PALMPILOT TABLES HOWTO V1.0

1. Administration

1.1 DISCLAIMER

This tutorial will explain how I use tables. It will present a very simple application that uses a table. The method presented in this memo is not the only way, may not be the best way, and may not even be a good way. However people have been asking questions about how to implement simple PalmPilot table resources, so I offer this HOWTO.

I haven't done everything with tables, so this HOWTO will not explain everything about a table. In fact I have done very little with tables, so this HOWTO explains very little. This HOWTO gives the recreational programmer a starting point. It is expected that after mastering the techniques described here, the reader will find better and more sophisticated ways to manipulate tables. If you improve on these methods, please send me a note describing your techniques and I will immortalize your achievement in an updated version of this HOWTO.

1.2 PLATFORM

This memo will focus on the GNU SDK. The techniques described here may be adaptable to other SDKs. All references to PalmOS API functions, events, data structures, et cetera refer to the PalmOS 2 manuals. If it makes a difference, I use the GNU SDK on Win95, debug on Copilot using a PalmOS 1.0 ROM, and carry a Pilot 1000 with a PalmPilot Professional ROM upgrade.

The HOWTO was written in MS Word 97 and (tediously) converted to HTML for cross-platform distribution.

1.3 PREREQUISITES

Readers should master the techniques discussed in the GNU Pilot SDK Tutorial before advancing to tables. The example application in this HOWTO conforms to and extends the simple application structure discussed in the tutorial. This HOWTO will explain line-by-line all the code in the example which is directly relevant to the table resource. This HOWTO will not explain the simple application code and resources discussed in the tutorial. For instance, this HOWTO will not explain the PilotMain, EventLoop, and FormEventHandler functions, nor will it explain the resource definitions used in the example. The reader should be able to read and understand these source code elements without assistance.

1.4 THANK YOU

Most of the procedure explained here has been adapted from the Memopad.c source code copyright PalmComputing. Thanks PalmComputing, and special thanks to DevSupport for the time they've spent promoting recreational programming of the Pilot.

1.5 UPDATES

The most recent version of this HOWTO will be available at my website <<www.iosphere.net/~howlett>>. If my site moves, then check at Roadcoders or query your favorite search engine.

2. Organize your data

2.1 WHAT IS A TABLE?

A table is a way of organizing data on the screen. For instance the Address List (press the Address hardware button on your Pilot) presents your address data using a table of three columns and eleven rows. An intersection of a column and a row is called a cell. Cells may also be called "table items". According to the PalmOS manual, part 1 page 114, the cells may contain other UI objects such as fields and buttons. This HOWTO is limited to cells containing text.

The PalmOS manual states that the table may be larger than the LCD panel. The table implementation described in this HOWTO applies to tables that fit on the LCD panel.

When you tap in one of the table's cells PalmOS enters a tblSelectEvent on the event queue. The tblSelectEvent data structure includes the row and column of the cell that was tapped. This allows you to execute different actions for each column, row, and/or cell. In this way the table can be thought of as a grid of buttons.

2.2 ORGANIZING THE DATA

Before writing any code, you must plan the presentation of your data. I use pencil and paper and draw what I expect the screen to look like. Graph paper is helpful. The table below illustrates another planning tool I often use.

Data Source	Character Width	Pixel width		Comments
		Required	Assigned	
Field One				
Field Two				
Field Three				
…				
Total:	about 32		160	

In my tables, each row is associated with a record of information. The record has several fields. The problem is to figure out which fields will be displayed, how many characters of each field will be displayed, the number of pixels required to display the entire field and the pixel width assigned to the field.

Some things to remember when planning your column widths:

- In standard font characters average about five pixels wide. So pixel width required is typically 5 times the character width. This means you've got about 32 characters width on the screen.
- If your field is long, then you can truncate it and append an ellipsis (ellipsis are the three dots "…"). However there must be meaningful data in front of the ellipsis or the screen space is wasted.
- Use the minimum number of characters. Use the symbol character sets where appropriate. For instance, if a note is attached, use the note symbol instead of the four characters "Note". Be creative.

Next is to figure out the height of the table. The PalmPilot has 160 pixel-rows. If you want the form to have a title bar, subtract 15 pixel-rows. Many people want on-screen buttons or a message area at the bottom of the LCD panel – subtract another 15 pixel-rows. Now you're down to 130. Standard and bold font are both 11 pixels high, so you can fit 11 rows in the table. Some applications may require column headings. If so, subtract a row for that.

2.3 DEFINE THE RESOURCE

If you have organized your data properly, then you will be able to define the table UI object. From PilRC documentation the Table prototype is

```
TABLE ID <Id.n> AT (<Left.p> <Top.p> <Width.p> <Height.p>)
                   ROWS <NumRows.n> COLUMNS <NumCols.n>
                   COLUMNWIDTHS <Col1Width.n> <Col2Width.n>...
```

2.4 EXAMPLE

For our example, consider an application that lists the data bases on the Pilot memory card. We will call the application “File Manager”. We want the list to show the dbID, whether it’s a Resource database or a “data” database, the Type of database, the CreatorID, and the database Name. Furthermore, if we tap on one of the databases in the list, we want to see more details such as the number of records in the database, the total length and the data length, and the database attributes.

We decide to use a table. Each row of the table will show information for a different database. The 160 pixel-width will be divided as follows:

Data	Character Width	Pixel width		Comments
		Required	Assigned	
dbID	3	15	15	
Resource/Data	1	6	10	R for resource, D for data. Bold font 6 pixels wide.
Type	4	20	30	
CreatorID	4	20	30	
Name	variable	variable	75	Suppose we need at least six characters to identify a database name, plus room for the ellipsis.

Our form will need a title bar, and we want some room at the bottom of the form to display other information and maybe controls. So we have 130 pixel-height to work with, which gives us eleven rows in the table (using standard and bold fonts).

We now have all the information we need to define the table resource and it's form.

```
FORM formID_FileList 0 0 160 160
MENUID  menuID_MainMenu
USABLE
NOFRAME
BEGIN
  TITLE "File Manager 1.0"
  TABLE tableID_FileList AT (0 20 160 140) ROWS 11 COLUMNS 5 COLUMNWIDTHS 15 10 30 30 75
END
```

3. Drawing the Table

3.1 GENERAL

In the SDK Tutorial drawing the form was simple. We defined our UI objects and used the FrmDrawForm API call. If your form contains a table, then the drawing requirements are more complicated. We must create a custom "draw form" function for the form containing the table. I usually take the name of the form and suffix "DrawForm" to it. The DrawForm function must do at least six things:

1. set rows usable/non-usable
2. mark rows invalid
3. set columns usable/non-usable
4. set the style for each cell
5. set the draw procedure for each cell
6. call FrmDrawForm

3.2 SET ROW USABLE

PalmOS displays only rows that are usable. The number of usable rows may change while your application is running. For instance, if you have twenty items to list and only eleven row in your table, then your application will initially display the first eleven items. So initially all eleven rows are usable. Then the user presses the page down key and your application displays the final nine items. So the top nine rows are usable (rows zero to eight) and the bottom two rows are non-usable (rows nine and ten).

You set rows usable/non-usable with the TblSetRowUsable API call. Its prototype is:

```
void TblSetRowUsable ( TablePtr table, Word row, Boolean usable)
```

Row is zero based (first row is row zero) and usable is 1 for usable, 0 for non-usable.

3.3 MARK ROW INVALID

If you set the row usable, then PalmOS will draw that row. But if you need to redraw the table later on, Palm OS will only redraw rows that are marked "invalid". "Invalid" means that the data source has been changed implying that the data displayed by the table in that row is no longer the same as the data source. You mark a row as "invalid" using the TblMarkRowInvalid API call. Its prototype is:

```
void TblMarkRowInvalid (TablePtr table, Word row)
```

3.4 SET COLUMN USABLE

In the same way that PalmOS only displays "usable" rows, PalmOS only displays "usable" columns. So you can turn entire columns off if you want to. Consider a sophisticated table application where the user gets to choose which data will be represented in each column. The user may choose to turn a column off. In my applications all the columns are always usable. The call is TblSetColumnUsable and its prototype is

```
void TblSetColumnUsable ( TablePtr table, Word row, Boolean usable )
```

3.5 SET ITEM STYLE

The DrawForm function must specify the style of each usable cell. Styles are defined in "table.h". I always set the item style to "customTableItem". This forces PalmOS to use my custom cell draw procedure. TblSetItemStyle must be called for every usable cell. Its prototype is:

```
void TblSetItemStyle ( TablePtr table, Word row, Word column,
                       TableItemStyleType type )
```

3.6 SET CUSTOM DRAW PROCEDURE

If the item style is "customTableItem" then PalmOS must use the application defined custom cell drawing procedure. I always set the item style to "customTableItem" and my custom cell drawing procedures will be explained later. You have to define the custom cell drawing procedure for each column. The prototype is:

```
void TblSetCustomDrawProcedure( TablePtr table, Word column,
                                VoidPtr drawCallback )
```

3.7 DRAW FORM

After all the table parameters have been set, our DrawForm procedure uses the FrmDrawForm API call to draw all UI objects, including the table.

3.8 EXAMPLE

Our form is called "File List" so we will call our custom draw form function "FileListDrawForm". First it will erase the window. Then it gets pointers to the form and to the table.

```
static void FileListDrawForm(void)
{
  FormPtr     frm;
  TablePtr    tableP;
  UInt        dbIndex;
  Word        row, numRows;
  int         currFont;
  char        string[30];

  WinEraseWindow();

  frm = FrmGetActiveForm();
  tableP = FrmGetObjectPtr (frm, FrmGetObjectIndex (frm, tableID_FileList));
```

NumDatabases is a global variable containing the number of databases on card zero. If there are no databases on card 0, then return from the function now.

```
   if (NumDatabases == 0)
   {
      FrmDrawForm (frm);
      return;
   }
```

Now we get to the guts of the function. TopRow is a global which is equal to the dbIndex of the database which will be drawn in the top row (row zero) of the table. Get the number of rows in the table using TblGetNumberOfRows. Stuff the value of TopRow into the RowData storage location that is part of the table structure (I'll explain why we have to do this in section 4).

```
   dbIndex = TopRow;
   numRows = TblGetNumberOfRows (tableP);
   TblSetRowData ( tableP, 1, TopRow);
```

Next we loop through all the rows and set them usable/non-usable, and if usable set them invalid and set the item style. We have to set the values for all the rows, but we may not have enough databases on the card to populate the entire table. If dbIndex is less than NumDatabases, then we have an entry for that row so we set the row usable, mark the row invalid, and set the item style of every cell in the row to customTableItem. If dbIndex is not less than NumDatabases, then we set the row non-usable.

```
  for (row = 0; row < numRows; row++, dbIndex++)
  {
     if (dbIndex < NumDatabases)
     {
        TblSetRowUsable ( tableP, row, true);
        TblMarkRowInvalid ( tableP, row);
        TblSetItemStyle ( tableP, row, 0, customTableItem);
        TblSetItemStyle ( tableP, row, 1, customTableItem);
        TblSetItemStyle ( tableP, row, 2, customTableItem);
        TblSetItemStyle ( tableP, row, 3, customTableItem);
        TblSetItemStyle ( tableP, row, 4, customTableItem);
     }
     else
     {
        TblSetRowUsable ( tableP, row, false);
     }
  }
```

We have to set the custom cell draw procedure for each column. We decide now that the name of the custom cell draw function will be "FileListDrawCell". We also have to set all the columns to usable. When that's done we can call FrmDrawForm.

```
  TblSetCustomDrawProcedure ( tableP, 0, FileListDrawCell);
  TblSetCustomDrawProcedure ( tableP, 1, FileListDrawCell);
  TblSetCustomDrawProcedure ( tableP, 2, FileListDrawCell);
  TblSetCustomDrawProcedure ( tableP, 3, FileListDrawCell);
  TblSetCustomDrawProcedure ( tableP, 4, FileListDrawCell);
  TblSetColumnUsable ( tableP, 0, true);
  TblSetColumnUsable ( tableP, 1, true);
  TblSetColumnUsable ( tableP, 2, true);
  TblSetColumnUsable ( tableP, 3, true);
  TblSetColumnUsable ( tableP, 4, true);
  FrmDrawForm (frm);
```

When using a scrollable table it's a good idea to indicate the total number of rows in the data source. You can do this in several ways. Some apps use FrmCopyTitle to indicate the number of records right in the form title. This app uses WinDrawChars to print a string at the bottom of the LCD panel. Because we use the WinDrawChars API call, we had to include the WinEraseWindow call at the beginning of the function.

```
  StrIToA (string, NumDatabases);
  StrCat (string, " databases on card 0");
  WinDrawChars (string, StrLen(string), 0, 149);
}
```

4. Custom Cell Draw Procedure

4.1 GENERAL

When PalmOS draws the table it calls the custom cell draw procedure. Your application does not call this procedure - PalmOS calls this procedure. This statement has an important implication - none of the global variables in your application will be accessible to the custom cell draw procedure. For instance, in our example the TopRow global variable will not be accessible to the custom cell draw procedure. That's why we stuffed it into the row data area using TblSetRowData when we drew the form. Any data that you need to pass from the application to the custom cell draw procedure must be communicated using the row data area.

When PalmOS calls the custom cell draw procedure PalmOS passes the table pointer of the relevant table, the row and column of the cell to be drawn, and a Rectangle structure which describes the location and size of the cell. If you have multiple tables in your application, it is conceivable but not recommendable that you could use one custom cell draw procedure and switch on the table pointer to implement different functions for different tables. It would be way easier to write different custom cell draw procedures for different tables.

Your cell draw procedure processes the row and column arguments to determine how to populate the cell. Exactly how the row and column arguments determine the contents of the cell depends on how you choose to organize your data. PalmOS leaves a lot of room for programmer imagination.

4.2 BEWARE THE BOOLEAN!

The data to be displayed in a cell often requires more pixels than the pixel width of the cell. Memopad.c and my first custom cell draw procedures used the FntCharsInWidth function call to trim strings such that they would fit in the cell. The last argument of FntCharsInWidth is a boolean. Beware the boolean when using GCC! I compiled many PRC files that worked fine on Copilot but resulted in Fatal Errors on PalmPilot. These are obviously word alignment errors. Fooling around with booleans, for instance declaring a second boolean variable, often removed the fatal error. Later on I invented an alternative to the FntCharsInWidth function in order to avoid the boolean. The alternative function is:

```
/***********************************************************************
 *
 * Trim String To Fit Cell
 *
 * I don't want to use the FntCharsInWidth function for two reasons:
 *
 * 1.  The use of a boolean causes strange problems with GCC
 * 2.  I want ellipsis ("...") to appear at the end of truncated strings
 *
 ***********************************************************************/

void TrimStringToFitCell (char *string, int cell_width_pixels)
{
   if (FntCharsWidth(string, StrLen(string)) > cell_width_pixels)
   {
      cell_width_pixels -= 6;
      do
      {
         string[StrLen(string) - 1] = 0;
      } while (FntCharsWidth(string, StrLen(string)) > cell_width_pixels);

      StrCat(string, "...");
   }
}
```

4.3 EXAMPLE

The custom draw cell procedure will be called FileListDrawCell.

```
static void FileListDrawCell (VoidPtr tableP, Word row, Word column,
                              RectanglePtr bounds)
{
   DWord           dbID;
   short           font;
   char            DBName[32], *display_string, string[5];
   short           TextLen;
   FontID          currFont;
   unsigned long   type, creator, RowTop;
   unsigned short  attributes;
```

We want to get the database id of database to be displayed in this row. To figure this out, we need to know the dbIndex of the row. From the Form Draw routine we know that the dbIndex is equal to TopRow plus row. But first we have to get the value from where we stashed it – that is, out of the row data area of row 1. I call TopRow RowTop within the custom cell draw procedure to remind me that it isn't really the global variable.

```
RowTop = TblGetRowData(tableP, 1);
dbID = DmGetDatabase (0, RowTop + row);
```

Once we have the dbID, we can retrieve information about that database.

```
DmDatabaseInfo (0, dbID, (char *) &DBName, &attributes, NULL,
                NULL, NULL, NULL, NULL, NULL, NULL, &type, &creator);
```

Now we must display that information. What we display depends on which column PalmOS wants us to populate. So we switch on column. For each case, we need three pieces of information: a pointer to a string, the length of the string, and which font to use for the string.

```
switch (column)
{
case 0:   // column 0 shows index
   StrIToA(string, RowTop+row);
   display_string = string;
   TextLen = StrLen(string);
   font = 0;
   break;

case 1:   // column 1 shows Res/Data character
   if (attributes && dmHdrAttrResDB) string[0] = 'R';
   else string[0]='D';
   display_string = string;
   TextLen = 1;
   font = 1;
   break;

case 2:   // column 2 shows type
   display_string = (char *) &type;
   TextLen = 4;
   font = 0;
   break;

case 3:   // column 3 shows creatorID
   display_string = (char *) &creator;
   TextLen = 4;
   font = 0;
   break;

case 4:   // column 4 shows dbName
   display_string = (char *) &DBName;
   TrimStringToFitCell(display_string, bounds->extent.x - 2);
   TextLen = StrLen(DBName);
   font = 0;
   break;
}
```

And now all we have to do is display the string.

```
  currFont = FntSetFont (font);
  WinDrawChars(display_string, TextLen, bounds->topLeft.x, bounds->topLeft.y);
  FntSetFont (currFont);               // Restore the font.
}
```

For columns one to four I know that the string will fit in the cell, but in the last column the string might be larger than the cell. Therefore I used the TrimStringToFitCell function when processing column five (case 4).

5. Scrolling the Table

Scrolling the table is easy. Of course, the user has to request a scroll, perhaps by depressing the page down or page up hardware keys, or by tapping an on-screen button, or by selecting a menu item. You have to catch these events in your FormEventHandler. I usually catch the hardware keys, but if you prefer on screen buttons you could create buttons and catch the ctlSelectEvent or you could put the scroll commands in a menu and trap the menuEvent. Here is the code used in FileListEventHandler to trap the hardware keys:

```
   case keyDownEvent:
      if (event->data.keyDown.chr == pageUpChr)
      {
        FileListScroll (pageUpChr);
        FileListDrawForm();
        handled = true;
      }
      else if (event->data.keyDown.chr == pageDownChr)
      {
        FileListScroll (pageDownChar);
        FileListDrawForm();
        handled = true;
      }
      break;
```

You see that I implement scrolling using a FileListScroll function. I pass one of two values: pageUpChr or pageDownChr. These values are defined in PalmOS header files. After FileListScroll does its thing, I redraw the form, using the custom form draw function discussed in section 3.

So what exactly does FileListScroll do? Not much. If the argument was pageUpChr then FileListScroll subtracts numRows from the TopRow global, where numRows is the number of rows in the database. If the argument was pageDownChr, then FileListScroll adds numRows to TopRow. Of course, we don't want TopRow to be greater than our maximum number of data rows. In the example NumDatabases is the maximum number of rows, so if TopRow is larger than NumDatabases, then we set TopRow equal to NumDatabases minus numRows. In fact, there's a better way to do this. You can make sure that the last page displayed is always a full page by using the expression "((TopRow + numRows) >= NumDatabases)". I'll leave it for you to figure out why that works.

Similarly, we don't want TopRow to be negative, so if TopRow is less than zero we set it equal to zero. Notice that the negative test comes after the maximum test. This is important. Consider the case where TopRow=0, NumDatabases=5 and numRows=11. Someone pushes the pageDownChr key. FileListScroll adds numRows to TopRow, so TopRow=11. But TopRow + numRows is greater than NumDatabases, so FileListScroll sets TopRow = NumDatabases - numRows = 5 - 11 = -6. But now TopRow is negative, so the negative test has to come at the end. Since TopRow is less than zero, FileListScroll sets TopRow = 0.

```
static void FileListScroll (int direction)
{
   TablePtr    tableP;
   Word        numRows;
   FormPtr     frm;

   frm = FrmGetActiveForm();
   tableP = FrmGetObjectPtr (frm, FrmGetObjectIndex (frm, tableID_FileList));
   numRows = TblGetNumberOfRows (tableP);
   if (direction == pageUpChr) TopRow = TopRow - numRows;
   else TopRow = TopRow + numRows;

   if ((TopRow + numRows) >= NumDatabases) TopRow = NumDatabases - numRows;
   if (TopRow < 0) TopRow = 0;
}
```

6. Selecting a Table Item

If the user taps a table cell and the both the column and the row have been set to USABLE, then PalmOS adds a tblSelectEvent to the queue. Your FormEventHandler traps the tblSelectEvent and takes appropriate action. When PalmOS enqueues a tblSelectEvent, it includes the following data in the event data structure:

- **TableID**. Developer-defined ID of the table. You might have several different tables in your form. Or you might have more than one form sharing the same FormEventHandler. In either case, you will need to know which table the user tapped. If you only have one table in your application, as in the example, then you don't need this information - you know that there's only one table which could generate a table event, so you can assume the TableID.
- **pTable**. Pointer to a table structure (TableType). This isn't really necessary - since PalmOS gives you the TableID, you could get the table pointer easily using:

```
pTable = FrmGetObjectPtr (FrmGetActiveForm(), FrmGetObjectIndex(FrmGetActiveForm(), TableID));
```

- **row**. Row of the item. Very important. In all my applications, **row** determined what the application does.
- **column**. Column of the item. Might be useful in your application. Not used in the example.

The following code traps and actions the tblSelectEvent in our example's FileListEventHandler function:

```
case tblSelectEvent:
    Show_database_index = TopRow + event->data.tblSelect.row;
    FrmGotoForm(formID_ShowDatabase);
    handled = true;
    break;
```

In the example, when the user taps one of the rows in the table we must show the database information for the corresponding database. The global integer "Show_database_index" contains the index value of the database to show. The ShowDatabase form provides the user interface for displaying this information. The function ShowDatabaseDrawForm shows the information and the function ShowDatabaseEventHandler handles the events. The ShowDatabase form and its two functions are very simple and have little to do with table implementation, so they will not be discussed here.

7. Conclusion

That's all you have to do to implement a simple table. It's not hard. As mentioned in the introduction, you may wish to become more sophisticated. For instance, you might highlight the most recently selected field, using the TblSetRowSelectable call in the custom form draw procedure. Or you might implement tables that are larger than the LCD panel. If you develop interesting tips and tricks, please consider sending them to me so that I might incorporate them in future versions of this HOWTO.

Andrew Howlett
howlett@iosphere.net
Ottawa, Canada
March 1998