



Classic Mac OS 7 Application Basics

Toolbox Initialization and Setup

Developing a windowed application on System 7 (such as 7.5.3) requires initializing the Mac Toolbox managers. This is typically done with a sequence of calls at startup, before creating any windows ¹. For example, a minimal `main` might include:

```
InitGraf(&qd.thePort);      // Initialize QuickDraw (graphics)
InitFonts();                // Font Manager
InitWindows();               // Window Manager
InitMenus();                 // Menu Manager (even if no menus yet)
TEInit();                   // TextEdit (for text fields)
InitDialogs(nil);           // Dialog Manager (nil=system alert proc)
InitCursor();                // Set the cursor to arrow by default
```

These calls prepare the system for GUI operations ². Even if your app doesn't directly use fonts or dialogs, other parts of the system might rely on them, so it's standard to call them all. After this initialization, the application can create its main window.

Creating a Window

In classic Mac OS, windows can be created either from resources or programmatically. Using resources (with `GetNewWindow`) is common, but here we show creating a window in code using `NewWindow`. First, define a `Rect` for the content area (the interior drawable region) and optionally a drag area rectangle (for unconstrained dragging):

```
Rect windowRect, dragRect;
SetRect(&windowRect, 10, 40, 310, 240);          // size of content
area
SetRect(&dragRect, -32767, -32767, 32767, 32767); // no drag limit
(full desktop)
WindowPtr myWin = NewWindow(
    nil, &windowRect, "\pMy App Window", true,           // title (Pascal
    string), initially visible                         // string), initially visible
    noGrowDocProc, (WindowPtr)-1L, true, 0);
```

Here `noGrowDocProc` specifies a standard document window that cannot be resized (no grow box ³ ⁴). The title is a Pascal string (prefixed with a length byte), denoted by `"\p...` in MPW/Think C. Passing `nil` for the Window record tells the OS to allocate memory for it. The last parameters set this as the front window (using `(WindowPtr)-1L`) and enable the window's close box ³. After creation, the window is visible (since we passed `true`).

Note: In System 7, windows don't automatically update their content when uncovered. You must handle repaint (update) events. Also, using Pascal strings is required for many Toolbox calls (e.g., window titles and certain text functions), so remember to use the `\p` prefix or otherwise construct a `Str255` when needed.

Event Loop Basics

Classic Mac applications run an event loop to process user input and system events. System 7 introduced `WaitNextEvent`, but it's common (and simpler for a quick start) to use the older `GetNextEvent` along with `SystemTask` ⁵. A basic event loop looks like:

```
EventRecord event;
Boolean quit = false;
while (!quit) {
    SystemTask(); // allow system to handle housekeeping (network, etc.)
    if (GetNextEvent(everyEvent, &event)) {
        switch (event.what) {
            case mouseDown:
                // handle mouse click...
                break;
            case keyDown:
            case autoKey:
                // handle keyboard input...
                break;
            case updateEvt:
                // handle window redraw...
                break;
            case activateEvt:
                // handle activation (activate/deactivate window)...
                break;
            case kHighLevelEvent:
                // handle AppleEvents (in System 7) if needed...
                break;
        }
    }
}
```

`SystemTask()` gives time to background processing (like networking or drive I/O) on cooperative multitasking systems ⁶. `GetNextEvent(everyEvent, &event)` returns the next event (or a null event if none). In System 7, you could use `WaitNextEvent` for better CPU idle behavior, but that requires also handling suspend/resume events. The above loop is sufficient for a basic app, though it will run at full CPU when idle (to improve, consider `WaitNextEvent` in a real app).

Handling Common Events

Update Events: When `event.what == updateEvt`, the OS is telling your app a window needs redrawing (e.g., after being uncovered or exposed). The `event.message` field carries the WindowRef. Handling this involves setting the current graphics port to that window, beginning an update, redrawing content, then ending the update:

```

case updateEvt: {
    WindowPtr win = (WindowPtr) event.message;
    BeginUpdate(win);
    SetPort(win);
    EraseRect(&win->portRect);           // clear the content region
    // Draw content, e.g., text:
    MoveTo(10, 20);
    DrawString("\pHello from macvox68!");
    EndUpdate(win);
    break;
}

```

This example erases the window's content area and draws a string [7](#). In a real app, you would draw whatever UI elements or text are needed (ensuring to confine drawing to `win->portRect` or the update region).

Mouse Events: On a classic Mac window, you must handle clicks in the title bar, close box, etc., manually. For a `mouseDown` event, use `FindWindow` to determine where the click occurred, then respond accordingly [8](#) [9](#):

```

case mouseDown: {
    WindowPtr clickWin;
    short part = FindWindow(event.where, &clickWin);
    switch (part) {
        case inContent:
            if (clickWin != FrontWindow()) {
                SelectWindow(clickWin); // bring background window forward
            }
            break;
        case inDrag:
            DragWindow(clickWin, event.where, &dragRect); // allow window
dragging
            break;
        case inGoAway:
            if (TrackGoAway(clickWin, event.where)) {
                quit = true;           // close box clicked -> signal quit
            }
            break;
        case inSysWindow:
            SystemClick(&event, clickWin); // pass to system (for desk
accessories)
            break;
    }
    break;
}

```

This ensures the user can drag the window, activate it, or close it properly [8](#) [9](#). The call to `TrackGoAway` checks if the click on the close box is a true click (not just a mousedown without

release). If it returns true, we set `quit = true` to break out of the main loop and exit. (Alternatively, you might call `DisposeWindow` here and set a flag.)

Keyboard Events: If your application has text input fields or expects key commands, handle `event.what == keyDown` or `autoKey`. The `event.message` has the key code and ASCII char (in `hiWord/loWord`). You might use `char c = event.message & 0xFF;` to get the character and check for Return or command-key shortcuts (modifier bits are in `event.modifiers`). If you have an editable text field (see next section), you would forward the character to it.

Idle Time: The loop above doesn't include an idle handler, but you can add code in the `while` loop (outside of `if(GetNextEvent)`) to do background tasks when no events are pending. Alternatively, use an `osEvt` (null event) or simply the `!GetNextEvent` branch for idle processing. Ensure to call `SystemTask()` periodically as shown.

Once the user triggers quit (e.g., clicking the close box or selecting a Quit menu in a more complete app), break out of the loop, perform cleanup (dispose windows, etc.), and return from `main` to let the application terminate.

Adding UI Controls (Buttons and Text Fields)

System 7's Control Manager allows adding buttons, text fields, and other controls to windows. Controls can be created from resources ('CNTL' resources with `GetNewControl`) or via code with `NewControl`. Using `NewControl` is straightforward for basic controls. For example, to add a "Speak" push-button in our window:

```
Rect btnRect;
SetRect(&btnRect, 20, 160, 100, 180); // position within window
ControlHandle speakBtn = NewControl(
    myWin, &btnRect, "\pSpeak",           // title text (Pascal string)
    true, 0, 0, 1,                      // initially visible, value=0, min=0,
    max=1
    pushButProc, (long)0);
```

This creates a control of type "push button" (`pushButProc`) with the given bounds and title ¹⁰. We set `visible=true`, initial `value=0` (unpressed), and a `min..max` range of 0..1 (for a button, 0 means up, 1 would mean pressed). The last parameter (`refCon`) is an application-defined reference value (here 0). The button appears immediately in the window (the Control Manager draws it when created).

For a **text input field**, the Control Manager provides an edit-text control definition. We can similarly create an editable text box:

```
Rect textRect;
SetRect(&textRect, 20, 40, 300, 140);
ControlHandle textCtrl = NewControl(
    myWin, &textRect, "\p",             // no initial text (empty Pascal string)
    true, 0, 0, 0,                      // visible, value=0 (not used), min=0,
```

```
max=0  
editTextProc, (long)0);
```

Here `editTextProc` (constant 64) specifies an editable text control (the Control Manager internally uses `TextEdit` for this control) ¹¹. This will create a bordered text field that the user can click and type into. The Control Manager automatically handles drawing the text and basic editing behaviors. You should call `SelectWindow(myWin)` after creating controls if you want keyboard focus to be in your window. Also, when processing `event.what == keyDown`, you can let the Control Manager handle keystrokes by using `HandleControlClick` or sending the event to the control. However, the `editTextProc` control typically takes input if it is active. You might need to call `TakeFocus` or similar (depending on Toolbox version) to ensure the text control is active. In many cases, clicking the text control once will focus it, and then keystrokes go to it by default.

After creating controls, remember to update/redraw them during update events. A convenient way is to call `DrawControls(window)` inside your update handler, which will redraw all controls in that window ¹². For example, in the update event case:

```
case updateEvt:  
    BeginUpdate(win);  
    EraseRect(&win->portRect);  
    DrawControls(win);           // draw buttons/text fields  
    // ... draw any additional content ...  
    EndUpdate(win);  
    break;
```

This ensures controls like buttons and edit fields are refreshed when needed ¹².

Control Events: To detect button presses, use `TrackControl`. In a `mouseDown` handler, if `FindWindow` returns `inContent` for your window, you can further call `FindControl(event.where, myWin, &ctrl)` to see if a control was clicked ¹³. If `ctrl` is not `NULL`, call `TrackControl(ctrl, event.where, NULL)` to handle the interaction ¹³. `TrackControl` will track the mouse for you and return true if the control was "hit" (e.g., button released while still inside it) ¹³. At that point, you can take action (e.g., start speaking if the Speak button was pressed). For example:

```
ControlHandle ctrlHit;  
if (FindControl(event.where, myWin, &ctrlHit)) {  
    if (TrackControl(ctrlHit, event.where, NULL)) {  
        if (ctrlHit == speakBtn) {  
            // Speak button was clicked and released  
            DoSpeakText();  
        }  
    }  
}
```

For the edit text control, basic text editing doesn't require your code for each key (the control definition handles it). But if you want to retrieve the text (e.g., when Speak is pressed), use `GetControlData` or `TextEdit` functions. A simple way is `TEHandle hTE = (TEHandle)(**textCtrl).ctrlData;` to

get the TextEdit handle, then access `(*hTE)->hText` for the handle to the text, etc., or use `GetControlData(textCtrl, kControlEditTextTextTag, ...)` on newer OS. On System 7, you might use `GetControlText` to copy out the text. For example:

```
Str255 contents;
GetControlText(textCtrl, contents);
```

That will give the current text in Pascal-string format (ensure the buffer is large enough to receive it). Now you have the string the user typed, which you can send to the Speech Manager to speak.

In summary, by using `NewControl` with appropriate control types (`pushButProc`, `editTextProc`, etc.), you can build a simple UI with buttons and text fields entirely in code. The Toolbox will handle much of the user interaction if you use the provided control definitions, leaving your code to respond to completed actions (button clicks, etc.).

With the basics of UI in place, we can move on to integrating the Mac **Speech Manager** for text-to-speech and handling network input via sockets, which will be covered in the next document.

Sources:

- Michael Martin, *Classic Mac Development: Beginning in Earnest* 1 7 8 9
- Inside Macintosh: Toolbox Essentials – Control Manager (for `NewControl` and control types)

10 12

1 2 3 4 5 6 7 8 9 Classic Mac Development: Beginning in Earnest | Bumbershoot Software

<https://bumbershootsoft.wordpress.com/2023/02/03/classic-mac-development-beginning-in-earnest/>

10 12 13 BlobDemo1 - Blob Manager Demo source (part 1 of 5)

https://groups.google.com/g/mod.mac.sources/c/oIDcmGbmD_A

11 TextControls.c

<https://github.com/Kelsidavis/System7/blob/cd2bdd951cf7a8982f6aab11676df79f56ee1618/src/ControlManager/>

TextControls.c