A Win32 Programming Interface for SML / NJ

Sheng Liang* Lorenz Huelsbergen†

August 1995

Abstract

We have built a Win32 API function call interface for SML / NJ, making it possible to write ML applications with a Microsoft Windows user interface. The interface includes the *complete* support for all major components of the Win32 system, as well as two high-level libraries for constructing *menus* and *controls*.

In this document, we give an overview of the SML Win32 interface, discuss how it was designed and implemented, and show how it can be used to write a simple application.

*Department of Computer Science, Yale University, New Haven, CT 06520-8285. liang-sheng@cs.yale.edu
†AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974. lorenz@research.att.com

1 Introduction

This document gives an overview of a Win32 API function call interface for the Standard ML of New Jersey (SML/NJ). The interface makes it possible to write ML applications with a Microsoft Windows user interface. By using ML instead of, for example, C, programmers are able to take advantage of ML's strong static typing, higher-order functions, the module system and automatic garbage collection. These features increase productivity and result in more reliable code. We will argue that the SML Win32 interface meets the usability and reliability goals, and provides the right level of abstraction and functionality. Here are the two major characteristics of our system:

- Our interface is *complete*. It includes support for window management, graphics, system services (processes and threads, file I/O, memory management, etc.[1]), security, multimedia, and extension libraries (such as the Common Dialog Library). In total, there are more than 1300 functions and 500 Windows specific data structures.
- We follow a layered design. A *low-level interface* provides the full functionality and preserves the naming conventions and function arguments of the Win32 C interface [1]. As a result, the Win32 API manuals [1] are a useful reference for Windows programming in ML. Familiarity with Windows programming in C carries over to the ML framework. Our low-level interface can be used to build various *high-level libraries*; e.g., a menu library constructs menus from easy-to-read specifications in ML.

Win32 is the native system call interface in Windows NT. Windows 95 leaves out certain features (such as security and Unicode), while providing a few extensions of its own (such as the Direct Draw Game API). In addition, the Win32s compatibility package supports a subset of Win32, and allows Win32 programs to run under the 16-bit Microsoft Windows 3.1.

The next section is a tutorial in which we design and implement a simple application. Section 3 provides an overview of the SML Win32 interface, followed by a discussion on its design and implementation in Section 4.

We assume that the reader is familiar with ML [4]. This document only covers a small part of Windows programming; the reader should consult one of many Windows programming books (such as [3]) for more information.

2 An Example

In this section we construct a Win32 program in ML which allows the user to freely draw in a window by pressing down the left mouse button while moving the mouse. To simplify the presentation, we first introduce a version with minimum capabilities, and later add a more sophisticated user interface with menus and dialog boxes.

2.1 A Simple Drawing Program

Our simple program displays a drawing window with a title bar labeled "`Draw`" (Figure 1). The user draws with the mouse, and terminates the program by double clicking the upper-left corner of the window (as in a conventional Win32 application).

We will present the code in a literate-programming style, with code segments separated by explanations. The entry point of our program, the `draw` function, begins with:

```
fun draw () =
```

[1] Some of these are not necessary for SML/NJ, whose standard library already provides support for various operating system services.

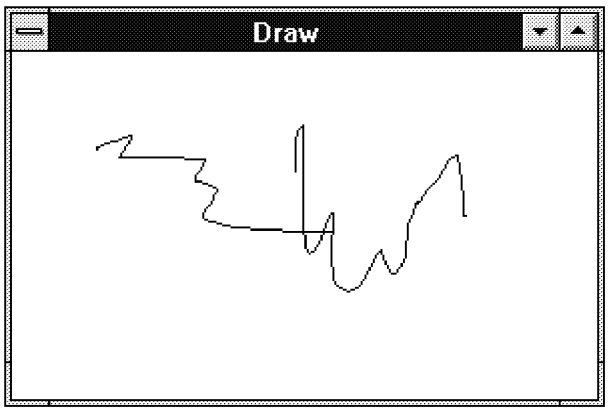

Figure 1: A Simple Drawing Program

```
let
  val className = Win32.RawString.make "My Draw"

  val wc = Win32.makeWndClass {
             style = 0,
             lpfnWndProc = Win32.makeWndProc wndProc,
             cbClsExtra = 0,
             cbWndExtra = 0,
             hInstance = Win32.getInstance (),
             hIcon = CI.NULL,
             hCursor = Win32.loadCursor(CI.NULL, Win32.IDC_ARROW),
             hbrBackground = Win32.makeIntHandle (
                               Win32.COLOR_WINDOW + 1),
             lpszMenuName = CI.NULL,
             lpszClassName = className}

  val res = Win32.registerClass (Win32.addr wc)

  val _ = Win32.freeWinStruct wc
```

Besides `Win32`, we use the SML/C interface module `CI`. As with most Win32 applications, we must first construct and register a *window class*. The code is slightly complicated by the fact that Win32 functions frequently take pointers into the C heap as arguments, and return C strings and structures. For example:

```
Win32.RawString.make : string -> t_char ptr
```

converts an ML string to a C string (i.e., a character pointer). The C string is in turn passed (with other required arguments) to:

```
Win32.makeWndClass : {..., lpszClassName : t_char ptr} -> wnd_class
```

which allocates, initializes, and returns a window class structure (`wnd_class`) in the C heap. Type `wnd_class` is not just a C pointer to the structure, but also contains other bookkeeping information needed by the SML/C interface. We can extract the C pointer to the structure by using:

```
Win32.addr : wnd_class -> wnd_class ptr
```

Due to the limitations of the current SML/C interface, we must explicitly free objects in the C heap.[2] For example, after registering the window class, the C structure passed to `makeWndClass` is no longer needed. We call `Win32.freeWinStruct` to release the C heap memory occupied by `wc`.

The most important field in a window class structure is `lpfnWndProc` — the address of a programmer defined *call-back* function invoked by Win32 to process the messages sent to any window in this class. We will explain messages and define the `wndProc` function later; now we return to the second part of the `draw` function:

```
val windowName = Win32.RawString.make "Draw"

val wnd = Win32.createWindow (
            className,                  (* class *)
            windowName,                 (* name *)
            Win32.WS_OVERLAPPEDWINDOW,  (* style *)
            100,                        (* x *)
            100,                        (* y *)
            300,                        (* height *)
            200,                        (* width *)
            CI.NULL,                    (* parent *)
            CI.NULL,                    (* menu *)
            Win32.getInstance (),       (* app. instance *)
            CI.NULL)                    (* extra info. *)

val _ = app CI.free [className, windowName]

val _ = (Win32.showWindow (wnd, Win32.SW_SHOW);
         Win32.updateWindow wnd)
```

The above code creates a window and calls `showWindow` and `updateWindow` to make it appear on the screen. After the window is created, the C strings holding the names of the window and window class can be freed. Note that C pointers are freed by `CI.free`, whereas Win32 structures (e.g., `wnd_class`) are freed by `Win32.freeWinStruct`.

To build an interactive drawing application, we must handle user inputs such as mouse clicks and movements. The Win32 system generates a *message* for each input event, and manages per-thread[3] *message queues* to store unprocessed messages. The last part of the `draw` function simply enters a loop which repeatedly fetches and dispatches on the messages from the application's message queue:

```
val msg = CI.malloc (CI.sizeof Win32.msgT)

fun eventLoop _ =
       if Win32.getMessage (msg, CI.NULL, 0, 0)
       then
         (Win32.dispatchMessage msg;
          eventLoop ())
       else
         ()
in
```

[2] In the future, one could incorporate a weak pointer finalization mechanism to make the explicit deallocation unnecessary.

[3] In contrast, the earlier 16 bit Windows 3.1 has one global message queue.

```
    eventLoop ();
    Win32.unregisterClass(className, Win32.getInstance());
    CI.free msg
  end
```

We allocate an uninitialized message structure in the C heap using the SML/C interface functions `malloc` and `sizeof`. `Win32.msgT` contains the C type information about a Windows message structure.

The `Win32.getMessage` function retrieves a message from the message queue and places it in the specified structure (`msg`). The event loop terminates when `getMessage` sees a `WM_QUIT` message and returns `false`.

Before exiting the program, we unregister the window class (making it possible to later register a class with the same name, i.e., "`My Draw`"). Finally, we free the storage used to hold messages.

The event loop passes all messages (other than `WM_QUIT`) to `Win32.dispatchMessage`, which then invokes the pre-registered call-back function `wndProc` to carry out the main functionalities of the drawing application. The `wndProc` function is defined as follows:

```
val lastPoint = ref {x = 0, y = 0}    (* remember the last point *)

fun wndProc {window = hWnd, message = message,
             w_param = wParam, l_param = lParam} =
  if message = Win32.WM_DESTROY then
    (Win32.postQuitMessage 0; 0)
  else if message = Win32.WM_LBUTTONDOWN then
    (lastPoint := {x = Win32.loWord lParam,
                   y = Win32.hiWord lParam};
     0)
  else if message = Win32.WM_MOUSEMOVE then
    if (Bits.andb (wParam, Win32.MK_LBUTTON) <> 0) then
      let val dc = Win32.getDC hWnd
          val {x = x, y = y} = !lastPoint
          val x' = Win32.loWord lParam
          val y' = Win32.hiWord lParam
      in
        Win32.moveToEx(dc, x, y, CI.NULL);
        Win32.lineTo(dc, x', y');
        Win32.releaseDC(hWnd, dc);
        lastPoint := {x = x', y = y'};
        0
      end
    else
      0
  else
    Win32.defWindowProc (hWnd, message, wParam, lParam)
```

The system passes a window call-back function the handle of the affected windows, the message identifier, and two parameters. The exact interpretations of `l_param` and `w_param` vary with messages, and could be ignored or treated as integers, pointers, etc.

When the application is terminated (by, for example, double clicking the upper-left corner of the main window), the system generates a `WM_DESTROY` message. `WndProc` responds by calling `Win32.postQuitMessage` which puts a `WM_QUIT` in the message queue, causing the event loop to terminate.

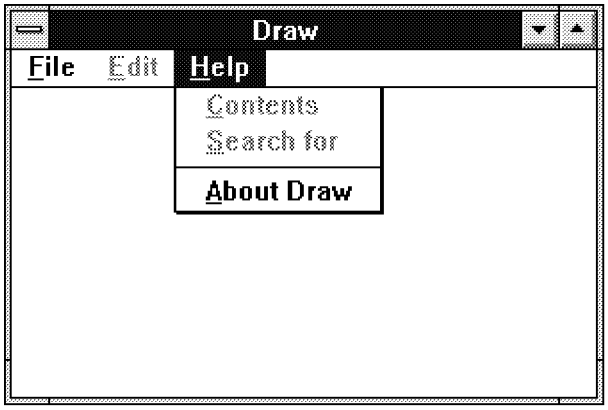

Figure 2: A Drawing Program with Menu

`WndProc` handles two mouse related messages (`WM_MOUSEMOVE` and `WM_LBUTTONDOWN`) to perform the drawing. Note how the high and low words of the `l_param` argument encode the mouse cursor position. The actual drawing is performed by four Win32 graphics API functions: `getDC`, `moveToEx`, `lineTo`, and `releaseDC`. A call-back function must always pass the messages it does *not* handle to `defWindowProc` to invoke default actions.

We now have a simple but working Win32 program. Next we will make it more sophisticated by adding menus and dialog boxes.

2.2 Menus, Controls and Dialog Boxes

Win32 menus are usually attached to the menu bar at the top of a window.[4] The programmer assigns a unique identifier (represented as an integer) to each item on the menu. Later, when the user selects a menu item, the system generates a `WM_COMMAND` message, passing along the identifier as an argument.

Win32 provides built-in support for a wide variety of *controls*, such as buttons, check boxes, text editors, list boxes and scroll bars. A properly designed Win32 user interface should always put controls in *dialog boxes*, a special kind of temporary pop-up window that receives user inputs. The call-back function of the dialog box receives `WM_COMMAND` messages when the user operates on a control. As with menu items, every control sends its parent (usually a dialog box) a unique identifier along with the `WM_COMMAND` message.

It is tedious and error-prone to directly use the Win32 API to create menus, controls and dialog boxes. Indeed, most Windows software development environments provide tools that automatically generate the low-level data structures needed to build the desired user interface. The Microsoft Resource Compiler, for example, takes a high-level *resource definition file* describing user interface objects, and outputs a binary file which can later be linked into the application's executable and loaded at run-time.

We have written a high-level interface to create menus and dialog boxes containing controls. The interface consists two structures (`Menu` and `Dialog`), and is documented in Appendices A and B. To demonstrate its use, we add a menu (shown in Figure 2) to the drawing program. With our interface, the menu is specified as follows:

```
val ID_HELP_CONTENTS  = 40010   (* unique within the
val ID_HELP_SEARCH    = 40011      application *)
```

[4] Win32 also supports *floating pop-up menus*.

```
val ID_HELP_ABOUTDRAW = 40012

val menu = [POPUP {text = "&File",
                   flag = 0,
                   menu = [ ... ]},
            POPUP {text = "&Edit",
                   flag = Win32.MF_GRAYED,
                   menu = [ ... ]},
            POPUP {text = "&Help",
                   flag = 0,
                   menu = [MENUITEM {text = "&Contents",
                                     id = ID_HELP_CONTENTS,
                                     flag = Win32.MF_GRAYED},
                           MENUITEM {text = "&Search for",
                                     id = ID_HELP_SEARCH,
                                     flag = Win32.MF_GRAYED},
                           SEPARATOR,
                           MENUITEM {text = "&About Draw",
                                     id = ID_HELP_ABOUTDRAW,
                                     flag = 0}
                          ]}
           ]
```

A menu consists of a list of menu items. A menu item is either a `MENUITEM` with text, id and flag fields, a `SEPARATOR`, or a `POPUP` submenu which in turn contains a list of menu items. Menu items that are gray (initially set through the `flag` field and later toggled by the Win32 API function `enableMenuItem`) do not generate `WM_COMMAND` messages, and are used for disabled or unimplemented features. With our high-level interface, we only need to make one change in the `draw` function to add a menu to the main drawing window:

```
val wnd = Win32.createWindow (
              className,
              windowName,
              Win32.WS_OVERLAPPEDWINDOW,
              100,
              100,
              300,
              200,
              CI.NULL,
              Menu.create menu,      (* add menu *)
              Win32.getInstance (),
              CI.NULL)
```

`Menu.create` builds a menu from its SML description, and is a function implemented using low-level calls to the Win32 API. Selecting a menu item often causes a dialog box to pop up. As an example, we will attach to the menu item "About Draw" a dialog box containing some information about the application, as shown in Figure 3.

We must specify the geometry and style of the dialog box and all the controls it contains:

```
infix ||
val op || = Bits.orb
```

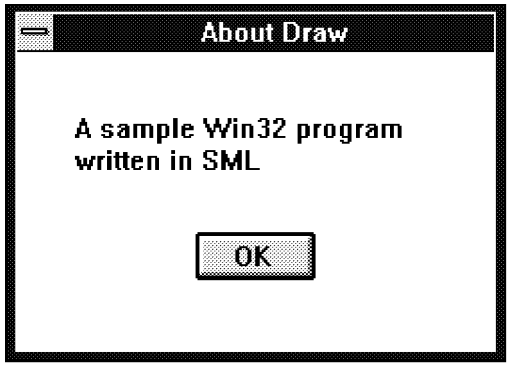

Figure 3: The “About Draw” Dialog Box

```
val aboutDlgBox =
      { style = Win32.WS_BORDER || Win32.WS_SYSMENU ||
                Win32.DS_MODALFRAME || Win32.WS_CAPTION,
        pos = {x=40, y=40},
        size = {w=120, h=75},
        title = "About Draw",
        font  = NONE,
        controls = [{ style = Win32.WS_VISIBLE || Win32.SS_LEFT,
                      pos = {x=15, y=15},
                      size = {w=95, h=30},
                      id = 0,
                      class = "STATIC",
                      text = "A sample Win32 program written in SML" },
                    { style = Win32.WS_VISIBLE ||
                              Win32.BS_DEFPUSHBUTTON,
                      pos = {x=45, y=45},
                      size = {w=30, h=12},
                      id = Win32.IDOK,
                      class = "BUTTON",
                      text = "OK" }
                   ]
      }
```

“STATIC” and “BUTTON” are two *built-in* controls in Win32. A static control consists of a piece of text or graphics, never generates a `WM_COMMAND` message, and therefore does not need a unique identifier. The push button is labeled “OK”, and when pressed, generates a `WM_COMMAND` message with the control identifier `IDOK`.

The call-back function `wndProc` must now include an extra case to handle the `WM_COMMAND` messages generated by the menu:

```
    ...
    else if message = Win32.WM_COMMAND then
         if loWord wParam = ID_HELP_ABOUTDRAW then
           Dialog.dialogBox(aboutDlgBox, aboutDlgProc, hWnd)
         else
           Win32.defWindowProc (hWnd, message, wParam, lParam)
    else
```

```
    ...
```

When a WM_COMMAND message arrives, we check to see if it contains the identifier of the menu item "About Draw". If so, we create a dialog box, and register aboutDlgProc as the dialog box call-back function; if not, we invoke the default message handler.

Dialog.dialogBox creates a dialog box which captures the input focus until the user clicks the "OK" button. Thus all aboutDlgProc has to do is to wait for the IDOK message, and calls Win32.endDialog to close the dialog box:

```
fun aboutDlgProc {window = hDlg, message = message,
                  l_param = lParam, w_param = wParam} =
    if message = Win32.WM_COMMAND andalso wParam = Win32.IDOK then
      (Win32.endDialog(hDlg, 0); true)
    else
      false
```

Unlike window call-back functions, a dialog call-back function returns false to invoke default message handling.

3 The SML Win32 Interface

In this section we give an overview of the SML Win32 interface, an ML module Win32, that consists of constants (e.g., WM_COMMAND), functions for the Win32 API (e.g., createWindow), types (e.g., t_char), and functions for manipulating C structures (e.g., wnd_class).

We built the SML Win32 interface using the recently developed SML/C interface [**?**], which supports calling C functions from ML and vice versa, and automatically converts arguments into the correct format. Part of the SML/C interface is as follows:

```
structure CI :
  sig
      type caddr                (* a C address *)
      val malloc : int -> caddr
      val free : caddr -> unit

      datatype ctype = CintT
                     | CstructT of ctype list
                     ...

      val sizeof : ctype -> int
  end
```

CI.ctype describes the type of C data. For example, the structure type:

```
struct {
  int x;
  int y;
};
```

is represented as "CstructT [CintT, CintT]." Ctype provides information for the SML/C interface to convert between ML and C data, as well as to determine the size of C data.

The complete signature for Win32 is large and is not listed or explained in detail here; instead, we will describe the conventions we followed to map C entities to ML counterparts. This, along with the interface signatures, allows one to deduce how to use a particular feature in the SML Win32 interface.

3.1 Constants

All Win32 constants are ML values, and have exactly the same name and capitalization as those in C Win32 API. Most have type `Win32.flags`, which is a type abbreviation for 32-bit integers. A few constants have pointer (`CI.caddr`) or string types. For example:

```
val WM_COMMAND : Win32.flags  (* Windows message *)
val RT_CURSOR : CI.caddr      (* a predefined Win32 resource type *)
val LBSELCHSTRING : string    (* a string used to obtain unique window
                                 message for communication between
                                 between dialog and caller. *)
```

3.2 Functions

Function names follow a simple mapping scheme shown in the following two examples:

SML name	C name
`createWindow`	`CreateWindow`
`mouseEvent`	`mouse_event`

Arguments to functions are passed in tuples, in the same order as in C. For example:

```
unregisterClass : t_char ptr * h_instance -> bool
```

corresponds to:

```
BOOL UnregisterClass(LPCTSTR, HINSTANCE);
```

In C, the system passes four arguments to the main entry point of a Win32 program. In SML, we provide access to these arguments using four functions:

```
getInstance : unit -> h_instance     (* handle of current instance *)
getPrevInstance : unit -> h_instance (* handle of previous instance *)
getcmdLine : unit -> string          (* command line *)
getCmdShow : unit -> int             (* show state of window *)
```

3.3 Types

We use the SML/C interface address type `CI.caddr` to denote all pointer types. For example, `LPCTSTR` is a pointer to character strings, and is mapped to ML as "`t_char ptr`:[5]"

```
type 'a ptr = CI.caddr
```

The SML/C function call interface currently cannot pass structures and floating point numbers as values; they must be passed by reference. Fortunately, very few functions in Win32 take arguments of such types. For those which do, the ML side instead accepts a C pointer; and a C stub function dereferences the pointer before entering the Win32 API function. When a Win32 function returns a structure by value (on the C stack), a C wrapper function allocates memory from the C heap, copies the structure, and returns the C heap pointer.

For clarity, we use a set of ML type synonyms for all basic C types used in Win32:

[5] The Win32 interface does not automatically convert between ML and C string types because C strings admit operations (e.g., casting to integers and overwriting an existing string) that are not available to ML strings. See the next section for details.

C type	ML type synonym	ML definition	notes
BYTE	byte	int	
DWORD	dword	Word32.word	
float	float ptr	CI.caddr	must be passed by reference
long	long	Word32.word	
int	s_int	Word32.word	
UINT	u_int	Word32.word	unsigned
short	short	int	
USHORT	u_short	int	unsigned
UCHAR	u_char	int	unsigned
WCHAR	w_char	int	Unicode char
TCHAR	t_char	char	or int when using Unicode
WORD	word	int	16 bit Win32 word
void	void	unit	
<...>PROC	c_proc ptr	CI.caddr	call-back functions
H<...>	h_<...>	CI.caddr	handles (e.g., HWND $\rightarrow$ h_wnd)
<type> *	<type> ptr	CI.caddr	

For example, the Win32 API function for drawing an arc has the following C type:

```
BOOL AngleArc (
    HDC  hdc,     /* handle of device context */
    int x,        /* x-coord of circle's center */
    int y,        /* y-coord of circle's center */
    DWORD radius, /* circle's radius */
    float start,  /* start angle */
    float sweep,  /* sweep angle */
  );
```

The SML type signature for the same function is:

```
val angleArc : h_dc *
               s_int *
               s_int *
               dword *
               float ptr *
               float ptr
               -> bool
```

3.4 Structures

Every C structure used in Win32 has a corresponding ML type. We provide a set of functions to construct and inspect C structures.

Conversion

We provide the necessary types and functions to convert between ML records and C structures. For example, the Win32 `POINT` structure:

```
typedef struct tagPOINT {
  LONG x;
  LONG y;
} POINT;
```

has the following ML counterparts:

```
val pointT : CI.ctype
type point = CI.caddr * CI.caddr list
type point_t = {x: long,
                y: long
               }
val makePoint : point_t -> point
val extractPoint : point ptr -> point_t
```

`Win32.pointT` describes the layout of the C structure `POINT`. `Win32.point` is a pair consisting of the actual pointer to the C structure and other book-keeping information needed by the SML/C interface. We can retrieve the C pointer using:

```
fun addr (cptr, _) = cptr
```

In ML, we can construct Win32 C structures in two ways, depending on whether or not they need initialization. For example, we can use:

```
makePoint {x = 0, y = 0}
```

to create a point at coordinate $(0, 0)$, or build an uninitialized structure:

```
CI.malloc (CI.sizeof Win32.pointT)
```

Structures of Undetermined Size

Win32 occasionally uses C structures of undetermined size. For example, the structure that denotes a menu item varies in size depending on how much text the menu item contains:

```
typedef struct {
    WORD mtOption;
    WORD mtID;
    WCHAR mtString[1];     /* unknown size */
} MENUITEMTEMPLATE;
```

The `mtString` slot contains a Unicode string of arbitrary length. ML interface functions for `MENUITEMTEMPLATE` take additional arguments that specify the size:

```
val menuItemTemplateT : int -> CI.ctype
type menu_item_template = CI.cobj
type menu_item_template_t = {mtOption: word,
                             mtID: word,
                             mtString: w_char Array.array
                            }
val makeMenuItemTemplate : int -> menu_item_template_t
                              -> menu_item_template
val extractMenuItemTemplate : int -> menu_item_template ptr
                                  -> menu_item_template_t
```

Unions

We use SML data types to represent Win32 unions. For example, for the Win32 type:

```
typedef union {
    WCHAR UnicodeChar;
    CHAR  AsciiChar;
} UNICODE_OR_ASCII_CHAR;
```

the SML interface provides:

```
datatype unicode_or_ascii_char_t = UnicodeChar of w_char
                                 | AsciiChar of char
```

Data constructors correspond to field names in the union. The conversion functions are similar to those for structures, except that C unions lack a tag indicating the actual type, making it impossible to convert them back to SML data types. Therefore calling an "`extract`..." function for unions raises an exception.

4 Design Rationale and Implementation Strategy

In this section we explain the design and implementation decisions in the SML Win32 interface, which in turn determine the power and limitations of the current approach.

4.1 Scope and Level

Win32 is large. If we choose to only support a subset with limited functionality, we could build a much cleaner and elegant interface. However, after investigating the organization of Win32 API as well as existing Win32 applications written in C, we opt for a complete, albeit low-level interface that is easy to implement, debug, and document. It must be complete because we cannot foresee what a Windows application might need. A low-level interface is relatively easy to construct and test, and is already well documented by the existing Win32 documentation.

The abstraction mechanisms in ML enable us to build various higher-level packages on top of the lower-level Win32 interface. For example, the `Menu` and `Dialog` libraries used in Section 2 are implemented using the lower-level interface in less than 200 lines of ML code.

Win32 is an integral part of Microsoft operating systems, and does not have a rigorous low-level specification like the X protocol. Therefore we cannot follow the approach used in eXene [2], where the C API layer is bypassed altogether, and a cleaner interface is built from scratch. Such an endeavor would not only require us to reimplement a large part of Microsoft operating systems, but also result in poor interoperability with existing Win32 applications.

4.2 Organization

The Win32 API consists of 5 parts: window management, graphics device interface (GDI), system services, multimedia and extension libraries (e.g., the Common Dialog Library).

```
structure Win32User : WIN32USER        (* window management *)
structure Win32GDI : WIN32GDI          (* GDI *)
structure Win32Sys : WIN32SYS          (* system services *)
structure Win32MM : WIN32MM            (* multimedia *)
structure Win32CommDlg : WIN32COMMDLG  (* common dialog lib *)
...                                    (* other extensions *)
```

Each module can be further divided into smaller parts. Although such finer organization makes the interface specification itself cleaner, it is hardly useful for programmers, because the Win32 documentation provides little indication of where each function or type belongs. Instead, to correspond to the C world, we provide a flat `Win32` structure containing all features:

```
structure Win32 =
  struct
    open Win32User
    open Win32GDI
    open Win32Sys
    open Win32MM
    open Win32CommDlg
    ...
  end
```

4.3 Functions

C functions in the Win32 API pose two difficulties:

1. They are not type-safe. The programmer must frequently perform coercions.
2. Some functions take a large number of arguments, which are difficult to manage.

At the first glance, it seems we could use ML's strong type system and record types to overcome these problems. In the end, we do not take advantage of either, for the reasons stated below.

Type Safety

In C, the first argument to the `CreateWindow` function is the name of the window class, usually a text string. Indeed, the argument is marked as having type `LPCTSTR`, meaning a pointer to a constant string. But the programmer can also choose to pass a 16-bit integer denoting a Win32 global atom. Therefore the ML version of the function must instead take arguments of the following type:

```
datatype string_or_int = STRING of string
                       | INT of int
```

Alternatively, we can have multiple instantiations of the same function with different types. The problem is that the number of instantiations grows exponentially when we have more arguments of such types.

Things get even more complicated in the `WinHelp` function:

```
BOOL WinHelp(
  HWND hWnd,
  LPCSTR lpszHelp,
  UINT uCommand,
  DWORD dwData
  );
```

Depending on the value of the third argument, the fourth argument can be ignored, an integer, a string or a pointer to structures. We cannot define the SML version of this function without a careful reading of its documentation, and combine the third and fourth arguments into a single data type.

This issue becomes unsolvable when we consider the default Windows call-back function:

```
LRESULT defWindowProc(
  HWND hWnd,
  UINT msg,
  WPARAM wParam,
  LPARAM lParam
  );
```

Here the value of `msg` determines what `wParam` and `lParam` stand for. Unlike the `WinHelp` structure, however, the system sends many different kinds of messages, thus resulting in huge data types for `WPARAM` and `LPARAM`. Furthermore, the number of possible messages increases as Win32 evolves, and programmers are allowed to send arbitrary user-defined messages to any application in the system. Therefore a completely type safe approach will not work without an extensive reorganization of the existing Windows programming framework.

Our solution is to let the ML interface mimic the C world, and represents, for example, `WPARAM` as a 32-bit word. As in C, we provide coercion functions between integer and pointer types. This approach is hardly satisfying, but we choose it because of the lack of a better alternative, and the low-level nature of our interface.

Tuples or Records?

For functions taking a large number of arguments, the logical choice is to pass them in a labeled record, so that the programmer need not worry about argument order. Uncertainties arise, however, for functions with two or three arguments, where tuples and records are equally applicable. In general, there is no right solution to this issue, which is largely a matter of taste. Instead of making the choices for the programmer, we decided to be predictable and always use tuples to pass the same number of arguments in the same order as in C. Furthermore, a uniform tuple-based interface eliminates the need to invent and document field names.

4.4 Character Strings

The string type in C is an ordinary pointer to characters. Win32 functions usually expect NULL terminated character strings, but there are a number of situations where they expect other formats or overwrites an existing string. For this reason — and the difficulties of strong typing mentioned above — we use C style strings for all Win32 API functions, and provide an ML module to convert between C and ML strings:

```
structure RawString :
  sig
      type raw_string = CI.caddr

      val alloc : int -> raw_string         (* uninitialized *)
      val make : string -> raw_string       (* initialized *)
      val get : raw_string -> string
      val set : raw_string * string -> unit
  end
```

Note that just like an ordinary C address, a `raw_string` can be coerced to and from an integer, and must be explicitly reclaimed using `CI.free` when no longer in use.

A higher-level library built on top of the SML Win32 interface can hide the details of dealing with `raw_string`s. For example, the higher-level `Menu` and `Dialog` modules used in Section 2.2 only require the programmer to supply ordinary ML strings.

4.5 Documentation

ML signature files provide a summary of the SML Win32 interface, while the standard Win32 documentation gives precise and detailed information on every aspect of the system. It is easy for the programmer to locate the needed information because of two reasons: 1) constants, functions and types follow a simple name mapping scheme, and 2) the SML and C versions of Win32 functions take the same number of arguments in the same order.

4.6 Implementation

Even with the above mentioned design simplifications, we are still faced with the daunting and error-prone task of writing the SML/C interface stub functions. A better alternative is to generate such functions from type descriptions. It is even better if we need not write the type descriptions ourselves. Indeed, the Win32 header files (e.g., `windows.h`) always contain the necessary information about Win32 constants, types, structures, and function prototypes.

Although automatic stub generation from header files is possible in theory, the C preprocessor complicates the task of building such a tool. We cannot generate stubs after running preprocessor because, in Win32, all constants and many functions are defined as macros. On the other hand, directly processing the header files is also difficult. This, for example, would require a global analysis to determine that one of the following macros defines a constant, the other a function:

```
#define WH_MAXHOOK      WH_MAX          /* constant */

#define CreateWindow    CreateWindowA   /* function */
```

Rather then writing a sophisticated analyzer, we modified the header files by hand to make sure that they are reasonably well formed. Our stub code generator is therefore relatively simple (about 2,700 lines of SML code).

Assuming that the Win32 header files have been tested, bugs can only be introduced by modifying the header files or through an incorrect stub code generator. Fortunately, we found that the Win32 header files need only go through minor hand modifications. The stub code generator detects most errors in the header files. Although only a fraction of the generated ML and C stub code has actually been tested, we are quite confident of the correctness of the entire interface, which is generated mechanically by the same process. We have written two sample Win32 applications in SML, and did not encounter a single bug in the interface itself.

4.7 Future Improvements

C pointer finalization The major inconvenience of this SML Win32 interface is that we must to some extent explicitly manage C heap objects. Representing C heap objects as weak pointers will allow the system to free them automatically.

Higher-level interfaces Taking advantage of the powerful abstraction mechanisms in SML, it is possible to build a much cleaner interface which frees programmers from low-level details. The `Menu` and `Dialog` modules are a start, and many more useful higher-level libraries remain to be constructed.

More library support At present, the interface supports windows management, GDI, system services, multimedia and the Common Dialog Library. New extensions to Win32, such as the Common Control Library, can be incorporated into our system.

Better Unicode support To support Unicode in the current version, we need to set a special flag and recompile the run-time system. It is possible in the future to let Unicode and ASCII versions of the interface be two instances of the same signature.

5 Conclusion

The SML Win32 interface provides support for developing ML programs under the Microsoft Windows environment. ML programmers can incorporate a sophisticated user interface into their

applications. Meanwhile, C programmers can migrate to ML and continue to follow the familiar way of user interface construction, while taking advantage of ML's power to implement the core functionalities.

We have made compromises to accommodate the irregularities in C, so that we can incorporate all the Win32 features without a dramatic reorganization of the overall programming framework. Overall, we believe that we have obtained an immediately useful system with room for future improvements.

References

[1] Microsoft Corporation. *Win32 programmer's Reference*, volume 1-5. Microsoft Press, 1993.

[2] Emden R. Gansner and John H. Reppy. eXene. In *CMU Workshop on SML*, 1991.

[3] Charles Petzold. *Programming Windows 3.1*. Microsoft Press, 1992.

[4] Jeffrey Ullman. *Elements of ML Programming*. Prentice-Hall, Englewood Cliffs, N.J., 1993.

A Menus

A menu consists of a list of menu items. A menu item is either a `MENUITEM`, a `SEPARATOR`, or a `POPUP` submenu. The `Menu.create` function takes the ML description, and returns a Win32 menu handle `h_menu`.

```
signature MENU =                    (* signature for Menu *)
sig

  datatype menu_item = MENUITEM of {id : int,
                                    text : string,
                                    flag : int}
                     | SEPARATOR
                     | POPUP of {menu : menu_item list,
                                 text : string,
                                 flag : int}

  val create : menu_item list -> h_menu

end
```

Each menu item must have an application-wide unique `id`, which is used by the call-back procedure to distinguish what menu item is selected. The `flag` field controls the appearance of the menu item, and can be the bit-wise *or* of `MF_CHECKED`, `MF_GRAY`, etc. The Win32 documentation for `MENUITEMTEMPLATE` contains a full description of the `flag` field.

B Dialogs

A dialog is a temporary window consisting of a list of controls. The `dialogBox` function creates a *modal* dialog box which captures the input focus, and does not return until `endDialog` function is called. The `createDialog` function, on the other hand, creates a *modeless* dialog box which functions like an ordinary pop-up window.

```
signature DIALOG =                    (* signature for Dialog *)
sig

  type dialog_proc_t = {window : h_wnd,
                        message : u_int,
                        w_param : w_param,
                        l_param : l_param} -> bool

  type control = {style : flags,
                  pos : {x : word, y : word},
                  size : {w : word, h : word},
                  id : word,
                  class : string,
                  text : string}

  type dialog = {style : flags,
                 pos : {x : word, y : word},
                 size : {w : word, h : word},
                 title : string,
                 font : (word * string) option,
                 controls : control list}

  val dialogBox : dialog * dialog_proc_t * h_wnd -> s_int

  val createDialog : dialog * dialog_proc_t * h_wnd -> h_wnd

end
```

The Win32 documentation gives the complete list of dialog box and control styles. The programmer can specify a custom font for a dialog; otherwise the default system font is used. The predefined control classes include “BUTTON”, “LISTBOX”, “EDIT”, “COMBOBOX”, “SCROLLBAR”, and “STATIC”.