OPL

DATA FILE AND DATABASE HANDLING

CONTENTS

DATA FILE HANDLING

You can use OPL to create data files (databases) like those used by the Data application. You can store any kind of information in a data file, and retrieve it for display, editing or calculations.

This section covers:

- **Creating data files**
- **Adding and editing records**
- **Searching records**
- **Using a data file both in OPL and in the Data application**

The Series 5 and the Series 3c database models differ quite substantially. However, the Series 3c method of database programming (except for some removed keywords as indicated) is completely understood by the Series 5 model and any existing code will not have to change. However, it is very strongly recommended that on the Series 5 you use the new keywords INSERT, MODIFY, PUT and CANCEL, along with bookmarks and transactions, rather than using APPEND, UPDATE, POS and POSITION.

If you are using the Series 5, it is recommended that you read this section for a description of simple database use, and then the following section of this document which refers specifically to features available on the Series 5.

FILES, RECORDS AND FIELDS

Data files (or *databases*) are made up of *records* which contain data in each of their *fields*. For example, in a database of names and addresses, each record might have a name field, a telephone number field, and separate fields for each line of the address.

In OPL you can:

- Create a new ***file*** with CREATE, or open an existing file with OPEN, and copy, delete and rename files with COPY, DELETE and RENAME.
- Add a new ***record*** with APPEND, change an existing one with UPDATE, and remove a record with ERASE.
- Fill in a ***field*** by assigning a value to a field variable.

CREATING A DATA FILE

Use the CREATE command like this:

```
CREATE filename$,logical name,field1,field2,...
```

For example:

```
CREATE "clients",B,nm$,tel$,ad1$,ad2$,ad3$
```

creates a data file called `clients`.

The file name is a string, so remember to put quote marks around it. You can also assign the name string to a string variable (for example `fil$="clients"`) and then use the variable name as the argument - `CREATE fil$,A,field1,field2`.

LOGICAL NAMES

❺ You can have up to 26 data files open at a time. Each of these must have a logical name: A to Z.

❸ You can have up to 4 data files open at a time. Each of these must have a logical name: A, B, C or D.

The logical name lets you refer to this file without having to keep using the full file name.

A different logical name must be used for each data file opened - e.g. one called A, one called B and one called C. A file does not have to be opened with the same logical name as the last time it was opened. When a file is closed, its logical name is freed for use by another file.

FIELDS

`field1`, `field2`,... are the field names - up to 32 in any record. These are like variables, so - use % & or $ to make the appropriate types of fields for your data. You cannot use arrays. Do not specify the maximum length of strings that the string fields can handle. The length is automatically set at 255 characters.

Field names may be up to 8 characters long, including any qualifier like &.

When referring to fields, add the logical file name to the front of the field name, to specify which opened file the fields belong to. Separate the two by a dot. For example, `A.name$` is the `name$` field of the file with logical name A, and `C.age%` is the `age%` field of the file with logical name C.

The values of all the fields are 0 or null to start with. You can see this if you run this example program:

```
PROC creatfil:
     CREATE “example”,A,int%,long&,float,str$
     PRINT “integer=”;a.int%
     PRINT “long=”;a.long&
     PRINT “float=”;a.float
     PRINT “string=”;a.str$
     CLOSE
     GET
ENDP
```

OPENING A FILE

When you first CREATE a data file it is automatically open, but it closes again when the program ends. **If a file already exists, trying to CREATE it again will give an error** - so if you ran the procedure `creatfil:` a second time you would get an error. To open an existing file, use the OPEN command.

OPEN works in the same way as the CREATE command. For example:

```
OPEN “clients”,B,nm$,tel$,ad1$,ad2$,ad3$
```

- You must use the same filename as when you first created it.
- You must include in the OPEN command each of the fields you intend to alter or read. You can omit fields from the end of the list; you **cannot** miss one out from the middle of the list, for example nm$,,ad1$ would generate an error, whereas nm$,tel$,ad1$ would be fine. They must remain the same type of field, but you can change their names. So a file created with fields name$,age% could later be opened with the fields a$,x%.
- You must give the file a logical name. See ‘Logical names’ above. You can’t have two files open simultaneously with the same logical name, so when opening the files, remember which logical names you have already used.

You might make a **new** module, and type these two procedures into it:

```
PROC openfile:
  IF NOT EXIST(“example”)
    CREATE “example”,A,int%,lng&,fp,str$
  ELSE
    OPEN “example”,A,int%,lng&,fp,str$
  ENDIF
  PRINT “Current values:”
  show:
  PRINT “Assigning values”
  A.int%=1
  A.lng&=&2**20            REM the 1st & avoids integer overflow
  A.fp=SIN(PI/6)
  PRINT “Give a value for the string:”
  INPUT A.str$
  PRINT “New values:”
  show:
ENDP
```

```
PROC show:
   PRINT “integer=”;A.int%
   PRINT “long=”;A.lng&
   PRINT “float=”;A.fp
   PRINT “string=”;A.str$
   GET
ENDP
```

NOTES

OPENING/CREATING THE FILE

The IF...ENDIF checks to see if the file already exists, using the EXIST function. If it does, the file is opened; if it doesn’t, the file is created.

GIVING VALUES TO THE FIELDS

The fields can be assigned values just like variables. The field name must be used with the logical file name like this: `A.f%=1` or `INPUT A.f$`.

If you try to give the wrong type of value to a field (for example “`Davis`” to `f%`) an error message will be displayed.

You can access the fields from other procedures, just like global variables. Here the called procedure `show:` displays the values of the fields.

FIELD NAMES

You must know the type of each field, and you must give each a separate name - you cannot refer to the fields in any indexed way, e.g. as an array.

OPENING A FILE FOR SHARING

The OPENR command works in exactly the same way as OPEN, except that the file cannot be written to (with UPDATE or APPEND), only read. However, more than one running program can then look at the file at the same time.

SAVING RECORDS

The last example procedure did not actually save the field values as a record to a file. To do this you need to use the APPEND command. This program, for example, allows you to add records to the `example` data file:

```
PROC count:
     LOCAL reply%
     OPEN “example”,A,f%,f&,f,f$
     DO
          CLS
          AT 20,1 :PRINT “Record count=”;COUNT
          AT 9,5 :PRINT “(A)dd a record”
          AT 9,7 :PRINT “(Q)uit”
          reply%=GET
          IF reply%=%q OR reply%=%Q
               BREAK
          ELSEIF reply%=%A OR reply%=%a
```

```
            add:
        ELSE
            BEEP 16,250
        ENDIF
    UNTIL 0
ENDP

PROC add:
    CLS
    PRINT “Enter integer field:”;
    INPUT A.f%
    PRINT “Enter long integer field:”;
    INPUT A.f&
    PRINT “Enter numeric field:”;
    INPUT A.f
    PRINT “Enter string field:”;
    INPUT A.f$
    APPEND
ENDP
```

BEEP

The BEEP command makes a beep of varying pitch and length:

```
BEEP duration%,pitch%
```

The duration is measured in 1/32 of a second, so `duration%=32` would give a beep a second long. Try `pitch%=50` for a high beep, or `500` for a low beep.

THE NUMBER OF RECORDS

The COUNT function returns the number of records in the file. If you use it just after creating a database, it will return 0. As you add records the count increases.

HOW THE VALUES ARE SAVED

Use the APPEND command to save a new record. This has no arguments. The values assigned to `A.f%`, `A.f&`, `A.f` and `A.f$` are added as a new record to the end of the `example` data file. If you only give values to some of the fields, not all, you won’t see any error message. If the fields happen to have values, these will be used; otherwise - null strings (“”) will be given to string fields, and zero to numeric fields.

New field values are always added to the end of the current data file - as the last record in the file (if the file is a new one, it will also be the first record).

At any time while a data file is open, the field names currently in use can be used like any other variable - for example, in a PRINT statement, or a string or numeric expression.

APPEND AND UPDATE

APPEND adds the current field values to the end of the file as a new record, whereas UPDATE deletes the **current** record and adds the current field values to the end of the file as a new record.

MOVING FROM RECORD TO RECORD

When you open or create a file, the first record in the file is current. To read, edit, or erase another record, you must make that record current - that is, move to it. Only one record is current at a time. To change the current record, use one of these commands:

- POSITION 'moves to' a particular record, setting the field variables to the values in that record. For example, the instruction POSITION 3 makes record 3 the current record. The first record is record 1.
- You can find the current record number by using the POS function, which returns the number of the current record.
- FIRST moves to the first record in a file.
- NEXT moves to the following record in a file. If the end of the file is passed, NEXT does not report an error, but the current record is a new, empty record. This case can be tested for with the EOF function.
- BACK moves to the previous record in the file. If the current record is the first record in the file then that first record stays current.
- LAST moves to the last record in the file.

DELETING A RECORD

ERASE deletes the current record in the current file.

The next record is then current. If the erased record was the last record in a file, then following this command the current record will be empty and EOF will return true.

FINDING A RECORD

FIND makes current the next record which has a field matching your search string. Capitals and lower-case letters match. For example:

```
r%=FIND("Brown")
```

would select the first record containing a string field with the value "Brown", "brown" or "BROWN", etc. The number of that record is returned, in this case to the variable `r%`. If the number returned is zero, no matching field was found. Any other number means that a match was found.

The search includes the current record. So after finding a matching record, you need to use NEXT before you can continue searching through the following records.

`FIND("Brown")` would not find a field "Mr Brown". To find this, use wildcards, as explained below.

You can only search string fields, not number fields. For example, if you assigned the value 71 to the field `a%`, you could not find this with FIND. But if you assigned the value "71" to `a$`, you could find this.

WILDCARDS

`r%=FIND("*Brown*")` would make current the next record containing a string field in which `Brown` occurred - for example, the fields "MR BROWN", "Brown A.R." and "Browns Plumbing" would be matched. The wildcards you can use are:

?	matches any one character
*	matches any number of characters.

Once you've found a matching record, you might display it on the screen, erase it or edit it. For example, to display all the records containing "BROWN":

```
FIRST
WHILE FIND("*BROWN*")
     PRINT a.name$,a.phone$
     NEXT
     GET
ENDWH
```

MORE CONTROLLED FINDING

FINDFIELD, like FIND, finds a string, makes the record with this string the current record, and returns the number of this record. However you can also use it to do case-dependent searching, to search backwards through the file, to search from the first record (forwards) or from the last record (backwards), and to search in one or more fields.

```
f%=FINDFIELD(a$,start%,no%,flag%)
```

searches for the string `a$` in `no%` fields in each record, starting at the field with number `start%` (1 is the number of the first field). `start%` and `no%` may refer to string fields only and other types will be ignored. The `flag%` argument specifies the type of search as explained below. If you want to search in all fields, use 1 as the second argument and for the third argument use the number of fields you used in the OPEN/CREATE command.

`flag%` should be specified as follows:

search direction	`flag%`
backwards from current record	0
forwards from current record	1
backwards from end of file	2
forwards from start of file	3

❺ Constants for these flags are supplied in Const.oph. See the 'Calling Procedures' section of the 'Basics.pdf' document for details of how to use this file and see Appendix E in the 'Appends.pdf' document for a listing of it.

Add 16 to the value of `flag%` given above to make the search *case-dependent*, where case-dependent means that the record will exactly match the search string in case as well as characters. Otherwise the search will be *case-independent* which means that upper case and lower case characters will match.

For example, if the following OPEN (or CREATE) statement had been used:

```
OPEN "clients",B,nm$,tel$,ad1$,ad2$,ad3$
```

then the command

```
r%=FINDFIELD("*Brown*",1,3,16)
```

will search the `nm$`, `tel$` and `ad1$` fields of each record for strings containing "Brown" searching case-dependently backwards from the current record.

If you find a matching record and then you want to search again from this record, you must first use NEXT or BACK (according to the direction in which you are searching) to move past the record you have just found, otherwise the search will find the same match in the current record again.

CHANGING/CLOSING THE CURRENT FILE

Immediately after a file has been created or opened, it is automatically current. This means that the APPEND or UPDATE commands save records to this file, and the record-position commands (explained below) move around this file. You can still use the fields of other open files, for example `A.field1=B.field2`

USE makes current one of the other opened files. For example `USE B` selects the file with the logical name B (as specified in the OPEN or CREATE command which opened it).

If you attempt to USE a file which has not yet been opened or created, an error is reported.

In this procedure, the EOF function checks whether you are at the end of the current data file — that is, whether you've gone **past** the last record. You can use EOF in the test condition of a loop UNTIL EOF or WHILE NOT EOF in order to carry out a set of actions on all the records in a file.

EXAMPLE - COPIES SELECTED RECORDS FROM ONE FILE TO ANOTHER

```
PROC copyrec:
     OPEN "example",A,f%,f&,f,f$
     TRAP DELETE "temp"      REM If file doesn't exist, ignore error
     CREATE "temp",B,f%,f&,f,f$
     PRINT "Copying EXAMPLE to TEMP"
     USE A                   REM the EXAMPLE file
     DO
          IF a.f%>30 and a.f<3.1415
          b.f%=a.f%
          b.f&=a.f&
          b.f=a.f
          b.f$="Selective copy"
          USE B              REM the TEMP file
          APPEND
          USE A
          ENDIF
          NEXT
     UNTIL EOF               REM until End Of File
     CLOSE                   REM closes A; B becomes current
     CLOSE                   REM closes B
ENDP
```

This example uses the DELETE command to delete any `temp` file which may exist, before making it afresh. Normally, if there were no `temp` file and you tried to delete it, an error would be generated. However, this example uses TRAP with the DELETE command. TRAP followed by a command means "if an error occurs in the command, carry on regardless". The error value can then be found using ERR.

There are more details of ERR and TRAP in the 'Errors.pdf' document.

CLOSING A DATA FILE

You should always 'close' a data file (with the CLOSE command) when you have finished using it. Data files close automatically when programs end.

❺ You can use up to 26 logical names (files or *views* — see the 'Series 5 Database Handling' section of this document) at a time - if you are using 26 logical names and you want to use another one, you must close one of the open files or views first. CLOSE closes the file or view referred to by the *current* logical name.

❸ You can only have 4 files open at a time - if you already have 4 files open and you want to access another one, you must close one of the open files first. CLOSE closes the *current* file.

KEEPING DATA FILES COMPRESSED

When you change or delete records in a data file, the space taken by the old information is not automatically recovered.

❺ **By default**, the space is **not** recovered when you close the file, unless you have used the SETFLAGS command to enable auto-compaction on closing a file.

❸ **By default**, the space is recovered when you close the file, provided it is on 'Internal drive' or on a **RAM** SSD (i.e. it is not on a Flash SSD).

Closing a very large file which contains changed or deleted records can be slow when compression is enabled, as the whole file beyond each old record needs copying down, each time.

❸ You can **prevent** data file compression on the Series 3c if you wish, with these two lines:

```
p%=PEEKW($1c)+$1e
POKEW p%,PEEKW(p%) or 1
```

(Use any suitable integer variable for `p%`.) Files used by the current program will now **not** compress when they close.

Use these two lines to re-enable auto-compression:

```
p%=PEEKW($1c)+$1e
POKEW p%,PEEKW(p%) and $fffe
```

Warning: be careful to enter these lines exactly as shown. These examples work by setting a system configuration flag.

If you have closed a file **without** compression, you can recover the space by using the COMPRESS command to create a new, compressed version of the file. `COMPRESS "dat" "new"`, for example, creates a file called `new` which is a compressed version of `dat`, with the space which was taken up by old information now recovered. (You have to use COMPRESS to compress data files which are kept on a Flash SSD.)

❺ On the Series 5, you can use the COMPACT command when the database is closed. See the 'Series 5 Database Handling' section of this document.

SERIES 3C AND SIENA DATA FILES AND THE DATA APPLICATION

The files you use with the Data application (listed under the Data icon in the System screen) often called *databases* or *database files* - are also just data files.

Data files created by the Data application can be viewed in OPL, and vice versa.

In OPL: to open a data file made by the Data application, begin its name with `\DAT\`, and end it with `.DBF`. For example, to open the file called `data` which the Data application normally uses:

```
OPEN "\dat\data.dbf",A,a$,b$,c$,d$...
```

Restrictions:

- You can use up to 32 field variables, all strings. It is possible for records to contain more than 32 fields, but these fields cannot be accessed by OPL. It's safe to change such a record and use UPDATE, though, as the extra fields will remain unchanged.
- The maximum record length in OPL is 1022 characters. You will see a 'Record too large' error (-43) if your program tries to open a file which contains a record longer than this.
- The Data application breaks up long records (over 255 characters) when storing them. They would appear as separate records to OPL.

In the Data application: to examine an OPL data file, press the Data button, select 'Open file' from the 'File' menu and Control+Tab to type in a file name, and then type the name with `\OPD\` on the front and `.ODB` at the end for example:

```
\opd\example.odb
```

Restrictions:

- All of the fields must be string fields.
- You can have up to a maximum of 32 fields, as specified in the CREATE command. If you view an OPL data file with the Data application, and add more lines to records than the number of fields specified in the original CREATE command, you will get an error if you subsequently try to access these additional fields in OPL.

In both cases, you are using a more complete *file specification*. There is more about file specifications in the 'Advanced.pdf' document.

❺ For details of using Data application files in OPL on the Series 5, see the next section of this document.

SERIES 5 DATABASE HANDLING

The Series 5 uses the relational database management system (DBMS) of EPOC32 which supports SQL (Standard Query Language).

Apart from the removed keywords RECSIZE, COMPRESS and ODBINFO, the Series 3c methods of database programming are completely understood by the Series 5 model and existing code will not have to change. However, it is very strongly recommended that you use INSERT, MODIFY, PUT and CANCEL along with bookmarks and transactions, rather than using APPEND, UPDATE, POS and POSITION.

See also the 'Alphabetic Listing' section of the 'Glossary.pdf' document for some more detailed description of the use of new and changed database commands and 'Database OPX' in the 'OPX.pdf' document.

THE SERIES 5 DATABASE MODEL

As has been stressed previously, it is very strongly recommended that you use this Series 5 specific model on the Series 5, despite the fact that the data file handling methods of the Series 3c may still be used on the Series 5. The reasons for this are as follows:

- the new keywords closely reflect the underlying EPOC32 database model supplied by DBMS. They are therefore more efficient than the Series 3c keywords on the Series 5.
- to emulate the Series 3c behaviour, APPEND has to create an intermediate copy of the record which is erased on completion of the keyword. This ensures the rather strange requirement that the field values of the previous APPEND are used as the initial values for the current APPEND. This can make a database grow far larger than on the Series 3c. You can, however, use COMPACT or SETFLAGS to remove erased records from a database.
- without transactions, writing a large number of records to a database is far slower on the Series 5 than on the Series 3c. However, with transactions it is far faster.
- the Series 5 model is superior.

DATABASES, TABLES, VIEWS, FIELDS AND FIELD HANDLES

To describe the new model it is necessary to expand upon the terminology that was used in the previous section.

A Series 3c data file corresponds more or less to a single *table* in a DBMS database file. A database can contain one or more tables. A table, like a data file on the Series 3c, contains records which are made up of fields. Unlike the Series 3c, however, the field names as well as the table names are stored in the database.

CREATING DATABASES AND TABLES

With the statement:

```
CREATE "datafile",A,f1%,f2%
```

as described in the previous section, the Series 5 creates a database called `datafile` and a table with the default name `Table1` would be added to it. The field names are derived from the `f1%` and `f2%` which are called *field handles*. The type of the field, as always, is defined by these handles.

With the Series 5 it is also possible to use, for example,

```
CREATE "people FIELDS name, number TO phoneBook",A,n$,number$
```

This will create a table called `phoneBook` in the database called `people`, creating the database too if it does not exist. The table will have fields `name` and `number`, whose respective types are specified by the field handles `n$` and `number$`, both strings in this example.

Note that CREATE creates a **table**. An error is raised if the table already exists in the database. DBMS does not allow the database to be open when a table (or an index: see 'Database OPX' in the 'OPX.pdf' document) is created in it, so you should first close the database, i.e. close any tables previously opened in it, before using CREATE.

LOGICAL NAMES

You can have up to 26 views on tables open at a time on the Series 5. Each of these must have a logical name: `A` to `Z` (the Series 3c only supported 4 files open at one time).

FIELDS

On the Series 5, field names may be up to 32 characters long, including any qualifier like &.

OPENING DATABASES AND TABLES

With the Series 3c OPEN statement,

```
OPEN "datafile",A,f1%,f2%
```

the Series 5 would open the default table `Table1` and provide access to as many fields as there are handles supplied.

On the Series 5, it is also possible to open multiple *views* on a table simultaneously and to specify which fields are to be available in a view, e.g.

```
OPEN "people SELECT name FROM phoneBook",A,n$
```

This view gives you access to just the `name` field from the `phoneBook` table.

The string from `SELECT` onwards in the OPEN statement forms an SQL query which is passed straight on to the underlying EPOC32 DBMS. The SQL command-set is specified in Appendix F in the 'Appends.pdf' document.

A more advanced view, ordered by an *index* (described later), would be opened as follows,

```
OPEN "people SELECT name,number FROM phoneBook ORDER BY name ASC, number
DESC",A,n$,num%
```

This would open a view with `name` fields in ascending alphabetical order and if any names were the same then the number field would be used to order these records in descending numerical order.

TRANSACTIONS

A set of related records should be committed only on successfully PUTting the last one. Otherwise all new records may be discarded using ROLLBACK. This ensures the atomicity of the whole transaction.

Transactions allow changes to a database to be *committed* in stages. It is necessary to use transactions in database operations to achieve reasonable speeds.

Transactions are a truly fundamental part of the DBMS model, so much so that without the use of transactions you will find that writing to a DBMS database is in fact slower than the equivalent operations in on the Series 3c. With transactions however, the Series 5 database handling is far faster than that of the Series 3c.

A transaction is carried out using the following commands:

- BEGINTRANS begins a transaction on the current database. Once a transaction has been started on a view (or table) then all database keywords will function as usual, but the changes to that view will not be made until COMMITTRANS is used.
- COMMITTRANS commits the transaction of the current view.
- ROLLBACK cancels the current transaction on the current view. Changes made to the database with respect to this particular view since BEGINTRANS was called will be discarded.
- INTRANS finds out whether the current view is in a transaction.

RECORD POSITION

In the DBMS model, as with most modern relational database models, absolute record position does not have much significance.

Bookmarks can be assigned to particular records to provide fast record access **and should be used in preference to POS and POSITION** when opening views using the Series 5 `OPEN...SELECT..` or `CREATE ... FIELDS...` statements. POS and POSITION can be used safely on tables opened or created using a Series 3c-style OPEN or CREATE statement. However, POS and POSITION should **not** be used in conjunction with bookmarks as bookmarks can cause these keywords, kept mainly for Series 3c compatibility, to become inaccurate. Note that if bookmarks are used in conjunction with POS and POSITION accuracy can be restored by using FIRST or LAST on the current view.

The new commands provided for the use of bookmarks are as follows: BOOKMARK puts a bookmark at the current record of the current database view. The value returned can be passed to GOTOMARK to make the record current again and to KILLMARK to delete the bookmark.

SAVING RECORDS

When using the Series 5 extensions to CREATE and OPEN, you should also use the new MODIFY, INSERT, PUT and CANCEL keywords in preference to the APPEND and UPDATE Series 3c commands. APPEND and UPDATE will still work as expected, but do not naturally fit in the DBMS model.

- MODIFY allows records to be changed without being moved to the end of the set (as UPDATE still does).
- Instead of copying the current record to the end of the set as APPEND does, INSERT appends a new record to the end of the set with numeric fields set to 0 and string fields empty if values have not been assigned to them.
- PUT marks the end of a database's INSERT or MODIFY phase and makes the changes permanent.
- CANCEL marks the end of a database's INSERT or MODIFY phase and discards the changes made during that phase.

THE NUMBER OF RECORDS

The COUNT function returns the number of records in the file. If you try to count the number of records between assignment and APPEND/UPDATE or between MODIFY/INSERT and PUT an 'Incompatible update mode' error will be raised.

CLOSING VIEWS AND DATABASES

CLOSE closes the current view on a database. If there are no other views open on the database then the database itself will be closed.

INDEXES

Indexes can be constructed on a table using several fields as *keys.* These indexes are subsequently used to provide major speed improvements when opening a table or views on them.

Further database functionality is provided in the Database OPX, discussed in the ''OPX.pdf' document.

COMPACTION

COMPACT replaces the COMPRESS command on the Series 5. This compacts a database, rewriting the file in place without any removed or deleted data. All views on the database and the hence the file itself should be closed before calling this command. Compaction may also be done automatically on closing a file by setting the automatic compaction flag using SETFLAGS. See the 'Alphabetic Listing' section of the 'Glossary.pdf' document for full details of this.

OPENING A DATABASE CREATED BY THE DATA APPLICATION

It is currently not possible to open an OPL database from the Data application. You can however open a file created by the Data application in an OPL program. The file is opened for reading only, because if it were written to, OPL would have to discard all the formatting characters and prevent the Data application from reopening the file subsequently. An OPL program can create a new OPL database and copy the Data application records into it if necessary.

To open a Data application database that has one string field which you need to access, you could use:

```
OPEN "file",a,a$
```

Types not supported by OPL will be ignored. Note that integer fields in the Data application correspond to long integer fields in OPL: the Data application does not support (16-bit) integer fields. The types and order of the OPL field handles must match the fields in the Data file. For example, if the data file `Data2` contains:

1. long integer field
2. date/time field (ignored by OPL)
3. string field
4. floating-point number field

you could access the fields supported by OPL using:

```
OPEN "Data2",A, f1&,f2$,f3
```

It would be better, however, to use the SQL SELECT clause to name the required Data file fields explicitly. For this to be possible it is necessary to use table name and the same field names as are used by Data. All Data files have a single table called `Table1`. The fields (referred to internally as columns in Data) are named `ColA1`, `ColA2`, etc.

So, with the field types from the previous example, the Data file could be opened using:

```
OPEN "Data2 SELECT ColA1,ColA3,ColA4 FROM Table1",a,f1&,f2$,f3
```

INDEX

R

T

U

W