From M to Windows: One Step or Two?

by Max Rivers

Abstract

This paper postulates the need for a new transitional technology to aid in reengineering existing M software which was originally written for character-based displays, into graphical-user interfaced applications. One solution, Simple Windows in M (SWIM), is described in detail.

Windows has finally arrived. Why won't my customers use it?

There is a great deal of concern about the future of M. While M still excels in database management and fast development, its reputation continues to focus on its one major drawback: M "doesn't do windows."

Nearly all of the major M implementers offer state-of-the-art Windowing solutions. So why is it that not one of my customers has plans to convert their systems?

It is because they perceive the conversion as either too complex, too costly and too dangerous, or all three. They are concerned that they will have to switch from the simple M environment they are currently using to the more complex world of the client/servers; that they will need to buy a new version of M and in many cases will need new networking hardware; that they may have to hire new people to install and then maintain these new networks; and that they will either have to hire new programmers who know languages other than M, or go through the time and expense of retraining their existing staff.

And this is prior to the really frightening part of the conversion: they will have to rewrite their thousands (or millions) of lines of mission-critical code. All of this, just to pop up a few windows.

No wonder M continues to be windowless!

Converting Legacy Systems vs. New Development Continuity vs. Innovation

I have not hesitated to recommend any of the M vendor-windowing solutions to customers who are considering developing new M systems from scratch. However, converting **existing** M applications involves a completely different set of issues, and therefore a very different technology—a transitional conversion technology.

The primary concern of any institution which is dependent on its existing computer systems for doing business is that their applications continue to perform without interruption, even during the implementation of new technologies. The need for continuity creates a very conservative atmosphere, even in the most forward-thinking IS departments.

This means that the most important aspect of any conversion technology is that it must allow for small, incremental changes, while all unchanged routines which are part of the same application must be able to continue running as they always have. The perception by management that this transition can be accomplished without threatening the well-being of their mission-critical software cannot be overstated. Basically, any conversion whose risks (real or perceived) outweigh the rewards will cause the conversion project to never get off the drawing board.

Since existing systems which are candidates for conversion generally do the job they were intended for, conversion is seen as an enhancement, rather than a replacement. As a result, the budgets of both time and costs tend to be considerably less than budgets allotted for new development. This severely limits access to new hardware, software licenses and training.

Therefore, an ideal CHUI to GUI conversion solution would:

- Utilize existing hardware, O/S and M implementations
- Allow conversion of small amounts of code at a time

- Allow unconverted software to continue to run as is
- Be optimized for ease of learning
- Be inexpensive

The Missing Link: A Transitional Technology

In order to satisfy this very real-world set of concerns, we set out to write a programming utility designed specifically for converting existing systems from character interaction (CHUI) to windows (GUI).

The two primary considerations in designing this Application Programmer's Interface (API) were: 1) it needed to be simple enough for any M programmer to use without a lot of new training, and 2) it had to minimize the amount of code that needed to be changed in order for an existing M program to pop up a window.

The Simple Windows API was designed as a simple set of function calls written in standard M so that programmers would be completely familiar with the format and so that the API could run on all existing M implementations.

A separate, compiled application (SWIMMER.exe) runs in the background on the displaying PC's or Mac's and handles all the interactions between the end-user, the window, and M. The windows which SWIM draws are designed as very-smart-dumb-terminals, or very-thin-clients. As a result there is no need for any window-specific programming, no programming in any language other than M, and no need to port M code up into the window.

Because the API is written in M, it was designed as an integrated extension to M's functionality. So data returned from the user's interaction with the window comes in the form of a local M array. Another example of this M-centric approach is that only the name of a global is needed to pass a list of data to the window. The API handles the $Ordering for the application programmer, including getting more data from the global if the user scrolls down past the first batch of data sent to the window.

All interaction with the user which requires information not already in the window (program flow, syntax checking, help prompts) is handled from inside the M environment. In most cases, the original code which accomplished this for the character-based interaction can be used by changing the user interaction from READs and WRITEs to calls to the DRAW function.

One of the trickiest parts of writing this type of API is the age old trade-off between complexity and flexibility. We wanted the novice GUI M programmer to be able to create fully functioning windows with as little as a single line of code, while enabling the more advanced GUI programmers to have access to the full, exciting, and complex range of attributes available in the windows environment.

To this end, the API defaults every attribute, allowing application programmers to set only those qualities that are unique to their application, with the most prominent attributes (location, size, etc.) readily available as arguments to the function call. Then, for the more sophisticated GUI programmer, the final argument in every function allows for direct calls into the Windows Toolbox. A Rapid Application Development (RAD) tool, called Visual SWIM is also available which lets programmer's create their windows by drawing them and then accessing the complete set of attributes from selectable lists.

Simple Windows: One Example of CHUI-to-GUI Transition Technology

When designing any piece of software, a report for instance, it is always a good idea to start with the final product and then work backwards through the flow of data: from final report to data structures to data collection. In this same way, by focusing on the target users, namely application programmers involved in conversion of legacy systems, we were able create a design specification which uses a very different approach from windowing solutions that were designed with new development in mind.

To this end, we have developed a CHUI-based windowing utility which we call Simple Windows In M (SWIM). SWIM is:

- A set of 10 standard function calls: one for each widget type (window items like buttons and text fields are all called widgets in the windows parlance)
- Two functions for communicating between M and displayed windows: one for requesting data back from a window (RTN), and one for sending data to a window (SET)
- A compiled executable program, SWIMMER.exe, which runs in the background on the displaying com pute.
- A set of auxiliary functions for advanced GUI programming (such as managing multiple windows at the same time)

Creating a Window with SWIM

Every SWIM widget function call has seven arguments which allow the programmer to define its place in the window (X,Y), its size (WIDE,HI), label parameters (LABEL, LABEL POSITION) and what action(s) (such as returning data to M) it will make in response to users (COMMAND).

A typical function call would look like this:

```
set VAR=$$WIDGET^%SWIM(X,Y,LABEL,LABELPOS,
WIDE,HI,VALUE,COMMAND)
```

where WIDGET can be either:

1. BUTTON
2. TEXT
3. LABEL
4. CHECKBTN
5. RADIOBTN
6. DROPLIST
7. LOOKUP
8. LINE
9. BOX or
10. PIX (for picture)

Calls to any function returns the operating system's internal identifier for that widget. The application programmer need never know what this value is, because they have this data stored in a local variable (VAR in the example above) of their choice.

Each call to a widget defines, but does not display the widget. Once the whole window is defined, the application program calls the **DRAW** function, which alerts the background job to display the window, and interact with the user.

A functioning window can be created with the program:

```
set BUT1=$$BUTTON^%SWIM(10,10,"Hello World.")
set x=$$DRAW^%SWIM()
```

Notice that there is no need to specify the window's dimensions or attributes (though this is possible with the WINDOW function). SWIM automatically creates and sizes the window for all the widgets defined, in an effort to minimize what the application programmer has to include in order to create the window they desire.

Interacting with a Window: RTN and SET

If the COMMAND argument of any widget is set to **RTN**, that widget will return its name and value (and the names and values of any other widgets specified) to M in the **%rtn** array. This data, now in a local M variable, can then be processed exactly as if it had been input through a read statement in a character-based interface.

To make our Hello World program interactive, let's include an **RTN** command:

```
set BUT1=$$BUTTON^%SWIM(10,10,"Hello
World.",,,,,"RTN")
```

Sending data back to the window is done with the SET function. Let's add a text field to receive our data (we'll store the name of this text widget in the variable "T1"):

```
set T1=$$TEXT^%SWIM(10,50,"M answers:")
```

Since the name of that new field is in the M variable T1, the following code will write "Hi Back!" into that text field:

```
set x=$$SET^%SWIM(T1,"Hi Back")
   set x=$$DRAW^%SWIM()
```

Figure 1

With only a handful of functions, SWIM can produce all the major widgets (see Figure 2): displaying any GIF image, lines of any size or color, boxes, text fields with scrollbars, push buttons, checkbuttons and radiobuttons, pop-up fields and lookup fields, menus, as well as encoded password windows, help and error windows, and multiple windows at the same time.

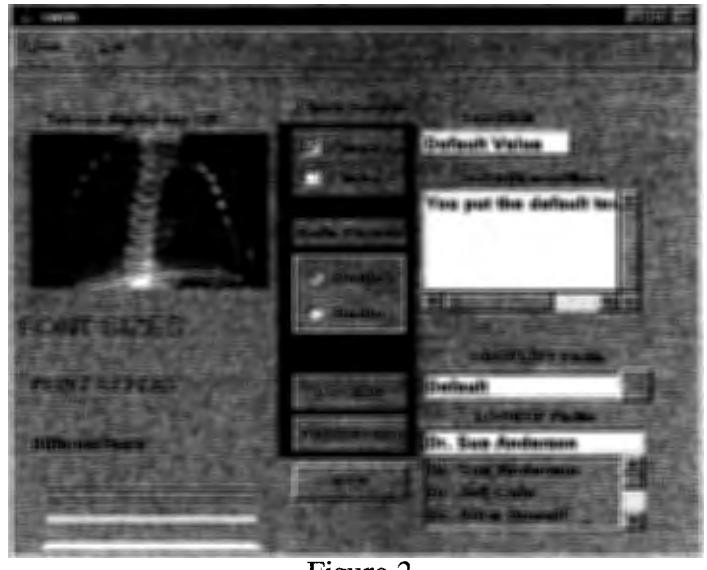

Figure 2

Converting CHUI to GUI

The real test of a transitional technology is how well it does on conversions.

The main challenge in converting code written for a character interface into a windows program is that the structure of the programs needs to be different.

There are three major types of program structures found in M programs: block, modular and object oriented (OOP), in ascending order of complexity (and therefore of flexibility).

Most character-based programs are written in block form. They generally have three different blocks which keep repeating: the first block, the user-interaction block, writes out the prompt and reads the response; the second block checks the user's response for syntax and appropriateness and then either returns to a previous prompt, quits or goes to the third block, which files the data (see Figure 3).

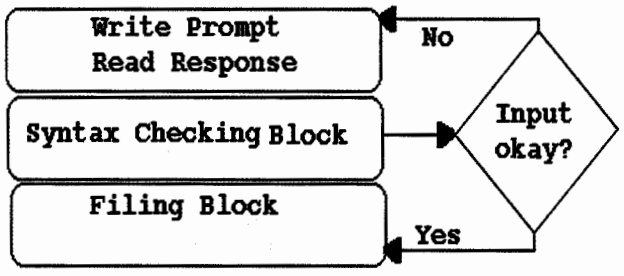

Figure 3

The two major variations on the block structure are:
1) Prompt and Syntax blocks repeat until all the data is collected, and then all the filing is done in one block which follows (Figure 4); or

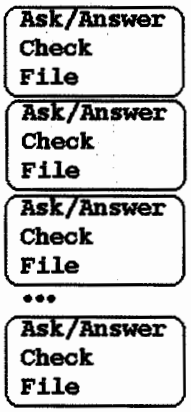

Figure 4

2) All the prompts are written out first, in a "form" and then all the reads, syntax and filing blocks follow (Figure 5).

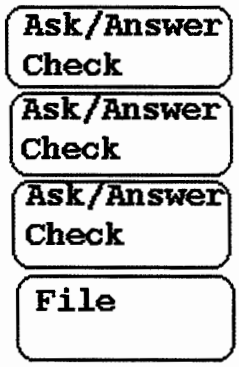

Figure 5

What all of these block structures have in common is that regardless of when the prompt is written out, each read happens in a linear order. The program determines which question comes next, and the user's only choice is to respond to the computer's next question. This type of interaction is called "modal."

Because of this modal interaction, the read blocks (and often the writes as well) occur every 10-30 lines, with syntax and filing blocks interspersed.

Windows programs tend to be non-modal. The program displays all the prompts (similar to the modal "form" structure) but the user can interact with the widgets in any order. In traditional window solutions, this is handled by porting the syntax code up into the widgets in the window, so that each object does its own syntax checking (and sometimes its own filing). In most Windows/M hybrid solutions, this means having some code up in the window, and other subroutines down in the database. This tends to make maintenance more difficult, and the design specification more complex. This type of structure is the kind found in Object Oriented Programming (OOP).

SWIM programs might be described as a half-step between the block and OOP structures. Simple Windows requires a modular structure: while there is no need to move code up to the window, but the block structure does need to be reorganized to accommodate the non-modal interaction allowed by windows.

By cutting and pasting all of the user interaction blocks into a "windows definition" module, and moving the syntax and filing blocks into a "RTN" module, a SWIMified routine restructures a CHUI program for GUI, using

most of the CHUI code unchanged.

Because the data is returned from the window in a local M variable, the original syntax and filing blocks will need little or no change. **In conversions done to date, CHUI M programs requires changing only about 10% of the code in order to interact with SWIM.**

Transition to What?

Since "SWIMifying" a set of routines requires moving from block to modular structure, it is a good investment even if the ultimate solution turns out to be adopting a more integrated, vendor's windowing solution. The time spent adjusting the flow of control to accommodate non-modal user interaction will lessen the amount of changes needed in the code even if the end platform is significantly different from the current one. In the meantime, users get the advantage of the GUI-interface, and system designers have a real-world prototype to work from. Transitional conversion can be used to move up the implementation of windowing technology before an institution is ready to commit to an overall conversion effort, without closing the doors to any new technology which may come along in the near future.

Converting Modules Incrementally

Another major concern for conversion to a new technology is how much of the original application to convert at one time. New technologies that require new hardware, versions of M, and/or new programming languages may require that the entire application be converted at once, or at the very least, modified to accommodate the new platforms. Since Simple Windows was designed to run in the existing environment, it is possible to implement on a single screen by screen basis, leaving the rest of the application completely unchanged.

Modules that would benefit most by the non-modal window's environment (like patient registration or accessioning) could be converted first. Then modules that are particularly easy to convert, ones that are already in the "forms" structure could be done next. Then, other modules could be converted on an "as needed" basis, perhaps because there is new development in that module, or they are simply next in priority.

It is also completely conceivable that some aspects of an institution's legacy system will remain CHUI-based indefinitely, while others are completely converted into windows-based interaction.

Hardware Configurations

Another significant concern about conversions, is the question of hardware. Many M implementations which are still using character-based interaction are running on PC's. Whether the PC has the M database on its own disks, or the database is on a server somewhere else, if the PC is running any windowing operating system, and both the M database and the displaying PC share access to any disk, SWIM can be used to display windows. This is true for any windowing O/S including Unix, Microsoft Windows, or Apple's Mac O/S. It is important to understand that it is the O/S of the displaying computer that is important, not the operating system of the server. So applications running on mainframes or DEC mini-computers can upgrade to windows using SWIM as long as the end-users are running on computers with one of the windowing O/S's.

Conclusion

Converting existing character-based M applications to windows is a significantly different process than designing window's applications from scratch. This paper introduced the concept of a transitional CHUI-to-GUI technology for applications written in M. Simple Windows is one example of transitional conversion technology which provides the capability to incrementally upgrade mission-critical software to GUI with relatively minor changes to the existing applications using standard M as the only programming tool necessary, and with little or no hardware changes required. **M**

Max Rivers is CEO of Simple Windows, Inc. located in western Mass., as well as being an M consultant for over 15 years. He can be reached at maxrivers@aol.com.

Information about SWIM can be found at http://www.SimpleWindows.com, or through email at SWIM@SimpleWindows.com.
