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. All other chapters are new.

Revision 0.0, May 2011.

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

1. *Reducing Software Complexity*, part of the philosophy behind XXICC.

2. *XOE User Guide*, which describes how to use the XXICC Object Edtor.

3. *The GalaxC Simplified Window Manager (G-SWIM)*, a programmer's guide to the graphics library used by XOE and other GalaxC programs to construct GUIs.

4. *Advanced G-SWIM*, a continuation of the last chapter.

5. *XXICC Objects*, which describes how to include graphics in GalaxC programs and use them as building blocks for GUIs.

6. *Decoded XXICC Objects*, which describes how to use XXICC Objects at run time.

7. *XXICC Tables*, which describes how XXICC represents tables and how to use them with GalaxC to perform spreadsheet-style computing.

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.

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 85] 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.

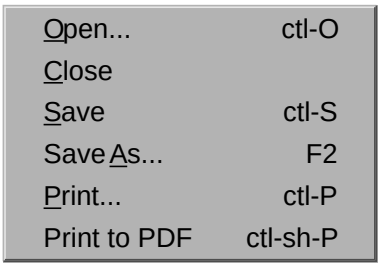
Basic XOE editing should be very familiar and intuitive to people used to modern GUI-based editors like LibreOffice and Microsoft Word. Whenever possible, XOE uses standard cursor keys and mouse operations. However, its initial appearance may be a bit startling to the new user in that menus and scroll 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 is quite stable and was used to write this book. Editing dialogs and tables is less stable since they are relatively recent additions. Figure editing is unstable and not enabled in the current version. 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 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. Alternatively, you can select Open at any time (whether the menu is exposed or not) by pressing `ctl-O`. In this case the keyboard equivalent is case sensitive: `ctl-sh-O` means something different. It's only the underlined letters that ignore case.

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 say 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 XOE 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... F2

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 XXICC objects (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

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. 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. *Horizontal scrolling is not currently implemented.*

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

3. XOE implements the standard scrolling keys:

   `PRIOR`, `NEXT` (Page Up, Page Down): Scroll 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 to previous or next page, and extend selection.

   `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 part of the page. *Not yet implemented.*

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.

2.5. 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 (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 currently selected 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 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.

Times ctl-T

Switch to Times Roman or similar serif font.

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.

Need to add menu items to enlarge font (`ctl-+`*), reduce font (*`ctl--`*), superscript (*`ctl-sh-UP`*), and subscript (*`ctl-sh-DOWN`*).*

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. 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:

- Section
- Subsection
- Subsubsect
- Chapter
- Appendix
- Figure caption
- Table caption

2.10. The Table Menu

Need to add Table Menu. For now, here are the keyboard equivalents for table editing:

- If you click an unselected cell in a table with mouse button 1, you select the cell. If you shift-click or shift-drag, you select a rectangular sub-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.
- The `r` command extends the current selection to one or more rows. The `c` command extends the current selection to one or more columns.

- 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).

- 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. The Figure Menu

Need to add Figure Menu. Figure editing is disabled until it is more stable.

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.

`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.

`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 file that is part of a multi-file program using GalaxC compiler. *Needs documentation.*

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

`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.

`F11` Toggle between displaying cell expressions or their values in compiled and executed tables. See Chapter 7.

`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.

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 coordinate 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 coordinates. 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` coordinates `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 coordinates 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` coordinates `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 Points:

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

You can access the individual coordinates 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 coordinates 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 coordinates increase from left to right and y coordinates increase from top to bottom, which is flipped from standard mathematical coordinates. By convention, r.z1 is the top left (upper left) coordinate and r.z2 is the bottom right (lower right) coordinate which means r.x1 ≤ r.x2 and r.y1 ≤ r.y2. This is reflected in the alternate coordinate 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` coordinates:

```
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 Coordinates

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

Each UWM has its own conventions regarding coordinates 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) coordinates 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* x *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 x *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.

3.2. 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 coordinates 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 coordinates 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 coordinates 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 function 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, and perhaps update the current position to `z2`. They 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 coordinate 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 coordinates must be sorted.

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

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

`draw arc (z, r) from z1 to z2`
`draw arc (z, r) from z1 thru z2`
Draw a circular arc with `Point` center `z` and `int` radius `r` from `Point z1` to (or through) `Point z2`, using the selected pen. The arc direction is clockwise, which is more natural for a flipped coordinate system with *x* increasing to the right and *y* increasing down. 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 `r` 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.

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.

"`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 coordinates *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`" 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

To get the size of a text string using the selected font call:

```
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.

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 coordinates and avoids character-by-character kerning. Future UWMs should be careful to maintain this property.

The height of `text` is currently ambiguous. On some UWMs the height of `text` always equals the height of its font, but others may reduce this if the characters of the string do not use the full ascent and/or descent, e.g., "`access`". XOE avoids this ambiguity by always using font dimensions for height.

G-SWIM has a specialized text width function used to format text left to right with automatic word wrapping. 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* coordinate 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 coordinate systems, both in pixels. *Screen coordinates* 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 coordinates* 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 coordinate.
    short   win.screenY,         // Window client area screen Y coordinate.
    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 coordinates `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`	`WHEEL`	"`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 coordinates 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 coordinates `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 coordinates of the upper left corner of `win`'s client area. These can be used to convert the window coordinates `x` and `y` of a mouse event to screen coordinates by adding `[win.screenX, win.screenY]`. They may also be used to convert coordinates 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 coordinates of window. If negative, use defaults. If primary or pop-up window, `x` and `y` are screen coordinates. 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 coordinates `[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.

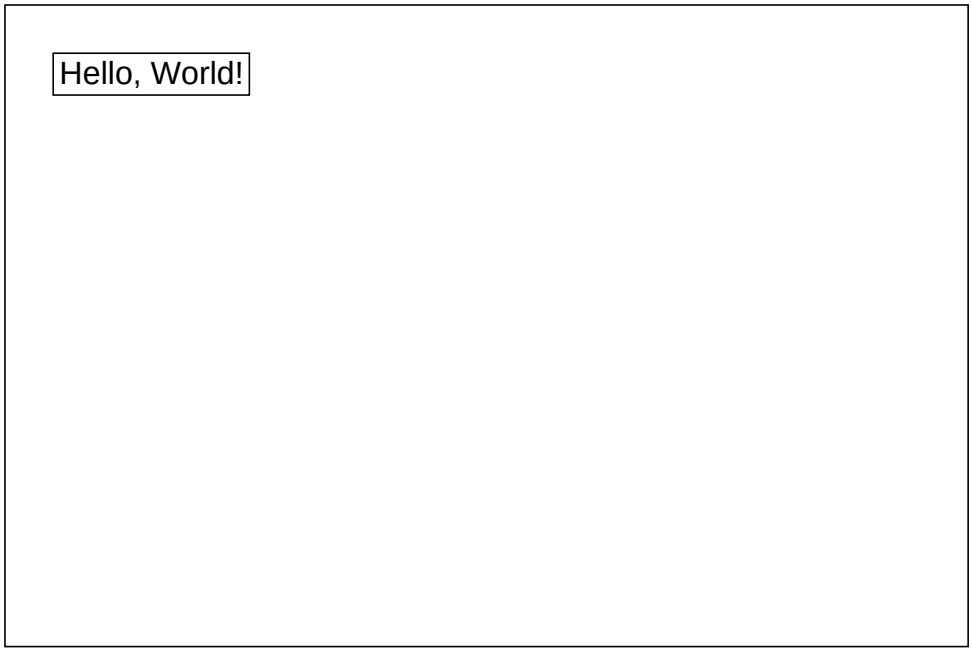
The last step calls "`create window`", which creates a nice window on the screen containing "`Hello, World!`" using the system font, looking something like:

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]` coordinates from the previous call to `sample mouse`. When the user first presses the left mouse button (`LBUTTONDN`), we initialize `prev` to the current coordinates. Then for each mouse movement event with the left button pressed (`LBUTTON`), we draw a line from `prev` to the current coordinates 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 coordinates must be sorted. All coordinates are relative to the upper left corner of the window, i.e., they are *client* or *user* coordinates.

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 coordinates.

`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 coordinates 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) coordinates passed to a mouse event are the client coordinates 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.

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.

`can render all`

This `Boolean` constant indicates whether the UWM can render all formats in response to `PUT_CB_EVENT` or impending loss of advertised data. It is `TRUE` for Win32 and currently `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 `string` 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.

"`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 opcode such as `RECT` or `TEXT`. An XOST consists of instructions to specify XOST properties such as cell width, followed by the XOST's opcode (`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 opcode. 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:

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).

LINETO(Δx, Δy)

Line segment from current position z_0 to $z_0 + [\Delta x, \Delta y]$, updating z_0 to the latter. Specifying line color and style TBD.

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(Δz_0, Δz_2)

Circular arc from z_0 to $z_0 + \Delta z_2$, with center at $z_0 + \Delta z_0$. The arc is drawn in the clockwise direction. Update z_0 to $z_0 + \Delta z_2$. Specifying arc color and line style TBD.

The following XOs are used in schematic diagrams, which are a kind of FIGURE. *Move these to a future chapter on schematics.*

DOT

Filled circle with center at z_0 and pre-defined radius. DOTs are used mostly for connection dots. Specifying dot color TBD.

DIR *dir*

Specify the origin of a SUBMOD (submodule). If the submodule is an I/O port, *dir* > 0 is the direction of that port. *Dir* is 0 for non-port submodules.

`INST(`*submod*, *rotmir*`)`

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

Submodules in a XOE file are numbered from 1: *submod* is the index of the submodule. Scaling is TBD.

`PIN` *net*

Indicates that the next pin of an `INST` is connected to `NET` *net*. `NET`s are numbered from 1 in a `FIGURE`: *net* is the index of the connected `NET` or 0 for an unconnected pin.

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 `CELL`s cannot contain as many kinds of XOSTs. `CELL`s are primarily used within `DIALOG`s for editable fields, within `FIGURE`s for placing text, and for constructing `TABLE`s.

A positive *width* is the width of the `CELL`. In a `TABLE`, the width of a column is set by the `CELL`s 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 `CELL`s are called `FIGtext`.

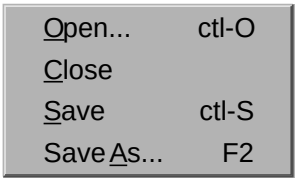
XOE sets a `TABLE` `CELL`'s height to its `TABROW` height -- i.e., the height of the parent `HSTACK` -- less separator line thickness. XOE sets a `FIGURE` `CELL`'s height to its contents plus CellBorder, and other `CELL`s' height to the current font height plus CellBorder.

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 schematic diagrams, which are a kind of FIGURE. *Move these to the schematics chapter.*

SUBMOD(*name, options*)

A SUBMOD is a schematic symbol such as a logic gate, a circuit element, or a port. It contains leaf XOs for drawing the symbol, and may also contain instances of other SUBMODs such as connection points (pins). The origin of the SUBMOD is usually specified with a DIR XO. *Name* is the name of the submodule, e.g., “Nand”.

COMP

A COMP (schematic component) contains an INST of a SUBMOD along with PIN XOs that indicate which NETs the pins of the instance connect to. A COMP may also contain FIGtext for associated text such instance name or properties such as resistor value.

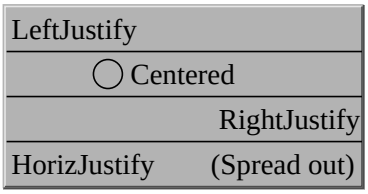
NET

A NET contains the LINETOs and DOTs for the wires that interconnect instance pins. It may also have FIGtext such as a net name or other properties. A NET does not have any connectivity information: that is derived from COMPs.

5.3. Options

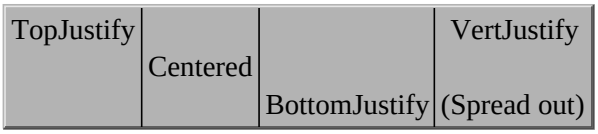
Many XOs have an *options* operand, which is a bit vector of additional properties. In the current implementation, *options* is a 16-bit number, of which only the low 14 bits are used so that they fit into a PSI small integer. The low 8 bits are for generic XO options:

- 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.

  If there is only one non-`SPACER` HorizJustify looks the same as LeftJustify. Menu buttons normally use HorizJustify as in 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.

  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 other 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. 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.

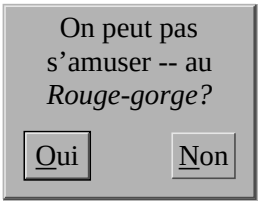
- RETisTAB: treat `ENTER` or `RET` as if it were a `TAB` character, i.e., go to the next field instead of closing the dialog. XOE uses this in its Open/Save File dialog.

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

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

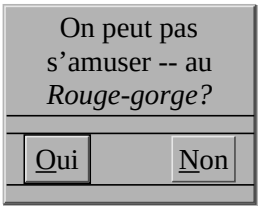
Here is a sample `DIALOG` box to illustrate the concepts of leaf XOs and subtrees:



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 coordinates, 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. `FIGURE`s show selected DXOs by displaying vertices that the user may drag to move or copy the DXO.
- AbridgedObject: part or all of a DXO subtree's contents are absent to save storage and processing.
- ShowValue: show the computed value of a `CELL` instead of the expression used to compute that value. This is used for displaying spreadsheet results. Similarly, show the value of an `XREF` instead of its symbol.
- HiddenObject: do not display or select this DXO.
- 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.
- MarkedObject: DXO is marked for special processing.

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 `CELL`s in a `FIGURE`, so that the `CELL` structure is transparent to the user. XOE sets the FIGtext option to identify these `FIGtext CELL`s.

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 coordinates. By using document coordinates, XOE does not need to recompute the coordinates of DXOs within `FIGURE`s, `DIALOG`s, 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 coordinate: $x = z.x$, $y = z.y$.

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 coordinates 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.

w, *h* DXO width and height, same as *size.x* and *size.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.

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).

cd General-purpose 32-bit field used for various purposes by different DXOs.

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 coordinates. 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*) coordinates. 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 coordinates 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*:

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:

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.
```


```
;
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.
```


```
;
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*.

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 XOE 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 really at the "proof of concept" stage at the present time. 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:

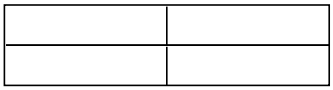
a[0,0]	a[0,1]	a[0,2]
a[1,0]	a[1,1]	a[1,2]

```
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 may display a GalaxC expression or its calculated 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 using 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 that naming cells is usually not much of 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.
- *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 *deadpan* cell. It contains the value 0, or `NULL` for `strings`. Some spreadsheet functions such as computing averages do not count deadpan 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 empty or may be labels, which have the same value as an empty cell. An *unnamed block* is a subtable with the same type (which may be `void`) and data format. Each cell in a table must be in exactly one block.

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

name`:` *expr*`;` *format*	Specify name, expression, and data format explicitly. Deduce type from *expr*, which may include typecast.
name`:` *type*`;` *format*	Deadpan 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 deadpan cell: copy data format from previous cell.

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 use 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 the name 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.
type	Specify type for deadpan cell: copy name and data format from previous cell. Empty cell: copy name, type, and data format from previous cell. Assume value is zero.
`:`	Unnamed empty cell: copy type and data format from previous cell. Assume value is zero.

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 `for` loops.

A cell may contain a *label* instead of an expression. The label is displayed in the table, but has no value and does not enter calculations. Labels are usually outside of blocks, but they may be inside a block to comment a row, column, or cell. Here are the notations for labels:

`,`*label*	Show *label* as value of deadpan 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 deadpan 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.

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.
`^`	Empty cell: copy name, type, and data format from cell to left of this one.

Yes, I know '`^`' is counter-intuitive since it suggests "up" rather than "to the left". However, I want to save '`<`' for other things and at least '`^`' does suggest a change of direction. Perhaps it's best to consider Unix regular expressions where '`^`' denotes "begin of line", which is to the left.

Each block has *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

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

Compiling a table requires two passes. Pass 1 determines the name, type, size, and position of each block in the table. Pass 1 also allocates a *helper* array which XOE needs to display table values. Pass 2 generates the code to evaluate a table and fills in the helper arrays.

GalaxC represents values of a document with tables as a `struct`, with each block (named or unnamed) as a field within the `struct`. The pointer to the `struct` is implied and details are TBD. We don't want to consider blocks as global variables, since there may be multiple instances of the document with different values. On the other hand, we don't want to consider blocks to be local variables, since they are allocated on a heap instead of the stack. Probably what is needed is the concept of *persistent storage* associated with a window that retains the current values of tables until the window is deallocated. Details TBD.

Blocks appear in the `struct` in the order defined by their upper left cell in row major order.

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 encode the byte offset of the block within the document `struct`, but in Pass 1 we won't know the offset until the blocks preceding the block are complete. So .value assumes 0 as the size of preceding blocks of the current table and updates .value at the end of Pass 1.

GalaxC also keeps track of unnamed blocks as variable patterns, but closes them when it declares a new named block, updating the current `struct` offset. Unnamed blocks are usually columns, but may be merged with adjacent columns if they have the same type and data format. [*Need to keep track of data format somewhere.*]

The helper array converts a (*row*, *col*) address into a pointer into the document value `struct`. In principle, this array could simply be an array of pointers, one for each (*row*, *col*), but we can save a lot of space in most applications by having the helper array contain a descriptor for each named or unnamed block, with the following fields:

- Starting row and column.
- Number of rows and columns.
- Data type.
- Pointer to first data item.
- Pointer to data format.

XOE does not need to use a helper array to see if a cell is empty or not, since it can determine this from the text in the cell, i.e., if it's a null string or begins with comma or semicolon.

In Pass 2, GalaxC generates actual code to update the values in the document `struct`. It also fills in the

actual values in the helper array.

7.5. Sample Table

Here is a simple table that calculates and displays squares and cubes for `i` = 0 through 6.

i	i*i	i*i*i
"	"	"
"	"	"
"	"	"
"	"	"
"	"	"
"	"	"

The first row defines expressions `i`, `i` squared, and `i` cubed. The remaining rows are copies of the expressions in the first row. When you compile and run this table, and then show table values using `F11` (*in the current implementation*), 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, and then press `F11`, XOE displays:

0	0	X
1	1	j
2	4	b
comment	9	c

As before, `int` values are displayed by default data format "`%d`". `Char` values are shown using format "`%c`". The comment in cell (3, 0) is displayed 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`.

Bibliography

[B&M 85] 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.

[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.

[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.