Writing Extensions

What is an extension?

Account supports up to *eight* plug-in "extensions". These may be used to provide additional functionality to the software.

How are extensions implemented?

Extensions are written in OPL, and are translated to be an *OPO* file.
The filename must be eight characters in length, in the form *xxxx*AEXT.OPO where the "*xxxx*" can be anything you choose.

There must be a minimum of four procedures in the program. The procedures begin with the same four characters of the filename.

1. *xxxx*ID$
   Returns a string of up to 32 characters. When the extensions are scanned at *Account* start-up, each extension is loaded and this procedure called. The text returned is used in the "Extensions" selection dialogue to describe what the extension does.
   This procedure should be quick and simple, an idea could be:
   ```
   PROC TestID$:
     RETURN "This is a test extension."
   ENDP
   ```

2. *xxxx*INI%
   This is the initialisation call. It is called just prior to the main entry, and it should return 0 for *okay* and non-zero for *failure*.
   If your module needs any specific memory claims, etc, then it should set them up here. This includes things like new 'windows'.

3. *xxxx*DO%
   This is the main entry. This should get any required user input (using dialogues, *not* console input/output), and do whatever the extension is supposed to do.
   Returns 0 for *okay* or non-zero for *failure*.

4. *xxxx*FIN%
   This is the finalisation entry. It is *always* called prior to the extension being unloaded. Here, you should close all graphics windows and release all memory claims.
   You must be aware, and able to cope with, it being called in cases such as a failed initialisation (i.e. not enough memory). It should return 0 or non-zero, but because the extension is immediately unloaded, the return value is ignored.

It is *strongly* recommended that you provide a procedure at the start of the program to act as a "catch" for if the user attempts to run the program. Refer to the CSV export example.

Calling convention, pseudo-code

This should help you to better understand how extensions are called.

During start-up:
    Load extension
    Description$ = xxxxID$:
    Unload extension

During extension execution:

```
Load extension
If (xxxxINI%: < > 0)
  xxxxFIN%:
  Unload extension, and return
ENDIF
If (xxxxDO%: < > 0)
  xxxxFIN%:
  Unload extension, and return
ENDIF
xxxxFIN%:
Unload extension
Return
```

Global variables

Global variables are not possible in loaded modules.
For this reason, *Account* provides nine global variables exclusively for use by extension modules:

EMIa%, **EMIb%**, and **EMIc%**
These are (16 bit) integer values, for use for things such as file handles.

EMLa%, **EMLb%**, and **EMLc%**
These are (32 bit) long values, for numbers and/or pointers with a value over 32K.

EMRa, **EMRb**, and **EMRc**
These are real values (i.e. numbers which may have a decimal fraction like 12.34).

Wherever possible you should use local variables, and pass variables to procedures.

Errors

Account has an error-catcher, however you should implement your own error trapping.
Account behaviour upon an error is to report the error, call the finalise entry, then unload the extension.

Account variables

The following *global* variables are accessible:

EM*??*
The supplied int, long, and real variables described above.

C.UName$
The name of the account holder (64 chars).

C.IBAN$
The IBAN or account number (48 chars).

C.Bank$
Details of the bank (64 chars).

C.IniBal
The initial balance of the account.

C.CredLim
The credit limit of the account, usually 0 or a negative value.

C.NxtChq&
The number of the next cheque.

C.LastCls&
The date of the last time *Account* was exited.

The '**C**' database is read-only.

DatBase$(128**)**
The path and filename of the account being accessed, *without* extension.

pdd%, **pmm%**, **pyy%**
May be used if you need some integer values.

ax%, **bx%**, **cx%**, **dx%**, **si%**, **di%**, **fl%**
Values, plus *fl%* for flags, for OS calls.

TopLine%, **BotLine%**, **CurLine%**
The top and bottom lines of the display, and the current line. You should alter these with *extreme* care; and if you do it is imperative that you call `DoFudge:` afterwards.

AppN$(12**)**, **AppV$(**6**)**, **AppD$(**12**)**
Application name, version, and date, in case you need them, perhaps to test the version or somesuch?

DtForm%, **DtSep%**
The date format (**0** = American MM/DD/YYYY, **1** = European DD/MM/YYYY, or **2** = Japanese/ISOish YYYY/MM/DD); and the ASCII value of the date separator ('/', '-', etc). These are read from the organiser's system settings.

Buffer$(128**)**
May be used if you require a string buffer.

Accessing account entries

The account entries are implemented as a database, providing the following items of information:

A.Date&
The date of the entry.

A.Item$(64**)**
The description of the account entry.

A.Amount
The amount of the entry. There is no flag for Debit or Credit, these are inferred by whether the amount is positive or negative.

Unless you have *extremely good reason*, the account entries should be treated as *read only*. This is because there are a number of formalities after modifying the database (such as date sorting and updating the account settings). *Contact me if you would like further information*.

Procedures provided within Account

There are various procedures within *Account* that may be of use to you:

Contrast:(*Adjust%*)
If you wish to trap Acorn-< and Acorn-> to adjust the contrast (yes, it seems the OS may expect the current application to do it!), then call this procedure.
Adjust% = **-1** to make contrast *lighter*, any other value for *darker*.

DoFudge:
If you alter TopLine%, BotLine%, or CurLine% (the display position values), you will *need* to call DoFudge to sanitise the values.

DoYMD$:(*year%, month%, day%*)
Given a broken-down date, this will return a string containing the date in a form such as "*2006/12/16*". The actual format is determined by the system configuration.

DyToDat:(*days&*)
Given a number of days (i.e. the account entry date value), this will return the date in broken-down form in the global variables **pyy%**, **pmm%**, and **pdd%**.

Compatibility

Apart from any bug fixes, the extension system is "finished", so there are not envisaged to be any problems using extensions with future versions of *Account*.
The only anticipated 'problem' is that *Account* only supports up to eight extensions… Most future additions to the software will be implemented in the form of extension modules.

Distribution

You are completely free to distribute your extension modules; however I would appreciate if you could provide me with the URLs so I can link directly to the software, and to your site, from my website alongside the software itself.
While I wil not disallow charging for your extensions, I will state that doing so goes against the ethos of the PB2 website.

Support

What sort of support you offer for your extension is a matter for you to decide.
I will only be able to provide a limited amount of support for problems is developing extension modules on a 256Kb PocketBook II or series 3A. The support is completely informal and provided as a favour to you. The quality of support greatly depends upon whether or not you are willing to share your OPL source with me.
No support will be provided to the end-user for extension modules that I did not write.

Example

Provided is a CSV exporter, along with source. This is provided so that you may see an extension in action.

Contact

The website is:

http://www.heyrick.co.uk/software/pb2/

My email address is:

heyrick -at- **merseymail** -dot- **com**

Disclaimer

This information is believed to be correct and is provided in good faith, however I accept no liability or responsibility for error and/or omission. E&OE.
If you suspect an error in this documentation, please email me.

[end of document]