# *WorldTime: A Pilot World Clock*

*Steve Mann*
*sem@cdpubs.com*

It started out as a simple idea – create a straightforward Palm Pilot application to demonstrate some of the basics of building, testing, and deploying Pilot software. But it quickly got out of hand. I didn't mean for this to happen, it just did. What follows is the sordid story.

For some helpful background information, you may want to read "Pilot Programming Primer" in the May/June issue of *PDA Developers* for some useful background information. It's available in text format, without graphics, at http://www.cdpubs.com.

## World Time, Take One

I constantly rely on a PDA to calculate dates and times in foreign countries so that when I place an overseas telephone call, I know I'm not waking someone up in the middle of the night or disturbing them on a weekend. Unfortunately, the Pilot doesn't have a world time calculator. If I'm going to carry a Pilot with me and actually use it, I need at least a basic time zone application. It seemed like a simple first-time program.

My first design was much like the time zone calculators on most other PDAs – there's a world map and a home location. You select a point on the map by tapping on it, and the program tells you the date and time of that location. The immediate problem with this traditional implementation is that the Pilot doesn't really have a home location that it saves. It just stores a date and time. To deal with that issue, I decided that the user could enter two locations. The program would assume that the first is their current location, whether it's home or not, and assign it the current date and time from the device. The second location would be the location the user was interested in knowing about.

The first job was to find some time zone data. After scouring CompuServe, I found a world map with reasonably clearly marked time zones and about 100 cities. This data seemed like a good starting point. There were a few strange markings like "+30" in a few spots, but being naive about time zones, I didn't worry about those. I downloaded the map as a GIF file and set it aside.

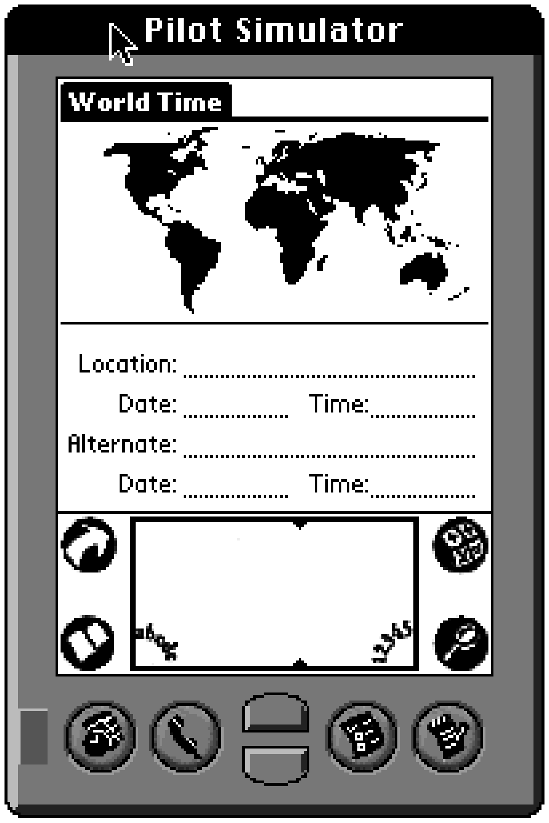


Figure 1 - The original World Time.

### *The Interface*

Figure 1 shows my first interface design. The user is supposed to tap on one of the location fields to indicate which location they want to change, and then tap on the world map. The program finds the city nearest the tap point and displays the date and time for that location. If the primary location (the one on top) is selected, the date and time are set to the Pilot's date and time. If the secondary location is selected, and the primary location is already identified, the program calculates the date and time of the secondary location using it's knowledge of the time zone difference between the two locations.

### *The Resources*

The interface uses a simple set of resources:

- A "tFRM" form that has a list of all the other interface elements except the world bitmap;
- A "tFBM" form bitmap resource, which points to the map;
- The world map "PICT" graphic, which was created from the times zones map I downloaded from CompuServe;
- A "tTTL" form title resource; and
- A set of fields and labels ("tFLD" and "tLBL" resources) for the data-entry area at the screen's bottom.

### *Source Code*

Here's the source code for my first pass at World Time. Note that this is not finished code. For reasons I explain shortly, this first prototype was never finished. I'm including it here for instructional purposes only. (That's another way of saying this code isn't very good, doesn't work very well, and shouldn't be taken very seriously).

```
/***********************************
 * Pilot World Time Application - take one
 * File - WorldTime.c
 ***********************************/
#include <Pilot.h>       // system toolbox headers
#include "WrldTimeRsc.h" // application resource defines
#include "WrldTime.h"    // application constants, etc.
#include <math.h>        // temp include for square root

/***********************************
 * Global defines for this module
 ***********************************/
#define WrldTimeAppType 'WTim'    // application type
#define WrldTimeDBType  'Data'    // app database type
#define WrldTimeDBName  "WrldTimeDB" // database name

struct cityinfo {
  unsigned char zone, Xloc, Yloc;
  char name[22]; };
struct cityinfo cityrec;

/***********************************
 * Global variables for this module
 ***********************************/
static MenuBarPtr CurrentMenu = NULL;
static Word      CurrentView;
static DmOpenRef WrldTimeDB;
static Word      CurrentRecord;
static VoidPtr   CityDataPtr;
static short     CityCount = 0;
static VoidHand  WrldTimeRec;
static FieldPtr  CurrentFld;

/***********************************
 * Prototypes for internal functions
 ***********************************/
```

```
static void    StartApplication(void);
static Boolean  EditFormHandleEvent(EventPtr event);
static Boolean  MainFormHandleEvent(EventPtr event);
static Boolean  ApplicationHandleEvent (EventPtr event);
static void    EventLoop(void);
static void    CreateRecord (short zone, short Xloc,
                 short Yloc, char * name);
static void    MarkMapLocation (int Xloc, int Yloc);
static void    DisplayLocation (int Xloc, int Yloc);
static void    GetCityName (short Xloc, short Yloc);

/************************************
 * CreateRecord
 * create a database record
 ************************************/
static void CreateRecord (short zone, short Xloc,
  short Yloc, char * name) {
  UInt    index;

/*  load the zone and coordinate information */

  cityrec.zone = zone;
  cityrec.Xloc = Xloc;
  cityrec.Yloc = Yloc;
  StrCopy (cityrec.name, name);

/* create the new record and write it out */

  index = DmNumRecords (WrldTimeDB);
  WrldTimeRec = DmNewRecord (WrldTimeDB,
    &index, StrLen (cityrec.name) + 4);
  CityDataPtr = MemHandleLock (WrldTimeRec);
  DmWrite (CityDataPtr, 0, &cityrec,
    strlen(cityrec.name) + 4);
  MemHandleUnlock (WrldTimeRec);
  DmReleaseRecord (WrldTimeDB, index, true);
}

/************************************
 * StartApplication
 ************************************/
static void StartApplication(void) {
  Word    error;
  UInt    mode;

/*  open the cities database if it's there */

  mode = dmModeReadWrite;
  WrldTimeDB = DmOpenDatabaseByTypeCreator(
    WrldTimeDBType, WrldTimeAppType, mode);

/*  if not, create it */

  if (! WrldTimeDB) {
    error = DmCreateDatabase(0, WrldTimeDBName,
      WrldTimeAppType, WrldTimeDBType, false);
    ErrFatalDisplayIf(error,"Couldn't create new DBMS.");

    WrldTimeDB = DmOpenDatabaseByTypeCreator(
      WrldTimeDBType, WrldTimeAppType, mode);

/*  create and load a test subset of the location data */

    CreateRecord (4,24,23,"San Francisco");
    CreateRecord (5,29,22,"Denver");
    CreateRecord (6,33,27,"Dallas");
    CreateRecord (6,39,21,"Chicago");
    CreateRecord (7,45,21,"New York");
  }
/*  init DB rec count and start main form processing */

  CityCount = DmNumRecords (WrldTimeDB);
  CurrentView = mainForm;
  FrmGotoForm(CurrentView);
}

/************************************
 * GetCityName
 * find out which city is closest to the
 * user's pen tap location.
 ************************************/
static void GetCityName (short Xloc, short Yloc) {

  Int winner, WinDist = 999, CurrentDist = 0;
  Int Xdist, Ydist, DBindex, WinX = 0;

/*  loop thru database looking for the closest city */

  for (DBindex = 0; DBindex < CityCount; DBindex++) {

    WrldTimeRec = DmQueryRecord (WrldTimeDB, DBindex);
    if (WrldTimeRec != NULL) {

/*  get data for next record */

      CityDataPtr = MemHandleLock (WrldTimeRec);
      StrCopy ( (char *) &cityrec, CityDataPtr);
      MemHandleUnlock (WrldTimeRec);

/*  compute the distance from the pen tap */

      Xdist = cityrec.Xloc - Xloc;
      Ydist = cityrec.Yloc - Yloc + MapVStart;
      CurrentDist = sqrtf ((Xdist*Xdist)+(Ydist*Ydist));

/*  save new values if it's closer to this city */

      if (CurrentDist < WinDist) {
        winner = DBindex; WinDist = CurrentDist;
  } } }

/* retrieve the winner for displaying the name */

  WrldTimeRec = DmQueryRecord (WrldTimeDB, winner);
  CityDataPtr = MemHandleLock (WrldTimeRec);
  StrCopy ( (char *) &cityrec, CityDataPtr);
  MemHandleUnlock (WrldTimeRec);
}

/************************************
 * DisplayLocation
 * display the closest city's name in the
 * currently selected location field.
 ************************************/
static void DisplayLocation (int Xloc, int Yloc) {
  VoidHand NameHand;
  VoidPtr NamePtr;

/* check for fld's text handle. if not there, make one */

  NameHand = FldGetTextHandle (CurrentFld);
  if (NameHand == NULL)
    NameHand = MemHandleNew (22);

/* put the new city name's text in the field */

  GetCityName (Xloc, Yloc);
  NamePtr = MemHandleLock (NameHand);
  NamePtr = StrCopy (NamePtr, cityrec.name);
  FldSetText (CurrentFld, NameHand, 0, 22);
  MemHandleUnlock (NameHand);

/* Draw the new value and disable the insertion point */

  FldDrawField (CurrentFld);
  InsPtEnable (false);
}

/*************************************
 * MarkMapLocation
 * draw an asterisk at the location the user just
 * tapped, wait for a bit, and then unmark it.
 *************************************/
static void MarkMapLocation (int Xloc, int Yloc) {
  float        j;
  int          i;
  WordPtr      error;
  CharPtr      Asterisk = "*";
  WinHandle    SaveRegion;
  RectanglePtr source;

/* preserve the bitmap around the location to mark */

  source->topLeft.x = Xloc - 10;
  source->topLeft.y = Yloc - 10;
  source->extent.x  = Xloc + 10;
  source->extent.y  = Yloc + 10;
  SaveRegion = WinSaveBits (source, error);

/* draw an asterisk at the mark point. wait. */

  WinDrawChars (Asterisk, 1,Xloc, Yloc);
  for (i = 1; i < 20000; ++i) {j = i * I;};

/* restore the original bitmap */

  WinRestoreBits (SaveRegion, Xloc - 10, Yloc - 10);
}

/*************************************
 * MainFormHandleEvent
 *************************************/
static Boolean MainFormHandleEvent(EventPtr event) {
  Boolean   handled = false;

  switch (event->eType) {

/*  check for a form open event */

    case frmOpenEvent:
      FrmDrawForm (FrmGetActiveForm());
      handled = true;
      break;

/*  check for a pen tap within the world map */

    case penDownEvent:
      if ((event->screenY >= MapVStart) && (
        event->screenY <= MapVStop)) {
        MarkMapLocation (event->screenX, event->screenY);
        handled = true;
        if (CurrentFld != NULL)
          DisplayLocation(event->screenX,event->screenY);
      }
      break;
  } return(handled);
}

/*************************************
 * ApplicationHandleEvent
 *************************************/
static Boolean ApplicationHandleEvent (EventPtr event)
{
  FormPtr frm;
  Word    formId;
  Boolean handled = false;

  switch (event->eType) {

/*  check for a form load event */

    case frmLoadEvent:
      formId = event->data.frmLoad.formID;
      frm = FrmInitForm(formId);
      FrmSetActiveForm(frm);
      FrmSetEventHandler(frm, MainFormHandleEvent);
      handled = true;
      break;

/*  check for a field - switch the current field */

    case fldEnterEvent:
      CurrentFld = event->data.fldEnter.pField;
      handled = true;
      break;
  }
  return handled;
}

/*************************************
 * EventLoop
 *************************************/
static void EventLoop(void) {

  EventType event;
  Word      error;

  do {
    EvtGetEvent(&event, evtWaitForever);
    if (! SysHandleEvent(&event))
      if (! MenuHandleEvent(CurrentMenu, &event, &error))
        if (! ApplicationHandleEvent(&event))
          FrmDispatchEvent(&event);
  }
  while (event.eType != appStopEvent);
}

/*************************************
 * PilotMain
 *************************************/
DWord PilotMain(Word cmd, Ptr cmdPBP, Word launchFlags) {
  StartApplication();
  EventLoop();
  DmCloseDatabase (WrldTimeDB);
  return 0;
}
```

This example uses a database which gets initialized in the `StartApplication` routine with a small subset of the final data. Each record contains a city name, a time zone, and X and Y coordinates for that city. I got the coordinates by clicking in a graphics program on map locations on the original, large-scale map I downloaded from CompuServe, and then manually mapping the coordinates into the 160 x 120 coordinate space of the Pilot map graphic. If I had ended up using this application interface, I would have eventually included the coordinate mapping as part of the application start-up code.

Once the data is all set up, the user proceeds as I described earlier. When they tap on the world map, the `penDownEvent` is intercepted in `MainFormHandleEvent`. The program marks the map at the tap location. Then, if a field has been selected by the user, the program determines the closest city name and displays it in the current field. The distance to each city is determined by calculating the hypotenuse of the triangle formed by the city location and the tap location, and the horizontal and vertical axes.

### *Pros and Cons*

Once I actually got this code running, it became immediately obvious the user interface doesn't work for one basic reason – the screen is too small and the resolution is not fine enough. As a result, a pen tap doesn't let you get close to a target location with any accuracy, especially in the Pilot simulator where you use a mouse instead of a pen. If you tap in the Rocky Mountains, for instance, you're equally likely to get Denver, Dallas, San Francisco, or Los Angeles. Also, if your pen calibration is not perfect, which is likely, then the accuracy is worse.

There are two other reasons to consider alternate implementations:

- The triangulation algorithm uses a square root function and floating point numbers, which are not available as part of the Pilot ROM-based libraries. In a case like that, the linker takes floating point functions from the standard desktop libraries. Not only might they be error-prone on the Pilot, but they could use substantial amounts of memory.
- There's a basic problem with handling the location data. In order to use a database, there should really be a separate program that just creates the database. Otherwise, with the static allocation approach I use, the data is both embedded in the program and the Pilot's dynamic data store, using twice as much memory it needs. The database functions are useful for searching and if you want to let the user add their own locations (something I was considering). Having two programs for a simple function like calculating world time zones strikes me as very awkward.

The only interesting part of this sample is the database code. I was surprised at how easy it was to set create, load, and search a database. I didn't test this code on an actual Pilot device, only with the simulator, so I don't know how well it actually works in real life (more about that later).

### *Lessons Learned*

With all that in mind, I decided on a new approach. First, the next revision would be based on a scrolling text list of locations instead of a world map. The Pilot has two list-like structures: lists and tables. At first glance, tables have a fairly complex API, so I decided to use lists instead.

Second, I decided to use static data storage instead of a database. I could easily search through string arrays and look up locations – I didn't need the Pilot's database routines just for that. Also, I could avoid having two programs (one for initialization and one for daily use) or double data storage in one program. The only restriction with static storage is that I can't easily set up the application to let users enter their own locations without a database. The logical decision is to make sure that I have a very comprehensive set of locations, of all types, so that no one ever needs to add any new sites. (As it turns out, everyone wants more locations).

## World Time, Take Two

I started off phase two of this project looking for more complete time zone data. After scouring CompuServe and trying to search the web (for you Mac users, Open Transport 1.1 isn't all it's cracked up to be), I resigned myself to some old-fashioned research – I went to the library. There, in a matter of minutes, I found a fairly complete reference to world times zones.

Unfortunately, that complete reference included lots of details. It turns out that:

- Every country, and certain time zones and locations within countries have different starting and ending dates for daylight savings time.
- The northern and southern hemispheres have daylight savings time during the summer and the winter respectively.
- There are certain countries that have their time set forward a fraction of an hour from the rest of the world (that explained the "+30" markings I found on the original maps).

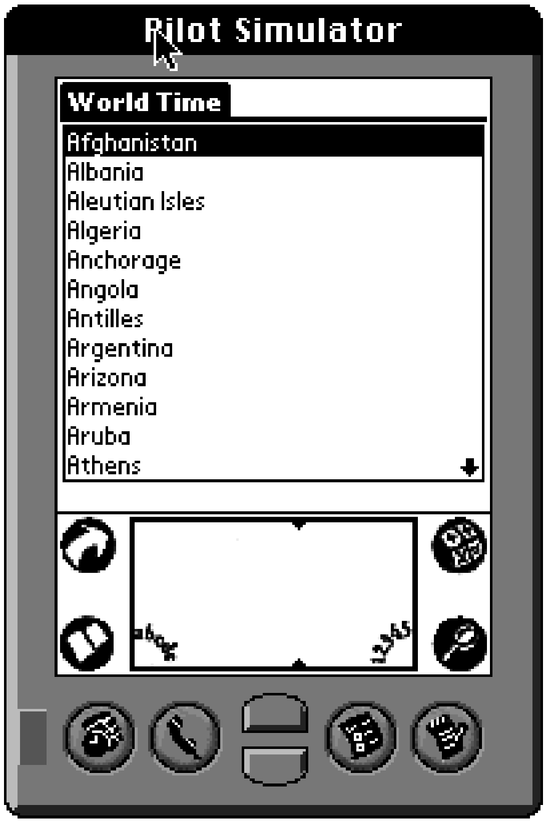
Armed with this knowledge, I made some simplifying assumptions. First, I decided to set daylight savings time in the northern hemisphere to run from May 1 through September 30 for all countries, and October 1 through April 30 for all southern-hemisphere countries. These dates are close to the actual dates used by all countries that indulge in daylight savings time. This simplification saves storing at least four bytes of extra data per location, saving at least 2K, and guarantees that the calculated times are inaccurate by no more than one hour for just a few days close to the transition dates.

I also decided that I would only use countries, not cities, for locations. I later changed my mind and added in the 50 or so original cities I had collected.

Finally, I also rounded off all offsets for countries whose normal times don't fall on an even number of hours from Greenwich Mean Time to 30-minute offsets. This only applies to Nepal and Guyana. Both are 45 minutes ahead of other countries in their time zones.

### *The Interface*

Figure 2 shows the interface for World Time, take two. It's a simple scrolling list. Once you set up the list for the Pilot ROM routines, scrolling, item selection, and most of the other boring housekeeping details are taken care of for you. After using the scrolling list of the program for a few hours, I decided to add some additional navigation aids: if you enter a letter in the Graffiti pad, the list scrolls to the first item starting with that letter. Also, I enabled the scroll buttons so that they work just like the scroll arrows.



Afghanistan
Albania
Aleutian Isles
Algeria
Anchorage
Angola
Antilles
Argentina
Arizona
Armenia
Aruba
Athens

Figure 2 - World Time with a scrolling list.

Figure 3 shows what happens when you tap an item in the list – the location is treated as the home location, and is assigned the current date and time as set by the user with the Pilot's Prefs program. The date and time for all the other locations are then calculated relative to the home location. Daylight savings time is also factored in. Those locations currently operating under daylight savings time are highlighted with an asterisk. As the user scrolls through the list, using the Graffiti pad or the scroll arrows or buttons, the home location stays the same until they tap another list item.

### *The Locations Data*

The data about each of the locations is stored in a separate source code file that is included in the main World Time program. Here's its structure:

```
struct LocInfo {
  char  name[18];
  short zone;
  short DST;
  } LocationInfo [248] = {

"Afghanistan ",46,0,
"Albania ",13,6,
  .
  .
  .
"Zambia",14,0,
"Zimbabwe",14,0,
"Zurich",13,60};
```

Each location includes its name, a time zone indicator ranging from 1 - 54, and a daylight savings time indicator. The time zone indicator is normally a value between 1 and 24, but if a location is shifted ahead 30 minutes, the difference is added to the time zone. For instance, most of India is 30 minutes ahead of the other countries in its time zone. Consequently, all locations in India have a time zone of 47. World Time interprets this value as time zone 17 plus 30 minutes.

Likewise, the daylight savings time value has some embedded coding. A zero value indicates that the location never uses daylight savings time. A value of 60 (indicating a 60 minute shift) indicates locations that run under daylight savings from May through September. A value of -60 indicates locations that run under daylight savings between October and April.

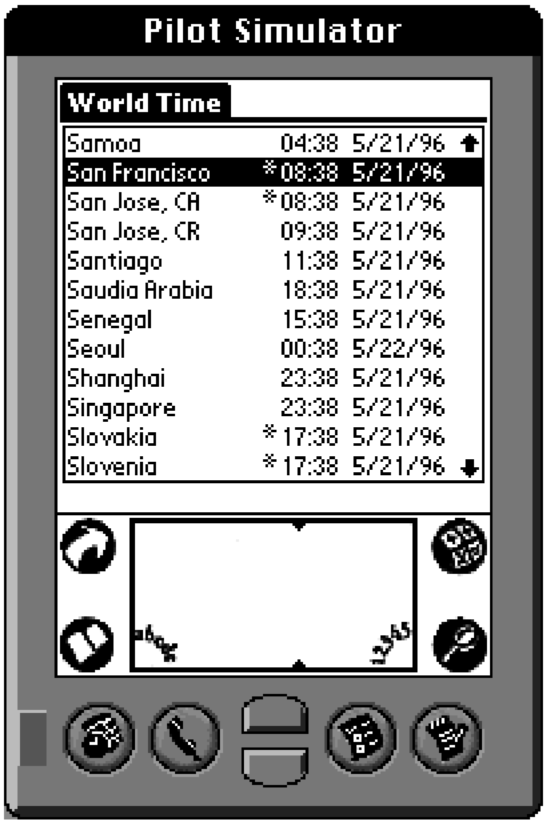
For data fetching reasons on the Pilot, with the data structure I use, each character string must be padded out to an even number of characters. That's why there's a blank at the end of "Albania " but not "Zambia".

### *The List Structure*

One of the most important parts of World Time is the list-handling code. Lists are interesting resources. You start off by specifying the list parameters – the location, the font, and the number of visible items (the list height) – using a "tLST" resource. You then point to the list resource from a "tFRM" resource, your primary container view for the list. You initialize the list, set up an optional callback function for drawing a single line of the list on the screen, and then the Pilot OS takes care of the basic list handling. Every time the list is updated, the callback function is called to draw individual lines.

To initialize a list you need to create an array of pointers to the list's strings (see Figure 4). Fortunately, there's a Pilot ROM routine to set up this structure from a packed list of null-terminated strings

Here's the code for setting up the list, plus the World Time's main routine and an initialization function. I leave out all the declarations, include files, and other miscellany. For the full details, you should look at the complete project on the source code disk for this issue of *PDA Developers*:



```
Samoa             04:38 5/21/96
San Francisco    *08:38 5/21/96
San Jose, CA     *08:38 5/21/96
San Jose, CR      09:38 5/21/96
Santiago          11:38 5/21/96
Saudia Arabia     18:38 5/21/96
Senegal           15:38 5/21/96
Seoul             00:38 5/22/96
Shanghai          23:38 5/21/96
Singapore         23:38 5/21/96
Slovakia         *17:38 5/21/96
Slovenia         *17:38 5/21/96
```

Figure 3 - World Time in action.

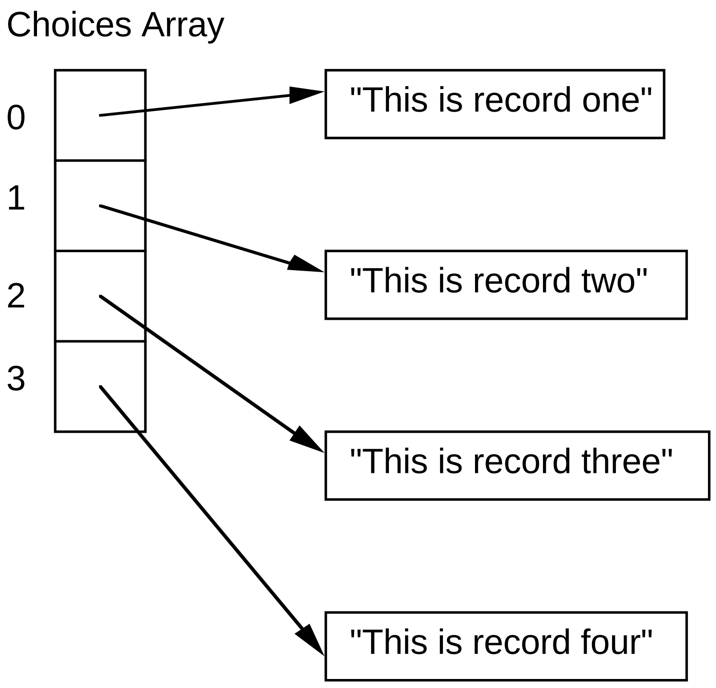


Figure 4 - A List data structure.

```
/************************************
* MainFormInit
* create the locations list string array
 ************************************/
static void MainFormInit (void)
{
  CharPtr errmsg;
  Int   choicesOffset = 0, half;

/*  get a pointer to the list resource */

  frm = FrmGetActiveForm ();
  itemIndex = FrmGetObjectIndex (frm, LocationsList);
  lst = FrmGetObjectPtr (frm, itemIndex);

/*  allocate dynamic memory variables */

  MemAllocate ();

/*  add the location names to the list setup string*/

  for (itemIndex=0;itemIndex<numLocations;itemIndex++) {
    textLen = StrLen (LocationInfo [itemIndex].name);

/*  error check for odd-length string */

#if EMULATION_LEVEL == EMULATION_MAC
    half = textLen / 2;
    if ((half * 2) != textLen) {
      errmsg = StrCat ("Odd-length string - ",
        LocationInfo [itemIndex].name);
      ErrFatalDisplayIf (1, errmsg);
    }
#endif

/* add new location to choices list, NULL terminated */

    error = MemPtrResize (
      choicesPtr, textLen + 1 + choicesOffset);
    ErrFatalDisplayIf (error,
      "Couldn't grow locations list.");
    for (i = 0; i < textLen; i++)
      choicesPtr [choicesOffset + i] =
        LocationInfo [itemIndex].name [i];
    choicesOffset += textLen;
    choicesPtr [choicesOffset++] = 0;
  }

/*  list complete - convert it to a list data structure
  and set the list drawing function callback */

  choicesPtrsHandle = SysFormPointerArrayToStrings
    (choicesPtr, numLocations);
  LstSetListChoices (lst,
    MemHandleLock (choicesPtrsHandle), numLocations);
  LstSetDrawFunction (lst, DrawListLine);
}

/************************************
 * MemAllocate and MemFree
 * Routines to handle most dynamic memory globals.
 ************************************/
static void MemAllocate (void){

  strHandle = MemHandleNew (10);
  strPtr = MemHandleLock (strHandle);
  choicesHandle = MemHandleNew (sizeof (char));
  choicesPtr = MemHandleLock (choicesHandle);
  *choicesPtr = 0;
}
```

```
static void MemFree () {
  error = MemHandleFree (strHandle);
  error = MemHandleFree (choicesHandle);
  error = MemHandleFree (choicesPtrsHandle);
}

/************************************
 * MainFormHandleEvent
 * handle events for the main (and only) form.
 ************************************/
static Boolean MainFormHandleEvent(EventPtr event) {

  Boolean   handled = false;
  char      key;
  FormPtr   tempfrm;

/* form open - initialize the list */

  switch (event->eType) {
    case frmOpenEvent:
      MainFormInit ();
      FrmDrawForm (frm);
      handled = true;
      break;

/* a list item has been tapped - set current location */

    case lstSelectEvent:
      SetHereZone ();
      handled = true;
      break;

/* keystroke - move to matching location if alphabetic */

    case keyDownEvent:
      key = event->data.keyDown.chr;
      handled = ProcessKeystroke (key);
      break;

/*  menu event - show the About box */

    case menuEvent:
      MenuEraseStatus (currentMenu);
      FrmEraseForm (frm);
      tempfrm = FrmInitForm(AboutForm);
      FrmDoDialog (tempfrm);
      FrmDeleteForm (tempfrm);
      FrmDrawForm (frm);
      handled = true;
  }
  return (handled);
}

/************************************
 * ApplicationHandleEvent
 * handle application events.
 ************************************/
static Boolean ApplicationHandleEvent (EventPtr event) {

  Word    formId;
  Boolean handled = false;

/*  check for a form load event */

  if (event->eType == frmLoadEvent) {
    formId = event->data.frmLoad.formID;
    frm = FrmInitForm (formId);
    FrmSetActiveForm (frm);
/*  if so, load the appropriate form event handler */

    switch (formId){
      case mainForm:
        FrmSetEventHandler (frm, MainFormHandleEvent);
        break;
    }
    handled = true;
  }
  return handled;
}

/***********************************
 * EventLoop
 * handle program events.
 ***********************************/
static void EventLoop(void)
{
  EventType event;

  do {
    EvtGetEvent (&event, evtWaitForever);
    if (! SysHandleEvent (&event))
      if (!MenuHandleEvent (currentMenu, &event, &error))
        if (! ApplicationHandleEvent (&event))
          FrmDispatchEvent (&event);
  }
  while (event.eType != appStopEvent);
}

/***********************************
 * PilotMain
 ***********************************/
DWord PilotMain(Word cmd, Ptr cmdPBP, Word launchFlags) {

  if (cmd == sysAppLaunchCmdNormalLaunch) {
    FrmGotoForm (mainForm);
    currentMenu = MenuInit (mainMenu);
    EventLoop ();
    MemFree ();
  }
  return (0);
}
```

If you read the "Pilot Programming Primer" article I referenced earlier, you should recognize a lot of this code. `PilotMain`, `EventLoop`, `ApplicationHandleEvent`, and `MainFormHandleEvent` are very similar to routines with the same name in the Hello World program in that article. `PilotMain` controls the application flow at a very high level. `EventLoop` is a generic event-handling loop that is found in an almost identical format in all Pilot programs. `ApplicationHandleEvent` loads the main form and sets its handler (if there are multiple forms their handlers are all set here). And finally, `MainFormHandleEvent` intercepts keystrokes, pen taps, and menu events for the main form.

The list initialization is done in `MainFormInit`, which is called when a `frmOpenEvent` event is intercepted for the main (and only) form. First, the list dynamic memory variables are allocated. Then the routine loops through the list of locations, adding each one to the packed names list, resizing the packed list for each new addition. Finally, once the packed list is built, the program calls `SysFormPointerArrayToStrings` to create the pointers list, `LstSetListChoices` to initialize the pointer to the list in the list data structure, and `LstSetDrawFunction` to initialize the callback function for drawing each list line.

The code surrounded by

```
#if EMULATION_LEVEL == EMULATION_MAC
#endif
```

is designed to catch any odd-length location names that make it through the source code editing process. It generates an exception, only while running under the Pilot simulator, and is not compiled for the final device upload. More about this later.

## *Setting the Primary Location*

When the program first starts, the display shown in Figure 2 appears. Once the user selects a primary location, the list shown in Figure 3 appears. Both of these displays are created by the list drawing callback routine.

Here's the function that sets the home location. It's called from `MainFormHandleEvent` in response to a `lstSelectEvent` which indicates that the user has tapped a list item:

```
/***********************************
 * SetHereZone
 * there's been a pen tap on the list.
 * set the primary time zone to that location.
 * all subsequent times/dates are calculated
 * relative to that location.
 ***********************************/
static void SetHereZone (void) {

  currentRecord = LstGetSelection (lst);
  hereZone = LocationInfo [currentRecord].zone;
  hereTime = TimGetSeconds ();

/* convert the current time to discrete values and
  determine DST period. */

  TimSecondsToDateTime (hereTime, &dateTime);
  DSTseason = spring;
  if ((dateTime.month >=10) || (dateTime.month <= 3))
    DSTseason = fall;
  hereDST = LocationInfo [currentRecord].DST;

/* check for a partial hour adjustment */

  if (hereZone > 24) {
    hereZone = hereZone - 30;
    hereTime = hereTime - 1800;
  }

/* force a redraw of the list to update all the other
locations */

  LstEraseList (lst);
  LstSetSelection (lst, currentRecord);
  LstDrawList (lst);
}
```

The primary location is called the Here zone. All other zones, as calculated in the list callback function, are There zones. Setting the Here zone is straightforward. First, `SetHereZone` gets the tapped list location and the current date and time, which are converted to an appropriate data type. Then the routine determines the daylight savings status of the new primary location, and checks for any partial hour adjustment. Finally, it erases the current list, sets the list selection to the new primary location, and redraws the list. `LstDrawList` sets in motion a drawing callback for every list item that is now visible.

## *Drawing Each Line*

The callback function is responsible for drawing each line of the list display. Although this happens in response to setting a new primary location, scrolling and entering a letter using the Graffiti pad also cause the list view to change. All list-change events eventually call the drawing callback routine.

My first callback routine merely concatenated the calculated date and time for each line onto the location name and displayed the result. The first time I got that code running, it became clear that approach wouldn't work – Pilot fonts are all proportionally spaced, making it impossible to get the date and time columns to line up vertically. Instead, I switched to drawing characters directly on the screen at specific horizontal locations. The vertical locations are calculated by determining the distance of the current line from the list's first visible line, multiplying it by the height of the current font, and adding an offset to compensate for the start of the list from the top of the device display. To simplify matters, the Pilot coordinate drawing system is all relative to the top left-hand corner of the screen. There aren't separate coordinate systems for each view.

Here's the routine to draw a string at a specific screen location:

```
/************************************
 * DrawListChars
 * draw a character string at the current list location.
************************************/
static void DrawListChars (CharPtr string, int location,
  short Xloc) {

  Yloc = location - lst->topItem;
  Yloc = (Yloc * fontHeight) + lstTop;
  textLen = StrLen (string);
  WinDrawChars (string, textLen, Xloc, Yloc);
}
```

`DrawListLine`, the callback procedure, uses `DrawListChars` to draw all the components of each list item. `DSTAdjust` handles the various daylight savings time adjustments that need to be made.

```
/************************************
 * DrawListLine
 * draw one full line for the current list item. if a
 * current time zone is selected,include the date
 * and time, relative to the selected zone.
 ************************************/

static void DrawListLine (Word location,
  RectanglePtr bounds, CharPtr *itemsText) {

  short ThereZone;
  short ZoneDiff, AbsZoneDiff;
  ULong ThereTime, adjust;

/*  draw the current city name, no more if virgin list */

  fontHeight = FntLineHeight ();
  DrawListChars (LocationInfo[location].name,location,3);
  if (currentRecord < 0) return;

/*  current time zone is active. Draw time & date too */

  ThereZone = LocationInfo [location].zone;
  currentDST = LocationInfo [location].DST;
  if (ThereZone > 24) ThereZone = ThereZone - 30;
  ZoneDiff = ThereZone - hereZone;
  AbsZoneDiff = ZoneDiff;

/* get absolute value the old-fashioned way */
/* to avoid linking in a C desktop C library */
  if (ZoneDiff < 0) AbsZoneDiff = - ZoneDiff;
  adjust = 0;

/* adjust time using addition - bug in multiplication */

  for (i = 0; i < AbsZoneDiff; i++)
    adjust = adjust + hoursInSeconds;

/* check for 30-minute adjustment (zone > 24) */

  if ( LocationInfo [location].zone > 24)
    adjust = adjust + hoursInSeconds/2;
  if (ZoneDiff > 0)
    ThereTime = hereTime + adjust;
  else
    ThereTime = hereTime - adjust;

//  adjust for daylight savings time for all
//  locations except Home */

  if (currentRecord != location)
    ThereTime = DSTAdjust (ThereTime, location);

/* convert time/date to a useful format */

  TimSecondsToDateTime (ThereTime, &dateTime);

//  get date string & zero-fill on the left if needed

  TimeToAscii (dateTime.hour, dateTime.minute,
    tfColon24h, strPtr);
  if (StrLen (strPtr) == 4) {
    for (i = 5; i >= 1 ; i–)
      strPtr [i] = strPtr [i - 1];
    strPtr [0] = blankChar;
  }

/* draw DST indicator and the current time */

  if ((currentDST == hereDST) && (hereDST != 0))
    DrawListChars ("*", location, DSTLoc);
  if ((hereDST == noDST) && (currentDST != noDST))
    DrawListChars ("*", location, DSTLoc);
  DrawListChars (strPtr, location, timeLoc);

/* get current date and draw it too */

  DateToAscii (dateTime.month, dateTime.day,
    dateTime.year, dfMDYWithSlashes, strPtr);
  DrawListChars (strPtr, location, dateLoc);
}

/************************************
 * DSTAdjust - adjust for daylight savings time.
 * simplified DST adjustment. assume that May - Sept and
 * Oct - March are the two DST periods.
 ************************************/
static ULong DSTAdjust (ULong Time, Word location) {

//  if primary zone is same DST setting as the
//  current location, do nothing

  if (hereDST == currentDST) return (Time);

/*  if primary zone has no DST, bump everyone in the
  active DST season up 1 hour */
  if (hereDST == noDST)
    if (currentDST == DSTseason)
        return (Time + hoursInSeconds);

//  if primary zone is the active DST, bump everyone back
//  1 hour except those with the same DST season

  if (hereDST == DSTseason)
    if (currentDST != DSTseason)
      return (Time - hoursInSeconds);

//  if primary zone has DST but it's wrong season, bump
//  opposite DST season up in fall, back in spring

  if ((hereDST == spring) && (currentDST == fall))
    if (currentDST == fall)
      return (Time + hoursInSeconds);
  if ((hereDST == fall) && (currentDST == spring))
    if (currentDST == spring)
      return (Time - hoursInSeconds);

  return (Time);
}
```

## *Navigation*

The last bit of code of any consequence is the keystroke and scroll-button processing. It's handled by `ProcessKeystroke`, which is called from the main form event handler when there's a keystroke:

```
/************************************
 * ProcessKeystroke
 * check for an alphabetic keystroke. if so,
 * set the list to point to the first matching item.
 ************************************/
static Boolean ProcessKeystroke (char key) {
  Boolean handled = false;

  if (key != NULL) {

/*  map lower case into upper case */

    if (key >= lowerA && key <= lowerZ)
      key = key - lowerUpperDiff;

/*  search location array for match & set list pointer */

    if (key >= upperA && key <= upperZ) {
      for (i = 0; LocationInfo [i].name [0] < key; i++);
      LstEraseList (lst);
      LstSetTopItem (lst, i);
      LstDrawList (lst);
      handled = true;
    }

/* check for scroll up button */

    if (key == scrollUp) {
      i = lst->topItem;
      i = i - visibleItems;
      if (i < 0) i = 0;
      LstEraseList (lst);
      LstMakeItemVisible (lst, i);
      LstDrawList (lst);
      handled = true;
    }

/* check for scroll down button */

    if (key == scrollDown) {
      i = lst->topItem;
      i = i + visibleItems;
      if (i > (numLocations - 1)) i = numLocations - 1;
      LstEraseList (lst);
      LstMakeItemVisible (lst, i);
      LstDrawList (lst);
      handled = true;
  } }
  return (handled);
}
```

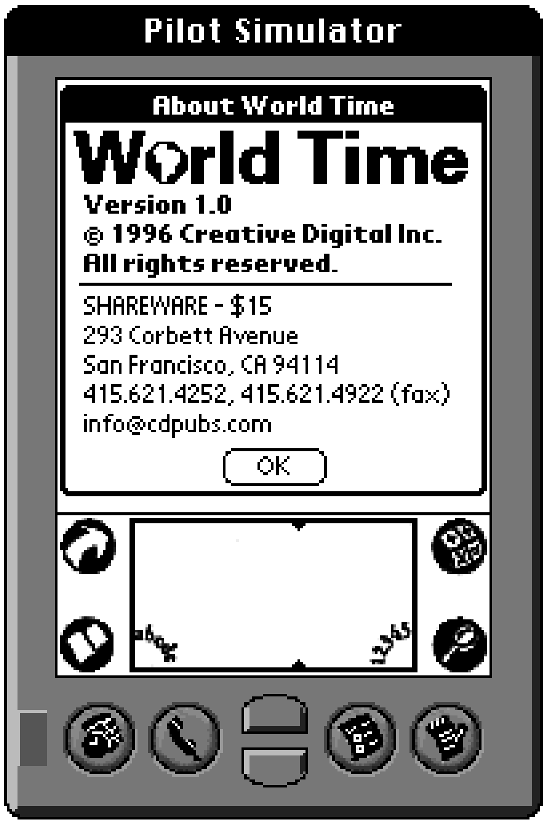
Graffiti keystrokes arrive as characters. This function merely maps them into the uppercase alphabet, finds the appropriate list item (if it's an alphabetic key), and repositions the list at the new location. Note the simple code for repositioning the list. It gives you an idea of how easy it is to use a Pilot list:

```
  LstEraseList (lst);
  LstSetTopItem (lst, i);
  LstDrawList (lst);
```

The list is also redrawn in response to the scroll buttons. The only difference is that the function has to figure out the new list top before setting it.

## *The About Box*

Most of the Pilot built-in applications have an About box that is displayed when the user selects the About command on the Options menu. I decided to add an About Box to World Time. You've already seen the code; it's in `MainFormHandleEvent` in the `switch` statement:



Version 1.0
© 1996 Creative Digital Inc.
All rights reserved.
SHAREWARE - $15
293 Corbett Avenue
San Francisco, CA 94114
415.621.4252, 415.621.4922 (fax)
info@cdpubs.com

Figure 4 - World Time's About box.

```
  case menuEvent:
    MenuEraseStatus (currentMenu);
    FrmEraseForm (frm);
    tempfrm = FrmInitForm(AboutForm);
    FrmDoDialog (tempfrm);
    FrmDeleteForm (tempfrm);
    FrmDrawForm (frm);
    handled = true;
```

Since there's only one menu command, I assume it's the About command. I erase the menu and the main form, display the About dialog using `FrmDoDialog` (which doesn't return until a dialog control is pressed), then delete the dialog and redraw the main form.

Figure 4 shows the About dialog. It's simply two 72-dpi graphics, one for the title and one for the horizontal line, plus a set of "tLBL" label resources and a default OK button.

## World Time Meets the Real World

Once I got all this code running under the simulator, it was time to turn it into something that could be run on a real Pilot device. On the surface, this seems very straightforward. First, you create a Rez file that defines the application's resources as required by the Pilot, not the desktop. Here's the World Time Rez file as created from a sample that is included with the Pilot SDK:

```
/**********************************************
 * FileName: WorldTime.r
 * Resource description for generating the
 * World Time resource database
 **********************************************/
//———————————————————————
// Include definitions for basic resources.
//———————————————————————
#include <BuildRules.h>
#include <SystemMgr.rh>

//———————————————————————
// Code and Globals resources
//———————————————————————
// Include the main code resource
include "WorldTime.code" 'CODE' 1 as sysResTAppCode 1;

// Include CODE 0 which has global size information in it
include "WorldTime.code" 'CODE' 0 as sysResTAppCode 0;

// Include DATA 0 which has initialized global data info
include "WorldTime.code" 'DATA' 0 as sysResTAppGData 0;

//——————————————————————————————
// Include the MemoPad UI resources
//——————————————————————————————
#if LANGUAGE==LANGUAGE_ENGLISH
include ":Rsc:WorldTime.rsrc";
include ":Rsc:WorldTimeAbout.rsrc";

#elif LANGUAGE==LANGUAGE_GERMAN
include ":Rsc:German:WorldTime.rsrc";
include ":Rsc:German:WorldTimeAbout.rsrc";

#elif LANGUAGE==LANGUAGE_FRENCH
include ":Rsc:French:WorldTime.rsrc";
include ":Rsc:French:WorldTimeAbout.rsrc";

#else
#error "The compiler variable LANGUAGE must be defined"
#endif
```

```
//——————————————————————————————
// PREF resource. The current version of the OS does not
// use the ionfo in this resource. it should be included
// for future compatibility.
//——————————————————————————————
resource sysResTAppPrefs 0 {
  30,              // priority
  0x1000,            // stack size
  0x1000             // minHeapSpace
  }
```

Every place you see the name "WorldTime" you need to substitute the name of your application. Also, you need to include each of your ResEdit resource files in the `LANGUAGE` sections.

Next, you have to create a Makefile, a set of commands that is used by the MPW shell to compile, link, and set up your application for downloading and debugging on a real Pilot. Here's the World Time Makefile:

```
#######################################################
# Set up paths
#######################################################
LIB_DIR = :::Libraries:PalmOS:
INC_DIR = :::Incs:
SRC_DIR = :Src:
OBJ_DIR = :Obj:
DBG_DIR = :::

#######################################################
# Set up Compiler
#######################################################
# Use Metrowerks' compiler
CC = MWC68K
CPP = MWC68K
LINK = MWLink68K

#######################################################
# Compile Options
# The most likely options you might change are COUNTRY,
# LANGUAGE, and ERROR_CHECK_LEVEL.
#######################################################
C_OPTIONS = ∂
        -d COUNTRY=0  ∂
        -d LANGUAGE=0 ∂
        -d ERROR_CHECK_LEVEL=2 ∂
        -d CMD_LINE_BUILD ∂
        -d EMULATION_LEVEL=0 ∂
        -d ENVIRONMENT=0 ∂
        -d MEMORY_FORCE_LOCK=1 ∂
        -nosyspath ∂
        -i ":Src:" ∂
        -i "{INC_DIR}" ∂
        -i "{INC_DIR}System:" ∂
        -i "{INC_DIR}UI:" ∂
        -i "{INC_DIR}Hardware:" ∂
        -model near ∂
        -intsize 2 ∂
        -maxerrors 3 ∂
        -opt speed -opt global -opt peep ∂
        -mbg on ∂
        -b ∂
        -d PILOT_PRECOMPILED_HEADERS_OFF
LINK_OPTIONS = -single -custom
```

```
########################################################
# Object List
########################################################
OBJECTS = ∂
       "{LIB_DIR}StartupCode.c.o" ∂
       "{OBJ_DIR}WorldTime.c.o"

########################################################
# Compiles
########################################################
"{OBJ_DIR}WorldTime.c.o" ƒ MakeFile
"{SRC_DIR}WorldTime.c"
   {CPP} -o "{OBJ_DIR}WorldTime.c.o" ∂
      "{SRC_DIR}WorldTime.c" ∂
      {C_OPTIONS}

########################################################
# Final Link
########################################################
WorldTime ƒƒ MakeFile {OBJECTS} "{SRC_DIR}WorldTime.r"
  {LINK} {LINK_OPTIONS} -t rsrc -c RSED ∂
    {OBJECTS} ∂
    "{MW68KLibraries}ANSI (2i) C.68K.Lib" ∂
    "{MW68KLibraries}console.stubs.c.o" ∂
    -o WorldTime.code
  Delete -i "WorldTime.prc"
  {CC} -d RESOURCE_COMPILER ∂
      {C_OPTIONS} ∂
      -e ∂
      "{SRC_DIR}WorldTime.r" > WorldTime.i
  PilotRez -backupBit -v 1 -t appl -c CDI1 -it
WorldTime.i -ot "WorldTime"
  Rename -y "WorldTime" "WrldTime.prc"
  Duplicate -y "WrldTime.prc" "{DBG_DIR}"Debugger
```

I'm not a serious MPW-head (and I don't really want to be), so I don't really know what a lot of these commands do. Basically, you need to use the name of your application every place you see the word "World-Time", except for the last two references where I revert to DOS file-naming conventions for compatibility with Windows systems.

There's one other change you also need to make – in the `Pilot-Rez` line, use your own application's creator ID instead of `CDI1`, which is Creative Digital's creator ID. Creator IDs are designed to prevent applications from accidentally messing with each other's data and resources during loading and at runtime. You need to request a unique creator ID from Palm Computing tech support.

To compile and link your application, you start up the MPW shell, set the current directory to point to your applications directory, and chose the **Build** command. The script is run, and if all goes well, your application is created and a copy is put in the Pilot Debugger folder in your Pilot SDK folder.

To download the program, you start up the Debugger and open a serial connection between your cradled Pilot and your Mac. There are currently no options for selecting the port (the Debugger assumes that you're connected to the modem port), just the baud rate. To open the connection, you start the Pilot Prefs application and enter the ".2" shortcut. (You initiate a shortcuts by first entering a tall lower-case cursive "l" with very long beginning and ending strokes, then enter the shortcut name by tapping twice in the left-hand side of the Graffiti pad and entering a "2" on the numeric side of the pad.) The Pilot should beep once, indicating that the connection is open.

Once you establish the connection, go back to the debugger and enter the phrase "import 0 WrldTime.prc" in the Debugger's Console window (not the Debugger window). If you have a connection, the resources are downloaded and you're all set for some real testing. The ".prc" extension is added for compatibility with the Pilot desktop, whose installation file open dialog filters for that extension.

## *Crash-and-Burn City*

When I first loaded one of the earlier version of World Time onto my Pilot, I was greeted with some immediate serious crash-and-burn exceptions. I had to hard reset the device to get any response from it. I still had lots of work to do on the program, so I sent off an e-mail to Palm's tech support, pretended nothing had happened, and went back to work on the desktop.

In a subsequent conversation with tech support, I learned there are several things that you can do on the desktop that you definitely can't do on the Pilot. Here's an abbreviated list:

- Don't use any library routines except those that are explicitly defined for the Pilot. That rules out many standard C runtime library calls that we take for granted, including memory allocations, string operations, and arithmetic functions. There are Pilot equivalents for many, but not all, standard library routines.
- Don't write to storage RAM unless you use the database routine `DMWrite`. The simulator doesn't have write protection, but the Pilot does.
- Don't access 16 or 32-bit values at odd addresses. This generates a bus error on the Pilot but not on the simulator. This can happen when working with packed data structures.
- Don't use intra-application jumps of more than 32K. You will get linking errors when you try to create your Pilot-bound application with MPW.
- Applications running on a Pilot only have a 2K stack. Avoid storing a lot of data on the stack.

Pilot errors are generally much more destructive than simulator errors. Expect everything to be destroyed all the time.

After learning about these crash-and-burn problems, I went back and checked my code more carefully. I'm not a great code reader – I'm more inclined to let my machines, not my brain, do my debugging. With these gotchas in mind, and by liberally sprinkling system sounds throughout the code, creating an audio audit trail, I was eventually able to track down my errors. There was one serious problem: some location names were an odd number of characters, resulting in time zone data being fetched from odd locations on the Pilot. Once I cleared that up, most of my problems went away. I could have avoided this problem by putting the time zone information first in the location structure. By the time I realized this, however, I didn't want to go back and re-edit 250 locations, introducing new editing errors that I would have to track down. Instead, I just added blanks where needed and included the error-checking code for odd-length strings.

One other note about errors: while testing my daylight savings time code, I kept coming up with ridiculous time zone calculations. For a long time, knowing the level of my programming skills, I figured it was my code. After several hours of careful checking, however, I managed to track down a compiler error in the `Ulong` multiplication operator. Since then I've also found at least one other bug having to do with static string declarations. These types of errors can be real hair pullers. The moral of the story – don't assume that the compilers are perfect.

## *Simulator Testing*

Getting your program to run on a real Pilot is no guarantee of success, as I found out. The other thing you need to do is run your program under the simulator and start up a Gremlin. Gremlins generate semi-random system events that test many difference facets of your application. After I successfully got World Time running on my Pilot, I went and tried a Gremlin at the urging of Palm's tech support. Within five seconds World Time was seriously dead in the water.

Again, the problem was essentially one item – action codes. Action codes are global system events that are passed to your `PilotMain` routine, whether you like it or not. They can arrive when your application is active or inactive. They are used primarily for interapplication communication, communication between the operating system and your application, and communication between your application and the Pilot desktop. Palm suggests you set up your application to respond to as many action codes as possible.

There are two very basic action codes that almost all applications should support in order to act like a real Pilot application. The first is `sysAppLaunchCmdFind`, which tells your application to go find a particular text string in your database. This is used for global searching. The second is `sysAppLaunchCmdGoTo`. This ones tells your application to launch itself and go to the record (with the appropriate view) that contains a particular string. This is used to implement the "Go to" command on the global search dialog.

There's a whole section in the beginning of the Pilot SDK documentation that explains actions codes, and tell you how to offensively and defensively deal with them. My problem was that I didn't even set up my application to ignore the actions codes, let alone respond to them. The simplest thing to do is bracket your PilotMain routine with a defense barrier:

```
DWord PilotMain (Word cmd, Ptr cmdPBP, Word launchFlags)
{
  if (cmd == sysAppLaunchCmdNormalLaunch) {
    FrmGotoForm (mainForm);
    currentMenu = MenuInit (mainMenu);
    EventLoop ();
    MemFree ();
  }
  return (0);
```

That way none of your code gets executed unless it's part of a normal launch. Of course, this also makes your application a bit brain dead in the Pilot sense of the phrase, but it's quick and easy.

## Ready for Prime Time

Once you think you have a bullet-proof, or relatively wound-proof application, you need to prepare it for the outside world. The only remaining step (assuming you already have a unique Creator ID) is to give it an icon and an application name, which may or may not be different from your file name.

Creating an icon is easy. First, you create an "ICON" resource. Next, in its Info window, give it the name "App Icon" (see Figure 5). That's it.

Creating the application name is relatively easy too. You create a "tAIN" resource and set its contents to the name you want to appear on the Pilot desktop. The `-ot` item on the `PilotRez` line of the Makefile also comes into play with the application name, but I couldn't quite figure out all the details (tech support explained them to me, but I failed to grasp all the ramifications). To be on the safe side, I used the same name in the Makefile and the "tAIN" resource.

## Pilot Rules

This is the second Pilot project I've done. The first was a very lame and simple Hello World. This is more like a real application. In fact, World Time is the first publicly released third-party Pilot client application (as far as we know – by the time you read this there will probably be more). You can find World Time on our web site at http://www.cdpubs.com. The full source code is also on the source code disk for this issue of *PDA Developers*.

There are several things that World Time needs. First of all, I need to go back to Plan A and put all the data in a database. That's the only way you can implement global find, and World Time is a natural for global find support. I may have to create a separate database initialization application, which makes for a very awkward user experience. (It looks like there may be a way to initialize a database from the desktop, using HotSyncing, without writing a separate application. One of the action codes sounds suspiciously like it might be designed to do that.) Once I have the data stored in a bona fide Pilot database, it also makes sense to let users add their own locations, delete useless locations, and provide full data access.

I've also gotten a series of suggestions from end users about providing more locations, location filters, a persistent Home location, alternate interface suggestions, and much more. There's a lot of work that could be done. Based on my experience with World Time, I think anyone with reasonable programming skills (and I'm not sure I would classify my skills as reasonable) can turn out a full-featured Pilot application with just several weeks of full-time effort. That estimate is based on the current state of the SDK documentation, which as I write this are minimal. With real docs, anything is possible.

The Pilot developer tools, even in their current beta form, with bugs and all, are definitely usable. However, there are two additional, very good reasons you might want to consider writing Pilot applications.

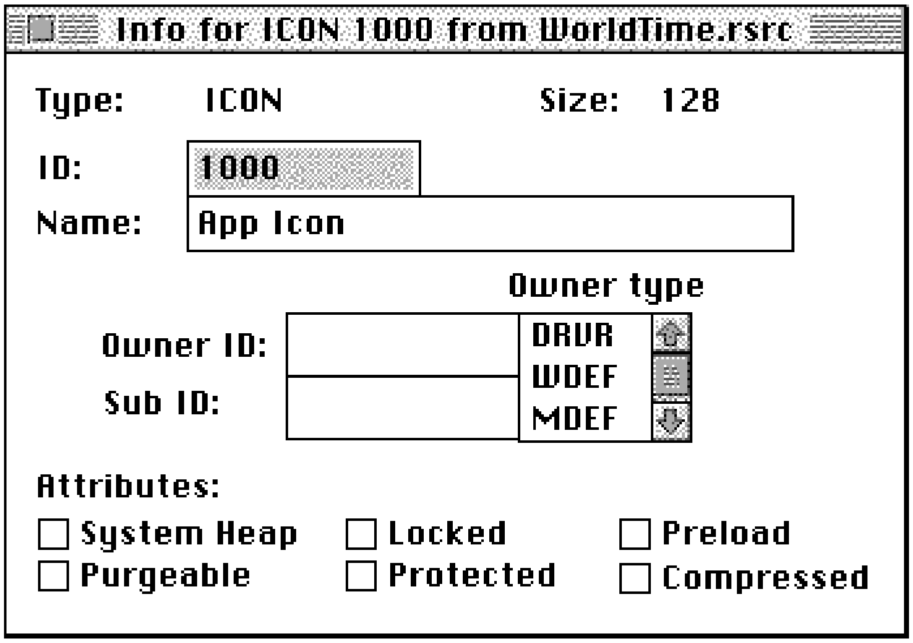
The first is the Pilot architecture. It's well designed, lean and mean, and simple, but not so simple that you have to do everything for yourself. It can be documented in a modest number of pages, understood with a reasonable amount of effort, and the System ROM contains a lot of high-level functionality, enough to save you a considerable amount of work. For a less-than-hard-core programmer like me, this is a definite plus. Palm has lowered, not raised the boundaries for programmers. Also, there's no dependence on C++ (a definite plus in my book).

Second, Palm has left lots of opportunities for third-party developers. Some other PDAs on the market are so full-featured that developers are hard-pressed to think of applications to write for them. Not so with the Pilot.

The Palm SDK, which requires Metrowerks CodeWarrior and a Macintosh computer, should be publicly available by the time you read this. I expect to see a lot of interesting Pilot applications, perhaps even a new version of World Time, in the coming months. ✔

## Acknowledgments

I would like to thank the tech support folks at Palm Computing for their assistance in writing this article and creating World Time. I would especially like to thank Chris Raff for taking the time to explain some of the finer points of Pilot development, while at the same time, working very hard on finishing the tools.


Info for ICON 1000 from WorldTime.rsrc

Type: ICON    Size: 128

ID: 1000

Name: App Icon

Owner type: DRVR, WDEF, MDEF

Owner ID:

Sub ID:

Attributes: ☐ System Heap ☐ Locked ☐ Preload ☐ Purgeable ☐ Protected ☐ Compressed

Figure 6 - A typical ICON resource.