w* > 0. *To be implemented.*
- LINE broken by LFST: ENDOBJ.*pc* points to LFST XO, .*len* = 0, .*w* = 0. Set the NTbreak bit in .*options*
- LINE broken by ENDOBJ XO: ENDOBJ.*pc* points to ENDOBJ XO, .*len* = 2, .*w* = 0. Set the NTbreak in .*options*.

A LINE may also be wrapped by an NT XO. *Need to document what happens to* ENDOBJ *in that case.*

PARAG

If it is formatting a PARABOX, XOE groups LINEs into PARAGs. PARAGs have two purposes: to manage Pstyle and to form groups of LINEs that can be abridged. Each PARAG contains one to 20 LINEs -- 20 is an arbitrary value approximating one window's worth of text. This way XOE only needs to store and process the contents of a few PARAGs: the rest are abridged and XOE only stores PARAG size, Pstyle, and initial and final character formatting. PARAG breaks also occurs when paragraph style (Pstyle) changes, or if Pstyle requires starting a new PARAG after each LF.

PARAG.*text* is always equal to the first LINE's .*text*, and the PARAG's ENDOBJ is always a copy of the last LINE's ENDOBJ.

6.3. Formatting Dialogs

In general, XO lists do not have explicit placement information. Instead, XOE automatically formats XOs and generates DXOs that have placement cöordinates. TEXT in a PARABOX is formatted using standard document formatting techniques. DIALOGs and other nested XOSTs use the approach in this section.

A DIALOG consists of a DIALOG XO which contains nested HSTACK, VSTACK, and PARABOX subtrees, eventually reaching fixed-size leaf XOs like TEXT and SPACER. XOE uses a recursive algorithm to calculate the dimensions of a DIALOG and each of its subtrees, and then places each XO at its appropriate (*x*, *y*) cöordinates. The algorithms are described in detail in xoedial.gal. Here is a brief summary.

If all non-stack XOs had fixed dimensions, the algorithm to calculate dimensions would be very simple:

- For each HSTACK, set HSTACK.*w* to the sum of the widths of its components and HSTACK.*h* to the maximum height of its components. Treat BUTTON or XREF like an HSTACK.
- For each VSTACK, set VSTACK.*w* to the maximum width of its components and VSTACK.*h* to the sum of the heights of its components. Treat DIALOG like a VSTACK.

However, we also have to consider variable-width PARABOXes. Each PARABOX has an ideal aspect ratio *a* and some contents, most likely TEXT. To calculate the ideal dimensions of a PARABOX, we first format its contents assuming the PARABOX is as wide as the whole document being displayed. The TEXT will most likely fit on a single line or perhaps a few lines. Let *A* be the area of this formatted text.

Let *W* and *H* be the ideal PARABOX dimensions to be calculated. The ideal aspect ratio *a is W/H*, so *H = W/a*. The area of the ideal PARABOX is *WH = W*(*W/a*) = *WW/a*. This should be equal to the *A* calculated earlier, so *WW/a = A*, *WW = aA*, or *W = sqrt*(*aA*). The ideal dimensions require perfectly rectangular formatting, which does not occur, especially if the PARABOX contains blank lines. The aspect ratio should be set to compensate for this.

After calculating the ideal dimensions for each variable-width PARABOX, we calculate width and height of all stacks for fixed-size non-stacks, using the above algorithm. Use the MatchWidths and MatchHeights options (§5.3) to increase DXO.*w* and DXO.*h* of stack components that are stretched by other components. We can increase dxo.*w* and dxo.*h*, but not reduce them. Consider all variable-width PARABOX.*w* and .*h* to be 0 until they are computed as follows.

Next, we make multiple passes through the DXO tree, each time calculating the final dimensions of one variable-width PARABOX. If the ideal width (height) of a PARABOX is less than the computed width (height) of its enclosing stack, then let the enclosing stack set PARABOX.*w* (.*h*) and recompute PARABOX.*h* (.*w*). If no PARABOX can be computed this way, set the deepest PARABOX to its ideal dimensions. Then recompute all stacks affected by this change. Eventually all PARABOXes and therefore all stacks will have defined dimensions.

Once we have computed all object dimensions, we can place them. Given the (*x*, *y*) upper-left cöordinates of a stack, we can place its components by adding widths to *x* (if an HSTACK) or adding heights to *y* (if a VSTACK). We also consider the horizontal and vertical justification options of each stack when placing its components. For example, when placing a HorizJustify HSTACK, any excess HSTACK.*w* is applied equally to the SPACERs between non-SPACER components.

This algorithm is very fast for reasonably-sized dialogs so there is no need to store them when they are not being used. Similarly, XOE abridges dialogs and other XOSTs embedded in a document when they are not visible, with only the top-level XOST retained so XOE knows their overall dimensions.

6.4. Button Actions

BUTTONs in XO lists do not have any behavior associated with them. On the other hand, a BUTTON in a DXO list can have an *action function* associated with it, which is called when the BUTTON is pressed. This is best illustrated by a simple example based on a gag from the *Hitchhiker's Guide to the Galaxy*:

```
include "xoedial.gi";

XOEwindow arg parent, DXOptr arg button,
```

```
fn punchline (parent, button) =
{
    if button == NULL then return;      // User canceled dialog using ESC key.
    create dialog;                      // Create punchline dialog.
```


```
    open dialog;
};

// Main program contains straight line, which calls punchline as button action.
create dialog;
buttonAction = fnptr {punchline (NULL, NULL)};
```


```
open dialog
```

The main program calls `create dialog` so that the following embedded `DIALOG` creates an DXO list by using `XOIstruct DXOcreate`. It sets global `fnptr` variable `buttonAction` to the `punchline` function. Next comes the XOST for the `DIALOG` as embedded graphics. The XOST and its components are interpreted by PSI which calls `DXOcreate` functions for each kind of XO, producing an undimensioned, unplaced DXO list. Any `BUTTON` components created in this list have `BUTTON`.*action* set to `punchline`. Finally, the main program calls `open dialog`, which dimensions and places the DXO list using the algorithm in §6.3 and then pops up the resulting DXO `DIALOG` on the screen.

Syntax note: GalaxC treats the graphical `DIALOG` as a statement. Normally there would be a semicolon between the `DIALOG` and the `open dialog` statements. Since this would look awkward, GalaxC automatically assumes a semicolon after an LFST.

When the user presses Let's see, XOE calls `BUTTON`.*action* (i.e., `punchline`) and closes the main dialog. Function `punchline` creates and pops up the "*Please do not press this button again*" dialog. This dialog goes away when the the user presses Sorry.

The general form of a button function call is *f* (*parent*, *button*), with the following arguments:

- Global variable `SelWin` (or its equivalents `SelXOE` and `SelDialog`) points to the `DIALOG`'s pop-up window that contains the button.
- *parent* is XOE window that created the dialog, or `NULL` if the dialog has no parent.
- *button* is the `BUTTON` DXO that was pressed to call *f*. If the dialog was closed by pressing `ESC` or by

the window mananger, *button* is `NULL`.

Button function *f* is usually the last code to use the dialog before it is closed and deallocated, so if the dialog contains any data that must be preserved *f* must copy it somewhere else.

Function `punchline` first checks whether `button` is `NULL`. This is not strictly necessary, since XOE should only call `punchline` if the user presses Let's see and not if the dialog is cancelled using `ESC`. However, it's good defensive programming practice to assume *f* might have *button* = `NULL`. `punchline` creates a pop-up dialog the same way as the main program, except that it does not specify `buttonAction`. In this case `buttonAction` is `NULL` and XOE does not call an action function when the user presses Sorry or cancels the `punchline` dialog using `ESC`: XOE just closes the dialog and we're done.

6.5. Check Boxes and Radio Buttons

A `BUTTON` or other DXO may have an associated variable that changes value when the button is pressed. One use is for check boxes, where the fact that a box is checked or not is reflected in a `Boolean` variable. For example, here are three check boxes for stacks which control the MatchWidths, MatchHeights, and SepLines bits of stack.*options*:

☐ Match Widths ☐ Match Heights ☐ Sep Lines

Check boxes are usually implemented as CheckButtons, CircleButtons, or TouchButtons. TouchButtons show the current checked/unchecked status by being selected or not.

Another use is for "radio buttons", where one of a group of buttons is selected and clicking a different button deselects the currently selected one. A group of radio buttons shares a `ushort` variable, which is set to the `index` field of the selected radio button. For example, here are the radio buttons which select one of five `BUTTON` kinds:

Button Kind: Push Touch Check Circle Menu

Radio buttons are usually implemented as CheckButtons, CircleButtons, or TouchButtons.

As with action functions, `BUTTONs` in XO lists do not have variables associated with them. A variable must be assigned to DXO.*v* after creating the `BUTTON`'s DXO. DXO.*v* is defined to be a `ubyte` pointer. `xoebase.gal` defines variant fields to point to other types: DXO.*bv* is `@Boolean`, DXO.*usv* is `@ushort`, etc.

Check boxes and radio button behavior is currently defined in `xoeprop.xoe`, which defines dialog boxes for editing XO options. That file uses a combination of graphics and text to define DXOs. Here is code that defines the three check boxes described earlier:

```
var b = DXOlast;            // Look for next button starting here.
☐ Match Widths  ☐ Match Heights  ☐ Sep Lines  ;
b = next PSI_BUTTON in b; b.bv = &StackOptionMatchW;
b = next PSI_BUTTON in b; b.bv = &StackOptionMatchH;
b = next PSI_BUTTON in b; b.bv = &StackOptionSepLines;
```

We first set local variable `b` to `DXOlast`, which is a global variable that is the last DXO created since `create dialog`. Next we create three `BUTTONs` enclosed in an `HSTACK`. GalaxC treats this `HSTACK` as a statement, which has a semicolon to separate it from the next statement.

For each `BUTTON`, we search starting at `b` for the next `BUTTON` DXO. Then we set `DXO`.*bv* to the address of a global `Boolean` variable that is set (cleared) when the corresponding `BUTTON` is selected (unselected). We can assume that DXO.*Vsize* is 0 which means `Boolean`.

Here is code for the above radio buttons:

```
var b = DXOlast;            // Look for next button starting here.
Button Kind: Push Touch Check Circle Menu ;
b = next PSI_BUTTON in b; b.index = PushButton;
b.usv = &ButtonOptionKind; b.Vsize = 2;
b = next PSI_BUTTON in b; b.index = TouchButton;
b.usv = &ButtonOptionKind; b.Vsize = 2;
b = next PSI_BUTTON in b; b.index = CheckButton;
b.usv = &ButtonOptionKind; b.Vsize = 2;
b = next PSI_BUTTON in b; b.index = CircleButton;
b.usv = &ButtonOptionKind; b.Vsize = 2;
b = next PSI_BUTTON in b; b.index = MenuButton;
b.usv = &ButtonOptionKind; b.Vsize = 2;
```

In this case the `BUTTONs` share a single `ushort` variable `ButtonOptionKind`. The .*Vsize* of this variable is 2 bytes. Each `BUTTON`.*index* is set to the button kind's option value. When the user selects one of the radio buttons, the others are deselected and `ButtonOptionKind` is set to the new `BUTTON.`*index*.

6.6. Zooming In and Out using DXOs

Window zoom in/out is simple in principle: software should just draw text and graphics with all cöordinates and images (including characters) enlarged or reduced by a scale factor. However, this does not work with G-SWIM since cöordinates and font metrics are integers, so naïvely rescaling to the nearest font size and rounding text DXOs to the nearest integer cöordinates results in badly-spaced text. XXICC therefore uses a more intelligent approach to zooming in and out, which is implemented in XOE.

DXOs provide an interesting way to implement zoom in/out. As discussed earlier, we would like to precompute as much DXO information as possible so that refreshing and selecting are fast. So instead of scaling cöordinates when it draws a DXO list, XOE temporarily *enlarges or reduces* XOs when it formats them into DXOs.

Since XOE documents should look as close as possible to how they will be printed, zooming in or out must not change how lines are split at word boundaries. Thus XOE makes line split decisions assuming the document is unzoomed. On the other hand, it does not matter if individual lines are a little longer or shorter when zooming in or out, since the edges are ragged anyway, so we won't worry if some characters widen differently than others.

Since zoomed fonts may be rounded differently from unzoomed fonts, XOE does have to be careful so that both displayed and printed text and graphics look good. Here are the details:

1. Each XOE window has an unzoomed font table `Ufonts` and a zoomed font table `Zfonts`, which

contain copies of font pointers from the master font table. When the user zooms in or out, XOE updates `Zfonts` from the master font table.

2. A DXO's dimension fields -- *z*, *size*, and *a* -- are usually for the zoomed DXO. The character format indexes zoomed fonts. While we are formatting a `LINE` or NTST the dimension fields may temporarily be unzoomed.

3. Each zoomed DXO has *unzoomed width, height, and ascent* (*uw*, *uh*, *ua*) fields which store the DXO's height, width, and ascent above baseline prior to zooming. The unzoomed width is used by DXOs placed in a `LINE` so that lines always split at the same boundaries. The unzoomed height is important for zooming abridged subtrees, as we will see below. All DXOs have a *uaw* field which includes both *ua* (MSbs) and *uw* (LSbs). Leaf DXOs that are placed in `LINEs` have *uh*. An NT subtree uses `ENDOBJ`.*uh* to store the unzoomed height and `ENDOBJ`.*euh* for the `ENDOBJ`'s unzoomed height.

4. The first time XOE formats an XO it uses unzoomed dimensions and calculates unzoomed *w*, *h*, and *a*. Then XOE makes a second pass to calculate zoomed dimensions. In some cases, XOE waits until it formats an entire XOST such as a `LINE` or `DIALOG` before zooming it. XOE tries to reuse subtrees whenever possible to avoid redundant formatting and zooming.

5. Zooming a window could take a lot of time if XOE had to reformat an entire document to determine the new sizes of paragraphs and NTSTs. Instead, XOE abridges all the top-level DXOs and estimates their new zoomed heights by multiplying their unzoomed heights by the new zoom factor. Then XOE reformats the visible DXOs, but not the ones outside the window. As the user scrolls up and down, XOE reformats abridged DXOs and calculates actual zoomed heights, which may be different from the estimated heights. XOE corrects the scrolling for the new heights.

6.6.1 Formatting `LINEs`

When formatting a `LINE`, XOE uses unzoomed DXO dimensions until the `LINE` is complete. Then XOE calls "`close dxo`" to complete `LINE` formatting, as follows:

1. Calculate the unzoomed `LINE` height `ENDOBJ(LINE)`.*uh*. This is non-trivial, since the DXOs in the `LINE` need to have their baselines aligned so fonts with different ascents and descents may increase `ENDOBJ`.*uh*, especially if there are subscripts and/or superscripts.

   NTs within the `LINE` also need to have their ascents aligned. XOE stores an NT's unzoomed ascent in NT.*ua* and its unzoomed height in `ENDOBJ`(NT).*uh*.

2. Calculate the zoomed `LINE` height `LINE`.*h*, using the same algorithm as step 1 but with zoomed fonts and zoomed NT dimensions.

3. Horizontally place `LINE` contents using zoomed fonts. If the `LINE` is centered or right-justified, adjust *x* cöordinates. XOE zooms label widths for list items and headings when it first formats them, which simplifies "`close dxo`".

6.6.2 Formatting NTSTs

Formatting NTSTs like `DIALOGs` and in-line `BUTTONs` is actually easier, since we don't have to deal with line wrapping and text ascent.

1. Recursively calculate NTST unzoomed dimensions. This includes any automatically-sized PARABOXes, which XOE must format using unzoomed dimensions so line breaks are consistent.

2. Recursively calculate NTST zoomed dimensions. If a DXO has fixed dimensions, just multiply by the zoom factor. If the DXO's size is based on its components, zoom first and then add them up.

3. Place NTST using zoomed dimensions. There is no need to place using unzoomed dimensions.

4. XOE saves unzoomed DXO dimensions in DXO.*uaw* and DXO.*uh* (if DXO is a leaf) or ENDOBJ(DXO).*uh* (if DXO is a subtree).

6.6.3 Formatting FIGUREs

XOs in FIGUREs have explicit cöordinates which are unzoomed in an XO list and zoomed when formatted as DXOs. When the user edits a zoomed FIGURE, XOE converts the zoomed cöordinates back to unzoomed cöordinates. If zoomed in, round-off errors do not occur. Round-off errors may occur if zoomed out. XOE corrects them in most cases by snapping unzoomed cöordinates to a grid, but to correct them properly XOE needs to store the LSBs of unzoomed cöordinates to ensure they're rounded the correct way. *This is not implemented yet, so do not edit figures while zoomed out.*

6.7. Formatting and Editing Rotated FIGtext

XOE formats rotated text by first formatting it into horizontal (unrotated) LINEs. Then when the FIGtext CELL is complete, XOE rotates the CELL and its contents all at once. *The following has not been implemented yet*: If you select rotated FIGtext for editing, XOE brings up an editing window with unrotated FIGtext. This makes editing rotated text more convenient for the user (and for XOE). When you are done editing, press ESC and XOE redraws the updated rotated FIGtext.

When rotating or mirroring FIGtext, XOE rotmirs the CELL's outline (two possible orientations) and then re-places the LINEs in the CELL. *The following has not been implemented yet*: XOE makes sure the top spacing of the first LINE and the bottom spacing of the last LINE is preserved by rotmir. It uses the glyph boxes of those LINEs to do the calculations. This ensures that if, for example, FIGtext is centered at a component pin then the FIGtext will be placed the same way when the component is rotated or mirrored.

Chapter 7
XOE Tables

Tables have two major purposes in XOE documents: formatting tabular information (as in Microsoft Word) and processing array-oriented data as spreadsheets (as in Visicalc or Microsoft Excel). The beauty of XXICC is that tables are just another kind of object that can be edited within a document, and spreadsheets are just another way of representing GalaxC code.

Executable Tables in GalaxC highly experimental and are still at the "proof of concept" stage. Many features are TBD.

7.1. Tables in XO Lists

Tables are defined hierarchically and are very similar to dialog stacks. A table consists of a `TABLE` instruction followed by one or more rows and terminated with `ENDOBJ`. Each row consists of `HSTACK` followed by one or more `CELL` objects and terminated with `ENDOBJ`. All tables are two-dimensional, i.e., a single row still starts with `TABLE` and a single column still has an `HSTACK`+`ENDOBJ` for each row. Here is a sample 2 x 3 table and its PSI code:

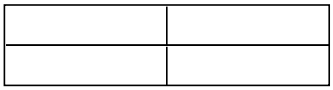
a[0,0]	a[0,1]	a[0,2]
a[1,0]	a[1,1]	a[1,2]

```
TABLE(Tstyle, options);
  HTABLE(height, options);              // First row.
    CELL(width, options); TEXT "a[0,0]"; ENDOBJ;
    CELL(width, options); TEXT "a[0,1]"; ENDOBJ;
    CELL(width, options); TEXT "a[0,2]"; ENDOBJ;
  ENDOBJ;                               // End of first row.
  HTABLE(height, options);              // Second row.
    CELL(width, options); TEXT "a[1,0]"; ENDOBJ;
    CELL(width, options); TEXT "a[1,1]"; ENDOBJ;
    CELL(width, options); TEXT "a[1,2]"; ENDOBJ;
  ENDOBJ;                               // End of first row.
ENDOBJ                                  // End of table.
```

`TABLE` is essentially the same as `VSTACK` except that `TABLE` has additional properties and must be nested two deep, whereas `VSTACK`s can be nested arbitrarily deeply. Like `VSTACK`, `TABLE` has an *options* operand specifying justification, frame width, and separation lines.

Each cell in the tree begins with a `CELL(`*w, options*`)` instruction where *w* is the cell width. Any objects between `CELL` and `ENDOBJ` are the contents of the cell, e.g., `TEXT`. The contents may be empty, i.e., `ENDOBJ` may immediately follow `CELL`. For document tables or literal data, the cell's contents are displayed. For spreadsheets, the cell can display a GalaxC expression or its calculated run-time value. In general, a cell property such as character format applies to all the remaining cells of the table unless overridden.

You can insert and edit tables using XOE. The Table button in the Insert menu (§2.5) creates an empty 2 x 2 table like this:



Editing tables is described in §2.10.

7.2. GalaxC Expressions in Cells

This section describes incorporating GalaxC expressions into table cells to provide functionality similar to spreadsheets. If you are familar with existing spreadsheets like Visicalc and Microsoft Excel, you will immediately notice that GalaxC tables use an entirely different approach to naming cells. The former implicitly name each cell with a column letter and a row number, while GalaxC requires that cells be given explicit names to use their values elsewhere within the same table or elsewhere in a GalaxC program. The Visicalc approach saves the trouble of explicitly naming cells, but makes it easy to name the wrong cell in complex expressions or when moving rows and columns. GalaxC's explicit cell names provide implicit commenting, e.g., calling a cell "AGI" is much more understandable than "J42". In practice, GalaxC cell names are usually defined once and then copied to other cells to form one- or two-dimensional arrays of cells (called *blocks*) so explicitly naming cells is usually not a burden.

In a GalaxC table, each cell may have the following properties:

- *Name*: name of cell so it can be referenced by other cells.
- *Expr*: GalaxC expression to compute the cell's value. The expression may be a constant or a function of GalaxC variables, which may be the values of other cells.
- *Type*: normally implied by expression, but may be given explicitly for an empty cell.
- *Data format*: `printf`-like format string specifying how to display the computed value as text. The data format string does not specify text formatting such as italics, bold, and centering -- XOE handles these.

A cell that does not have an expression is called a **blank cell**. It has by default the value 0 -- or `NULL` for `strings`. At run time it is displayed as an empty cell unless a run-time value is assigned (see §7.6). Some spreadsheet functions such as computing averages do not count blank cells.

An array of cells in a table with the same name is called a *named block*. All cells in the block must have the same type and data format. Cells in the block may be blank or may be label cells, described below. An *unnamed block* is a subtable with the same type (which may be `void`) and data format, but no name. Each cell in a table is in exactly one block.

XXICC encodes cell properties as cell `TEXT`, using the following notations:

name`:` *expr*`;` *format*	Specify name, expression, and data format explicitly. Deduce type from *expr*, which may include typecasting.
name`:` *type*`;` *format*	Blank cell with specified name, type, and data format.
name`:` *expr*	Specify name and expression: copy data format from previous cell.
name`:` *type*	Specify name and type of blank cell: copy data format from previous cell.

At run time, XOE displays the computed values of cell expressions using the specified format. Blank cells that have not been assigned run-time values are shown as empty cells (see §7.6).

In the unlikely case that the value of a cell is to be of type `type`, put *type* in parentheses so it is evaluated as an expression. Also do this if *expr* is a semicolon expression.

If you omit *name*, i.e., start text with ‘`:`’, then the cell is unnamed and any cells copying properties are also unnamed. You cannot specify *format* by itself: *format* must have at least a type.

If you omit *name* and ‘`:`’, GalaxC copies properties from the previous cell, usually the one above the current cell. This is the easiest way to create named blocks. Here are the possible notations:

expr	Just specify expression: copy name and data format from previous cell. Deduce type from *expr*, which may include typecast.
"*text*"	Literal `string` expression.
"	Ditto: copy name, expression, and data format from previous cell.
empty	Empty cell = blank cell with name, type, and data format from previous cell.
type	Specify type for unnamed blank cell: copy data format from previous cell.
`:`	Unnamed blank cell: copy type and data format from previous cell.

Ditto is normally used for columns that contain an expression which is exactly the same as the previous cell. Since expressions may use row and column index variables, there are lots of opportunities for using ditto. The compiler can optimize these into loops.

A cell may contain a *label* instead of an expression. At run time XOE displays the label and treats the labeled cell as a blank cell. Labels are usually in unnamed blocks, but they may be inside a named block to comment a row, column, or cell. Here are the notations for labels:

`,`*label*	Show *label* as value of blank cell. Copy name, type, and data format from previous cell. Copy expression, but don’t compile it. The next cell can copy the previous cell’s expression.
`;`*label*	Show *label* as value of unnamed `void` blank cell. Copy data format from previous cell.

Think of “comma” as denoting a “comment”. Think of “semicolon” as a combination of “comment” and the colon prefix for unnamed cells. XOE does not show the comma and semicolon at run time.

By default, expressions copy properties from the cell above the current cell, or to the left if this is the first row. You force the “previous” cell to be the cell to the left by inserting ‘`}`’ before the first character, e.g.:

`}"`	Left ditto: copy all properties from cell to left of this one.
`}`	Blank cell: copy name, type, and data format from cell to left of this one.

The default format is `"%d"` and the default type is `void`. *Improve and elaborate.*

Do not put spaces before prefix characters -- `,;}":` -- or between them.

Each block has two *implied variables*:

`i`	Row index, numbered from 0.
`j`	Column index, numbered from 0.

Cell [0,0] is the upper left element of the block. Empty cells and comment cells are included in the array, except for rows that are all comments: GalaxC skips those as far as row indexing is concerned. *We may add a way to select 1-based row and column indexing.*

A cell *name* corresponds to a GalaxC variable and may be a scalar or a one- or two-dimensional array,

depending on whether it is the name of adjacent cells. Use the notation *name*`[i]` to access row or column elements, and *name*`[i,j]` to access 2-D array elements.

7.3. Table Evaluation

At the present time, GalaxC evaluates a table in row major order, from the upper left cell to the bottom right cell evaluating each row from left to right before proceeding to the next row. A cell must not depend on a value that is computed in a later cell. When evaluating a cell, `i` and `j` are set to the block row and column of that cell. You can use the values of `i` and `j` to index blocks in the same table and/or in other tables.

At some point we plan to remove the row major order restriction, allowing any cells to be in any sequence as long as there are no circular dependencies. In fact, we plan to add this as a general capability of GalaxC where you can define an unsequenced block using the notation "`unseq {`*statements*`}`" or a similar notation. This will generate PSI `UNSEQ` tags which can be used to reorder the execution of PSI code.

7.4. Compiling Tables

To compile a GalaxC program that includes tables, press `F6` or `sh-F6` in XOE the same way you compile any GalaxC program. To run a GalaxC program that includes tables press `ctl-F6` as usual. XOE will automatically switch to "show cell values" mode and display cell values instead of expressions. To exit "show cell values" mode, press `ESC`. To edit the value of a table cell, select it as if you were editing its expression. You can then edit the value and press `ESC` or `RET` to update the cell, assuming you entered a legal value. XOE also updates the cell if you leave it using the mouse or cursor control keys.

Compiling a table requires two passes. Pass 1 determines the name, type, size, and position of each block in the table. Pass 2 generates the code to evaluate the table and tells XOE where to find the run-time values of cells.

In Pass 1, GalaxC builds a pattern table of named blocks, each one representated as a variable pattern (see `acg.h`) with `.size` = number of columns, `.depth` = number of rows, and `.aux` encoding the table coordinates of the upper left cell of the block. The `.value` field will eventually be the address of the block, but in Pass 1 we can't compute that address until all blocks are complete.

GalaxC also keeps track of unnamed blocks, but closes them when it declares a new named block. Unnamed blocks are usually columns, but may be merged with adjacent columns if they have the same type and data format.

In Pass 2, GalaxC allocates global memory (for now) to store table values and calls XOE for each named and unnamed block. XOE collects this information in a list of *hidden toons* (I'll explain the name in a later revision) which XOE uses to calculate the addresses of cell values so XOE can display or update them at run time.

Pass 2 also generates code to calculate cell values as described in §7.3. For the most part, this is code simply evaluates each cell and stores the result into table memory. This is fine for the first evaluation of a table, but a spreadsheet should allow users to change run-time cell values and watch the table immediately recalculate dependent cells using the new value. (GalaxC actually recalculates all cells, not just the dependent ones.) GalaxC implements dynamic recalculation by generating conditional code for cells with constant values -- i.e., those that can be calculated at compile time. These are *independent cells* because their values do not depend on other cells or GalaxC variables. The conditional code stores the initial values the first time the table is evaluated. However, the user can edit the values of independent cells using XOE

and subsequent evaluations of the table use the modified values. Cells with non-constant expressions -- *dependent cells* -- evaluate the usual way and XOE prevents users from changing their run-time values. (In the current implementation, a user can edit a dependent cell's run-time value but the cell will immediately revert to its previous value when XOE recalculates the table.)

7.5. Sample Tables

Here is a simple table which calculates and displays squares and cubes for `i` = 0 through 6.

i	i*i	i*i*i
"	"	"
"	"	"
"	"	"
"	"	"
"	"	"
"	"	"

The first row defines expressions `i`, i^2, and i^3. The remaining rows are copies of the expressions in the first row. When you compile and run this table (`F6` followed by `ctl-F6`), XOE displays:

0	0	0
1	1	1
2	4	8
3	9	27
4	16	64
5	25	125
6	36	216

The first column displays the value of row index `i`, which runs from 0 to 6. The second column display `i` squared, the third `i` cubed. None of the cells are named. Since no data format is specified, GalaxC assumes "`%d`" for `int` values.

Here is another simple example with named cells followed by some normal GalaxC code.

x: i	y: i*i	ch: char
"	"	'j'
"	"	'b'
;comment	"	'c'

```
// Here is text after the table.
int var i;
ch[0] = 'X';
for i = 0 thru 3
do printf("%d: x = %d, y = %d, ch = %c\n", i, x[i], y[i], ch[i]);
```

In this case column 0 is a one-dimensional `int` block named `x`, column 1 is `int` block `y`, and column 2 is `char` block `ch`. The expressions "`x: i`" and "`y: i*i`" define the values for columns 0 and 1. The

expression "`ch: char`" only defines the type of column 2. Its values are defined in the cells below. Since "`ch: char`" does not set the value of cell (0, 2), we use the GalaxC code "`ch[0] = 'X'`".

When you compile and run this table, XOE displays:

0	0	X
1	1	j
2	4	b
comment	9	c

As before, XOE displays `int` values using default data format "`%d`" and `char` values using format "`%c`". XOE displays the label cell [3, 0] with '`;`' removed.

Running the program also prints values to the Output window, as follows:

```
0: x = 0, y = 0, ch = X
1: x = 1, y = 1, ch = j
2: x = 2, y = 4, ch = b
3: x = 0, y = 9, ch = c
```

This shows that the values of the named blocks in the table are available outside the table. `x[3]` is not actually defined, so it doesn't matter what value is printed. Note that row index `i` is not defined outside the table: the main program had to declare its own copy of `i`.

Here is a table which computes the hypoteneuse length *c* of a right triangle given the lengths *a* and *b* of the other two sides. All values are single-precision `float`. The first two columns have independent cells `a[i]` and `b[i]`. The third column has dependent cells `c[i]` with values calculated using Pythagoras' formula.

;a	;b	;c = sqrt(a^2 + b^2)
a: float	b: float	c: sqrt(a[i]*a[i] + b[i]*b[i])
"	"	"
"	"	"

When you compile and run this table, XOE displays:

a	b	c = sqrt(a^2 + b^2)
0.00	0.00	0.00
0.00	0.00	0.00
0.00	0.00	0.00

You can modify the values in columns `a` and `b` and XXICC will automatically recalculate the values in column `c`:

a	b	c = sqrt(a^2 + b^2)
3.00	4.00	5.00
5.00	12.00	13.00
100.00	100.00	141.42

Press ESC to leave “show cell values” mode and display cell expressions, i.e., GalaxC source code.

7.6. Issues

1. At the present time XOE displays default values for blank cells that have not been assigned run-time values instead of showing them as empty cells. We need a bit map for each table indicating which cells are empty.

2. The current implementation does not allow explicit format strings: it uses "%c", "%d", or "%4.2f" according to the cell’s type.

3. GalaxC uses global memory to for storing table data. This will be generalized in later versions.

4. In the current implementation, a user can edit a dependent cell’s run-time value but the cell will immediately revert to its previous value when XOE recalculates the table. XOE needs a way to tell if a cell is dependent cell and if so forbid editing.

5. You cannot use copy/paste to copy the values shown in multiple table cells at this time. You can copy cell text within a single cell if you click into it to edit the cell’s value.

6. Compiling and running with multiple windows is not yet stable. Be sure to escape out of “show cell values” mode before compiling a different window.

Chapter 8
XOE Figure Editing

This chapter describes how to edit figures in a XOE document. XOE figure editing commands are based on the Galaxy Logic Editor (GLE) which was part of the Galaxy CAD System [JFB 92a]. Unlike GLE, figure editing is built into XOE instead of being a separate tool. Whenever possible, figure editing uses the same commands as text editing. For example, the Edit -- e.g., Undo -- and Style menu commands are mostly the same.

Figure editing commands are designed for efficiency and ease of use rather than ease of learning. Instead of drawing tool icons, XOE uses single-key commands such as **b** to create a rectangular box and **c** to create a circle. (*When it gets implemented, you will be able to use the Draw menu if you forget which keys to use.*) For best results, read the XOE User's Guide (Chapter 2) and this chapter instead of experimenting.

Revision 0.0h is the third released version of XOE figure editing. As such, some details have been deferred to later versions and users should expect occasional errors. Before using XOE figure editing, be sure to check the Issues list in §8.10.

To use figure editing, you must run XXICC with a target file of "`fig`", for example:

```
xxicc fig
```

If you run XXICC with the command "`xxicc xoe`" or just "`xxicc`", XOE will display figures but you will not be able to edit them. See [JFB11b] for details.

8.1. Creating and Sizing a Figure

To create a figure in a XOE document, select Figure from the Insert menu. XOE replaces the current selection with a default size figure and leaves the blinking text cursor after the figure. XOE treats the figure as a paragraph within the document so you can (for example) change its justification using commands from the Parag menu.

To select an entire figure in a XOE document, click mouse button 1 in the area to the right of the figure. You should see a blinking cursor at the upper left corner of the figure or at the end of the text line preceding the figure if that line doesn't end with an `LF` (created using `ENTER` or `RET`). Next, press `sh-RIGHT` to move the cursor one character or XO. This selects the entire figure, which is shown with a gray background. You can then select a paragraph format such as Center or Right from the Parag menu.

To resize a figure, move the mouse to the bottom or side of the figure and press mouse button 1. A (usually) red resize box appears and the mouse should change to tiny arrows pointing left, right, up, and down. Drag the mouse to the desired figure size and release mouse button 1.

You can always move the bottom edge of the figure. Which side(s) you can move depends on whether the figure is left-justified, right-justified, or centered within the document. If left-justified, you can move the right edge or bottom-right corner. If right-justified, you can move the left edge or bottom-left corner. If the figure is centered, you can move either side or either bottom corner.

As you move a figure edge or corner, XOE snaps it to the figure's grid.

8.2. Grids

You can select a grid to make it easier to align figure XOs. XOE displays the grid as an array of blue dots and snaps XOs to the grid when you create or move them. At the present time, XOE only allows you to select three grid spacings: 10, 4, and 1. The number indicates the number of pixels per grid dot if unzoomed. If you zoom in or out the grid expands or contracts accordingly. Grid spacing 10 is a good grid spacing for drawing schematic diagrams and is the default. Grid spacing 4 is good for more precise placement of text and for drawing schematic symbols. Grid spacing 1 turns off the grid and lets you place XOs anywhere.

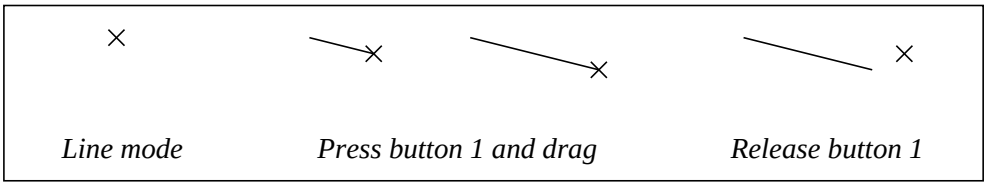
To toggle between grid spacings, press **#**.

8.3. Creating Basic Shapes

Once you have created a figure, the next step is creating basic figure elements like lines, boxes, circles, and arcs. Each XO type has a figure editing **mode** with a distinctive mouse shape. To begin editing a figure, click the mouse inside the figure at a location with no graphics. This enters *Select* mode and changes the mouse to an arrow. In addition, the blinking cursor vanishes -- it's only present in text editing modes. In figure editing modes the keyboard is used for single-character commands such as those described in the next sections.

8.3.1 Creating Lines

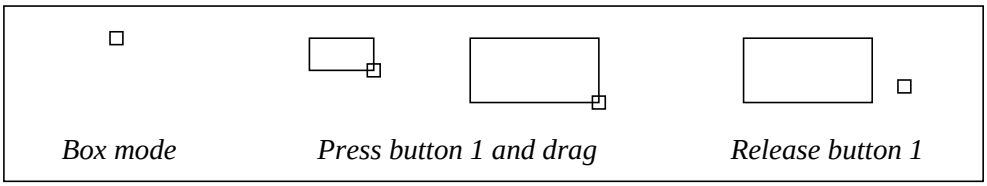
To create a line, first press **l** (lower-case **L**) to enter *Line* mode. The mouse changes to a small 'x' to show that diagonal lines are welcome. Next, move the mouse to the first vertex of the line and press mouse button 1. Finally, drag the mouse to the second vertex of the line and release button 1. The line stretches as you drag the mouse. Line vertices are snapped to the grid.



XOE stays in Line mode so that you can create a series of lines. When you are done creating lines, press ESC to return to Select mode. You can also press ESC while dragging the mouse to cancel the new line.

8.3.2 Creating Boxes (Rectangles)

To create a rectangle, first press **b** to enter *Box* mode. The mouse changes to a small square. Next, move the mouse to a vertex of the box and press mouse button 1. Finally, drag the mouse to the opposite vertex of the box and release button 1. The rectangle stretches as you drag the mouse. Box vertices are snapped to the grid.

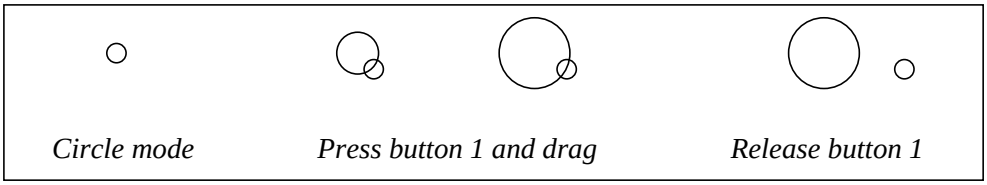


XOE stays in Box mode so that you can create a series of rectangles. When you are done, press ESC to

return to Select mode or a different mode key such as '**l**' to enter that mode. You can also press `ESC` while dragging the mouse to cancel the new box.

8.3.3 Creating Circles

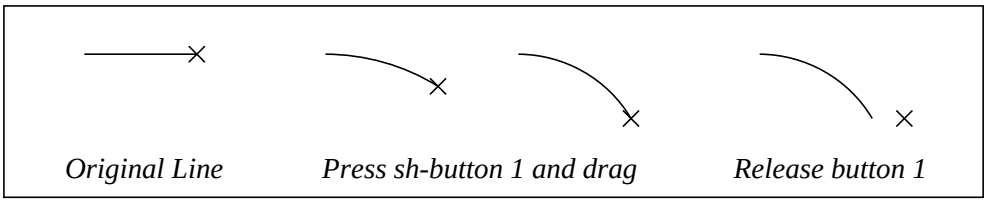
To create a circle, first press **c** to enter *Circle* mode. The mouse changes to a small circle. Next, move the mouse to the center of the circle and press mouse button 1. Finally, drag the mouse to the circumference of the circle and release button 1. The circle expands as you drag the mouse. The circle center and circumference point are snapped to the grid.



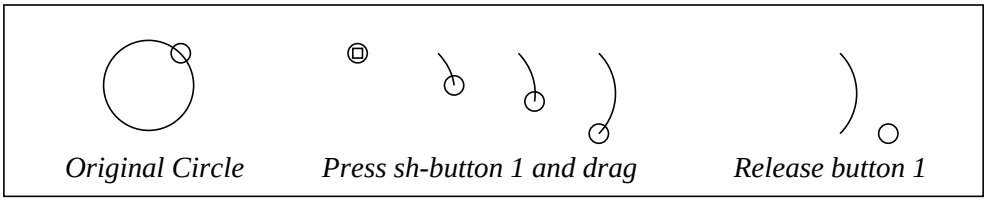
XOE stays in Circle mode so that you can create a series of circles. When you are done, press `ESC` to return to Select mode or a different mode key to enter that mode. You can also press `ESC` while dragging the mouse to cancel the new circle.

8.3.4 Creating Arcs

You can convert a line or circle into a circular arc. To convert a line to an arc, enter Line mode unless you're already in it. Next, move the mouse to an endpoint of an existing line and press mouse button 1 with `shift`. Then as you drag the mouse, the line changes to an arc with starting point equal to the other (fixed) point of the original line, initial arc direction the same as the original line, and radius calculated so that the arc ends at the mouse (snapped to the grid). The arc may be clockwise or counter-clockwise, determined by which side of the original line the mouse is on. When you're happy with the arc, release mouse button 1 and the line is now an arc.



To convert a circle to an arc, enter Circle mode unless you're already in it. Next, move the mouse to any location on the circumference of a existing circle and press mouse button 1 with `shift`. The circle disappears and a small square is shown at the selected point of the circumference which will be the starting point of the arc. (XOE adjusts the circle radius so that it matches the mouse, snapped to the grid.) Then as you drag the mouse, the small square turns into an arc with initial direction tangent to the original circle at the starting point and radius calculated so that the arc ends at the mouse (snapped to the grid). The arc may be clockwise or counter-clockwise, determined by which side of the tangent line the mouse is on. When you're happy with the arc, release mouse button 1 and the circle is now an arc.



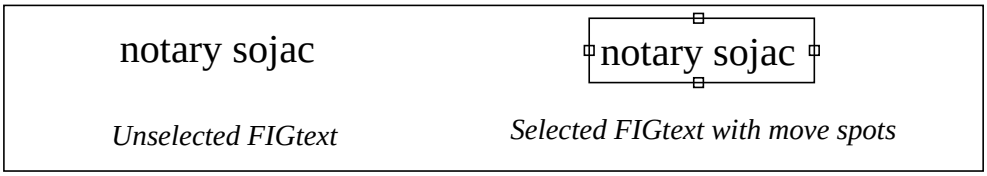
You can also modify existing arcs in both Line mode and Circle mode. Just press mouse button 1 with `shift` at either end of the arc and drag it to a new location. The other end of the arc remains fixed. `ESC` cancels a modification in progress.

8.3.5 Creating Text

To add text to a figure, press **t** to enter *Text* mode. The mouse changes to a vertical line. Move the mouse to where you want the text and press mouse button 1. A blinking cursor appears at that location. Then type in text using standard XOE commands as described in Chapter 2. When you are done editing figure text, press `ESC` to return to Select mode or click the mouse at a vacant location in the figure to start new figure text.

Editing figure text is similar to editing a table cell. You can change character formatting -- **bold**, *italics*, etc. -- but you cannot do paragraph formatting such as lists and headings. In fact, figure text is implemented as a special kind of cell called `FIGtext`. At the current time, `FIGtext` editing is limited: `FIGtext` is always left-justified and XOE automatically sizes the cell to fit the enclosed text. In the future, you will be able to select right- and center-justification, and specify cell width for automatic word wrap. For now, use `ENTER` or `RET` for multi-line text items. *You'll also be able to rotate `FIGtext` 90° some day, and specify a border.*

To edit existing `FIGtext`, first click it with mouse button 1 to select it. A rectangular border appears around the text with four *move spots*. If you click inside the border, the blinking cursor appears at that point and you can edit the `FIGtext` using standard XOE commands. If you press button 1 on one of the move spots, you can drag the `FIGtext` to a new location. When you release button 1, XOE places the `FIGtext` at that location, snapping the center of the move spot to the grid. We will explain selecting, moving, and copying in the next section.



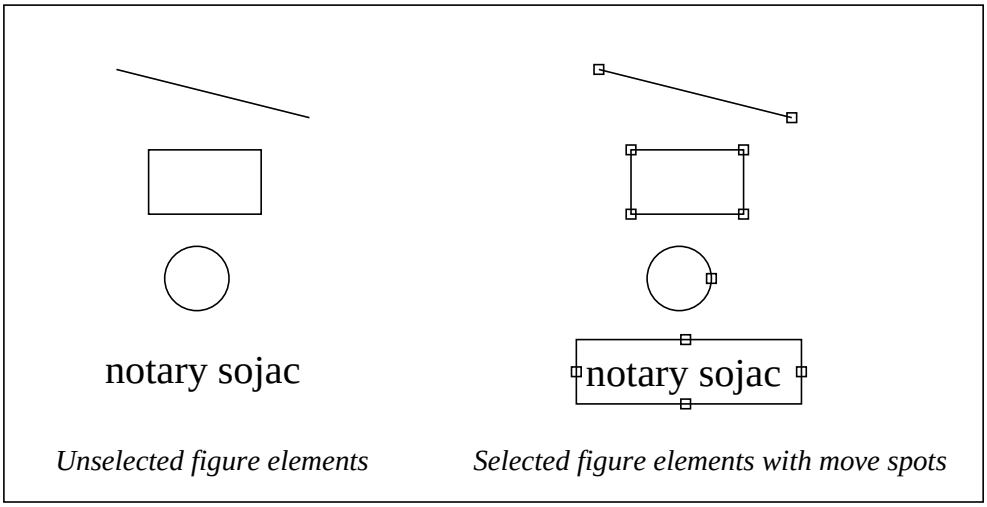
You can change `FIGtext` style by selecting one or more `FIGtext` objects and issuing a command from the XOE Style menu or its keyboard equivalent. All Style commands work except for Subscript and Superscript. You can change the font for all `FIGtext` in a figure by selecting the figure as described in §8.1 and issuing a Style command. You can also change fonts or styles -- including Subscript and Superscript -- when you are editing a single `FIGtext`. Changing `FIGtext` style has not been tested at length, so expect anomalies.

8.4. Selecting, Moving, and Copying

Selecting figure XOs is quite different from XOE text. In a figure, you can select any combination of figure

XOs whereas outside a figure you must select a contiguous sequence of characters or XOs. Here is the behavior you should expect in Select mode:

- Clicking mouse button 1 (with no shifts) on an unselected XO selects that XO and deselects everything else. XOE shows that a basic figure element -- a line, rectagle, circle, or arc -- is selected by displaying move spots at its vertices. XOE shows selected `FIGtext` by displaying a box with four *move spots*. XOE shows a selected schematic symbol or other component by giving it a gray background.



- Clicking mouse button 1 with `shift` on an XO *toggles* it: select it if unselected, and unselect it if selected.

- Clicking mouse button 1 (with no shifts) at a vacant spot in the figure deselects all selected figure XOs.

- Pressing `ESC` deselects all selected XOs.

- Pressing mouse button 1 (with no shifts) at a vacant spot and then dragging it creates a *group selection box*, usually shown red. When you release button 1, XOE selects all the XOs in the box and deselects everything else. If a line crosses into the box so that one vertex is in the box and the other outside the box, the line will be treated as a *rubber line* and stretch when you move the selection. The group selection box is not snapped to the grid.

- Pressing mouse button 1 with `shift` at a vacant spot and then dragging it also creates a group selection box, but in this case when you release button 1 XOE toggles all XOs in the box. Sometimes it's easier to select a group of XOs and then toggle off XOs that you don't want selected.

- If you click the mouse over several overlapping or abutting XOs, XOE prefers to select (1) a move spot, (2) an XO that is already selected, or (3) the smallest XO, in that order.

- Sometimes it's easiest to select XOs by first selecting a group and then toggling off the XOs you don't want selected.

Once you have one or more XOs selected, you can move, copy, and delete them.

8.4.1 Moving Figure XOs

All move operations use mouse button 1 with no shifts.

To move a selected XO (or XOs) other than `FIGtext`, press mouse button 1 over any drawn part of the XO(s) except a move spot, drag it (them) to a new location, and release button 1. XOE places the XO(s) at the new location. If the figure is a schematic, XOE also analyzes connectivity changes.

To move `FIGtext`, press mouse button 1 on one of its move spots, drag it to a new location, and release button 1.

If you press mouse button 1 on an XO's move spot (other than `FIGtext`), XOE deselects all other XOs and lets you stretch the XO as a *rubber shape* by dragging the mouse. When you release button 1, XOE updates the XO's size and shape, and deselects it.

As you drag the selected objects and when you release button 1, XOE snaps the mouse to the grid. If you selected multiple XOs, XOE retains the same grid alignment that the XOs had when you started the move. If you select a single XO, XOE snaps the XO to the grid before starting the move so that the XO ends up aligned to the grid. This is helpful for cleaning up diagrams that were edited before you decided on a new grid spacing.

If you selected the XOs using a group selection box, lines that cross into the box become rubber lines and stretch during the move. This allows you to move part of a schematic or other connected diagram while preserving connections.

Except when stretching a single rubber shape by dragging its move spot, XOE reselects the moved XOs at the end of the move. This allows you to move the same XOs without having to reselect. Pressing `ESC` or clicking button 1 in a vacant part of the figure deselects them.

8.4.2 Copying Figure XOs

To copy the selected XOs, press button 1 with `ctl` pressed over any drawn part of any selected XO. As you drag the mouse, a copy of the selected XOs moves with it. Releasing button 1 places the copy and leaves it selected for a subsequent move or copy. There is no special handling of `FIGtext` and XOE treats rubber lines as normal lines.

You only need to hold down `ctl` when you first press button 1. It does not need to be held as you drag or when you release.

As with moves, XOE snaps the mouse to the grid as you drag and when you release. If you selected multiple XOs, the copy has the same grid alignment as the original XOs. If you selected a single XO, XOE snaps the copy to the grid when it starts the copy operation so that the XO ends up aligned to the grid.

8.4.3 Deleting Figure XOs

To delete the selected XOs, press `DELETE` or `BACKSPACE`.

8.4.4 Origin Lines

In addition to the grid XOE lets you set **origin lines**, which act as an additional grid coördinates. To set the origin lines, issue the *Origin* command **o**. XOE displays red *x* and *y* axis lines -- initially at the center of

the figure -- which you can move independently using the mouse. When you draw a new XO or move an existing XO, XOE snaps to the *x* or *y* origin line when it's close. Origin lines have additional uses, which will be described later.

To remove the origin lines, press **o** twice. The first press sets them to the center of the figure (snapped to the grid) and the second press moves them to the upper-left corner of the figure where they are invisible.

8.4.5 Copy, Cut, and Paste

While mostly complete in 0.0g, Copy and Paste may still be unstable. Specifically, there may be problems pasting non-figure text and XOs into a figure. Save frequently, and make back-up copies of saved files to minimize lost work.

To copy or move figure XOs to another figure or document, use the Cut, Copy, and Paste commands which have the usual keyboard equivalents `ctl-X`, `ctl-C`, and `ctl-V`. The Copy command copies all selected XOs to the clipboard. If multiple XOs are selected, it calculates the selection's *bounding box* -- i.e., the smallest box containing all selected XOs -- and sets the *origin* of the clipboard copy to the center of that box, snapped to the grid. This origin will be used when placing copies of the clipboard. If a single component (§8.6) is selected, XOE sets the clipboard origin to the component's origin. If one or both origin lines (§8.4.4) are in the selection's bounding box, then XOE uses the origin line(s) as the clipboard origin.

Cut is the same as Copy, except that it deletes the selection after copying it to the clipboard.

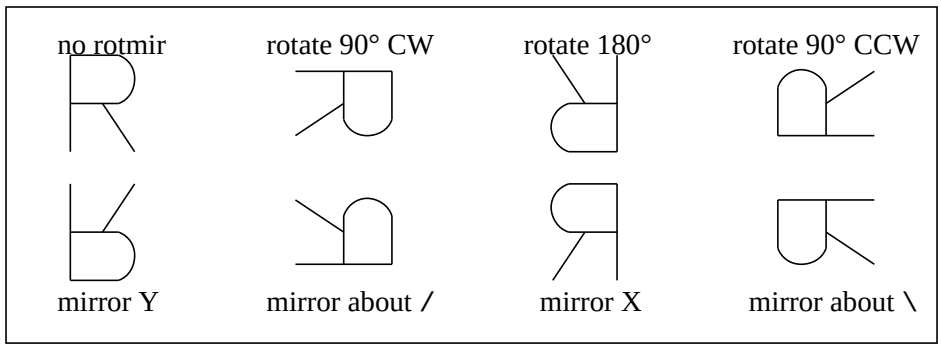
Pasting is a bit more interesting. With text, you paste at the blinking cursor or replace a continguous block of selected characters. In a figure, you must somehow tell XOE where to place the copied XOs. XOE handles this by initially placing the clipboard copy's origin at the mouse location where you pressed `ctl-V` or the last mouse location before selecting Paste from the Edit Menu. However, XOE does not place the copied XOs right away: instead it attaches them to the cursor as if a Move operation is already in progress. You can then move the XOs to where you want them and click mouse button 1 to place and connect them, or press `ESC` to cancel pasting.

XOE deselects any figure elements that were selected when you gave the Paste command. It does not replace them as it would with text.

Pasting component instances has additional functionality: see §8.6.3.

8.5. Rotating and Mirroring

XOE graphics can be rotated in 90° increments and mirrored horizontally or vertically. There are eight different *rotmir* orientations for graphics:



To rotate or mirror graphics, select one or more XOs and press one of the *rotmir keys*:

`@` Rotate the selected XOs 90° counter-clockwise, which is how one usually draws the `@` sign.

`|` Mirror the selected XOs about a vertical axis through the *rotmir center*, defined below. This is also called "mirror horizontally" and mirrors X coördinates.

_ Underscore: Mirror the XOs about a horizontal axis through the rotmir center. This is also called "mirror vertically" and mirrors Y coördinates.

You need to press a sequence of rotmir keys to get some combinations. For example, to mirror about `/`, you can mirror Y with '_' followed by one '`@`' rotation or mirror X with '`|`' followed by three '`@`' rotations.

The *rotmir center* is usually the center of the bounding box of the selected XOs, snapped to the grid. If you select a single component (§8.6) the center is the component's origin. If one or both origin lines (§8.4.4) go through the bounding box, XOE uses the origin line(s) instead.

You can also rotmir mobile graphics during a move, copy, or paste operation. For move and copy, keep holding down mouse button 1 when you press the rotmir keys. XOE ignores rotmir keys if you are adding a new basic shape or stretching a single rubber shape.

8.5.1 Rotating `FIGtext`

Most graphics is rotated and mirrored by transforming each vertex. This doesn't work well with text, since mirrored and upside-down text is hard to read. XOE only allows two text orientations: horizontal and rotated 90°. To minimize the amount of head tilting needed to read rotated text, XOE rotates all text either clockwise or counter-clockwise as selected by the user. This also minimizes the number of extra fonts that need to be created. The default is counter-clockwise; for clockwise, include the `-rtcw` option when starting XXICC.

Only `FIGtext` can be rotated or mirrored, and then only if it contains text and not other XOs. This is enough for XOE's primary purpose for rotmir text, which is pin labels on schematic symbols. `FIGtext` may contain `TAB` characters and may have multiple lines.

To rotmir text, XOE rotates or mirrors the `FIGtext` box that contains the text and then fits the text into that box. XOE displays unrotated text if the box is rotated 180° or mirrored horizontally or vertically, and displays rotated text using the 90° clockwise/counter-clockwise setting. See examples in §8.6.3.

At the present time, text can only be rotated as part of a component instance. At some point you will be

able to rotmir individual text such as component and net labels.

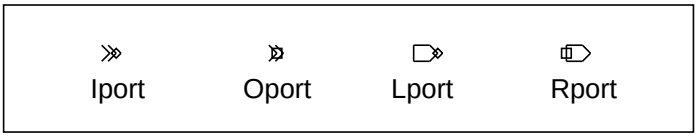
8.6. Schematics

Up to this point we have seen enough figure editing commands to create simple figures with lines, boxes, circles, arcs, and text. While these are useful for illustrations, XOE's real power is that you can create figures with connectivity, including schematic diagrams with electronic symbols, logic diagrams, flowcharts, and state diagrams. We will refer to them collectively as *schematics*.

A **schematic** is a collection of components and nets, defined circularly as follows:

- A **component** is an instance of a submodule such as a circuit element or a logic gate. A component may have connection points called **pins**, each of which can connect to a net. A component may have `FIGtext` attached to it, such as an instance name -- e.g., "R21" -- or a component value -- e.g., "10KΩ".

  Components may be instances of **port** submodules, which define the interface of a hierarchical module or act as "off-page" connections between multiple figures making up a complete design or a module. Here are the standard port symbols (*add bidirectional* `IOport` *which looks like* "<>"):

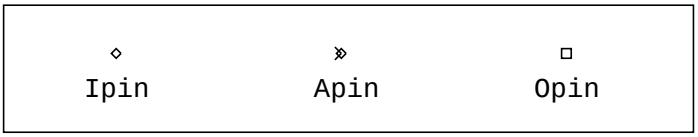


- A **net** is a connection of line segments called **wires** which connect component pins to each other. Some of the line segments may be **buses**, which are multiple wires running in parallel. When three or more wires join at a point, XOE automatically creates a **connection dot** to show that the lines connect to each other. A net may also have `FIGtext` for a net name, bus numbering, and other net attributes.

- A **module** is a connection of **components**, **nets**, and **ports** which form the building blocks of a **modular** or **hierarchical design**. A module may be shown as a single figure, or may consist of multiple figures connected by ports used as off-page connections.

- A **submodule** (or **submod**) is a symbol representing a lower-level (*child*) module that is instantiated in a *parent* module. We sometimes use "submodule" to refer to the child module itself, but usually we mean the symbol for it. A leaf submod does not have a corresponding module: it may be a primitive circuit or logic element.

  Each submodule symbol has graphics, plus **pins** for the connection points. The pins are usually named so it's clear which submod pins correspond to module ports. Some primitives do not have names and rely on the order of pins in the submod's XO list. Here are the standard pin symbols used in submods, from library file `ports.xoe` (*add bidirectional* `IOpin`):



  `Apin` has an arrow-head used for explicit inputs. When you draw a wire to a component pin, XOE

erases the small diamond or square so you can tell connection has been made. `Apin`'s arrow-head joins the wire to become an arrow.

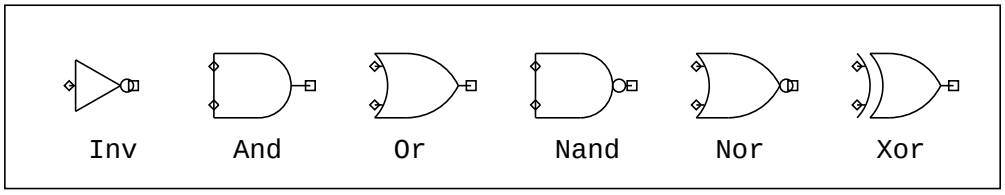
Module ports and submod pins are both represented internally as components with an assigned I/O direction. Submod pins always have names ending with "`pin`".

Submods can also be used in visual figures with no connectivity so that you can make multiple instances of the same group of graphical elements. This is like clip art in a drawing program. With XOE, you can make a change to one instance of a named submod and have XOE update all other instances automatically.

8.6.1 Drawing a Schematic

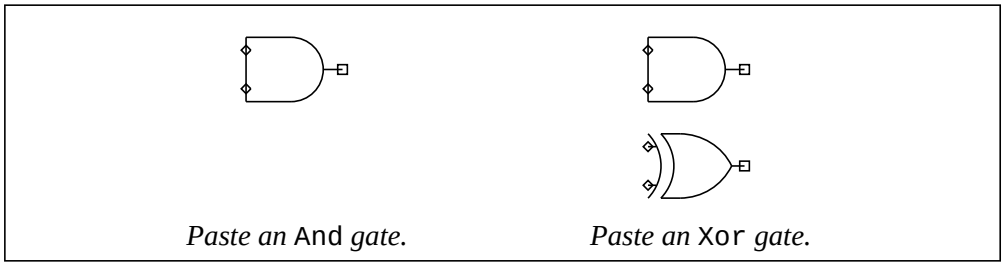
Here is a quick example of drawing a schematic to get you started. First, insert a blank figure into your XOE document as described in §8.1. It will normally have the default grid spacing of 10.

Next, add some components. You can copy and paste existing components from another figure or document. For logic diagrams, open the logic gate library `gates.xoe` in another XOE window. It will look something like this:



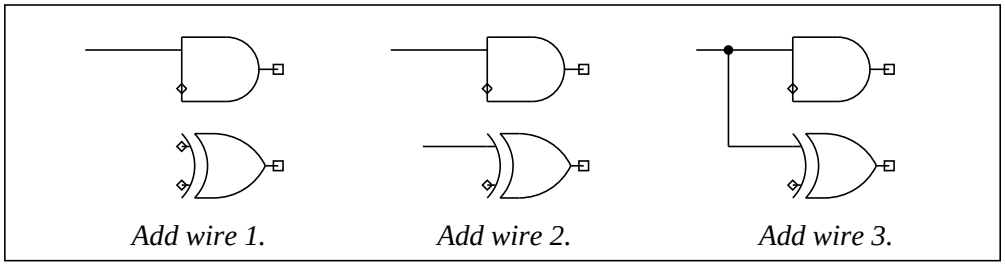
Now, use Copy and Paste to copy an `And` gate to your figure: (1) select the `And` gate in `gates.xoe` by clicking it with mouse button 1. (2) Press `ctl-C` to copy it to the clipboard. (3) Click in your blank figure so that it's selected for editing. (4) Press `ctl-V` to paste the `And` gate. XOE will attach the gate to the mouse. (5) Move the mouse where you want the gate and click mouse button 1 to place it there.

Repeat the process with the `Xor` gate:



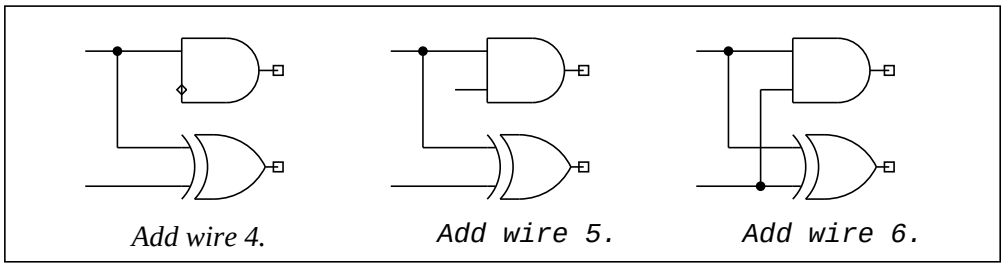
Note that the component pins are visible, which shows that they're unconnected. They will vanish when you connect wires to them, which is the next step. Note that some of the pins are off the grid.

Press **w** to enter Wire mode. Wire mode is like Line mode (§8.3.1), except that XOE analyzes connectivity as you add wires. To draw a wire, press mouse button 1 at the wire's starting point, drag the mouse to the wire's end point, and release mouse button 1. You should generally draw wires starting at component pins, since XOE will snap to them even if they're off-grid like some gate inputs. Let's draw some wires:

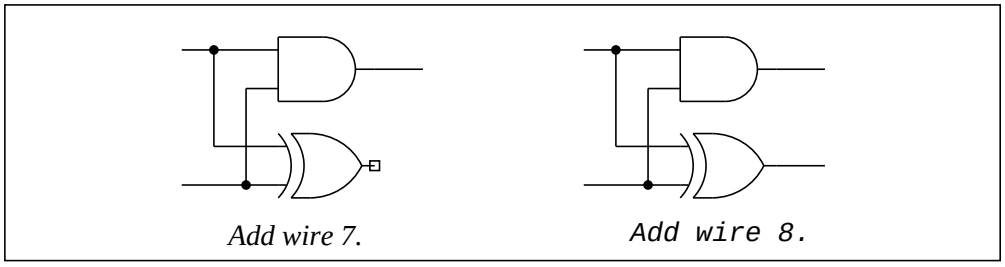


XOE draws the connection dot automatically when three or more wires meet at the same location.

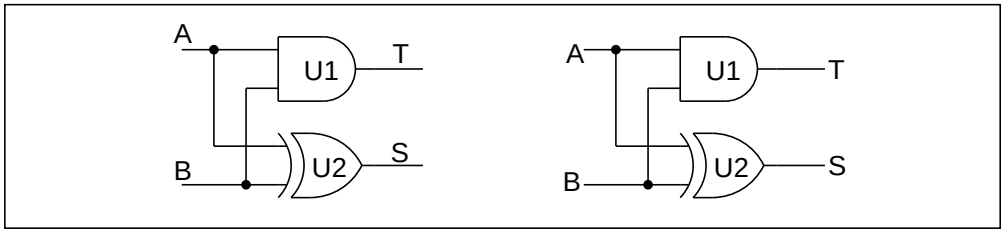
Now let's draw wires to the lower gate inputs:



If you make a mistake, use the Undo command `ctl-Z`. Finally, let's add output wires:



You now have a nice half adder. To add text to label the components and nets, press **t** to enter Text mode (§8.3.5), click where you want the text, and enter it at the blinking cursor:



Once you've entered the text XOs, you can move them around as desired.

8.6.2 Creating and Editing Submods

Unlike many schematic editors, XOE does not have a special editing mode for creating or editing component symbols. Instead, you draw the symbol shape using figure elements, add I/O ports, and use the *Group* command to collect the shapes into a submod. Here are the specific steps:

1. Draw the symbol using lines, rectangles, arcs, circles, and text.

2. Place I/O pins as instances of pin symbols `Ipin`, `Apin`, `Opin`, `IOpin`, etc. You will need to copy and paste them from the same or another figure. XXICC has a file `ports.xoe` with instances of each port and pin symbol.

3. Place pin names as `FIGtext` near existing pins. You can make a pin name *invisible* which makes XOE hide the text when it instantiates the submod, although it is visible as gray text when editing the symbol. To toggle `FIGtext` invisibility, select one or more `FIGtext` XOs and press **i** (*Invisibility* command).

   At some point you will be able to rotate pin names and other individual `FIGtext` *using the rotate key '@'. This has not been implemented yet.*

4. *Optional*: Create the submod's name as `FIGtext`, which may be invisible. A submod name is a *GalaxC identifier*, i.e., it starts with a letter or underscore '_' and contains letters, digits, and underscores. Submod names are case-sensitive. Some non-XXICC tools that use XOE schematics may not like an initial underscore, so we usually avoid it.

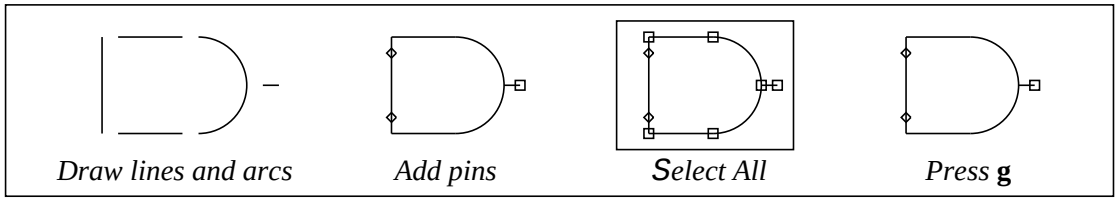
   If you want to create a new kind of submod pin, its name must end with "`pin`" and it must have a port direction, described in the next item.

5. By default, the submod's origin is the center of the bounding box that contains all symbol graphics, snapped to the current grid. However, you can specify a different submod origin -- e.g., the upper-left corner of the symbol or the center of the left edge -- by setting the origin lines (§8.4.4) to the desired submod origin.

   Use the *Direction* command **d** to specify a port direction when creating an I/O port or submod pin symbol. Each time you press **d** XOE toggles the port direction between *none*, *in*, and *out* (*I/O* to be added) and shows the port direction as an `Ipin`, `Opin`, or both at the intersection of the origin lines.

6. When the symbol is complete, select all of it (usually with a group selection box) and issue the *Group* command **g**. XOE collects the selection into a submod with origin specified by the origin lines if their intersection is in or near the bounding box of the selected objects. If the origin lines are not present, XOE uses the bounding box's center, snapped to the grid. If you specified a port direction using the Direction command, XOE makes the submod a port submodule. In this case, the origin lines must be present.

   XOE also creates an unrotated, unmirrored instance of the submod at the same location. The origin lines and invisible text vanish.



Some kinds of XOs cannot be grouped into submods. These include instances of anything other than submod pins (*this will be relaxed some day*), net XOs such as wires and net `FIGtext`, partially-selected lines, and text associated with a component.

To edit an existing submod, select any instance of it in a figure and issue the *Ungroup* command **G** (capital **g**). XOE replaces the instance with an unrotated, unmirrored copy of the contents of the submod, selects those XOs, and sets the origin lines to the submod's origin. All invisible text becomes visible.

Any `FIGtext` attached to the instance becomes unattached `COMPtext` and is shown using the grid color, normally blue. XOE will reättach `COMPtext` when you regroup the XOs, or you can move `COMPtext` elsewhere in the figure at which point it becomes normal `FIGtext`. You can also delete `COMPtext`. However, XOE does not attach `COMPtext` to an instance that is moved next to or on top of the `COMPtext` like normal `FIGtext` would be.

After ungrouping, you can modify the submod's figure elements, change its pins, and change its submod name if you like. When you are done editing the submod, select all of it and re-issue the Group command as described earlier. If the submod is named, XOE replaces an existing submod with the same name as described below. Otherwise XOE creates a new submod. When XOE creates an instance of the new submod, it reättaches `COMPtext` that is next to or on top of the new instance.

A XOE document has one set of submods which are shared by all figures in the document. A figure has only one version of a named submod. If you update a named submod with the Group command or paste an instance of a named submod, XOE replaces the existing submod with the new one and redraws instances of the submod. If the new submod's pins are placed differently from the old one, XOE reänalyzes connectivity and displays any updates.

Other figures in the same XOE document may have different submod versions, so they may still have instances of older versions. You can update them later by giving the *Update* command **u** in each figure. "**u**" updates all instances of named submods that have newer versions in the document, and reänalyzes connectivity. *At some point XOE may ask you if you want to update all instances throughout the document.*

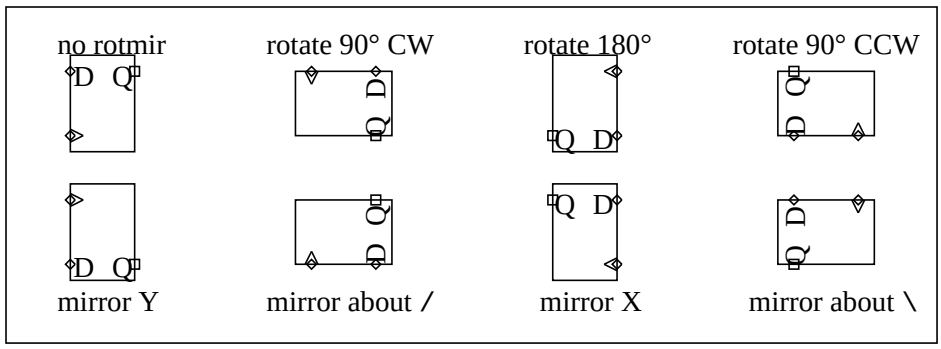
If an updated submod has all pins in the same locations, XOE can update other figures immediately since their connectivity is unchanged.

A figure may also have unnamed submods. If you ungroup an unnamed submod, modify it, and re-group it, XOE does not update any other instances of that submod.

8.6.3 Editing Components (Instances of Submods)

Once you have created submods, you can add more instances of them. If there is already an instance of the desired submod in the figure, you can select and copy it as described in §8.4.2. Otherwise, you'll need to use Copy and Paste to get it from another figure or document such as `gates.xoe` or `ports.xoe`.

You can *transform* an instance using the rotmir keys as described in §8.5. XOE rotates pin labels and other instance text with the instance and draws them either unrotated or rotated by 90°, with the rotation direction set by the user. Here are the eight possible rotmir transformations and how they display pin labels:



To attach `FIGtext` to a component, just edit the original text on or near the component or move existing text so that it overlaps or touches the component. Once `FIGtext` is attached, if you select a component by clicking on it XOE selects the attached text as well. You can deselect individual `FIGtexts` by clicking them while pressing `shift`. If `FIGtext` is a GalaxC identifier (§8.6.2), XOE considers it to be a component or port name when you compile the schematic for simulation.

XOE shows unconnected component pins as small diamonds for inputs (same as `Ipin`) and small squares for outputs and I/Os (same as `Opin` and `IOpin`). When you connect wires to them the connection spots vanish.

As mentioned in 8.6.2, when you paste one or more components into a figure XOE automatically updates submods with the same name to the new version just pasted. XOE updates all existing instances in the figure and reänalyzes connectivity. Paste then attaches the new XOs to the cursor and places them when you click mouse button 1. Pressing `ESC` cancels pasting, but the submod updates remain unless you undo them using `ctl-Z`.

XOE has special Paste behavior for replacing components with instances of a different submod or a different submod version. If you select one or more components and *only* select components, and the clipboard contains a single component, XOE assumes that you want Paste to change all the selected components into instances of the clipboard component's submod. In this case, XOE doesn't need to do the Move part of Paste: it just replaces the selected components with the new submod, with the submod origin moved to the instance origins. XOE does not change any attached `FIGtext`.

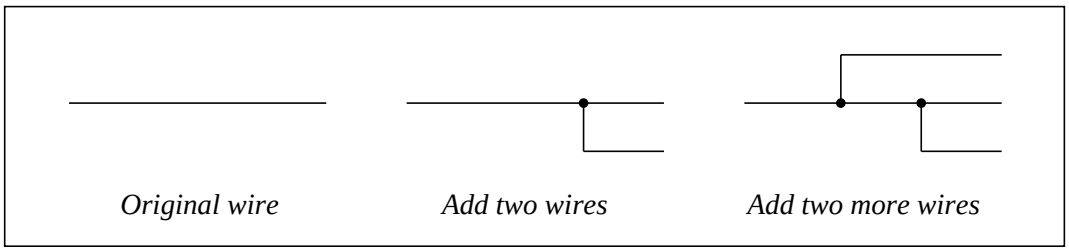
If the new submod has the same pins as the old ones, the connectivity doesn't change. Otherwise XOE reänalyzes connectivity.

8.6.4 Editing Nets

To create a net, press **w** to enter *Wire* mode. Wire mode is similar to Line mode, except that wires can connect to pins whereas lines are strictly visual. The mouse changes to a plus sign '+' to reflect that most wires are horizontal or vertical ("Manhattan") and rarely diagonal.

Once you have entered Wire mode, create wires just as you would lines: press mouse button 1 at the first vertex, drag to the next vertex, and release button 1. *At some point we will automatically create Manhattan wires unless you hold down* `shift` *at the first vertex.* Wire vertices are normally snapped to the grid, but if you begin a wire at a component pin or another wire, XOE will snap to that pin or wire instead of the grid. This allows you to connect wires to off-grid pins, giving you more flexibility as to where you place component pins. XOE also snaps to the origin lines (§8.4.4) and snaps the end point of a wire to an existing component pin or wire.

When you add a wire, XOE automatically updates connectivity. If the wire touches a component pin, the connection spot vanishes so that you can see whether XOE thinks there is a connection. When you connect a wire to the middle of an existing horizontal or vertical wire, or a point where two existing wires meet, XOE automatically creates a connection dot.



XOE continuously updates connectivity as you edit a schematic with wires. If you delete a wire, XOE automatically deletes connection dots if three wires no longer meet at that location. In some cases, you might find it easier to draw wires past where they should go and delete the excess later.

To attach `FIGtext` to a net, just edit the original text on or near a wire of the net or move the text so that it overlaps or touches a wire. Once `FIGtext` is attached, if you select a wire by clicking on it XOE selects the attached text as well. You can deselect individual `FIGtexts` by clicking them while pressing `shift`.

If `FIGtext` is a GalaxC identifier (§8.6.2), XOE considers it to be a net name when you compile the schematic for simulation. If `FIGtext` is '`0`' or '`1`', XOE recognizes it as a `Boolean` constant.

8.6.5 Uses of Schematics

Schematics are more than just pretty pictures with automatic connection dots. You can use them to provide connectivity information to other XXICC tools, e.g., simulation, netlist generation for FPGA (Field-Programmable Gate Array) design, and graphical representation of software behavior such as flowcharts and state diagrams. These capabilities will be available in future XXICC releases.

8.7. Cartoon Simulation

XOE has a wonderful educational and debugging tool for logic design called **Cartoon Simulation**, first developed for the Galaxy CAD System [JFB 92a]. With cartoon simulation, net values are displayed directly on schematics. You can change input values, and watch the effects propagate through the circuit as with a spreadsheet. You can quickly see if there are any obvious problems. When you detect bugs you can quickly isolate and fix them.

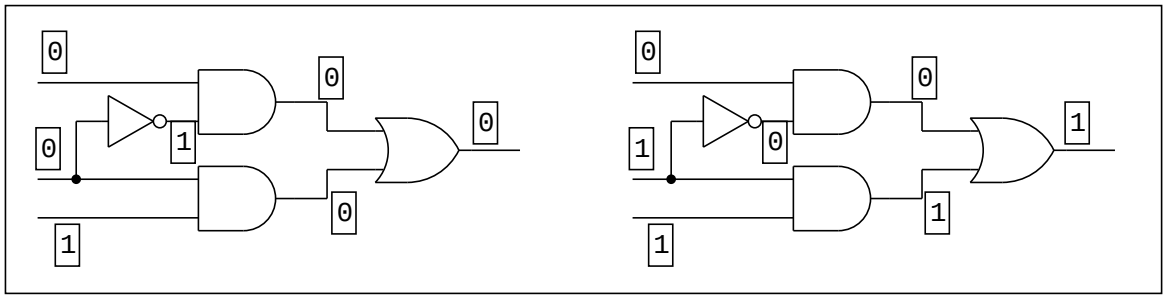
The current version of XOE figure editing includes simple logic simulation to demonstrate the concept. The current implementation is limited and experimental, with the following known limitations:

- Only combinational logic: there are no flip-flops and you cannot simulate logic that contains loops.
- You are limited to a few hundred nets.

To simulate a logic diagram, press **s** to enter Simulate mode. XOE compiles the logic diagram into executable PSI code using a conventional topologic sorting algorithm. If there are any loops or other errors, you will get an error message, but at the current time the offending components are not shown. If compilation is successful, the mouse changes to a large '`S`'.

Next, place **toons** next to the nets you want to monitor. A toon is a small box that displays a net's value. (The name "toon" is borrowed from *Who Framed [or Censored] Roger Rabbit?*) To create a toon, click mouse button 1 on or near a net. If the net can be observed by the simulator, XOE will show a small box with the current net value, initially '`0`' or '`?`' (unknown).

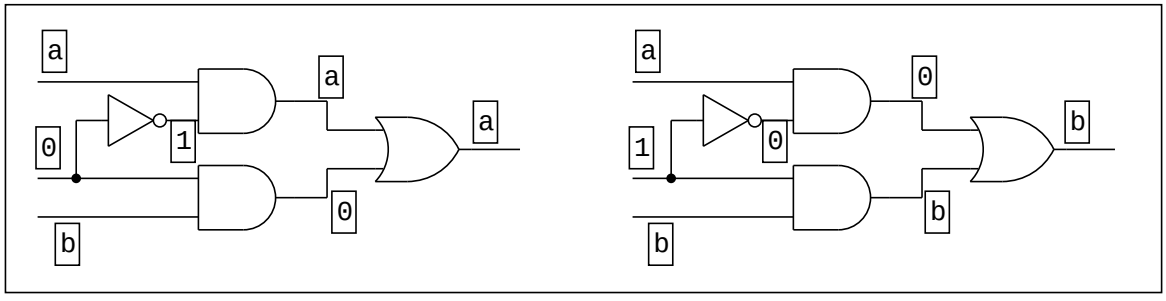
Once you have placed toons, you can change values by clicking input toons with mouse button 1. XOE usually toggles between '`0`' and '`1`', and toggles '`?`' to '`0`'. Here is a simulation of a 2-1 multiplexer:



XOE's simulator has an addition treat: **Simple Symbolic Simulation (S^3)**. This feature allows you to debug certain kinds of logic using Boolean algebra instead of just `0` and `1` binary values. The idea is to express a net's value as a single-character symbolic value '`a`'-'`z`' or its complement '`A`'-'`Z`', and whenever possible have the result of a logic operation be a single-character symbol or the constant '`0`' and '`1`'. If it's not possible to calculate a single-character value, the result is '`?`'. Here are the S^3 rules ('a' is any symbol, 'A' is its complement, and 'b' is a different symbol):

a & 0 = 0	a | 0 = a	!a = A
a & 1 = a	a | 1 = 1	!A = a
a & a = a	a | a = a	!? = ?
a & A = 0	a | A = 1	a & ? = ?
a & b = ?	a | b = ?	a | ? = ?

S^3 is particularly effective for multiplexers. For example, we can do a 100% test of the 2-1 multiplexer with just two tests, one with select = 0 and one with select = 1:



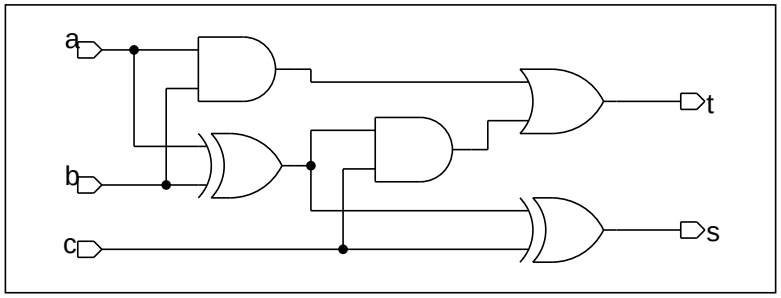
On the other hand, S^3 is not effective for arithmetic logic. Well, that's why tools boxes have more than one tool.

To use S^3, press **S** instead of **s** to enter Simulate mode. The initial values of S^3 nets is usually '`?`'. If you click on an input net with value '`?`', it changes to '`0`'. If a net is connected to an input port that has a single-character name such as '`a`', the net is initialized to that character.

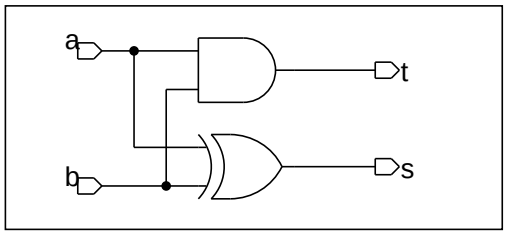
8.8. Hierarchical Schematics

Starting with 0.0h, you can simulate hierarchical schematics, with module ports and submodule pins used to connect between nested modules. Here is an example: a nine-input tally network which counts the number of inputs that are set to `1`. This example is copied from document `tally9.xoe`.

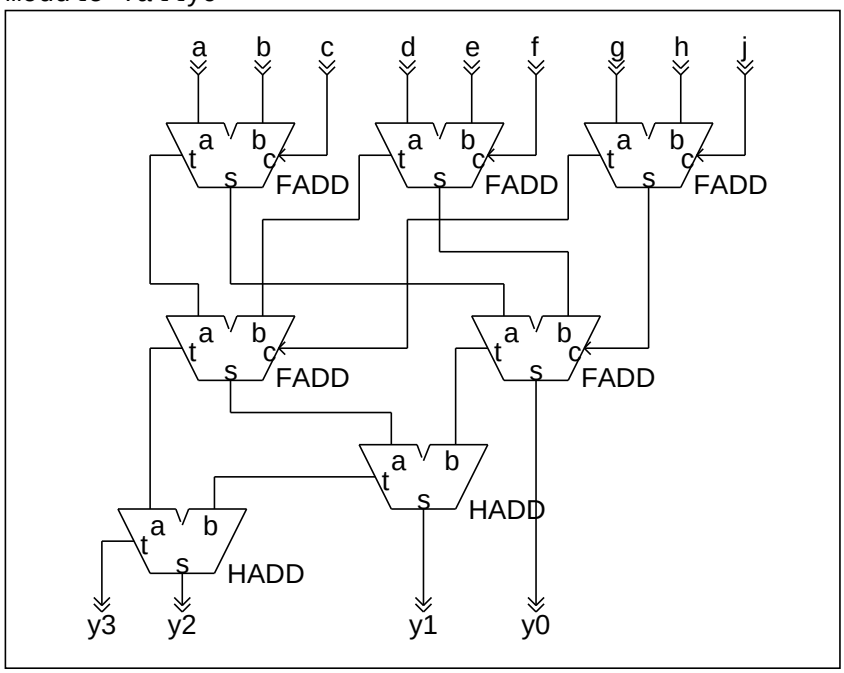
```
include "gchd.gi";

module FADD =
```




```
module HADD =
```




```
module Tally9 =
```



Let's take a look at each part of the example:

`include "gchd.gi";`

This includes *GalaxC for Hardware Design* (GCHD) constructs such as `module`, which are needed for hierarchical schematics. See Chapter 10 for details on GCHD.

`module FADD =` *figure*

This defines full adder module `FADD` to use the immediately-following `FIGURE` as its body. `FADD` adds inputs `a`, `b`, and `c` ("carry in") to produce a two-bit sum <t, s> with `t = a&b | (a ^ b)&c` = "carry out" and `s = a ^ b ^ c`.

`module HADD =` *figure*

This defines half adder module `HADD` with a `FIGURE` body. `HADD` adds inputs `a` and `b` (with no carry in) to produce <t, s> which only has values `00`, `01`, and `10`.

`module Tally9 =` *figure*

This defines module `Tally9` which adds inputs `a`-`j` to produce a 4-bit sum `y3`-`y0`. Tally9 uses `FADDs` to reduce the nine inputs weighted by 2^0 into four nets weighted by 2^2, 2^1, and 2^0. The four nets are then added by a ripple-carry adder implemented using `HADDs`. The figure shows all submod names and pin names. In practice, submod and pin names may be invisible to simplify the drawing. The case-sensitive submod pin names must match the module port names defined in the first two `module` definitions.

Since they only contain gates, you can simulate the `FADD` and `HADD` modules immediately using the Simulate commands **s** and **S**. On the other hand, since `Tally9` calls other modules you must first compile the entire `tally9.xoe` file using the GalaxC Compile command `F6` before simulating `Tally9`. (See [JFB 11a] for details on compiling and running GalaxC programs.) If a compilation error occurs, XOE selects the source code that generated the error and prints a message in the Output Window. *Currently, if there's an error in a* `FIGURE` *then XOE selects the entire* `FIGURE` *rather than individual figure elements.*

Once you have successfully compiled `tally9.xoe`, you can issue a Simulate command in any of the `FIGUREs`. XOE will treat that figure as the root module of a hierarchical design, and you can place toons to provide input stimulus and view output behavior. *At the present time you cannot monitor nested modules -- you can only view the root module.* Once a `.xoe` files has been compiled using `F6`, Simulate does not need to recompile each `FIGURE` -- it can reuse code that's already compiled.

Since `F6` compiles an entire `.xoe` file, this means that the entire file must contain valid GalaxC source code. Any non-program text must be contained in comments. On the other hand, a single-`FIGURE` schematic such as `FADD`, `HADD`, or the multiplexer in §8.7 can be included in any document, since in that case the Simulate command just compiles a single `FIGURE` at a time.

You can mix both schematics and GCHD textual modules in the same XOE file, and write a GalaxC program to generate root module inputs and print outputs using GCHD's "`digital hardware`" construct. See Chapter 10 for details.

In 0.0h, all figures must be in the root `.xoe` *file: you cannot write object code generated from figures into a* `.gi` *so it can be included in other files.*

8.9. Hints

- Figure editing is relatively new and not thoroughly tested, so expect a certain amount of instability. Save frequently, and make copies of saved files so that you can get back to a useful version if XOE writes bad data.

- When adding wires, connect to an off-grid component pin first so the wire snaps to the pin instead of the grid. You can also set the origin lines using one grid and then switch to a different grid. The origin lines will act as additional grid points.

- To create a new component instance, copy it from an existing figure or document, such as `gates.xoe` or `ports.xoe`.

- To move a component with wires attached, select it and the ends of the attached wires using a group selection box.

- To make a mirror-image copy of part of a schematic, select it using a group selection box, copy it by pressing mouse button 1 with `ctl` pressed, and while dragging it to the new position press the *mirror horizontally* key '`|`'. Alternatively, use the Copy and Paste commands and press '`|`' while placing the copy. This is a quick way to make symmetric drawings, such as parts of a differential circuit.

- To reälign objects to a different grid, first set the original grid and use the Copy command. The clipboard copy's origin is aligned to the original grid. Then switch to a different grid and use the Paste command. Paste will align the clipboard origin to the new grid. You can also make creative use of the origin lines.

- If you need to see which graphical objects are part of a component or net, select any object and press `F9`. XOE will select the entire component or net, including all `FIGtext`. Connection dots disappear when you select wires and reäppear when you unselect the wires.

- If several wires join at a connection dot, you may want to move that dot and have all the wires stretch together. To do this, first select the common vertex with a group selection box. The dot will vanish and you will see the overlapping move spots shown as a single move spot. Then press mouse button 1 on one of the wires *near but not right next to the move spot* and drag the wire to a new location. The other wires will stretch to retain the common vertex.

  If you press mouse button 1 *at the move spot*, XOE will instead select a single wire and let you stretch it by dragging the mouse. It will not stretch the other wires.

8.10. Issues

Revision 0.0h is the third released version of XOE figure editing, and there are a number of issues to be addressed in later versions. These issues include:

- The Draw menu is not implemented yet: use the single-key commands.

- You cannot set line color or style (dotted, dashed, etc.) at this time. All lines are solid black.

- The mouse only changes shape when editing a figure. Otherwise it looks like an arrow as with Select mode. This is confusing: it would be nice for the mouse to change to a vertical bar when editing text outside a figure and change to an arrow when selecting items in a dialog box.

- You cannot rotate individual text objects. Text that is part of a submod instance does rotate and mirror with the instance.
- You cannot scale individual component instances. You can zoom in or out, but all instances of a submod have the same size.
- Do not edit figures if zoomed out, i.e., reduced in size. XOE may not align coördinates to the grid properly and you'll have trouble making reliable connections.
- XOE does not prevent you from pasting document text into figures. This may confuse XOE.
- There may be bugs when editing nets, such as connection dots that aren't created and deleted properly, and component pins that are not drawn and erased properly. Please let us know if you can reproduce this sort of error.
- Specifying grids needs improvement. The current 10 spacing works well for most schematic editing, with 4 used for placing text. However, creating and editing symbols is tricky.
- Figure text character formatting has not been tested rigorously. There are known bugs, such as characters not being erased when deleted from the end of `FIGtext`. You can press `F5` (Refresh) to clean up the window.
- Invisible text may become visible if moved or otherwise manipulated. We need to fix this.
- XOE needs a way to specify `FIGtext` that doesn't bind to components and nets.
- If `FIGtext` binding changes, for example if previously-unbound `FIGtext` becomes bound to a component, the `FIGtext` is selected. We may fix this in a later version, or decide it's a nice "feature".
- Buses and control wires have not been implemented. All nets are drawn with the same line width and solid style.
- You cannot paste figure XOs as plain text, i.e., using `ctl-sh-V`.
- Toons (§8.7) do not move when zooming in and out: they are left at the same screen location, which is incorrect.
- There may be a problem with the Group and/or Ungroup commands which causes a segmentation violation at a future time. This is most likely a dangling pointer. Please let us know if you can reproduce this error.
- XOE has limited error checking when compiling for simulation and does not identify which figure elements caused the error.
- Simulation does not show the correct values of constant nets until after the first evaluation, so when you first create toons they may have the wrong values. Clicking on any toon should evaluate the figure and show the correct values.

Chapter 9
Implementing Schematics and Figures

This chapter describes how XXICC implements schematics and other figures internally. If you are only interested in editing them, see Chapter 8. Before reading this chapter, you should be familiar with XXICC Objects (XOs, Chapter 5) and figure editing (Chapter 8). This chapter uses schematic diagram terminology from §8.6 such as *submod*, *component*, *instance*, *port*, *pin*, and *net*. We also make a number of references to sections in Chapter 8, describing implementation details for figure commands.

XXICC figures consist of **figure elements**, which include basic graphical shapes such as lines, rectangles, circles, and arcs. XXICC represents figure elements as XOs, so they fit easily into XXICC documents. Multiple figure elements can be grouped into a **submodule**, which can then be instantiated one or more times in a figure. A figure can also include other XXICC objects such as dialogs, tables, and cells with text.

The main difference between figures and document text and graphics is that figure elements have explicit [*x*, *y*] coordinates while document text and graphics is formatted automatically.

9.1. Leaf XOs for Figures

This section describes how XXICC represents basic graphical shapes used in figures as leaf XOs. Leaf XOs are combined into various kinds of XO subtree: `FIGURE`, `SUBMOD`, `COMP`, and `NET`. Those will be described in the next section.

As in §5.1, we show how leaf XOs appear as `inline` calls in GalaxC program text, with operands following the XO opcode. Figure elements generally only make sense within a `FIGURE` or `SUBMOD` XOST, and are relative to the `FIGURE`'s upper-left corner or `SUBMOD`'s origin (usually its center).

The following XOs are used in both schematics and visual figures:

`MOVETO`(*x*, *y*)
: Pseudo-XO which sets the **current position** z_0 to [*x*, *y*], with coordinates relative to the `FIGURE`'s origin (upper left corner). `MOVETO` does not have a corresponding DXO.

`LINETO`(Δ*x*, Δ*y*)
: Line segment from current position z_0 to z_0 + [Δ*x*, Δ*y*], updating z_0 to the latter. Specifying line color and style TBD. Note that the figure element is `LINETO`: `LINE` is used for lines of *text*.

`RECT`(*width*, *height*)
: *Width* by *height* rectangle with upper-left corner at z_0. Do not update z_0. Specifying rectangle color and line style TBD.

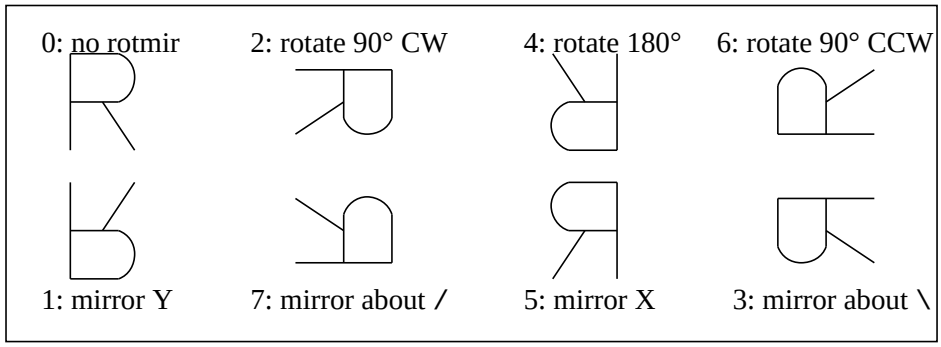
`CIRCLE`(*r*)
: Circle with center at z_0 and radius *r*. Do not update z_0. Specifying circle color and line style TBD.

`ARC`(*dir*, Δ*z*)
: Circular arc from z_0 to z_0 + Δ*z*, with initial direction *dir*. The direction is specified as a 15-bit signed BAM (binary angular measurement, see §3.1.2). The 15-bit BAM is sign-extended to 16 bits and fits in a PSI small integer. The arc may be clockwise or counter-clockwise depending on whether Δ*z* is

to the left or right of *dir*. Update z_0 to $z_0 + \Delta z$. Specifying arc color and line style TBD.

`INST(`*submod*, *rotmir*`)`

Instance of `SUBMOD` *submod* rotated and mirrored according to *rotmir* with `SUBMOD` origin moved to z_0. *Submod* is the index of the submodule, which is numbered from 1 in an XO list. The encoding of *rotmir* is shown in the following figure. Scaling is TBD.



rotmir is a 3-bit code consisting of a 2-bit *rot* code (BAM-2, §3.1.2) and a *mir* bit in the LSb. If *mir* is set (*rotmir* is odd), XOE first mirrors `INST` about the *x* axis. Then XOE rotates clockwise by *rot* x 90°.

An `INST` in a `FIGURE` is contained in a `COMP` subtree. At some point you'll be able to insert an `INST` into document text, where it formats like a word in a `LINE` subtree.

`FIGtext(`*width*, *options*`)`

All `TEXT` in a `FIGURE` must be contained in `CELLs` which specify justification and border. XOE automatically creates "`FIGtext`" `CELLs` for text as needed so that the fact that they're in `CELLs` is normally transparent to the user.

The following XOs are used in schematics for specifying connections:

`PIN` *net*

The first child of a `COMP` is always an `INST`, which is followed by zero or more `PINs`. They indicate which `NETs` connect to the component pins. `NETs` are numbered from 1 in a `FIGURE`: *net* is the index of the connected `NET` or 0 for an unconnected pin. `PIN` ignores z_0: XOE calculates `PIN` coordinates from the `INST`'s `SUBMOD`.

`DOT`

A `NET` may have connection dots where three or more wires converge at a single point. Each is represented by a `DOT` which is a filled circle with center at z_0 and pre-defined radius. Specifying dot color TBD.

9.2. XO Subtrees for Figures

This section describes the XO subtrees used by figures.

9.2.1 FIGURE(*width, height, options*)

A FIGURE is a line drawing made of LINETOs, RECTs, FIGtext, CIRCLEs, ARCs, etc. Unlike most XOSTs which automatically place their components using a formatting algorithm, FIGURE components have explicit coordinates set by MOVETO. FIGUREs are LFSTs, i.e., they cause paragraph breaks in a PARABOX.

9.2.2 SUBMOD(*name, options*)

A SUBMOD is a schematic symbol such as a logic gate, circuit element, or port. It may also just be a group of graphical elements in a visual figure. A SUBMOD contains leaf XOs for drawing the symbol, and may also contain instances of other SUBMODs such as ports used as submod pins. *Name* is the name of the submodule -- e.g., "Nand" -- and is a GalaxC identifier or the empty string.

Each INST XO is an instance of a SUBMOD, and provides a compact way to make an identical copy of the SUBMOD. You can later change a SUBMOD and all instances will be updated to the new version, unless you change the name. Each INST has a different location in the figure, set using MOVETO, and a *rotmir* transformation which combines rotation and mirroring.

A complete submodule begins with a SUBMOD instruction, followed by a sequence of XOs and terminated by ENDOBJ. The XOs in a SUBMOD are relative to the SUBMOD's **origin** [0, 0] which XOE uses to snap INSTs of the SUBMOD to a grid. The origin is also the center point for rotation and mirroring. A port SUBMOD such as Lport or Rport has a **port direction** coded in the LSbs of *options*. If the SUBMOD is not a port symbol its port direction is 0.

Options also has a SUBpin bit which means that this SUBMOD is a submod pin like Ipin and Opin. Currently only instances of submod pins can be included in a SUBMOD's symbol.

When XOE decodes the XO representation of a SUBMOD, it calculates the *bounding box* (BBox) of the SUBMOD and sets SUBMOD.*size* to the BBox dimensions. XOE sets SUBMOD.*z* to the upper left corner of the box, which has negative coordinates if the SUBMOD origin is inside the BBox.

When XOE formats an INST, z_0 corresponds to the SUBMOD origin. Any *rotmir* transformation is around z_0. XOE sets INST.*z* to the upper-left corner of the INST *after* transformation and INST.*size* to SUBMOD.*size*, also transformed. When placing an INST, XOE snaps the SUBMOD origin to the grid.

Zooming an INST is a little tricky because of rounding. INST.z is set according to the zoomed value of z_0 and it's tempting to zoom a SUBMOD by zooming each of its coordinates. However, if we add zoomed z_0 to a zoomed SUBMOD coordinate z_1, we could have inconsistent rounding because $round(z_0) + round(z_1)$ is not always equal to $round(z_0 + z_1)$. This causes misalignment between INST graphics and lines connecting to the INST. If zooming with anti-aliasing is performed by a graphics engine or otherwise built into G-SWIM, this is not a problem but it is with the current integer versions of G-SWIM for X11 and Win32.

XOE's current solution is to leave SUBMODs unzoomed and untransformed, and make a copy -- called an *expansion* -- of each zoomed or transformed instance of a SUBMOD. XOE then draws the expansions, so it only has to performs zooming and transformation when formatting. This fits the DXO philosophy of performing calculations once when formatting XOs instead of every time XOE draws them. SUBMOD

expansion is not needed if a INST isn't zoomed or transformed: in that case we just draw the SUBMOD with an [*x*, *y*] offset.

All SUBMODs used in anywhere in a document appear at the beginning of its XO list. All FIGUREs share the same set of SUBMODs. SUBMODs are followed by the root XOST. Each SUBMOD must be defined before it is used in another SUBMOD or within the root XO. In a schematic, the first SUBMODs are usually component ports (pins) such as Iport and Oport since they are normally used by all other schematic SUBMODs. Similarly, all SUBMODs appear at the beginning of a DXO list, followed by the root DXO.

When XOE makes a change to a SUBMOD *S* it replaces the entire SUBMOD instead of individual elements of *S*. If the INSTs inside the new *S* version only use SUBMODs before *S* then XOE can replace *S* in place. However, if *S* contains an INST of a SUBMOD after *S*, then XOE adds the new *S* version after existing SUBMODs. INSTs that use the old version will need to be updated at some point, after which XOE can remove the old version.

If a changed SUBMOD has pins at the same locations then connectivity is unchanged. However, if any pins are changed -- added, deleted, or moved -- XOE will need to reänalyze connectivity for all INSTs of the changed SUBMOD. A later version of XOE will automatically reänalyze the INSTs for a FIGURE, but for now XOE just adds a new version of the SUBMOD as in the last paragraph.

9.2.3 COMP

A COMP has an INST (usually) plus optional connectivity information and instance-specific text. The first child of a COMP is the INST (if present), which is followed by zero or more PINs. They indicate which NETs in a schematic connect to the component pins. COMP PINs are in the same order as SUBMOD pins. After the PINs a COMP may have FIGtext for an instance name and/or component properties. XOE usually selects and moves a COMP and its associated INST and FIGtext as a single unit.

A COMP may omit the INST and PINs and only have FIGtext. This is used for COMPtext that was attached to an COMP before it was ungrouped (§8.6.2).

A PORT is a COMP that is an instance of a port SUBMOD. XOE usually treats PORTs the same way as any other COMP. When XOE decodes a COMP it copies the SUBMOD.*options* port direction bits to COMP.*options* so XOE can easily see that a COMP with non-zero port direction bits is a PORT.

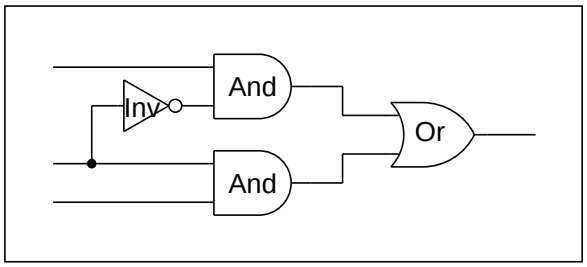
9.2.4 NET

A NET contains LINETOs and DOTs for wires that interconnect instance pins in a schematic. A NET may also have FIGtext such as a net name, bus numbering, and other properties. Each NET in a FIGURE has a net number, starting at 1.

A NET does not have any connection information other than its net number: actual connectivity is specified by COMPs and their PINs. When XOE deletes a NET it discards the contents of the NET but leaves the empty NET in the XO list. This avoids the need to update all PINs that reference NETs after the deleted one. When XOE needs a new net, it first sees if there's an empty one that can be reused. When XOE saves a document, it deletes empty NETs that aren't followed by non-empty NETs.

9.3. Sample Module XO List

Here is a small module containing a 2:1 multiplexer:



Here is the corresponding XO list, occasionally abridged. All `SUBMODs` must be defined at the beginning of the document. X coördinates increase to the right, Y coördinates increase downward.

```
// Here are the ports.  Neither Ipin nor Opin has any visible graphics.
SUBMOD("Ipin", 0x101); ... ENDOBJ;   // SUBMOD #1: Ipin with port direction 1
SUBMOD("Opin", 0x102); ... ENDOBJ;   // SUBMOD #2: Opin with port direction 2

// Here is the Inv submodule:
SUBMOD("Inv", 0);                    // SUBMOD #3: Inv
// Here is the graphics for an Inv gate.  MOVETO(x, y) is relative to SUBMOD center.
// LINETO(Δx, Δy) is relative to the previous MOVETO or LINETO.
MOVETO(-16, -16); LINETO(0, 32); LINETO(28, -16); LINETO(-28, -16);
MOVETO(16, 0); CIRCLE(4); MOVETO(-20, 0); LINETO(4, 0);
// Here are the SUBMOD pins, which are instances of Ipin and Opin:
COMP; MOVETO(-20, 0); INST(1, 0); ENDOBJ;    // Ipin.
COMP; MOVETO( 20, 0); INST(2, 0); ENDOBJ;    // Opin.
MOVETO(-16, -11); CELL(0, 1); TEXT("Inv"); ENDOBJ;  // FIGtext name.
ENDOBJ;

// The And and Or submodules are similar:
SUBMOD("And", 0); ... ENDOBJ;        // SUBMOD #4: And
SUBMOD("Or",  0); ... ENDOBJ;        // SUBMOD #5: Or

// Here is the FIGURE:
FIGURE(360, 160, 64);

// Here are the components: PIN(n) means component pin is connected to net n.
COMP; MOVETO(160,  50); INST(4, 0); PIN(2); PIN(7); PIN(3); ENDOBJ; // And
COMP; MOVETO( 90,  62); INST(3, 0); PIN(6); PIN(7); ENDOBJ;         // Inv
COMP; MOVETO(160, 110); INST(4, 0); PIN(6); PIN(1); PIN(4); ENDOBJ; // And
COMP; MOVETO(270,  80); INST(5, 0); PIN(3); PIN(4); PIN(5); ENDOBJ; // Or

// Here are the NETs, numbered from 1.
NET; MOVETO(130, 122); LINETO(-100, 0); ENDOBJ;             // NET #1
NET; MOVETO(130, 38); LINETO(-100, 0); ENDOBJ;              // NET #2
NET; MOVETO(190, 50); LINETO(20, 0);                        // NET #3
```

```
     LINETO(0, 18); LINETO(30, 0); ENDOBJ;
NET; MOVETO(190, 110); LINETO(20, 0);                          // NET #4
     LINETO(0, -18); LINETO(30, 0); ENDOBJ;
NET; MOVETO(300, 80); LINETO(30, 0); ENDOBJ;                   // NET #5
NET; MOVETO( 70, 62); LINETO(-16, 0); LINETO(0, 36); DOT;      // NET #6
     LINETO(-24,  0); MOVETO(54, 98); LINETO(76, 0); ENDOBJ;
NET; MOVETO(110, 62); LINETO(20, 0); ENDOBJ;                   // NET #7

ENDOBJ;
```

9.4. Creating and Editing SUBMODs

Section 8.6.2 describes how to create a SUBMOD. When you issue the Group command **g**, XOE collects the selection into a SUBMOD XOST with origin specified by the origin lines if their intersection is in or near the BBox of the selected objects. Otherwise XOE uses the BBox's snapped center. XOE subtracts the origin from all symbol graphics so that the SUBMOD origin becomes [0, 0]. If you specified a port direction using the Direction command **d** then XOE sets SUBMOD.*options* to that direction. This gives you a port SUBMOD with a pin at its origin. If a port SUBMOD's name ends with "pin", it is a submod pin and the SUBpin bit is set in SUBMOD.*options*.

Section 8.6.2 also describes how to edit an existing SUBMOD. When you select an untransformed instance of it in a FIGURE and issue the Ungroup command **G**, XOE replaces the INST with a (zoomed) copy of the contents of the SUBMOD, and sets the origin lines to the INST's origin. All invisible text becomes visible. Before it was ungrouped, the INST may have been in a COMP with attached FIGtext. We don't want that FIGtext to become part of every instance of the SUBMOD, so XOE keeps this COMPtext in a COMP with no INST. It will be reättached when you regroup the SUBMOD, or it can be moved or deleted like normal FIGtext.

When you are done editing the SUBMOD, select all of it and re-issue the Group command. XOE then replaces an existing SUBMOD with the same name (see §9.2.2), or creates a new one. When XOE creates a COMP for the new SUBMOD, it reättaches any nearby COMPtext.

9.5. FIGURE Copy and Paste

Copying and pasting figure XOs is described in Section 8.4.5. Here are some of the internal details.

When the Copy command copies all selected XOs to the clipboard, XOE also copies any SUBMODs used by selected INSTs or their SUBMODs. As with other XO lists, clipboard SUBMODs appear before other XOs. As with all XO lists, the clipboard XOs are unzoomed.

When XOE pastes an XO list from the clipboard, it must merge the clipboard SUBMODs into the current document. In many cases, some or all SUBMODs are already present. For example, it's rare to change standard I/O port symbols. XOE matches the clipboard SUBMODs to the current document by SUBMOD name. If the clipboard SUBMOD is an exact match, with graphics in exactly the same place, XOE simply uses the existing SUBMOD. Otherwise XOE tries to replaces the existing SUBMOD as described in §9.2.2. If that's not possible, XOE creates a new version of the SUBMOD and INSTs that use the old version will need to be updated.

9.6. DXO Fields for Figures and Schematics

Section 6.1 describes DXO fields used by all DXOs. Figures and schematics reuse some of those fields for figure editing and connectivity analysis.

`INST`.*expan*
: Points to an expanded copy of a zoomed or rotmir instance's `SUBMOD`. It reuses `INST`.*action*.

XOE defines `COMPstruct`, `NETstruct`, and `WIREstruct` to be subtypes of `DXOnode`. These subtypes are used in connectivity analysis (CXA), described in the next section. By making them subtypes instead of leaving them as DXOs, there's less chance of accidentally using the wrong type of DXO node. They're the same size as a DXO node, but use some DXO fields differently.

`COMP`.*pins*
: Points to a list of `PIN` DXOs for a `COMP`. They're currently the same list of `PINs` that follow a `COMP`'s `INST` DXO. `COMP`.*pins* reuses `COMP`.*v*.

`COMP`.*nextC*
: Points to the next `COMP` in a `FIGURE`, so that CXA doesn't have to search through a `FIGURE`'s DXO list to find the next `COMP`. `COMP`.*nextC* reuses `COMP`.*cd*.

`NET`.*wires*
: Points to a list of `WIRE` (`LINETO`) DXOs for a `NET`. They may be the `LINETOs` that are in a `NET` subtree, or may be scattered across several `NETs` as CXA finds new connections. `NET`.*wires* reuses `NET`.*v*.

`NET`.*nextN*
: Points to the next `NET` in a `FIGURE`, so that CXA doesn't have to search through a `FIGURE`'s DXO list to find the next `NET`, and skip empty `NETs`. `NET`.*nextN* reuses `NET`.*cd*.

`WIRE`.*group*
: Points to the next `WIRE` (`LINETO`) in a circular list of connected `WIREs`. They may be contained in a `NET` or scattered across several `NETs`. `WIRE`.*group* reuses `WIRE`.*v*.

`WIRE`.*nextW*
: Points to the next `WIRE` in a `NET`, so that CXA doesn't have to search through a `NET` to find the next `WIRE`, and skip hidden `WIREs`. `WIRE`.*nextW* reuses `WIRE`.*cd*.

`WIRE`.*conn*
: Bit vector that encodes the number of connections at each endpoint of a `WIRE`, and whether there are `DOTs`. `WIRE`.*conn* reuses `WIRE`.*Pstyle*.

9.7. Connectivity Analysis (CXA)

If a `FIGURE` has any `NETs`, XOE automatically analyzes connectivity after every graphical change that could affect connectivity. Changes include adding and deleting wires, adding and deleting components, moving and copying components, and updating `SUBMODs`.

Figure modifications that require CXA are Move operations or can be considered variations of Move. To

perform Move, first select individual DXOs using mouse clicks or multiple DXOs using using a group selection box. Then grab one of the selected DXOs with mouse button 1 and drag it to a new position. When you release button 1, XOE completes the move and reänalyzes connectivity.

To perform the Move, XOE copies the selected DXOs into a *mobile overlay* (MO) and hides the original DXOs in the *existing figure* (EF) so they aren't drawn while MO is present. During the drag, XOE adds the mouse's offset from its initial position when drawing MO so that MO DXOs follow the cursor. Each rubber shape has a fixed vertex that doesn't follow the cursor and a mobile vertex that does. When you release button 1, XOE adds the mouse offset to MO and merges MO into EF.

By hiding the original DXOs, they are still in EF so if you cancel the move XOE can immediately restore them. Also, in many cases XOE can modify the hidden DXOs instead of adding new ones. For example, if you move a component or wire, XOE can move and unhide the original DXO instead of adding a new one. In Move operations there is a one-to-one correspondence between hidden EF DXOs and MO DXOs which XOE uses to determine which EF DXO to update.

Other figure operations can be considered variations of Move. Copying the selected DXOs is basically the same as Move, except that XOE doesn't hide the original DXOs. Copying from the clipboard is very much like a normal Move: XOE copies the clipboard XO list into the `FIGURE`, creates DXOs from the copied XOs, and copies the new DXOs to MO, hiding the original DXOs.

Deleting hides EF DXOs like Move and then adds an empty MO. CXA deletes the hidden DXOs.

XOE adds new figure elements such as lines by creating a single rubber DXO in MO, with one vertex fixed and the other mobile. When you release button 1, the mobile vertex becomes fixed and XOE merges the single-DXO MO into EF, as with Copy. Modifying an existing figure element by dragging a vertex is the same, except that the original DXO is hidden as with Moves.

Updating components to a new `SUBMOD` version is basically the same as moving all instances of that `SUBMOD` by offset [0, 0]. Component `PINs` may be in different positions when reänalyzing connectivity.

In many cases, reänalyzing connectivity results in no changes. For example, just moving components around to improve æsthetics or to make room for new components does not change connectivity as long as all wires remain attached.

9.7.1 General CXA Rules and Terminology

1. CXA considers `COMPs` and `NETs` and ignores purely-visual graphics not in `COMPs` and `NETs`.

2. A `COMP` contains an `INST` of a `SUBMOD`, connection points represented as `PINs`, and `FIGtext` for component names and attributes. `COMPs` should not overlap or abut. *At some point XOE may automatically make sure they don't by refusing to complete operations that would cause them to.*

3. A `NET` contains wires in the form of `LINETO` instructions, `DOTs` to connect multiple wires, and `FIGtext` for net names and attributes. (`NET` `FIGtext` *has not been implemented yet.*) The **w** command creates data wires, **W** creates buses (thick wires), and `ctl-W` creates control wires (dotted). (**W** *and* `ctl-W` *aren't implemented yet*.) The **l** (lower-case **L**) command creates `LINETOs` outside `NETs` which do not have connectivity.

4. A *connection point* is a `COMP` pin or a `NET` wire endpoint. If two or more connection points overlap,

they are *connected* and the *degree* of the connection is the number of overlapping connection points. A connection of degree ≥ 3 or with two wires having different colors is shown with a `DOT` (standard-size filled circle).

5. Each `COMP` pin is either unconnected, touches one wire's endpoint, or touches a `DOT`.

6. Wires connect only at their endpoints. However, a new connection point may split a horizontal or vertical wire (*HV wire*) into two shorter wires joined by a `DOT`.

7. In a `NET`, each endpoint of each wire touches no wires, one other wire's endpoint, or a `DOT`.

9.7.2 DXO.state Bits

XOE uses DXO.*state* bits (§6.1) to manage EF and MO DXOs as they are being merged:

- Unchanged EF DXOs have DXO.*state* = 0.

- New DXOs are *selected* with DXO.*state* = SelectedObject. At the beginning of CXA, all MO DXOs are selected and all EF DXOs are unselected. CXA moves MO DXOs into EF but retains DXO.*state* so XOE can tell if a DXO is new. XOE will create new XOs for new DXOs.

- DXOs to be deleted are *hidden* with the HiddenObject bit set in DXO.*state*. In a Move, the original EF DXOs are hidden so that they aren't drawn as MO is dragged. CXA may *expose* -- i.e., un-hide -- EF DXOs if they can be reused. MO DXOs may also be hidden. For example, if MO has a rubber line which is unstretched to a single point during dragging XOE sets the HiddenObject bit so the rubber line is eventually deleted. XOE does not delete them immediately because Move needs a one-to-one correspondence between hidden EF DXOs and MO DXOs.

- If EF DXOs have been moved they are *marked* with the MarkedObject bit set in DXO.*state*. At the beginning of CXA, all DXOs are unmarked. During CXA, XOE replaces hidden EF DXOs with moved MO DXOs by hiding the MO DXO and exposing the EF DXO, marking the EF DXO so XOE can tell it has moved. XOE adds `MOVETO` XOs to move marked DXOs and other XOs for more complex modifications.

- If EF DXOs have been modified in other ways XOE sets the ShadowedObject bit in DXO.*state*. XOE uses this for `INSTs` with changed *rotmir* or *submod*, and for `FIGtext` with changed invisibility.

In addition to DXO.*state* bits, wires use `LINETO`.*a* bits to select individual vertices and for rubber lines. If `LINETO`.*a* = 0, the entire `LINETO` is selected or unselected, and has a fixed length. If `LINETO`.*a* = 1, vertex 1 is unselected and is fixed in MO, while vertex 2 is selected and mobile. If `LINETO`.*a* = 2, vertex 2 is unselected and is fixed in MO, while vertex 1 is selected and mobile. CXA uses `LINETO`.*a* = 3 for a formerly-rubber line that now has both vertices fixed.

9.7.3 Detailed CXA Algorithm

We now look at the detailed CXA algorithm. CXA uses a `FIGURE`'s DXO list as its data structure. DXOs already have coördinates calculated, so comparing coördinates is easy. XOE adds additional temporary pointers in `COMP`, `NET`, and `LINETO` (wire) DXOs for keeping track of connected neighbors (§9.6). When CXA is complete, XOE merges all DXO changes into the XO list.

Currently, CXA examines all COMPs and NETs in a FIGURE, which works fine for small figures and for verifying the basic algorithm. At some point CXA will use the bounding boxes of the COMPs, NETs, and MO to limit searches to just the COMPs and NETs that could be affected by the latest modification.

Here is the CXA algorithm:

1. Select all MO DXOs.

2. Extract net lists *EFnets* and *MOnets* from EF and MO for faster access. Each EF/MOnet has a wire list linking the exposed LINETOs within the net. The wire pointers are separate from the FIGURE's DXO list pointers.

3. Extract component lists *EFcomps* and *MOcomps*. Each EF/MOcomp has a pin list linking the PINs within the component. The pin pointer are separate from the FIGURE's DXO list pointers. EFcomps includes moved components: they were hidden during the move but now they're marked so XOE can tell they've moved. MOcomps only contains new components from a Copy or Paste.

4. Truncate, split, or hide wires in MOnets that overlap wires in EFnets. We will eventually merge MOnets into EFnets and we need to be sure each wire segment is represented exactly once. Two wires overlap if they're both horizontal or vertical, and have an overlapping segment. CXA truncates the MOnet wire if its first or second vertex is covered by an EFnet wire. CXA splits the MOnet wire if an EFnet wire is completely contained in the MOnet wire.

   If an MOnet wire is completely contained in an EFnet wire, or the MOnet wire's two vertices match an EFnet wire's two vertices in any order (in this case the wires can be diagonal), hide the MOnet wire and discard it later. We don't delete it at this time because Move requires a one-to-one correspondence between hidden EF DXOs (in the FIGURE DXO list) and MO DXOs.

5. Split HV wires in MOnets that touch connection points in EFnets and EFcomps: If a wire vertex in EFnets or a component pin in EFcomps is between the endpoints of an MOnet HV wire, split that HV wire at the EFnet vertex or EFcomp pin. We will add connection dots in a later step.

6. Split HV wires in EFnets that touch connection points in MOnets and MOcomps. This is the same as the last step except that it's MOnet wires and MOcomp pins that are splitting EFnet HV wires.

At this point, CXA does not know whether wires are connected to each other. EFnet wires were originally connected to other wires in their NETs, but if any NET wires have been deleted (and are now hidden) the remaining wires may now be isolated. *At some point we may save some time by recognizing that unmodified nets must have the same wire connectivity as before, but for now we just assume all wires may need to be reconnected.*

A group of connected wires use LINETO.*group* pointers to make a circularly-linked list. An isolated wire has LINETO.*group* pointing to itself. When we determine that two wires are connected, we merge their LINETO.*group* lists. During CXA a connected group may include wires from several nets in both EFnets and MOnets. CXA will eventually merge them into a single NET.

Each wire also has a LINETO.*conns* field which indicates whether its two vertices touch connection points, and if so, how many so CXA can create or delete DOTs. Initially all LINETO.*conns* are 0.

7. Find all connections between EFnet wires and other EFnet wires. Since we have already split all wires at interior connection points, we only need to check endpoints. For each pair of touching wires, increment the degree of the vertices that touch (one in each wire) and merge the `LINETO`.*group* lists.

8. Find all connections between MOnet wires and other MOnet wires. This is the same as the last step, except that CXA is looking at MOnets. Initially, each MOnet wire is in its own `NET`. This step groups MO wires that touch each other.

9. Find all connections between MOnet wires and EFnet wires. This groups new wires with existing wires that they touch. A new wire may join two `NETs`. The wires in a `NET` may have a loop, i.e., there may be more than one path between two connection points.

10. Expose hidden EF wires that correspond to MO wires, and hide (and eventually delete) the matched MO wires. This step relies on the one-to-one correspondence between hidden EF DXOs and MO DXOs. Unmatched hidden EF wires will be deleted later.

11. Find all connections between EFcomp and MOcomp component pins and wires in EFnets and MOnets. Increment degrees of the touching vertices. Mark EFcomps pins that have changed net number for a later step.

12. For each EFnet, merge all wires in its connected group into the EFnet, copying wires from other EFnets and MOnets, and hiding the originals. Merge colinear HV wires that share a connection point with degree = 2. Add `DOTs` at connection points with degree ≥ 3 and discard `DOTs` that no longer have enough connections. Add new `NETs` for unmerged EF and MO wire groups, reusing empty `NETs` whenever possible.

13. Reconnect all components to net lists and mark changed `PINs`.

14. Re-bind moved `FIGtext` to components. Do not re-bind if the `COMP`'s `INST` and `FIGtext` have moved by the same amount. *CXA does not currently handle* `FIGtext` *in* `NETs`.

15. Add any other MO graphics: we have already dealt with all `COMPs`, `NETs`, and `FIGtext`. XOE simply inserts the remaining MO at the end of the `FIGURE`'s DXO list.

At this point, the `FIGURE`'s DXO list has all the changed DXOs with DXO.*state* indicating whether DXOs are new (selected), moved or modified (marked and/or shadowed), or to be deleted (hidden). CXA now inserts the changes into the `FIGURE`'s XO by inserting new XOs, inserting `MOVETOs` to move existing XOs, and deleting existing XOs. These use `INSERT` and `DELETE` transactions -- all with the same transaction number -- so XOE can undo them later.

At some point, CXA will try to reduce the number of XOs needed to represent schematics. The most general form of a wire is `MOVETO`(*z1*) followed by `LINETO`(*z2*). The most general form of a `DOT` is `MOVETO`(*z3*) followed by `DOT`. However, we can save space and processing by chaining successive wires and `DOTs` together, e.g., `MOVETO`(*z1*), `LINETO`(*z2*), `LINETO`(*z3*), `DOT`, `LINETO`(*z4*), `LINETO`(*z5*). This optimization is not necessary, but may be useful when saving updated `NETs`. There is no canonical representation at this time.

Chapter 10
Using GalaxC for Hardware Design

Up to this point we have mostly seen GalaxC used as a programming language. In this chapter, we will see how GalaxC can also be used for hardware design. The ideas here are loosely based on the author's GHDL (Galaxy Hardware Description Language) described in *The Galaxy CAD System* [JFB 92a], which was a proof-of-concept hardware description language built on top of his Galaxy programming language [JFB 91]. The author's GHDL was never widely used or distributed, and the name "GHDL" has since been adopted for an open-source implementation of VHDL. To avoid trademark disputes, we will refer to GalaxC's hardware design technology as "GCHD" (GalaxC for Hardware Design). GCHD is *not* a separate language -- GCHD is a set of GalaxC extensions for hardware design. As we will see in this chapter, you can mix GalaxC software and hardware in the same program or document.

This chapter includes both introductory examples and GCHD implementation details for GCHD developers. At some point we will separate out and improve the tutorial material.

The current implementation of GCHD is at the "proof of concept" stage with many limitations: no clocked logic, no hierarchical topological sorting, `Boolean`-only values (no buses), and limited error checking. See §10.14 for more details.

To use GHDL, you must run XXICC with a target file of "`fig`", for example:

```
xxicc fig
```

If you run XXICC with the command "`xxicc xoe`" or just "`xxicc`", you will not be able to compile GCHD constructs. See *Installing and Running XXICC* for details.

10.1. Terminology and Example

GCHD uses the same hierarchical design terminology as schematics -- see Section 8.6 for definitions of **module**, **component**, **instance**, **submodule**, **port**, **pin**, and **net**. The term "module" can also refer to a separately-compiled GalaxC program unit [JFB 11a]. [*footnote:* When you combine hardware and software concepts into hardware/software co-design you can expect occasional terminology collisions.] In this chapter we always mean a *hardware* module unless we specify otherwise.

Here is a simple hierarchical design to show how GCHD uses these terms. This is the hierarchical schematic example from Section 8.8 rewritten as GCHD code.

```
// Full and half adders.
input {a, b, c}, output {t, s},
module {(t, s) = FADD(a, b, c)} =
{
    t = a & b | (a ^ b) & c;
    s = a ^ b ^ c;
},

module {(t, s) = HADD(a, b)} =
{
    t = a & b; s = a ^ b;
};
```

```
// 9-input tally module: count the number of inputs that are set.
input {a, b, c, d, e, f, g, h, j},        // 9 Boolean inputs.
output {y3, y2, y1, y0},                  // 4-bit total.
module {(y3, y2, y1, y0) = Tally9(a, b, c, d, e, f, g, h, j)} =
{
    // First level reduces 9 inputs to 6 using full adders.  s-u are weighted by 2, v-x by 1.
    net {s1, t1, u1, v0, w0, x0};
    (s1, v0) = FADD(a, b, c);
    (t1, w0) = FADD(d, e, f);
    (u1, x0) = FADD(g, h, j);
    // Second level reduces 6 nets to 4 using full adders.  p weighted by 4, q-r by 2.
    net {p2, q1, r1};
    (p2, q1) = FADD(s1, t1, u1); (r1, y0) = FADD(v0, w0, x0);
    // Half-adder chain to produce final result.
    net c2;
    (y3, y2) = HADD(p2, c2); (c2, y1) = HADD(q1, r1);
};
```

The `input` and `output` declarations define the I/O ports that may be used by the following module definitions. GCHD treats them as `arg` declarations. A `module` definition specifies the *pattern* to be used to instantiate the module, and a *body* that defines the contents of the module. The pattern does not need to include all the declared `args`. GCHD treats a `module` definition as a kind of `fn` definition.

The `FADD` and `HADD` bodies define their bodies using assignment statements and standard Boolean operators. In this case the operators are *primitive* or *intrinsic* operators at the bottom of the hierarchy.

The `Tally9` body has multiple instances of the `FADD` and `HADD` modules. Each instance is a copy of the module's pattern, with `Tally9` ports and nets replacing the `FADD` and `HADD` pin names. The `net` declarations define internal signals for connecting components. GCHD treats `net` declarations as local `vars`. All `nets` must be declared before they are used.

You can write the assignments in any order as long as nets are declared before used. Actual hardware automatically evaluates logic functions whenever inputs change, so the written sequence doesn't matter -- each element simply evalutes its function and propagates the results. It is implicitly parallel, like a human brain. In some cases hardware ends up doing redundant operations, which wastes power but does not produce the wrong result if properly designed.

Well-formed logic can be sorted so that each component only needs to be evaluated once for each set of changes to the inputs. This requires that combinational logic does not have any *circular dependencies*, also known as *loops*. Each loop must be broken by a register. GalaxC simulates well-formed logic by sorting components into functional dependency order so that all inputs of an operator are valid before evaluating that operator. We will consider this in detail later in the chapter, but first let's review the imperative programming model used by most programming languages for comparison.

10.2. The Imperative Programming Model

The standard programming model is **imperative** programming, where the programmer tells the computer what to do and what sequence to do it in, using either a high-level or low-level language. C is a typical imperative programming language where a program is made up of one or more functions, one of which is

`main`. Each function has a sequence of statements to be executed in the order specified, with various control statements to change the default order of execution explicitly. A primary advantage of an imperative language is that the same model is used by most computers, so there is an easy-to-follow mapping from high-level statements to machine or assembly language.

There have been many attempts over the history of computer hardware to design machines to use other programming models, such as data flow. However, none has had much success since simply telling a computer what to do next has huge performance advantages over having a computer spend time trying to figure out for itself what to do next.

Most GalaxC programs follow the imperative model. However, they can also follow other models as we will see in the next section.

10.3. The Functional Dependency Model

The imperative programming model is based on *sequential blocks* of the form {e_1; e_2; ... e_n} where e_1, e_2, ... e_n are expressions, usually assignment expressions, function calls, or control statements. The expressions are evaluated in consecutive order except when the sequence is explicitly changed by control statements. In this section we will borrow from the Occam language and write sequential blocks as `seq` {e_1; e_2; ... e_n} to make it clear that they're sequential.

In contrast, GCHD modules are based on *functional dependency blocks* of the form `fundep` {e_1; e_2; ... e_n} which evaluate the expressions in the order determined by their functional relationships. Usually this means that if expression *e* depends on the result of expression *f*, then *e* must be evaluated after *f*.

Here are some examples to clarify the differences. In a sequential block, each expression is completed before considering the next one. For example,

```
seq {x1 = x2 + x3; x2 = x1}
```

assigns a new value to `x1`, and then sets `x2` to a new value. The similar `fundep` block

```
fundep {x1 = x2 + x3; x2 = x1}
```

is not valid because there's a circular dependency: since `x1` depends on `x2` and `x2` depends on `x1`, there is no way to order the expressions so that each variable is assigned before it's used in other expressions.

The above `fundep` block is invalid because it uses *immediate assignment*, i.e., the target variables of the assignments are updated as soon as the expression on the right-hand side is evaluated. We can make the block valid by replacing the immediate assignment expressions by *simultaneous* (or *clocked*) *assignment* expressions like this:

```
fundep {x1:= x2 + x3; x2:= x1}
```

In this case the right-hand expressions are all evaluated and then assigned to the target variables simultaneously before reëvaluating right-hand expressions. (The assignment is only simultaneous in hardware. We simulate simultaneous assignment in software by updating all target variables before using any of their values.)

In hardware, an immediate assignment is usually just a connection between the output of an expression

(typically implemented as logic gates) and a named port or net. It may also assign a value to a *latch*. We will look at latches in §10.6. A simultaneous assignment implies the existence of an *edge-triggered register* which is updated by a clock edge. We will look at clocks later in the chapter.

Simultaneous assignments allow `fundep` blocks to express some kinds of data movement very cleanly. For example, you can exchange the values of two variables with just two assignments:

```
fundep {x1:= x2; x2:= x1}
```

To do this using a `seq` block, you need a temporary variable:

```
seq {t = x1; x1 = x2; x2 = t}
```

Another example is a 4-bit right shift register implemented as separate variables:

```
fundep {x1:= new_data; x2:= x1; x3:= x2; x4:= x3}
```

Each evaluation simultaneously shifts each `x` value to the higher-numbered `x`. You can implement this using a `seq` block, but you have to do it in the reverse order:

```
seq {x4 = x3; x3 = x2; x2 = x1; x1 = new_data}
```

If you do it in the same order as the `fundep` the value of `new_data` gets assigned to all four `x` variables.

In a `fundep`, an assignment of the form "`x = x` *op* `y`" must be a simultaneous assignment "`x:= x` *op* `y`", because in this case `x` depends on itself. Thus the C "*op*=" assignments are simultaneous assignments in `fundeps`, as are `x++` and `x--`. In a `seq` block, you only assign to one variable at a time so immediate and simultaneous assignments are equivalent.

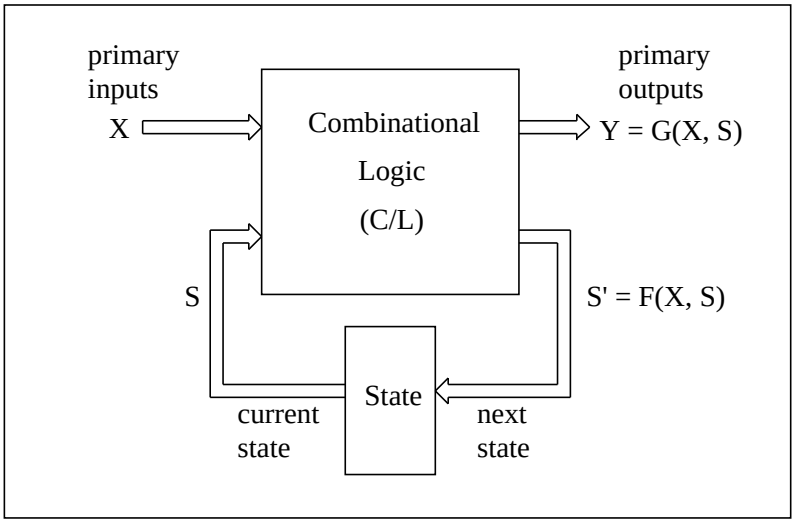
In a `fundep` each variable can only be assigned once, so this `fundep` block is incorrect:

```
fundep {x1 = x2 + x3; x1 = x4 + x5}
```

It would also be incorrect with simulataneous assignments. The use of variables in a `fundep` is similar to the Single Static Assignment (SSA) form in compiler optimization, which allows GalaxC to use that approach for both `seq` and `fundep` optimization.

10.4. Finite State Machines

GCHD semantics are based on Finite State Machine (FSM) theory and use the Mealy model. In the Mealy model, we separate hardware that performs a logical function (called **combinational logic** or **C/L**) from hardware that stores the current **state**.



X, Y, S, and S' are bit vectors with any number of bits, except that S and S' must have the same number of bits.

The combinational logic block computes two functions based on **primary inputs** X (inputs from outside the FSM) and **current state** S:

- G(X, S) calculates the **primary outputs** Y, which are the outputs of the FSM to the outside world.
- F(X, S) calculates the **next state** S', which will be written into the State at the next clock.

F and G are pure functions that always produce the same values for a given X and S. F and G must have no circular dependencies, i.e., they have no loops. C/L has no storage -- it only computes logic functions.

If X changes, the new value propagates through C/L to produce new values for Y = G(X, S) and S' = F(X, S).

"State" is the set of all storage elements in the FSM. These include multi-bit registers, single-bit flip-flops, and larger memories which may have multiple ports. State is updated by reset and clock signals:

- A *reset* signal initializes state to a particular value, e.g., all zeroes. The current state S is set to the initial state. This occurs immediately for an *asynchronous reset*. Synchronous resets are handled in C/L.
- A *clock* signal sets State to the value of S', the next state. If the storage elements are edge-triggered, all bits of State update simultaneously. The current state S become equal to the value of S' just before the clock edge. We will look at latches in §10.6.

If there are no active reset or clock signals, State retains its current value. State has no logic function -- it only has storage.

After a reset or clock occurs, the new value of S propagates through C/L to produce a new value for Y = G(X, S) and S' = F(X, S).

The above Mealy model is a clean way to define FSM behavior. Actual hardware may not look like the above diagram, but it's equivalent:

1. A primary output y_i may only depend on primary inputs. In this case, $y_i = g_i(X, S)$ is simply $y_i = g_i(X)$. If the hardware has no state then Y = G(X) and there is no F.

2. A primary output y_i may be connected directly to a state variable s_j. In this case, $y_i = g_i(X, S)$ is simply $y_i = s_j$. If all primary outputs are connected directly to state variables, then G is the identity function and Y = S.

3. A primary input x_i may be connected directly to a state variable s_j. In this case, $s_j' = f_j(X, S)$ is simply $s_j' = x_i$. If all primary inputs are connected directly to state variables, then F is the identity function and S' = X.

4. Some types of flip-flop have built-in combinational logic, for example TFFs and JKFFs. In the Mealy model, the combinational logic is moved to C/L so that the state is all DFFs. For example, a TFF is equivalent to a DFF plus an XOR gate connected to the DFF's current state. We can move that XOR to C/L.

5. A flip-flop may have clock enable input. We can replace this with an unconditional DFF plus a multiplexer connected to the DFF's current state, with the clock enable selecting the current state or the new state. We can then move the multiplexer to C/L. A register with clock enable is the same, except that we need a multi-bit multiplexer.

6. Different storage elements may use different clock signals or different phases. We can handle these by partitioning State into multiple blocks, each with a different clock.

7. A hierarchical design has many modules, each of which is an FSM. One way to handle this is to expand the hierarchy into primitive components, and then group all the functional elements into C/L and all the storage elements into State. In XXICC, we instead leave the hierarchy as is and avoid the expansion step which adds a lot of storage overhead for simulation and debugging. This creates the problem of *apparent loops*, which we handle using [JFB 92c].

Textbooks usually refer to logic that includes state as *sequential logic*. We're going to avoid this term since it conflicts with our software *sequential blocks* described in §10.3. We will instead call it **clocked logic**, since it always needs one or more clocks to update the state. (We're not going to consider asynchronous sequential logic for now.)

10.5. Writing Clocked Logic using GCHD

Now that we have seen the theoretical basis behind clocked logic, let's see how to express it in GCHD starting with a simple example, in this case a 4-bit Möbius counter:

```
clock clk,                          // Clock input.
input reset,                        // Asynchronous reset input.
outreg {s3, s2, s1, s0},            // Four registered output bits.
module {(s3, s2, s1, s0) = Moebius(clk, reset)} =
{
    if reset then s3 = s2 = s1 = s0 = FALSE else
    if clk rises then {s3:= s2; s2:= s1; s1:= s0; s0:= ~s3};
}
```

When `reset` occurs, all four bits are immediately set to 0. When `clk` rises, the four bits shift left simultaneously, with the LSb replaced with the complement of the MSb. The resulting sequence is 0000, 0001, 0011, 0111, 1111, 1110, 1100, 1000, and back to 0000.

The *if* statements indicate what assignments to perform for reset and clock inputs. Note that we use simultaneous assignments for `s3-s0` in the second *if* statement. This tells GCHD that they must have edge-triggered behavior.

The `reset` input has priority over `clk` and implies an *asynchronous reset*, i.e., the 4 bits are forced to 0 immediately no matter what the clock is doing. You can change this the reset to *synchronous* like this:

```
clock clk,                          // Clock input.
input reset,                        // Synchronous reset input.
outreg {s3, s2, s1, s0},            // Four registered output bits.
module {(s3, s2, s1, s0) = Moebius2(clk, reset)} =
{
    if clk rises then
    {
        if reset then s3:= s2:= s1:= s0:= FALSE
        else {s3:= s2; s2:= s1; s1:= s0; s0:= ~s3}
    };
}
```

In this case, `s3-s0` only update when `clk` rises and always use simultaneous assignment. The inner *if* statement implies a multiplexer for each `s` bit, with `reset` selecting either 0 or a different `s` bit, possibly inverted. If the target logic -- e.g., an FPGA -- has registers with synchronous reset then XXICC can use that feature instead of actual multiplexers.

In both examples the outer set of braces for each `module` body is optional.

Just for fun, let's convert `Moebius2` to use an `int` for `s` instead of individual bits:

```
clock clk,          // Clock input.
input reset,        // Synchronous reset input.
int outreg s,       // Four-bit output register.
module {s = Moebius3(clk, reset)} =
{
    if clk then s:= reset? 0: s << 1 & 0xE | ~s >> 3 & 1;
}
```

In this case we perform the shift using GalaxC left- and right-shift operators. We also use a conditional expresion instead of an *if* statement.

We can also put `s` inside the `module` and return its value using a `return` statement:

```
clock clk,          // Clock input.
input reset,        // Synchronous reset input.
module Moebius4(clk, reset) =
{
    int reg s;
```

```
        if clk then s:= reset? 0: s << 1 & 0xE | ~s >> 3 & 1;
        return s;
    }
```

This is functionally equivalent to `Moebius3` and which you want to use is a matter of personal taste.

As we saw in the above examples, *if* statements and conditional expressions are very important for expressing clocked logic. The semantics are quite simple:

1. Nested *if*s and conditional expressions behave as if the conditions are `AND`ed together.

2. Each simultaneous assignment needs a clock edge. The clock may be `AND`ed with Boolean values, but you cannot `AND` or `OR` two or more clocks or clock edges.

3. A module may have simultaneous assignments to the same register appearing at multiple places in the module. They must use the same clock edge, but it may `AND`ed with different Boolean conditions. GCHD creates one or more multiplexers controlled by the Boolean condition(s). If none of the `AND`ed conditions is true, the register retains its current state.

4. Immediate assignments can occur in *if* statements, and assignments to the same register may appear at multiple places in a module. They must have different Boolean conditions, which control implied multiplexers.

5. A register may have an immediate assignment for asynchronous reset and a simultaneous assignment for clock update. If it's a latch, it uses immediate assignment for clocked updates.

Synthesizing logic for *if* statements is very simple -- GCHD just creates multiplexers controlled by `AND`ing the nested Boolean conditions. This logic is then sent to gate-level optimization, perhaps in an FPGA vendor's tool suite.

10.6. Clocked Latches

The Mealy model in §10.4 assumes an edge-triggered state. Most clocked designs use edge-triggered logic, because it greatly simplifies behavior and timing, especially if you only have one clock phase. However, there are cases when *clocked transparent latches* (also known as *gated latches*) are very useful and can sometimes save a pipeline stage.

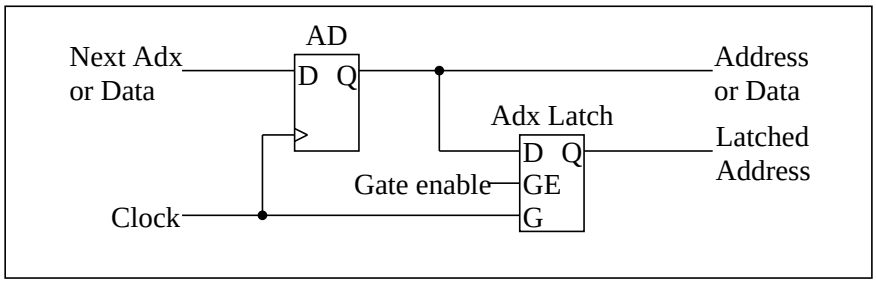
Before looking at two examples, we will define precisely what we mean by a clocked latch. First, a typical edge-triggered register samples its input at the rising edge of the clock and holds that value until the next rising edge. In contrast, a typical clocked latch samples its input when the clock rises and continues to sample the input as long as the clock remains high. During this interval the latch is *transparent* and propagates the input to the output with minimal delay. When the clock falls, the latch *holds* the value of the input and the output of the latch is stable until the next time clock rises.

A latch can sample the input on either clock level, just as edge-triggered registers may use either a rising or falling clock. The latch may also have a clock enable input so that the latch can ignore the clock and retain its current value over multiple cycles. The clock enable input must be stable before the clock rises (falls) and remain stable while the clock is high (low). This is a much longer interval than the clock enable input for an edge-triggered register -- that clock enable just needs to be stable for short set-up and hold intervals around the clock's rising (falling) edge.

FPGAs typically have storage elements that can be configured to be either an edge-triggered register or a latch. They also have configurable clock polarity and clock enable signals.

We will now look at two examples of situations where a latch can help with timing and/or saving pipeline stages. Both examples use a rising clock for edge-triggered registers, so the clock cycle is from a rising clock edge to the next rising edge.

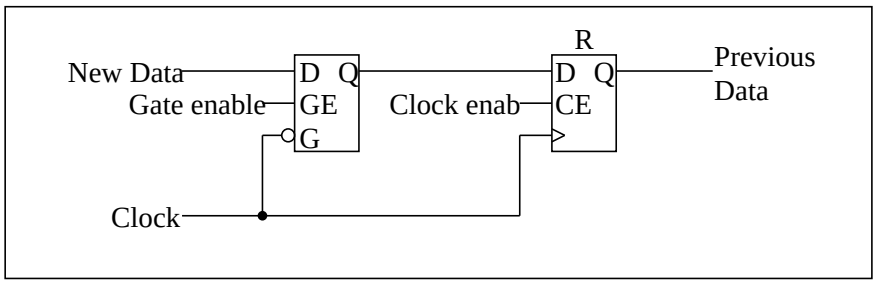
1. Let AD be an input register which updates at the beginning of each clock cycle. AD is either a read/write address or a data word to be written. By using a latch, we can save the address right after AD updates and decode the latched value in the same clock cycle. If we save the address in an edge-triggered register, we lose half a cycle (if the address register uses the falling clock) or a whole cycle (if the address register uses the rising clock). The address latch loads when the clock is high (the first half of the cycle) and propagates the address immediately so we can start using it near the beginning of the cycle. The latch holds the new value when the clock is low (the second half of the cycle).



The latch's clock is labeled *G* for "gate". The latch's clock enable signal *GE* needs to be designed carefully so that timing glitches don't cause errors. Using a flip-flop triggered on the falling clock edge works well in some cases.

2. Let R be an output register that may be loaded with a new value at the end of the clock cycle, but the signal that indicates whether or not it should be loaded is not available until near the end of the cycle. We need to generate the new value during the clock cycle, but we don't know if R is going to load it. If R doesn't, we need to save the new value somewhere. If we save the new value in an edge-triggered register, we either need to generate it one cycle earlier -- which may be impractical -- or register it half way through the cycle, which impacts timing.

   A latch provides a nice solution: we can latch the new value during the second half of the cycle when the clock is low. The latch is transparent during that interval, so it doesn't matter if the new data isn't ready until near the end of the cycle. The latch holds the value after clock rises. If R did not load the new value this cycle, the latch will hold it for the next cycle by inhibiting the latch's clock.

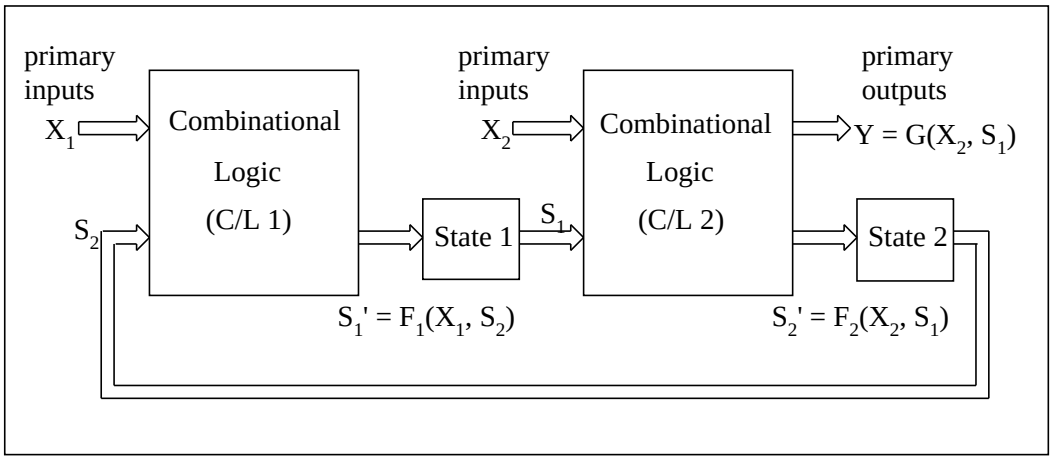


Again, we have to be careful with the latch's clock enable signal *GE*. In this case it must be stable

from the middle of the clock cycle until the next rising edge of the clock so it should be based on signals that are stable by the middle of the clock cycle.

Let's see how latches modify the Mealy model. Basically, when its clock is active, a latch is transparent and behaves like a buffer. So we will put latches in the combinational logic, even though they also have a storage function. The latches must not add any loops to the C/L, i.e., the input of a latch must not depend on the output of that latch. It can depend on other latches provided that the combination doesn't create a loop.

Some design methodologies -- notably IBM's Level-Sensitive Scan Design -- use latches for all storage and use multiple clock phases to ensure there are no C/L loops. Here is the Mealy machine representation for two clock phases:



Latches S_1 and S_2 are updated by non-overlapping phases of a single clock. During clock phase 1, S_1 loads a new value of F_1 which depends only on primary inputs X_1 and S_2 latches. Since S_1 is transparent, the new value of F_1 begins to propagate through C/L 2. However, it won't update S_2 so there are no loops. During clock phase 2, S_1 holds its current state and S_2 loads a new value of F_2 which depends only on primary inputs X_2 and S_1 latches. The new value of F_2 may propagate through C/L 1, but it won't update S_1 so again there are no loops.

To simplify the diagram, we have not shown primary outputs from C/L 1 which are certainly allowed. We've also shown separate primary inputs for C/L 1 and C/L 2. They could be the same set of primary inputs, provided that the timing is done correctly.

If you replace C/L 2 with wires that connect S_1 to S_2 -- which have the same width in this case -- the two latches with different clock phases become a *master-slave* register. This has exactly the same behavior as the simple Mealy machine from §10.4 -- indeed, edge-triggered registers are usually implemented as master-slave registers.

Using latches instead of registers can improve timing, so the multi-phase scheme is often used for high-performance logic. For example, you can often incorporate the latching function in the last layer of C/L so that it adds insignificant delay. In FPGAs there is usually no timing advantage to using latches instead of registers. FPGA registers are probably implemented as master-slave registers, and transparent latches probably as master-slave registers with the master forced to be transparent all the time.

If you have precise control over timing, it's possible to do the simple Mealy machine using only latches and

a single clock phase. This requires that the clock pulse be long enough to latch inputs reliably and at the same time short enough that the latching completes before the fastest signals propagate through C/L. This requires C/L to have a minimum delay, which is often harder to achieve than meeting a maximum delay requirement. The Cray-1 supercomputer used this approach and had various clever schemes to meet minimum delay.

As long as you meet the timing requirements, the latches with a single clock behave the same as if they were edge-triggered. This means we can use the simple Mealy model and assume all loops are broken by edge-triggered registers.

10.7. Simulating Clocked Logic

We now consider how to simulate clocked logic. We will be using FSM terminology and variable names from §10.4.

If a Mealy machine is implemented in hardware, C/L calculates new values whenever X or S changes. X changes whenever a primary input changes. Since there are no loops in C/L, the effects of the input change propagate through C/L and then it becomes **quiescent**. We handle asynchronous reset as part of C/L.

S changes whenever an asynchronous reset or clock signal updates State to a new value. For the Mealy model to work, all bits of State must update simulateously or nearly so, and State's set-up and hold requirements must be met so the update is complete before the new S propagates through C/L and changes S'.

To simulate a Mealy machine in software, we'd like to capture its functionality accurately without wasting CPU effort on details we don't need. One way to simulate logic is **event-driven** simulation, where each change of a net causes all components driven by that net to recalculate their logic functions and schedule changes to their output nets. Event-driven simulation can simulate any kind of logic, including C/L with loops, and provides an approximate simulation of the logic's timing if performed at the gate level. However, managing the event queue has a lot of overhead and there are better ways to simulate if you just care about function.

GCHD uses **compiled-code** simulation where the Mealy machine is compiled into code -- e.g., PSI code -- that can be executed on a general-purpose sequential computer. While we could execute this code whenever each input of X changes, that is wasteful and generally worse than event-driven simulation. Instead, we collect multiple input changes and process them all at once. Since we usually evaluate the entire C/L block once for each simulated clock cycle, this form of simulation is also called **cycle simulation**.

Basically, GCHD considers C/L to be a `fundep` block where each assignment expression calculates a logic function and assigns it to a net, either a primary output y_i, a next state s_j', or a net inside C/L. GCHD sorts the assignments into fundep order, which is always possible since C/L has no loops.

For a single module with only C/L, simulating a module is very simple: you set primary inputs to their new values and evaluate all the assignments in fundep order. This is a *fundep evaluation* or *feval.*

Simulating clocked logic is more interesting. The simplest case is a single phase edge-triggered clock with new values of X occuring near the beginning of the cycle:

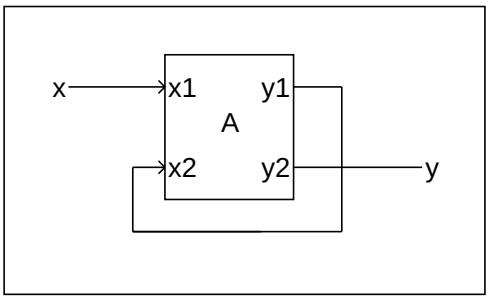
1. Copy S' to S to update State. This simulates the rising (falling) edge of the clock. If a storage element s_j has conditional assignment, only update s_j if its clock enable is active.

2. Propagate the new S and new X through C/L by evaluating C/L assignments in fundep order. This produces new values for S' = F(X, S) and Y = G(X, S). F(X, S) includes new clock enables for conditional assignments.

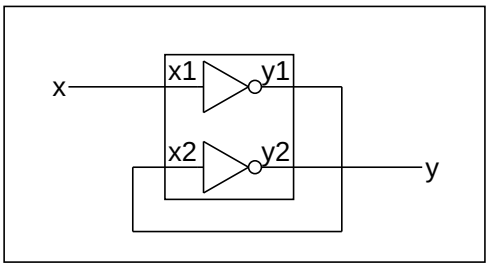
This is a *clocked evaluation* or *keval*. If there are multiple clocks or phases, a cycle requires multiple kevals.

If we change X after a keval, we need to reëvaluate C/L to produce new values of S' and Y. However, we only need to evaluate C/L that depends on X -- C/L that only depends on S can retain the results of its previous evaluation. This is a *partial eval* or *peval*.

Hierarchical designs may require additional pevals to handle *apparent loops*. To see this problem, consider this simple hierarchical design, where submod A contains combinational logic:

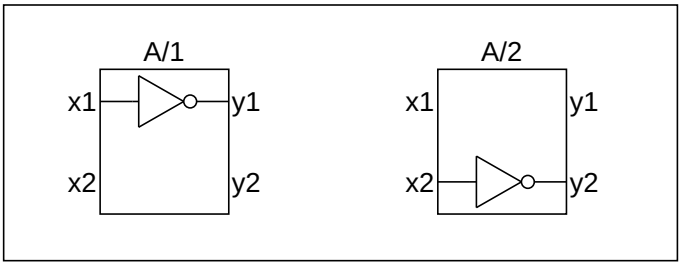


It appears that we cannot sort this design into fundep order, because we cannot evaluate A until both x1 and x2 are valid, and y1 isn't valid until we evaluate A. However, if we expand submod A we might get this:



In this case, we can sort the expanded components into fundep order.

While you can solve the problem of apparent loops by expanding the hierarchy, it would be nice to avoid doing so and [JFB 92c] describes in detail a method for dealing with apparent loops while keeping the hierarchy intact. This *Hierarchical Topological Sorting* algorithm splits blocks like A into several *partitions* each of which contains the subset of the module needed to calculate a subset of the outputs. In this simple example, there are two partitions:



Partition A/1 calculates y1 given x1. Partition A/2 calculates y2 given x2. We can now calculate the fundep order of the original block as A/1 followed by A/2.

Each partition has a corresponding peval. In the general case, a module could have so many partitions that you'd be better off expanding the hierarchy. In practice, modules only need a few partitions and thus only a few pevals. The first peval evaluates part of the module and updates the values of some module outputs and some internal nets. The next peval evaluates other parts of the module, using new module inputs and internal net values set by the previous peval(s). Eventually all nets including primary outputs have new values.

It is possible to avoid partitions and simply reëvaluate module A twice to calculate y2. The first evaluation would produce a valid y1 and the second would produce valid y1 and y2. That is correct, but for a large module it would be wasteful. It's also nice for each assignment to be evaluated only once per clock cycle because then if evaluation has any side-effects -- for example a debugging `printf` statement -- the side-effect only occurs once. Single evaluation is not always possible.

Clocked modules may require *partial clock evaluations* (*pkevals*) that update State but only propagate the new S and module inputs through part of the C/L.

When GCHD compiles a hierarchy of modules for simulation it automatically generates an *evaluation function* (evalfun) for each required peval and pkeval. Evalfuns update a common data structure called *persistent storage* which contains the values of registers, latches, and named nets. We will examine persistent storage in §10.8.

10.7.1 Simulating Latches

We simulate latches as part of C/L evaluation instead of as part of state update, and handle latch clocks differently. An edge-triggered register's clock is a special signal that bypasses C/L and only affects State update. A latch's clock is part of C/L and when it's active [*footnote*: i.e., high (low) for a latch with active-high (active-low) clock input.] during C/L evaluation the latch transparently copies its data input to its output net: the latch acts like a buffer. (If the latch has conditional assignment it only updates its output if its clock enable is active.)

When the clock is inactive, the latch is not simulated at all: we simply retain the value of the output net. This means that unless there is logic that depends on the inactive clock level, there's no reason to simulate C/L when the clock is inactive. Examples of logic that depend on the inactive clock level include latches with active-low clock and expressions that treat the clock as an ordinary signal instead of part of an *if* condition.

When GCHD compiles a module it determines the sensitivity of each storage element to clock edges and levels, and creates additional pkevals if needed. We then combine a rising clock plus its high level into a single pkeval that first updates positive edge-triggered registers and then evaluates C/L, including any active-high latches. Similarly, we combine a falling clock plus its low level into a single pkeval that first

updates negative edge-triggered registers and then evaluates C/L, including any active-low latches. If a design only uses a few latches to improve timing as in §10.6 then additional pkevals will probably be small.

10.7.2 Simulating Reset Logic

Reset logic is a bit more fun since a register may have both an edge-triggered clock and asynchronous reset. First, synchronous reset is combined into C/L and simply changes the data clocked into a register, so we don't need to do anything special. Asynchronous reset is also merged into C/L, but it treats the register as a latch if the reset is active and forces an initial value onto the register's output net. The reset also inhibits clocking of the register by forcing the register's clock enable low. Note that the generated code writes a new value into the register's output net at two places: a reset value as part of C/L evaluation and S' bits as part of State update.

Handling a latch with asynchronous reset is simpler: in this case the code to write a new value to the latch's output net is always part of C/L evaluation.

10.8. Persistent Storage

The imperative model permits a simple memory allocation model, used by both C and GalaxC. Variables are either global or local. Global variables are statically allocated. In C, a variable may be declared `static` which makes it private to the source file or function in which it is declared, but at run time it behaves like a global variable. Global variables are allocated when a program is executed and deallocated when execution ends.

Local variables are declared within functions. Local variables are automatically allocated on the stack when control enters the block containing the local variables and they are automatically deallocated when control leaves the block, i.e., their values are lost. A (mutually) recursive function may have multiple instances of local variables in existence at the same time.

In addition, an imperative program may allocate blocks of dynamic memory from the *heap*. Heap memory is accessed through pointers, which can be in local variables, global variables, or dynamic memory. The user must deallocate heap memory blocks explicitly unless the system has garbage collection. For best results you should free blocks opposite to the order in which you allocated them, i.e., the last allocated block should be the first deallocated. Otherwise you may get *memory fragmentation* which can lead to a program running out of useable memory because it cannot allocate a large enough block.

Evaluating GCHD modules requires a different approach to storage. Performing a module's peval or pkeval is basically the same as calling a function, except that since the local variables of a module are nets -- some of which store state -- their values must persist from one peval to the next. This means we can't deallocate the local variables when we're done with the peval: they must remain as long as we're still simulating the design that contains the module. [*footnote*: Actually, we don't need to retain all net values, as some are easily regenerated from nets that represent State. An optimizer can decide which nets to retain and which to regenerate as needed.]

The net values of a module are stored in a **persistent storage** (PS) `struct` which is hidden from the user. The module accesses PS using an **instance base** (IB) pointer, which points to the first net in the module's PS. IB is usually hidden from the user and may be implemented as a fixed register like the stack pointer or frame pointer. The inputs of a module are handled the same way as nets, with values stored in PS. The outputs of a module use a pass-by-reference scheme where the address of the net connected to the output is stored in PS. The inputs and outputs of a module have negative IB offsets.

If a module instantiates child modules, the parent module allocates PS for all its children and appends each child PS to its own PS. When the parent module calls a child module, it temporarily sets IB to the address of the child's first net when it evaluates the child.

The root module of the hierarchy allocates a single PS block for the whole design, usually on the heap. The size of this block is fixed for a given design because nesting of child modules in parent modules is static for hardware designs. You cannot have dynamic recursion as with software designs.

A module may call imperative functions, for example to simulate a large built-in component. These functions may contain dynamic recursion and automatically allocate and deallocate local variables, since they complete execution before returning control to the module that called them. Imperative functions use the normal program execution stack.

10.8.1 Passing Arguments to Components

As mentioned earlier, calling an evalfun to perform a peval or pkeval is basically the same as calling an imperative function. Like an imperative function, an evalfun has arguments for passing inputs from the parent module and for passing outputs back. Imperative functions normally use pass-by-value for input arguments and pass-by-reference for outputs. (C and GalaxC implement pass-by-reference by passing a pointer by value.) Evalfuns use the same scheme, except that input values and output pointers are copied to PS instead of the stack.

Here is how a module *M* evalutes a component *C*:

1. Evaluate each component argument. If it's an input argument, store its value in *C*'s PS. It it's an output argument, store its address in *C*'s PS.

2. Set IB to the first net in *C*. *C*'s input values and output addresses will be at negative offsets from IB. *C*'s nets and subcomponents will be at positive or zero offsets from IB.

3. Execute *C*'s evalfun as a normal GalaxC function.

4. Restore IB to point to *M*'s first net.

Modules that return a value leave it on the stack. *M*'s evalfun will copy it to a module net or to another component's input. *Details TBD.*

Imperative functions need to push the addresses of reference arguments on the stack, and discard them when the called function returns. This is because the stack is reused continually, and we don't know how many levels (mutually) recursive function may call. With hardware, the module hierarchy is fixed. This means that we can calculate the addresses of all output nets at initialization time and store them in PS once rather than storing them each time we call an evalfun. In essence, the reference pointers in PS is the static equivalent of a run-time stack. This saves a lot of unnecessary stack manipulation.

10.9. Mapping GCHD Constructs to GalaxC

GCHD is a set of language extensions for GalaxC. This section describes how GCHD constructs are defined as GalaxC extensions.

`[τ] module` *pattern* `=` *body*

A module definition is very much like a GalaxC `fn` definition. GalaxC compiles *body*, checking

GalaxC syntax and normal semantics, and produces a PSI function. Then GalaxC calls function "`module post processing`" which reänalyzes the PSI code as a GCHD module and sorts operations into fundep order. This converts the PSI function into an internal data structure that can be optimized and synthesized, and can be easily converted into evalfuns.

As with GalaxC `fns`, you can declare a return type τ for the module. Usually τ is missing, in which case GalaxC determines the type of the module from *body*. If *body* is a single expression, the module type is the type of that expression. If *body* has `return` expressions, GalaxC uses the common type of those expressions. Otherwise the module has type `void`. If a module is non-`void` it can be used in other expressions. *Currently all modules must have type* `void`.

At the present time, GalaxC does not allow recursive modules. We intend to relax this some day when we allow parameterized modules.

GalaxC defines the `module` construct as a special function which calls the same `DefineFnMac` function as `fn` and `def` definitions:

```
type arg T, spclarg {pattern, body},
void spcl {T module pattern = body} =
    DefineFnMac(MOD_KIND, T, pattern, body),
def {module pattern = body} = (unknown module pattern = body);
```

"`module`" is not a reserved word, so *pattern* syntax must be *prefix* or in braces.

While *pattern* can be any legal GalaxC expression, if you want to combine GCHD with `FIGUREs` then *pattern* must be one of several *standard forms* with a GalaxC identifier which acts as a unique module name. Here are the forms:

1. *`name`*`(`*`args`*`)`
2. `(`*`args`*`) = `*`name`*`(`*`args`*`)`
3. *`arg`* `= `*`name`*`(`*`args`*`)`
4. *`name`*

§10.1 uses form (2) for the `HADD`, `FADD`, and `Tally9` modules. By convention, outputs are on the left side of '`=`', but this is not required. "`args`" is usually a list of GCHD `inputs` and/or `outputs` separated by commas, but that's not required: you can have any legal GalaxC expression inside the parentheses as long as each `input` or `output` is listed at most once.

As shown in §8.8, *body* may also be a `FIGURE`. In this case GalaxC calls function "`compile FIGURE`" to compile the contents of the figure into PSI code. If *pattern* includes `inputs` and/or `outputs`, they define the ports of the module and any port symbols in the `FIGURE` must agree with them. If *pattern* is simply a module *`name`*, then the port symbols in the `FIGURE` define the ports of the module. If you want to include a `FIGURE` module in GCHD text, then *pattern* must include `inputs` and/or `outputs`. (*We plan to add an alternative way for GCHD to call submodules which will list pins connections by pin name. This will allow GCHD text to call a* `FIGURE` *module with* pattern = *`name`*.)

At some point we will allow body *to contain both GCHD text and FIGUREs. For now,* body *must be all GCHD text or a single* `FIGURE`.

- [τ] `input` *args*
  This defines one or more input arguments for a `module`. It's like an `arg` declaration, but the arguments are passed (by value) in persistent storage instead of on the stack. If τ is missing the arguments are `Boolean`.

  GalaxC defines the `input` constructs as:

```
type arg T, unknown spclarg name,
def {T input name} = in_attr T arg name,
def {input name} = in_attr Boolean arg name
```

  "`input`" is not a reserved word, so `name` must be *prefix* or in braces. "`in_attr`" is a storage attribute that modifies `T` and tells GalaxC to put the argument(s) into persistent storage and to treat them as module inputs.

  At some point add `inreg` *constructs for registered inputs. They may have* `init` *expressions for initialization.*

[τ] `output` *args*
  This defines one or more output arguments for a `module`. It's like an `arg` declaration, but the arguments are passed (by reference) in persistent storage instead of on the stack. If τ is missing the arguments are `Boolean`.

  GalaxC defines the `output` constructs as:

```
type arg T, unknown spclarg name,
def {T output name} = out_attr (&T) arg name,
def {output name} = out_attr (&Boolean) arg name
```

  "`output`" is not a reserved word, so `name` must be *prefix* or in braces. "`out_attr`" is a storage attribute that modifies `T` and tells GalaxC to put the argument(s) into persistent storage and to treat them as module outputs.

  At some point add `outreg` *constructs for registered outputs. They may have* `init` *expressions for initialization.*

At some point we need to add bidirectional `inout` *and* `ioreg` *constructs. Details TBD.*

[τ] `net` *nets*
[τ] `net` *nets* `=` *value*
  These define one or more nets of type τ. They are like local `var` declarations, but the nets are in persistent storage instead of on the stack. If τ is missing the nets are `Boolean`.

  If present, *value* is the continuously assigned value of the net(s) and can be any expression that uses ports and nets already declared. It's not just the initial value as in GalaxC. If *value* is present, you cannot assign any other value to the variable.

  You can define multiple nets that share a single value (not very useful and not currently

implemented) or define several nets with different values with the notations:

[τ] `net` {*list of nets*} `=` *value*
[τ] `net` {net_1 `=` $value_1$, net_2 `=` $value_2$, *etc.*}

GalaxC defines the `net` constructs as:

```
type arg T, unknown spclarg {name, value},
def {T net name} = persist T var name,
def {T net name = init} = (persist T var name = value),
def {net name} = persist Boolean var name,
def {net name = init} = (persist unknown var name = value)
```

"`net`" is not a reserved word, so `name` must be *prefix* or in braces. "`persist`" is a storage attribute that modifies `T` and tells GalaxC to put the net(s) into persistent storage.

[τ] `reg` *regs*
[τ] `reg` *regs* `=` *init*

These define one or more registers or latches of type τ. They are like local `var` declarations, but the registers are in persistent storage instead of on the stack. If τ is missing the registers are `Boolean`.

If present, *init* is the initial value of the registers. *Details TBD.*

You can define multiple registers that share a single initial value or define several registers with different values with the notations:

[τ] `reg` {*list of regs*} `=` *init*
[τ] `reg` {reg_1 `=` $init_1$, reg_2 `=` $init_2$, *etc.*}

GalaxC defines the `reg` constructs as:

```
type arg T, unknown spclarg {name, init},
def {T reg name} = reglatch T var name,
def {T reg name = init} = (reglatch T var name = init),
def {reg name} = persist (reglatch Boolean) var name,
def {reg name = init} = (reglatch unknown var name = init)
```

"`reg`" is not a reserved word, so `name` must be *prefix* or in braces. "`reglatch`" is a storage attribute that modifies `T` and tells GalaxC to put the register(s) into persistent storage and treat them as registers or latches.

`digital hardware` *body*
`digital hardware` (*options*) *body*

This construct is used to include GCHD logic into a GalaxC program. Usually the program is a "test bench" that provides input stimuli to simulated GCHD logic and then prints and/or verifies the results. For example, here is GCHD function that simulates the 3-input majority and minority functions:

```
fn simulate MajMin =
```

```
{   // Define logic for 3-input majority and minority functions.
    var M = digital hardware
    {
        net {a, b, c};
        net maj = a&b | a&c | b&c;
        net min = !maj;
    };
    if !M then return;      // Make sure "digital hardware" succeeded.
    // Simulate MajMin logic for all 8 input combinations.
    int var i;
    for i = 0 thru 7 do
    {   // Set nets a, b, and c to LSbs of i.
        a = i &? 4; b = i &? 2; c = i &? 1;
        evaluate M;         // Evaluate "digital hardware" for this
                            // combination of a, b, and c .
        printf("maj/min(%d, %d, %d) = %d %d\n", a, b, c, maj, min);
    };
};
```

A function with "`digital hardware`" is compiled (using `F6`) and called like any ordinary GalaxC `fn`. To run the above example, add the `fn` call "`simulate MaxMin`" at the end of the program and run the program using `ctl-F6`.

The `digital hardware` construct compiles *body* into normal PSI code, and than calls "`module post processing`" as if it were a `module` definition. GalaxC strips one level of braces around *body* so that nets defined in *body* remain defined. This allows later code to read and write their values in persistent storage. *Body* does not have any `input` or `output` arguments. Instead, it treats nets that have no value assigned as primary inputs, and allows any net to be a primary output.

In the above example, *body* is just some `net` declarations and assignment statements. A *body* may also contain module calls, making *body* the root of a hierarchical design.

At run time `digital hardware` returns a pointer to an internal data structure for simulating the module. If there was a problem creating that structure, `digital hardware` returns `NULL`. It also allocates persistent storage, initializes PS so that all non-constant nets are 0 (if binary) or '?' (see *options* below), and sets IB to point to PS.

In the current implementation, `digital hardware` *allocates PS in global memory. This is a temporary kludge, since global memory is limited to around 10KB per file, and it means that only one instance of* `digital hardware` *can be simulated at a time. At some point, we will have an option to allocate persistent storage dynamically.*

To perform the simulation, we loop through all possible values of `Boolean` nets `a`, `b`, and `c`. For each combination, we call "`evaluate M`" which executes the post-processed *body* using the current values of `a`, `b`, and `c`, writing the results to nets `maj` and `min`. Then we use `printf` to display the results, producing the following text in the Output window:

```
maj/min(0, 0, 0) = 0 1
maj/min(0, 0, 1) = 0 1
```

```
maj/min(0, 1, 0) = 0 1
maj/min(0, 1, 1) = 1 0
maj/min(1, 0, 0) = 0 1
maj/min(1, 0, 1) = 1 0
maj/min(1, 1, 0) = 1 0
maj/min(1, 1, 1) = 1 0
```

When we're done, we do not have to deallocate anything in version 0.0h. In the current implementation XOE keeps track of all GCHD modules created and deallocates them automatically.

The *options* argument tells `digital hardware` to do something other than building a simulation model for binary simulation. You can combine multiple option constants using the '`|`' operator. In the current version, the only option is "`simple symbolic`" which tells `digital hardware` to generate code to perform simple symbolic simulation (S^3 -- see §8.7). In S^3 mode the PSI code calls functions in `gchdmain.gal` to perform Boolean operations using S^3 algebra instead of the usual PSI `AND`, `OR`, and `NOT` instructions. In S^3 mode, all nets defined in *body* are converted from `Boolean` to type `S3char` (a subtype of `char`) and should be printed using format "`%c`". *Currently we do not support multi-bit nets, so all nets are* `Boolean` *in GHDL source code, and* `Boolean` *or* `S3char` *at simulation time.*

In the future, *options* will be used to generate modules for logic synthesis and other purposes.

`digital hardware` *name*
`digital hardware` (*options*) *name*

This is a simplified version of `digital hardware` which makes module *name* the root of a hierarchy for simulation. The module pattern must be one of the standard forms listed above with the `module` construct. At run time `digital hardware` returns the internal form of module *name* as the root of a hierarchy, allocating and initializing persistent storage as above.

This version automatically creates `nets` corresponding to the inputs and outputs of module *name*, and allocates space for the values of those nets in PS. Note that *name* could be module with a `FIGURE` body, in which case the inputs and outputs may be defined as module ports in the `FIGURE`.

If *options* includes "`simple symbolic`", the automatically-created nets are of type `S3char` instead of `Boolean`. In addition, if a root module input has a name that is a single letter -- like '`a`' -- the net is initialized to that letter instead of '`?`'.

`evaluate M`

Evaluate the combinational logic for module `M`, which is returned by "`digital hardware`". The above example shows how it's used. "`evaluate M`" also sets IB to the root of `M`'s PS.

`IB = PS`

Set the Instance Base to the beginning of persistent storage block `PS`, which is usually dynamically allocated. `PS` can be any pointer type, e.g., `@long`, and have `long` alignment. You can also set IB to a global or local array like this:

```
long var PS[100];
IB = PS;
```

You probably do not need to use "`IB = PS`" since "`digital hardware`" automatically sets `IB` to the correct location in persistent storage.

The GCHD constructs visible to users are in `gchd.gal`, which must be `included` explictly as `gchd.gi`. GCHD internals are primarily in `gchdmain.gal` and `figsim.gal`.

10.10. PSI Instructions for GCHD

GCHD requires a small number of additional PSI instructions, mostly for accessing persistent storage and manipulating IB. There are also some that tag nets and other variables so that "`module post processing`" knows what to do with them.

`PSVAR`

Convert the TOS byte offset to a persistent storage byte address relative to IB by adding IB to TOS. If TOS < 0, IB + TOS is the address of an input was passed by value or the address of an output pointer passed by reference. If TOS ≥ 0, IB + TOS is the address of a module's net or a component's PS. `PSVAR` is similar to `STAKVAR`.

`DECIB`

Decrement IB by TOS. This instruction has multiple uses. For example, to set IB to a new value `A`, we get the current value of IB using `PSVAR(0)` and then subtract IB `- A` from IB using `DECIB(PSVAR(0) - A)`. Another common use is with the `MCALL` instruction, described next.

`MCALL` *X*

This instruction is for calling a GCHD module. It's similar to the `CALL` instruction.

When a module *M* is defined using "`module` *pattern* = *body*", GalaxC first creates a standard PSI function from *body*. This *unprocessed* PSI function is never executed -- it's converted to executable PSI code in "`module post processing`". In an unprocessed PSI function, an `MCALL` indicates which submodule is called by a component, with *X* set to the address of the submodule's pattern. In a `.gi` file, *X* is a base-displacement value.

In a post-processed PSI function, an `MCALL` calls the post-processed submodule's PSI function at address *X*. PSI first adds `TOS` to IB to set IB to the component's PS. PSI leaves `TOS` on the stack, and when `MCALL` returns the caller executes `DECIB` to restore IB to its previous value.

The current version of `MCALL` only works for modules that do not return values.

`NETVAR`, `NETVAL`, `REGVAR`, `REGINIT`

These tag a GalaxC variable so that "`module post processing`" treats it as a net or register (or latch). `TOS` is set to the variable's size in bytes (0 if `Boolean`) shifted left two bits plus a 2-bit address alignment code (0 for byte alignment, 1 for short, 2 for long, and 3 for double). "`module post processing`" uses `TOS` to calculate the PS offsets of nets and registers within the module.

`NETVAL` has `NTOS` equal to the net's value. "`module post processing`" takes the code that calculates `NTOS` and converts it into an assignment so that it can be sorted into fundep order.

`REGINI` has `NTOS` equal to the register or latch's initial value. "`module post processing`" takes the code that calculates `NTOS` and converts it into a conditional assignment that is executed upon

module reset. *Details TBD.*

`SKIPHW` *X*

`SKIPHW` is for implementing "`digital hardware` *body*" at run time. It is followed immediately by unprocessed code for *body*, which it skips by setting PC to PC + *X* like a `GOTO` instruction. If *body* is a module identifier, the unprocessed code is a single `MCALL`. `SKIPHW` also calls GCHD function "`rebuild module`" which converts *body* into simulatable PSI code. This is usually fast since the module's internal data structures are usually available and do not need to be rebuilt from original PSI code using "`module post processing`". If `rebuild module` succeeds, `SKIPHW` pushes a pointer to root module *M* which GalaxC code can call using "`evalute` *M* ". If `rebuild module` fails, it prints an error message and `SKIPHW` returns `NULL`.

10.11. Compiling GCHD

This is a quick overview of how GalaxC compiles programs with GCHD constructs. Since GCHD constructs are GalaxC extensions, the GalaxC compiler does most of the work. Basically, GalaxC compiles each `module` definition as if it were a `fn` definition, except that `inputs`, `outputs`, and `nets` generate `PSVAR` addresses instead of `STAKVAR` and `GLOBVAR` like an ordinary `fn`, and instances of components generate `MCALL` instructions instead of `CALL`. When it has completed compiling a `module` body, GalaxC calls GCHD internal function "`module post processing`". This function takes the PSI object code and converts it into post-processed PSI code that can be simulated. Here are the main changes:

1. The original PSI code handles `MCALL` arguments like `CALL` arguments: it simply pushes them onto the stack. The post-processed PSI copies input arguments and output addresses into persistent storage instead of the stack.

2. In the original PSI code, `MCALLs` are in the same order as the source code. Post-processing sorts `MCALLs` into fundep order using topological sorting.

"`module post processing`" converts the original PSI into a tree structure for topological sorting. This same structure can be used for optimization, logic synthesis, and other CAD tasks. We can easily regenerate the original or post-processed PSI code from this tree. In the latter case, we can choose whether to generate code for binary simulation or simple symbolic simulation (§8.7).

To compile a `FIGURE`, GalaxC generates the same tree structure as for GCHD text. We can then use the same code for topological sorting and for generating PSI code. For a `FIGURE`, GalaxC generates original PSI code which looks the same as if the `FIGURE` was GCHD text instead. This original PSI code can be stored in a `.gi` file so that a `FIGURE` module can be called from other modules. (*This is not currently implemented.*)

10.12. GCHD versus Object-Oriented Programming

You can think of GCHD as a refactored form of Object-Oriented Programming (OOP). In OOP, each object has private data and "methods" that operate on that data. OOP private data corresponds to persistent storage inside GCHD modules. OOP messages correspond to evalfuns for pevals and kevals, since they update an instance's state according to current state values and additional inputs. The difference is that each OOP method requires an explicitly defined block of code, while GCHD evalfuns *implicitly* partition a GCHD module into sequences of code. GCHD lets you define a single module, and post-processing breaks that module into separate evalfuns.

The difference becomes more dramatic when you consider defining a hierarchical system using OOP. Each method for a module needs to send messages to the components of that module. For example, each kind of *peval* and *keval* is implemented as an evalfun that calls other *peval* and *keval* evalfuns. These evalfuns are rather repetitive, and you wouldn't want to write them yourself for a complex hardware system. GCHD post-processing automatically creates all the needed evalfuns, so you don't need to be concerned with the details.

10.13. Applying GCHD Concepts to Tables

This has not been implemented yet.

Up to this point, we have looked at using the fundep model for digital hardware. We can use the same model for evaluating tables (or spreadsheets, see Chapter 7). Each table cell contains an expression which may use the values of other cells as variables, or may use variables outside the table. GCHD allows tables to be sequential or fundep. A sequential table has its cells evaluated in row major order, i.e., left to right, then top to bottom. While cells usually use the values above and/or to the left, the sequential table can use later values -- which are the values from the previous evaluation of the table. If you want a table to be sequential, prefix it with "`seq`".

A fundep table requires GalaxC to sort the cells into fundep order, so there must be no circular dependencies -- GalaxC treats a table as combinational logic. This is the normal behavior of most modern spreadsheet programs. If can prefix a fundep table with "`fundep`" if you wish, or leave it off. *Currently all tables are sequential.*

GCHD designs can also include tables to specify logic function, where cells contain logic expressions. Some logic functions are easier to express and understand in tabular form. Most GCHD tables are fundep. In addition, XXICC uses tables to specify logic simulation input vectors and output results, and to specify pinouts for FPGAs.

10.14. GCHD Issues

This section describes issues with the current GCHD implementation and plans for improving it.

1. The current implementation only supports combinational logic.

2. The current implementation does not include hierarchical topological sorting, so you'll have to partition modules manually if they create apparent loops.

3. Modules currently must have type `void`. You cannot return a value from a module, and `return` expressions do not work.

4. We need to add simultaneous assignment versions of PSI `STOR` operators. For `seq` execution, immediate and simultaneous assignment are the same.

5. At some point I'd like to do parameterized modules, using regular arguments to specify parameters.

6. We need to add buses, using normal GalaxC integer variables like `ubyte` and `ulong`. Currently all I/Os and nets must be `Boolean`. You will be able to specify bit numberings like $a_{9:0}$ if you want, or let GCHD post-processing figure out how many bits you need. This can reduce the need to number bits and allow modules to be reüsed without explicit parameterization.

7. Need to add simple optimizations such as constant propagation to GCHD post-processing. At some point it will need to do more complex optimzations such as common sub-expression elimination. At some point we'd like to use this for regular GalaxC code as well.

8. All storage elements are updated simultaneously (in theory), though in practice they can be updated one at a time as long as updating one does not change the input values of another. So we may want to sort their updates as well. If there's a loop, we can use the regular stack to hold a value and break the loop.

9. XOE has limited error checking when compiling for simulation and does not identify which figure elements caused the error. It usually identifies which GCHD text caused a post-processing error, but does not identify which components caused a loop.

Bibliography

[B&M 86] Clifford Barney and Tom Manuel, "RISC: Is it a Good Idea, or just another Hype?", *Electronics*, May 5, 1986.

[BDM 87] A. Berenbaum, D. Ditzel, H. McLellan, "Architectural Innovations in the CRISP Microprocessor", *COMPCON*, pp. 91-95, February 1987.

[CP 99] Charles Petzold, *Programming Windows*, Fifth Edition, Microsoft Press, 1999.

[CH 81] C.A.R. Hoare, "The Emperor's Old Clothes", *Communications of the ACM*, 1981.

[DEC 73] Digital Equipment Corporation, *PDP-11 Peripherals Handbook*, 1973.

[JFB 11] John F. Beetem, *Programming in the GalaxC Language*, 2011-2013.

[JFB 11a] John F. Beetem, *Compiling and Running GalaxC Programs*, 2011-2013.

[JFB 11b] John F. Beetem, *Installing and Running XXICC*, 2011-2013.

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

[JFB 92a] John F. Beetem, *Galaxy CAD User Manual*, 1992.

[JFB 92b] John F. Beetem, "How Should Designs be Specified? Are HDLs Really the Answer?", *Proceedings of the First Open Verilog International User's Group Meeting*, March 1992.

[JFB 92c] John F. Beetem, "Hierarchical Topological Sorting of Apparent Loops via Partitioning", *IEEE Transactions on Computer-Aided Design*, vol 11, no 4, April 1992.

[LL 86] Leslie Lamport, *LaTeX: A Document Preparation System*, Addison-Wesley, 1986.

[SR 07] Scott Rosenberg, "Anything you can do, I can do Meta", *Technology Review*, Jan/Feb 2007.

[SR 08] Scott Rosenberg, *Dreaming in Code: Two Dozen Programmers, Three Years, 4,732 Bugs, and One Quest for Transcendent Software*, Three Rivers Press, 2008.

[Wiki 1] http://en.wikipedia.org/wiki/Allegory_of_the_cave.

[Wiki 2] http://en.wikipedia.org/wiki/Binary_scaling#Binary_angles.