Understanding Palm Applications generated with DB2 Everyplace Mobile Application Builder

Autor: Stev Witzel

Date: Jul 2005

Index

Special notices

The information in this publication is not intended as a substitution of the IBM DB2 Everyplace® product documentation provided by IBM. See the Library section of the IBM DB2 Everyplace Web Site for more information about what publications are considered to be product documentation. References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates.

Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program or service. Information in this publication was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

The information contained in this publication was derived under specific operating and environmental conditions. While IBM has reviewed the information for accuracy under the given conditions, the results obtained in your operating environments may vary significantly. Accordingly, IBM does not provide any representations, assurances, guarantees, or warranties regarding performance. Any information about non-IBM ("vendor") products, in this document, has been supplied by the vendor and IBM assumes no responsibility for its accuracy or completeness.

Trademarks IBM

DB2 Everyplace, DB2 are trademarks or registered trademarks of IBM Corporation in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

Intended audience

This document is intended for application developers who need to understand, debug, or extend the source code of an existing Palm application that was originally generated with DB2 Everyplace Mobile Application Builder (MAB). It gives an overview of the specific design characteristics of an application built with MAB. The document assumes that the underlying source code of the MAB project has been exported as described in "Porting a Mobile Application Builder Project to Codewarrior".

MAB essentials

DB2 Everyplace MAB is a rapid application development tool that is used to develop DB2 Everyplace applications for Palm OS. Because the application developer is always shielded from the underlying DB2 Everyplace CLI, the Palm OS 68K APIs, and APIs for barcode scanning and printing, developing applications in MAB requires very little programming experience.

In order to help users with knowledge of C programming, the Palm OS API, and DB2 Everyplace CLI to customize their applications to meet their specific business requirements, MAB provides so-called Custom Scripts. By using these scripts you can add custom C code to an MAB built application to do field validation, execute custom SQL statements, display alerts, pass parameters to the next form, or do almost anything else.

MAB also allows you to define custom global variables and functions.

Design characteristics

Source files

This section gives a short overview about the common source files MAB generates for a Palm OS application. If the MAB project uses multiple segments, you had to split the exported source files in order to move the project to Codewarrior (see "Porting a Mobile Application Builder Project to Codewarrior"). In this case, each of the files listed below might have been divided into several parts (e.g. `MAB_Forms.c` could have been separated into `MAB_Forms1.c`, `MAB_Forms2.c`, …).

MAB_Main.c

This source file contains the prototypes and implementations of the common functions that are listed in section of this document (for example, common database access and error reporting).

MAB_Tables.c

This source file contains the prototypes and implementations of all table-specific functions that are listed in section .

MAB_Forms.c

All form- and join-specific functions that are described in section and of this document are prototyped and implemented in this file. The file also includes the user functions and global variables that have been defined within Custom Scripts of the original MAB project.

MAB_FormHandler.c

This file contains prototypes and implementations of event handler functions. There is one event handler per form.

MAB_Common.h

This file defines the application-specific data structures that are explained in section , as well as some constants

(such as the different types of an SQL statement and font sizes) and macros (such as macros for error handling). If you defined custom preprocessor statements for the original MAB project, you can find these definitions inside this header file.

MAB_Tables.h

This header file contains the prototypes for table functions and other functions that are used within `MAB_FormHandler.c` and `MAB_Forms.c`.

MAB_Messages.h

This file defines the standard messages that are used within the dialogs of your application.

MAB_Events.h

This file defines events that can be used in a MAB project (such as scanner closed).

MAB.h

This file defines the resource identifiers of user interface elements that are used within the application's forms and dialogs.

MAB.rcp

This file is the PilRC resource file that MAB generated for the project. It is used by the PilRC resource compiler. The file contains definitions specific to the application's user interface (such as names of labels and buttons), as well as the SQL statements that are used by the application to select, delete, insert, or update rows.

Data structures

MAB generates specific data structures for each table, form, list, and join used in the application. This section gives an overview of these structures.

Table

For each table that is used by the application, MAB defines a specific structure that represents the columns of that table. This structure is used by the application as a buffer for selected or updated table rows.

The following example shows a table and the structure MAB generates for this specific table.

test	
c1	INTEGER
c2	VARCHAR(128)
c3	TIMESTAMP

```
typedef struct {
   Boolean dirty;            // dirty flag
   unsigned long position;         // current row position
   SQLCHAR *currentSelect;         // the current select statement
   SelectParms_Type parms[3]; // array of parameter information
   int numParms;             // number of paramters
   char Column1[20];               // buffer for 1st column's value
   SQLINTEGER Column1Len;          // length of 1st column
   char Column2[129];              // buffer for 2nd column's value
   SQLINTEGER Column2Len;          // length of 2nd column
```

```
   char Column3[27];              // ...
   SQLINTEGER Column3Len;         // ...
} Table1_Type;
```

The generated structure also contains the current select statement that is used to retrieve the buffered rows of the table (`SQLCHAR *currentSelect`), the number of parameters in the according query (`int numParms`), the current row's position in the table (`unsigned long position`), and a dirty flag (`Boolean dirty`). For parameterized SQL, the structure contains information to bind the parameters for each column (`SelectParms_Type parms[]`).

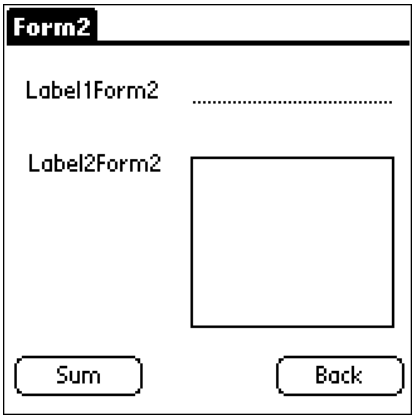
The described data structure is used to buffer both selected and updated rows of a table. More precisely, MAB declares two variables of this type for each table. One is used as a row buffer when interacting with the database, the other one functions as a update buffer that interacts with the application's user interface. In the example given above, MAB would declare the variables as follows:

```
Table1_Type Table1_RowBuffer, Table1_UpdBuffer;
```

Form

For each form in the application, MAB defines a specific structure that represents the fields and lists of that form. MAB then declares a variable of this structure type and uses it as a buffer for the particular form.

To understand the defined structure and its relation to the form it belongs to, consider the following example.



For this form, MAB defines a data structure as follows:

```
typedef struct {
   char Field1Form2_FLD[APP_Field1Form2_LEN+1];
   CharPtr List1Form2_Array;
   int List1Form2_Rc_Size;
} Form2_Record_Type;
```

The following pointer is then used as a buffer for the values of the form's fields and lists:

```
Form2_Record_Type *Form2_bufr;
```

List

For each list that is used in the application, MAB defines two data structures.

The first structure is used as a buffer for the columns of one row in this list. For a list with three columns, this structure could look like the following example:

```
typedef struct {
   char Table1Column2[20];
   char Table2Column2[40];
   char Table1Column1[9];

} List1Form1Cols_Type;
```

To buffer all rows in particular list, MAB defines a second data structure that is used to build a linked list of one or more variables of the row structure mentioned above. This linked list node is defined as follows:

```
typedef struct List1Form1_RCBuf_Type* List1Form1_RCBuf_Ptr;

typedef struct List1Form1_RCBuf_Type {
   List1Form1Cols_Type List1Form1_RCBuf;
   List1Form1_RCBuf_Ptr next;
} List1Form1_RCBuf_Node;
```

The linked list structure is then used in the application as follows:

```
List1Form1_RCBuf_Ptr List1Form1_RCBuf_Hdr;
```

Join

As for tables, forms, and lists, MAB also defines a specific data structure for each join statement that is used in the application. This structure is used as a row buffer and is therefore almost identical to the already shown data structure that is defined for each table.

The following example shows the defined structure for a join of two tables. In this example, the first table consists of three column, and the second table contains four columns.

```
typedef struct {
   Boolean dirty;                  // dirty flag
   unsigned long position;    // current row position
   SQLCHAR *currentSelect          // current select statement
   SelectParms_Type parms[5]; // array of parameter information
   int numParms;              // number of parameters
   char T1C1[20];             // buffer for 1st column's value
   SQLINTEGER T1C1Len;        // length of 1st column
   char T1C2[20];             // buffer for 2nd column's value
   SQLINTEGER T1C2Len;        // length of 2nd column
   char T1C3[20];             // ...
   SQLINTEGER T1C3Len;        // ...
   char T2C1[9];
   SQLINTEGER T2C1Len;
   char T2C2[40];
```

```
   SQLINTEGER T2C2Len;
   char T2C3[50];
   SQLINTEGER T2C3Len;
   char T2C4[25];
   SQLINTEGER T2C4Len;
} Form3JoinT1T2_Type;

Form3JoinT1T2_Type Form3JoinT1T2_RowBuffer;
```

Important: MAB supports joins between only two tables.

Common functions

This chapter gives a short overview of the most common functions that are generated by MAB.

Main

The following prototypes show the most common functions that are used by a MAB application. This set of functions is generated for each application.

```
static Err StartApplication(void);
```

Starts the application.

```
static void StopApplication(void);
```

Stops the application.

```
static void ReportError(SQLHSTMT handle);
```

Displays the error to user.

```
static int ConnectToDB(void);
```

Connects to the database.

```
static int DisconnectFromDB(void);
```

Disconnects from the database.

```
static void Prepare(SQLHSTMT stmtHandle, SQLCHAR* stmtString, Boolean
freeHandle);
```

Prepares a SQL statement for execution. Allocates the statement handle if not defined.

```
static void Execute(SQLHSTMT stmtHandle, Boolean freeHandle);
```

Executes the SQL statement.

Tables

The following prototypes show the set of functions that MAB generates for each database table that the application accesses.

```
static SQLRETURN SelectFrom_Table1(SQLCHAR *sqlStmt, Boolean fetch);
```

Execute Select statement and fetches the data from table.

```
static void ResetTable1();
```

Resets the row buffer and refreshes the statement handle.

```
static void CopyRow2Upd_Table1();
```
Copy data from the row buffer to the update buffer.

```
static void CopyUpd2Row_Table1();
```
Copy data from the update buffer to the row buffer.

```
static void BindParms_Table1(int parmOffset, Table1_Type *tbuffer);
```
Bind the column's data and parameters to corresponding buffer values.

```
static void BindParmsDelete_Table1(int parmOffset, Table1_Type *tbuffer);
```
Bind parameters for the delete statement.

```
static void BindParmsUpdate_Table1(int paramOffset, Table1_Type
*tbuffer);
```
Bind parameters for the update statement.

```
static void FetchTable1(SQLHSTMT stmtHandle, long irow);
```
Fetch the data for the row number that is passed.

```
static unsigned long NextRecord_Table1();
```
Increases the row position of the corresponding row buffer by 1 and sets its dirty flag to true.

```
static unsigned long PreviousRecord_Table1();
```
Decreases the row position of the corresponding row buffer by 1 and sets its dirty flag to true.

```
static unsigned long FirstRecord_Table1();
```
Sets the row position of the corresponding row buffer to 1 and sets its dirty flag to true.

```
static unsigned long LastRecord_Table1();
```
Sets the row position of the according row buffer to 32767 and sets its dirty flag to true.

Joins

The following set of functions is provided for each join statement that is used in the application.

```
static SQLRETURN SelectFrom_Form3JoinT1T2(SQLCHAR *sqlStmt,Boolean
fetch);
```
Executes the join select statement and fetches the data.

```
static void FetchForm3JoinT1T2(SQLHSTMT stmtHandle,long irow);
```
Fetches the data from the join select statement into the join buffer.

```
ResetForm3JoinT1T2();
```
Resets the join buffer.

Forms

For each form that is used in the application, the following function set exists:

```
static unsigned long FillIn_Form2Buffer();
```

```
static void FillIn_Form3JoinT1T2Buffer();
```

Fills the data in the form buffer. While the first function is provided for a form with an underlying select that doesn't use a join, the second function is used for a select with join.

```
static void Form2DrawForm();
```

Draws the form.

```
static Boolean Form2HandleEvent(EventPtr event);
```

Handles the events that occurred on the form.

```
static unsigned long SaveChanges_Form2();
```

Saves the changes that are made in the form.

Lists

For each list, MAB generates the following set of functions:

```
static SQLRETURN SelectFrom_List1Form1(SQLCHAR *sqlStmt, Boolean fetch);
```

Executes the select statement for the list.

```
static unsigned long SelectAndFillInList1Form1_Form1JoinT1T2();
```

Fetches the data and fills the list.

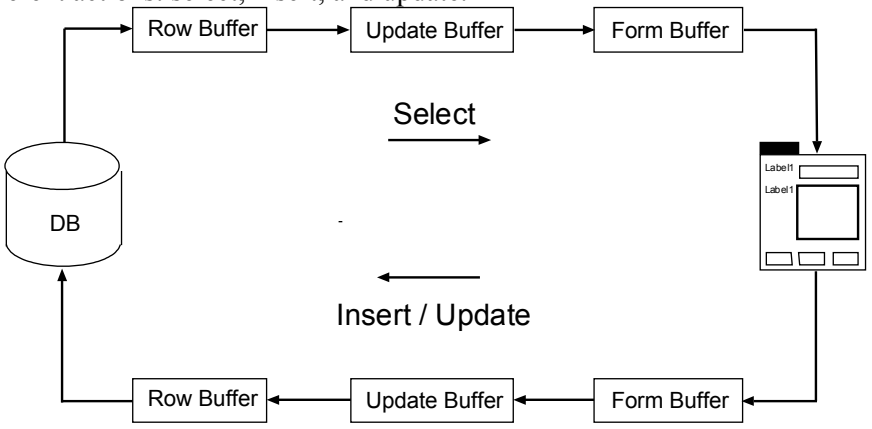
```
static void  List1Form1DrawList(Int16 id, RectangleType *bounds, Char
**data);
```

Draws the list on the form.

Data population and actions

This section gives an overview of the different actions in a Palm OS application that was developed using MAB.

The following graphic shows the structures that are involved in the processing of data. It illustrates three different actions: select, insert, and update.



Single table access

The following section shows the set of function calls made to process each of the common actions for a form that interacts with one table of the database.

Opening a form (frmOpenEvent)

- `SelectFrom_TableX()`
  Selects data from database and fetches the first row.
- `FillIn_FormXBuffer()`
  Fills the form buffer with the selected data.
- `FormXDrawForm()`
  Draws the form.

Inserting form data to a table (Insert Event)

- `SaveChanges_FormX()`
  Form data is copied to the update buffer and then to row buffer. The dirty flag of the update buffer is set to true.
- `CreateRecord_TableX()`
  Binds the columns to buffer values and inserts the record.
- `SelectFrom_TableX()`
  Selects and fetches the inserted data from the table.
- `FillIn_FormXBuffer()`
  Fills the form buffer with the selected data.
- `FormXDrawForm()`
  Draws the form.

Updating a table row with form data (Update Event)

- `SaveChanges_FormX()`
  Form data is copied to the update buffer and then to the row buffer. The dirty flag of the update buffer is set to true.
- `UpdateRecord_TableX()`
  Binds the columns to buffer values and updates the changes.
- `SelectFrom_TableX()`
  Selects and fetches the updated data from the table.
- `FillIn_FormXBuffer()`
  Fills the form buffer with the selected data.
- `FormXDrawForm()`
  Draws the form.

Deleting a row of a table (Delete Event)

- `DeleteRecord_TableX()`
  Binds the columns to buffer values and inserts the record.
- `SelectFrom_TableX()`
  Selects and fetches the previous record from the table.
- `FillIn_FormXBuffer()`
  Fills the form buffer with the selected data.
- `FormXDrawForm()`
  Refreshes the form.

Traversal actions (Next/Previous/First/Last Event)

- `NextRecord_TableX()` if the next record should be displayed or
  `PreviousRecord_TableX()` if the previous record should be displayed or
  `FirstRecord_TableX()` if the first record of the table should be displayed or
  `LastRecord_TableX()` if the last record should be displayed
  Each of these function calls sets the row buffer position accordingly and enables its dirty flag.
- `SelectFrom_TableX()`

The query is executed and the required record is fetched.

- `FillIn_FormXBuffer()`
  The form buffer is updated.
- `FormXDrawForm()`
  The form gets refreshed.

Single table access and form with list control

If there is a list control in the form, the *frmOpenEvent* that executes when the form gets opened is handled slightly different than shown above. For such a form, *lstSelectEvent* is executed when one of the elements in the list control was clicked by the user.

Opening the form (frmOpenEvent)

- `SelectAndFillInListXFormX()`
  Executes select statement on the table that is associated with the list. The following is done in this function:
  - `SelectFromTableX()`
    Execute the select query.
  - Fetch all records selected above and create a linked list. This linked list acts as a buffer.
  - Copy the data from the linked list buffer to the list array (`List1Form2_Array`) of the form buffer and set the list size (`List1Form2_Rc_Size`).
- `FormXDrawForm()`
  Draws the form.

Selecting a row within the list control (lstSelectEvent)

- Get the position of the row clicked and update the position in the row buffer accordingly.
- `SelectFrom_TableX()`
  Selects and fetches the required record from the database.
- `FillIn_FormXBuffer()`
  Updates the form buffer.
- `FormXDrawForm()`
  Refreshes the form.

Multiple table access and a form with list control

If the application uses a join query to retrieve the elements of a list control, *frmOpenEvent* and *lstSelectEvent* are handled as follows.

Opening the form (frmOpenEvent)

- Structure for join of two tables is generated.
- Joined select statement is executed.
- Data is populated in the join buffer.

Selecting a row within the list control (lstSelectEvent)

- Get the position of the row clicked and update the position in join row buffer.
- Select and fetch required record from database.
- Update join buffer with selected data.
- `UnJoinFormXJoinT1T2()`
  Updates the table buffers for the row clicked.

Peripheral support

Barcode scanning

MAB provides its own API for barcode scanning. This API enables the developer to be independent from a vendor- or device-specific implementation. MAB supports 1D barcode scanning (up to 255 characters) for the following scanning devices:

- SPT1500
- SPT1700
- CSM 150

MAB's barcode scanning API provides events that appear over a form, actions, and functions.

Events

The API provides the following events:

ScannerInitEvent
This event occurs just after the *frmOpenEvent*. It has to be overridden in order to make a form scan aware.

NoDecodeEvent
This event occurs if the barcode that was scanned was not able to be decoded.

ScannerCloseEvent
This event occurs after the *frmCloseEvent*.

Functions

The barcode scanning API consists of the following set of functions:

MAB_InitializeScanner
This function initializes the scanner hardware.

Prototype:
```
MAB_SCAN_STATUS MAB_InitializeScanner()
```

Arguments:
None.

Returns:
`MAB_SCAN_STATUS`

MAB_DecodeScan
This function will be called to decode the scan that just happened. This should be called from the *scanDecodeEvent* handler.

Prototype:
```
CharPtr MAB_DecodeScan
```

Arguments:
None.

Returns:
Pointer to the decoded value. If a successful decode did not take place `NULL` is returned.

MAB_DoSoftScan
This function should be called when the scanner should be triggered from a button or a menu item. This is called as soft-scan.

Prototype:
```
MAB_SCAN_STATUS MAB_DoSoftScan()
```

Arguments:
None.

Returns:
`MAB_SCAN_STATUS`

MAB_ScanGetLastErrorMessage
This function returns the error code of the last function called.

Prototype:
```
MAB_SCAN_STATUS MAB_ScanGetLastErrorCode()
```

Arguments:
None.

Returns:
`MAB_SCAN_STATUS`.

MAB_ScanGetLastErrorMessage
This function returns the error message of the last function called.

Prototype:
```
CharPtr MAB_ScanGetLastErrorMessage()
```

Arguments:
None.

Returns:
Pointer to the error message.

MAB_ScanErrorHandler
This functions handles all unsuccessful conditions.

Prototype:
```
Boolean MAB_ScanErrorHandler(UInt16 status)
```

Arguments:
An error code.

Returns:
Boolean that indicates whether the calling function should continue or not.

MAB_DeInitScanner
This function closes all scanner resources.

Prototype:
```
MAB_DeInitScanner()
```

Arguments:
None.

Returns:
`MAB_SCAN_STATUS`

Return codes

`MAB_SCAN_STATUS` is used as the standard return code of MAB's barcode scanning API.
`MAB_SCAN_STATUS` can have the following values:

- `MAB_SCAN_STATUS_OK`
  The function was executed successfully.
- `MAB_SCAN_UNKNOWN_ERROR`
  An unknown error occurred.
- `MAB_SCAN_BARCODE_NOT_SUPPORTED`
  The barcode format is not supported by the API.
- `MAB_SCAN_COMMUNICATIONS_ERROR`
  Unable to communicate with the scanner hardware
- `MAB_SCAN_BAD_PARAM`
  Incorrect parameters supplied to the scanner hardware.
- `MAB_SCAN_BATCH_ERROR`
  Batch too long.
- `MAB_SCAN_NODECODE`
  The barcode that was scanned was not able to be decoded.
- `MAB_SCAN_SOFTSCAN_FAILED`
  The soft scan initiation failed.
- `MAB_SCAN_CURRENT_FIELD_NOT_SCANAWARE`
  The current field is not scan aware.

Printing

MAB uses its own API for printing to hide the vendor-specific implementation and reduce complexity.

Functions

The API for printing consists of the following four functions:

MAB_Print_InitPrinter
This function initializes the printer for line-by-line printing. Pass the print library to be used for printing. You do not need to use this function for printing a buffer or printing a form.

Prototype:
```
MAB_PRINT_STATUS MAB_Print_InitPrinter (MAB_PrintLibrary lib)
```

Arguments:
The name of a valid printing library. Currently the only supported printing library is `MAB_PRINT_VIA_PALMPRINT`.

Returns:
`MAB_PRINT_STATUS`

MAB_ Print_Line
This function prints a line. The printer must be initialized prior to calling this function. The printer must be closed after using this function.

Prototype:
```
MAB_PRINT_STATUS MAB_Print_Line (char *buffer)
```

Arguments:
Pointer to the text to be printed.

Returns:
`MAB_PRINT_STATUS`

MAB_ Print_Buffer
This function prints a buffer. The buffer can contain multiple lines. You do not need to initialize or close the printer.

Prototype:
```
MAB_PRINT_STATUS MAB_Print_Buffer (
      MAB_PrintLibrary lib,
      char *buffer
)
```

Arguments:
1. A supported library name.
2. Pointer to the buffer that should be printed.

Returns:
`MAB_PRINT_STATUS`

MAB_ Print_ClosePrinter
This function closes all printer resources.

Prototype:
```
MAB_PRINT_STATUS MAB_Print_ClosePrinter ()
```

Arguments:
None

Returns
`MAB_PRINT_STATUS`

Return codes

The functions listed above use `MAB_PRINT_STATUS` as a return code. `MAB_PRINT_STATUS` can be set to the following two states:

- `MAB_PRINT_SUCCESS_CODE`
  The operation was successful.
- `MAB_PRINT_ERROR_UNKNOWN_CODE`
  The operation failed.

Usage

The following sample shows how to use MAB's print API.

```
if (MAB_Print_InitPrinter(MAB_PRINT_VIA_PALMPRINT))) {

    MAB_Print_Line((char*)(“Print this Text to test”));

    if (!MAB_Print_ClosePrinter()) {
          // Error in closing the printer
    }
}
```

References

Porting MAB projects to Codewarrior

To learn more about how to extract the underlying source code of a given MAB project, refer to the document “Porting a Mobile Application Builder Project to Codewarrior”.

Palm OS API

For information about the Palm OS 68K API, refer to the documentation given on http://www.palmos.com/dev/support/docs/palmos/PalmOSReference.

DB2 Everyplace CLI

The DB2 Everyplace information center which can be found at http://publib.boulder.ibm.com/infocenter/db2e82/index.jsp gives a detailed overview of all CLI functions provided by DB2 Everyplace. It also provides samples on how to use these functions.