End of an error

Mark Whitehorn shows you how to stop the crash before it happens with the art of error trapping.

In the last Psion programming session I explained how to build a program that, when given a date, works out on which day of the week it falls. It's fine, it's great, but it crashes as soon as it catches a glimpse of a user. So, this calls for error trapping: the art of stopping users from entering dates and other data that may confuse a program and cause it to collapse.

Trapping inappropriate values

The simplest form of error trapping is to put a limit on the values that are acceptable using the Do... Until control structure which we have looked at previously. To constrain the entries permitted into the Day% variable, you could use:

```
Do
   Print "and now the year",
   Input Year%
Until Year% > 1899
```

The above, as well as the other code fragments, are in a text file called PDACODE.TXT on our cover CD.

From a user's point of view this is less than ideal as a date can be rejected with no explanation whatsoever. Only by a process of trial and error can the user deduce that only dates after 1900 are acceptable. Actually, to be pedantic, the user would induce this information since the process of going from a set of specific observations to a general rule is induction, but hey — who loves a pedant? This would all be a bit of a blow if you wanted to find out what day Lloyd George was born.

A more helpful error trap would include guidance for users, displayed if they attempt to enter unacceptable data. To do this requires the If... EndIf command — a useful construct that lets your program behave differently depending on what the user has done.

```
Do
     Print "and now the year",
     Input Year%
     If Year% < 1900
   Print "Please enter a date
 in or after 1900",
     EndIf
Until Year% > 1899
```

When an unacceptable date is entered, the program now tells the user what has gone wrong and what to do about it. Sited inside the Do... Until loop, the If... EndIf message is repeated every time the year value is outside the specified range.

Of course, you might not want to restrict the range of years. You could instead use exactly the same Do... Until and If... EndIf construction to limit the contents of the Day% and the Month% variables — Day% requires a value between 1 and 31, and Month% a value between 1 and 12.

A date can be rejected with no explanation whatsoever

For the former, you might decide that a simple Do... Until loop will suffice without an If... EndIf to print a message such as this:

```
Do
Print "Enter the day element
 of your birthday"
      Input Day%
Until Day% > 0 AND Day% < 32
```

In the last line above, AND is used with the 'greater than and less than' operators to define the upper and lower values in an acceptable range.

Input in the wrong format

A useful command called TRAP and a useful function called ERR are available in OPL and together they let you handle situations where the user enters, for instance, a text string where a numeric value is expected.

With the Birthday program as it exists, typing 'May' when asked for the month leaves you looking at a question mark and a large blinking cursor. Pressing Enter in the hopes of being able to try again merely repeats the question mark/cursor combination. The program has not hung — it is waiting for an input it can process. To escape the loop you have only to type a number at the cursor — but there is nothing to tell the user this. It is much neater to add some lines like this:

```
Do
Print "now the month"
Trap Input Month%
Until Err=0
```

Err returns the number of the last error that occurred, or 0 if there were no errors, and Trap captures the input without halting the program, setting Err to the error code as it does so.

[FIG 1]

A graceful close

```
Proc EscTrap:  **Psion5: C/documents/mprog2**
Local a%, b%
Print "Enter a number or press Escape to stop"
Trap Input a%
              If Err=-114
      Goto Finish
             EndIf
Print "And another number"
Input b%
Print a%/b%, "is the result of dividing ", a%, "by ", b%
Pause 40

Finish::
      Print "The End"
Pause 40
EndP
```

In the code above, Trap captures the input on its way to the Month% variable and, if a number is captured, Err is set to zero. If anything other than a number is captured Err is set to the appropriate error code. In this latter case, the program asks 'now the month' again. When a number is entered, Err returns a zero and the program continues beyond the Do... Until loop.

This is not an ideal error trap because the user is simply asked the same question again, rather than being told unequivocally to type in a number, but using your skill and judgement you can embellish the code with If statements to take care of that.

Trapping specific errors

The Err function can also be used to identify specific error codes and act accordingly. For instance, the code -114 means that the Escape button has been pressed. On detecting this, the program can react by printing an appropriate message and closing the program gracefully [Fig 1].

The input to the b% variable could also be trapped and the If... EndIf loop inserted beneath it so that the user could leave the program at any stage. But the above should give you the idea.

For the propeller heads amongst us there is a complete list of the OPL error codes in the OPL programming manual.

You can also set up error handling routines whereby when an error occurs the program will skip to a label and proceed from there. A label is, in effect, a line identifier.

```
OnErr ErrorHandler ** full
 program: c/documents/mprog3**
Print a%/b%, "is the result of
 dividing ", a%, "by ", b%
ErrorHandler::
OnErr Off
If Err = -8
Print "You can't divide by zero"
EndIf
```

OnErr sets up an error handler called, in the example above, ErrorHandler. And the label to which it points is the line:

```
ErrorHandler::
```

When an error occurs, the program leaps to the error handler label where the code attached to the handler spots a -8 error code — which means that a divide by zero error has occurred — and prints the 'You can't divide by zero' message. OnErr should be deactivated with OnErr Off immediately after the label.

OnErr, Trap and Err are not covered exhaustively here and, as ever, the code fragments come with the usual health warnings, but they should point you in the right direction for keeping your users on the straight and narrow.

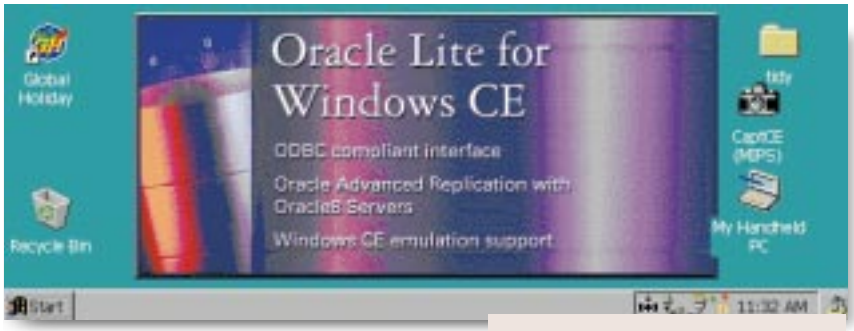
You need an RDBMS engine that runs in a tiny amount of RAM

Mobile computing

This month's column co-stars the *Hands On Databases* column because we are talking about mobile databases. Please flip to p242 and read the introduction there. Right. So from the PDA angle what do you need to create one of these mobile applications, how do you create it, and what is it like to use in practice?

What do you need?

You need an RDBMS engine that runs in an appallingly tiny amount of RAM.



▲ FIG 2 ORACLE LITE ALIVE AND RUNNING ON A WINCE MACHINE. NOTE THE GLOBAL HOLIDAYS ICON IN THE TOP LEFT-HAND CORNER

Three of the big database companies are either offering, or about to offer, such engines.

To give you a flavour of these I will use Oracle Lite as an example. Sybase has SQL anywhere, Oracle has Oracle Lite and IBM is about to launch DB2 Everywhere and DB2 Satellite.

Oracle Lite runs in well under 1Mb. You could be forgiven for wondering what has been removed from the product in order to achieve this. Indeed, you could be forgiven for asking if there is anything left except the grin — in fact, there are still enough teeth to give a serious bite, too.

There is a fully SQL 92 compliant engine but no support for PL/SQL — Oracle's own proprietary Procedural extensions to SQL. However, it does support Java which can be used in place of PL/SQL.

All the standard data types are supported including, rather surprisingly, BLOBs (Binary Large Objects). The rich support that Oracle 8 offers for objects is missing but this makes sense — anyone who expects to be able to store, classify, query and play video clips on a PDA is certainly using a more powerful PDA than mine.

Oracle Lite understands about transactions, so the current transaction can be rolled back. It also keeps a transaction log which can be used to step back through a series of transactions.

However, the extensive additional logging facilities found in the server-side product is missing. This is reasonable, though. Log files are traditionally

◄ FIG 3 THE BOOKING APPLICATION, RUNNING

kept on separate hard disks to guard against disk crashes. But how many hard disks does your PDA currently have? There is also no support for multiple users.

OK, that's the engine, but what about the application which runs on the PDA? If you're talking about a CE application, the development work can be done on a PC since Windows CE applications are typically developed there, anyway.

You can use the same database schema that you used for the Oracle backend database and build the UI for the PDA in the tool of your choice; say Visual C++. And because Oracle Lite is compatible with Oracle, all of the SQL 92 that was developed for the PC version of the application can be cut and pasted into the Visual C++ version. All of the existing queries that have taken so long to hand-craft should run without modification. Once the CE application is complete, it can be compiled for the target CE machine and transferred.

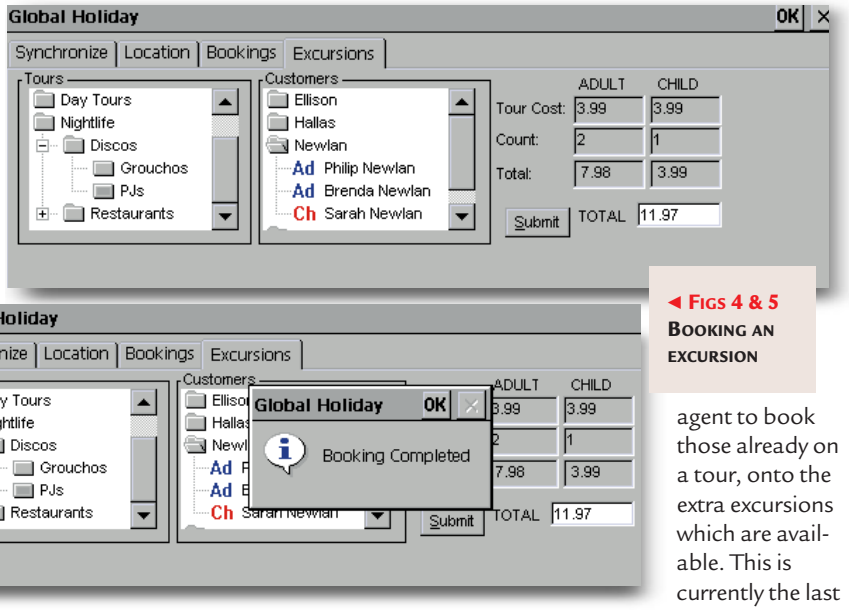
The development for other PDA machines may well follow the same path but the details will depend on the normal development path for that machine.

You also need a mechanism for connecting the PDA to the server. Oracle supports dial-in HTTP, LAN and wireless connections.

As discussed in the *Databases* column (p242) you also need a way to resolve the conflicts which arise when the data is synchronised back to the server, but we will cover that in the *Hands On Databases* column next month.

OK, so you've got your tiny, but perfectly formed, database engine on the PDA. You've developed your application, worked out how to resolve the conflicts when the data is synchronised and have got a reliable connection set up between the PDA and the server.

So, what is the whole thing like to use in practice? Fig 3 shows a simple application running on a WinCE machine. It is a holiday booking program which allows the on-site agent to book those already on a tour, onto the extra excursions which are available. This is currently the last order placed and if you turn to the *Hands On Databases* column *(p242)* you can see the underlying table as it exists on the server.

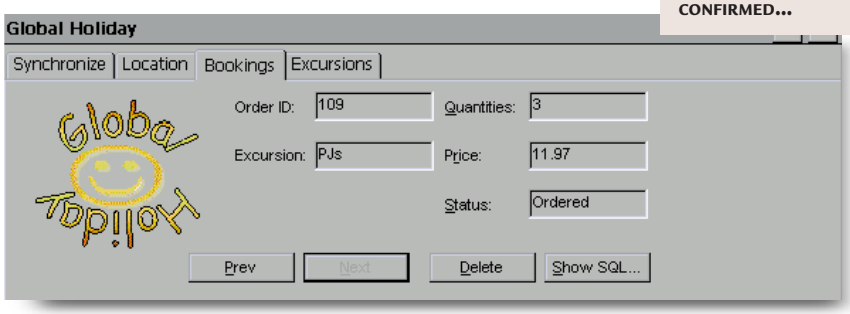


◄ **Figs 4 & 5** Booking an excursion

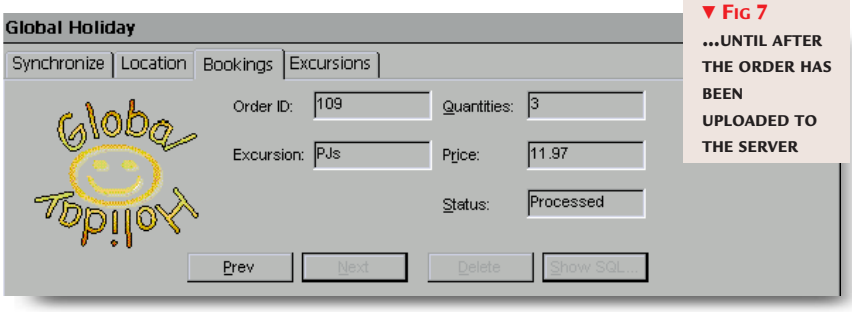
Figs 4 & 5 show the process of booking some new excursions. Here, two adults and their unfortunate offspring are destined for a night at PJ's.

Back at the Bookings tab, this excursion shows up with the status 'Ordered' [Fig 6]. After the database has been synchronised with the server-based data, this order appears on the server *(see p242)* and on the PDA its status changes to 'Processed' [Fig 7].

But all is not what it seems, which highlights the need for intelligent human design in these applications. This is simply a demonstration program, not intended for reality, which is just as well. It transpires that the change in status from 'ordered' to 'processed' is triggered in the PDA application by the process of requesting synchronisation with the server, rather than the successful completion of that process. I found that orders could acquire 'processed' status even when there was no cable between the PDA and the server. Either there was a wireless connection I didn't know about, or some redesign would be required before the system was used in anger.



▼ **Fig 6** The order is held on the PDA but not confirmed...



▼ **Fig 7** ...until after the order has been uploaded to the server

PCW CONTACTS

Mark Whitehorn welcomes your feedback on the PDAs column. He can be contacted via the PCW editorial office (address, page 10) or email pda@pcw.co.uk