OPL

USING OPXS ON THE SERIES 5

The Series 5 uses language extensions provided in separate DLLs written specially for OPL support. These DLLs have the file extension OPX.

It is not within the scope of this User Guide to cover how to develop OPXs yourself, as it requires knowledge of the C++ language. If you require details of how to do this, you should obtain a copy of the EPOC32 C++ Software Development Kit (SDK) from Psion Software plc.

OPX procedures for handling the following are provided in the ROM:

- **Date / time extras**
- **System controls**
- **Bitmaps**
- **Sprites**
- **Database extras**
- **Printing**

CONTENTS

OPX OVERVIEW

A maximum of 255 OPXs can be used in any one OPL module and any OPX may have up to 65535 procedures defined in it. If these values are exceeded then errors are reported at translate-time (see the 'Errors.pdf' document).

OPX HEADER FILES

The OPX supplier provides an OXH header file. This provides the declaration for the OPX, specifying its name UID and version number (see below), followed by its procedure prototypes.

The new INCLUDE keyword is used to include these header files. Alternatively, the declaration can be inserted directly into the OPL file, i.e.

```
DECLARE OPX <opxName>,<uid>,<version>
  <protoType1> : <ordinal1>
  <protoType2> : <ordinal2>
  ...
END DECLARE
```

where,

`<name>` is the name of the OPX without the .OPX extension. The OPX is stored in a `\System\Opx\` folder on any drive, with the drives scanned from `Y:` to `A:` and then `Z:` if no path is specified. This allows ROM OPXs (in `Z:`) to be overridden if required.

`<uid>` is the UID of the OPX. The specification of a UID as well as a name guards against OPXs with the same name being confused, which could otherwise cause serious problems. The UID is checked on loading the OPX and a 'Not supported' error will result if the UIDs in the header file and in the OPX itself do not match.

`<version>` is the version number of the OPX. The version number of an OPX will be increased when any new procedures are added. OPL will refuse to load an OPX which reports that it can't support the version number given in the declaration. The version number expressed in hex is e.g. $0100 to represent version 1.00, $0102 for version 1.02 etc. In general, an OPX supplier will increment the 2 low digits (the so-called minor version number) for backward compatible changes, and will increment the 2 high digits for major incompatible changes.

`<prototype>` specifies the name, return type and parameters of an OPX procedure in the same way as for an OPL procedure when using the EXTERNAL keyword. Numeric parameters to OPX procedure can be passed *by reference* using BYREF. This means that the value of the variable **can** be changed by the OPX procedure. The OPX procedure prototype is followed additionally by a colon and then the

`<ordinal>` specifies the ordinal number of the procedure in the OPX itself. This is used to call the correct procedure, i.e. the OPX is *linked by ordinal*.

If an OPX procedure name clashes with one of your OPL procedures, or with that of another OPX procedure, you can make a copy of the OXH file and change the name of the offending procedures, but **not** the return type, parameter list or ordinal. You should then include this new OXH file in your module and the new name can then be used to refer to the procedure in your code.

For example, consider the OPX declaration,

```
DECLARE OPX XXOpx,XXOpxUid&,XXOpxVersion&
  XXClose%:(id&) : 1
  ProcWithLongParam:(aLong&) : 2
  AddOneToParams:(BYREF par1%,BYREF par2&, BYREF par3) : 3
END DECLARE
```

If a short integer is passed to `ProcWithLongParam:`, the translator will automatically convert it to a long integer.

The `AddOneToParams:` procedure adds one to each of the parameters. This is possible because the parameters are passed by reference using BYREF. Parameters passed by reference must be variables rather than constant values because the OPX will write back to the variable. The variable type must always be the same as given in the declaration.

CALLBACKS FROM OPX PROCEDURES

An OPX procedure can call back to a module procedure. This is often useful if the OPX needs some information that is not known when the OPX procedure is initially called, or if it requires a large amount of data which needs to be sent piecemeal.

The OPX provider will specify the exact form of the procedure which you must provide.

OPXS INCLUDED IN THE SERIES 5

Procedures for handling the following are provided in the ROM:

- Date / time extras
- System - a variety of procedures for system control, e.g. backlight control, sound control, file and application control, etc.
- Bitmaps - for use with buttons and Sprites.
- Sprites
- Database extras

These OPXs and their OXHs have default paths in the ROM (`Z:`), but the full path for any OPX may be supplied. These are `\System\Opl\` for the header files and `\System\Opx\` for the OPXs themselves. The OPX header files are stored in the ROM, but may be created in RAM by using the ‘Create standard files’ option in the ‘Tools’ menu in the Program editor.

With these OPXs, the OPL programmer is sometimes given direct access to objects via pointers (for efficiency). Otherwise sprites, for example, could not be set up using an array of IDs. These objects can be explicitly deleted to free memory or else they will be deleted when the program exits.

DATE OPX

To use this OPX, you must included the header file Date.oxh, which contains the following declaration:

```
DECLARE OPX DATE,KUidOpxDate&,KOpxDateVersion%
   DTNewDateTime&:(year&,month&,day&,hour&,minute&,second&,micro&) : 1
   DTDeleteDateTime:(id&) : 2
   DTYear&:(id&) : 3
   DTMonth&:(id&) : 4
   DTDay&:(id&) : 5
   DTHour&:(id&) : 6
   DTMinute&:(id&) : 7
   DTSecond&:(id&) : 8
   DTMicro&:(id&) : 9
   DTSetYear:(id&,year&) : 10
   DTSetMonth:(id&,month&) : 11
   DTSetDay:(id&,day&) : 12
   DTSetHour:(id&,hour&) : 13
   DTSetMinute:(id&,minute&) : 14
   DTSetSecond:(id&,second&) : 15
   DTSetMicro:(id&,micro&) : 16
   DTNow&: : 17
   DTDateTimeDiff:(start&,end&,BYREF year&,BYREF month&,BYREF day&,
          BYREF hour&, BYREF minute&, BYREF  second&,BYREF micro&) : 18
   DTYearsDiff&:(st art&,end&) : 19
   DTMonthsDiff&:(start&,end&) : 20
   DTDaysDiff&:(start&,end&) : 21
   DTHoursDiff&:(start&,end&) : 22
   DTMinutesDiff&:(start&,end&) : 23
   DTSecsDiff&:(start&,end&) : 24
   DTMicrosDiff&:(start&,end&) : 25
   DTWeekNoInYear&:(id&,yearstart&,rule&) : 26
   DTDayNoInYear&:(id&,yearstart&) : 27
   DTDayNoInWeek&:(id&) : 28
   DTDaysInMonth&:(id&) : 29
   DTSetHomeTime:(id&) : 30
   LCCountryCode&: : 31
   LCDecimalSeparator$: : 32
   LCSetClockFormat:(format&) : 33
   LCClockFormat&: : 34
   LCStartOfWeek&: : 35
   LCThousandsSeparator$: : 36
END DECLARE
```

DTNEWDATETIME&:

Usage: `id&=DTNEWDATETIME&:(year&,month&,day&,hour&,minute&,second&,micro&)`

Creates a new date/time object which contains all the supplied date/time components and returns a handle `id&` for it.

The year is stored as the usual four figure year, e.g. 1997.

The month is stored as 1 for January, 2 for February, etc.

The day is stored as the day number in the month.

The hour is the hour of the day in 12 or 24 hour clock according to the system setting.

The minutes, seconds and microseconds are stored as the usual values 0 to 59 for minutes and seconds and 0 to 999 for microseconds.

See DTDELETEDATETIME:.

DTDELETEDATETIME:

Usage: `DELETEDATETIME:(id&)`

Deletes the date/time object with handle `id&`. This should be called when a date/time object is no longer needed. Date/time objects will be deleted automatically on unloading the Date OPX or the module which uses it.

See DTNEWDATETIME&:, DTNOW&:.

DTYEAR&:

Usage: `y&=DTYEAR&:(id&)`

Returns the year component `y&` which is stored in the date/time object with handle `id&`.

See DTNEWDATETIME&:.

DTMONTH&:

Usage: `m&=DTMONTH&:(id&)`

Returns the month component `m&` which is stored in the date/time object with handle `id&`.

See DTNEWDATETIME&:.

DTDAY&:

Usage: `day&=DTDAY&:(id&)`

Returns the day component `day&` which is stored in the date/time object with handle `id&`.

See DTNEWDATETIME&:.

DTHOUR&:

Usage: `h&=DTHOUR&:(id&)`

Returns the hour component `h&` which is stored in the date/time object with handle `id&`.

See DTNEWDATETIME&:

DTMINUTE&:

Usage: `m&=DTMINUTE&:(id&)`

Returns the minutes component `m&` which is stored in the date/time object with handle `id&`.

See DTNEWDATETIME&:.

DTSECOND&:

Usage: `s&=DTSECOND&:(id&)`

Returns the seconds component `s&` which is stored in the date/time object with handle `id&`.

See DTNEWDATETIME&:.

DTMICRO&:

Usage: `m&=DTMICRO&:(id&)`

Returns the microseconds component `m&` which is stored in the date/time object with handle `id&`.

See DTNEWDATETIME&:.

DTSETYEAR:

Usage: `DTSETYEAR:(y&,id&)`

Sets the year component which is stored in the date/time object with handle `id&` to `y&`.

See DTNEWDATETIME&:.

DTSETMONTH:

Usage: `DTSETMONTH:(m&,id&)`

Sets the month component which is stored in the date/time object with handle `id&` to `m&`.

See DTNEWDATETIME&:.

DTSETDAY:

Usage: `DTSETDAY:(day&,id&)`

Sets the day component which is stored in the date/time object with handle `id&` to `day&`.

See DTNEWDATETIME&:.

DTSETHOUR:

Usage: `DTSETHOUR:(h&,id&)`

Sets the hour component which is stored in the date/time object with handle `id&` to `h&`.

See DTNEWDATETIME&:.

DTSETMINUTE:

Usage: `DTSETMINUTE:(m&,id&)`

Sets the minutes component which is stored in the date/time object with handle `id&` to `m&`.

See DTNEWDATETIME&:.

DTSETSECOND:

Usage: `DTSETSECOND:(s&,id&)`

Sets the seconds component which is stored in the date/time object with handle `id&` to `s&`.

See DTNEWDATETIME&:.

DTSETMICRO:

Usage: `DTSETMICRO:(m&,id&)`

Sets the microseconds component which is stored in the date/time object with handle `id&` to `m&`.

See DTNEWDATETIME&:.

DTNOW&:

Usage: `id&=DTNOW&:`

Creates a new date/time object which contains all the date/time components of the current time and returns a handle `id&` for it.

Example: Timing a loop

```
...
start&=DTNow&:
WHILE condition
...
ENDWH
end&=DTNow&:
PRINT “Time to do loop was”,DTMicrosDiff&:(start&,end&)
...
```

See DTNEWDATETIME&:.

DTDATETIMEDIFF:

Usage: `DTDATETIMEDIFF:(start&,end&,BYREF year&,BYREF month&,BYREF day&,BYREF hour&,BYREF minute&,BYREF second&,BYREF micro&)`

Calculates the exact difference between two date/time objects with handles `start&` and `end&` in the form of a date/time object. The difference is returned in the variables `year&`, `month&` etc.

DTYEARSDIFF&:

Usage: `diff&=DTYEARSDIFF&:(start&,end&)`

Returns the difference `diff&` in whole years between the two date/time objects with handles `start&` and `end&`.

See DTNEWDATETIME&:.

DTMONTHSDIFF&:

Usage: `diff&=DTMONTHSDIFF&:(start&,end&)`

Returns the difference `diff&` in whole months between the two date/time objects with handles `start&` and `end&`.

See DTNEWDATETIME&:.

DTDAYSDIFF&:

Usage: `diff&=DTDAYSDIFF&:(start&,end&)`

Returns the difference `diff&` in whole days between the two date/time objects with handles `start&` and `end&`.

See DTNEWDATETIME&:.

DTHOURSDIFF&:

Usage: `diff&=DTHOURSDIFF&:(start&,end&)`

Returns the difference `diff&` in whole hours between the two date/time objects with handles `start&` and `end&`.

See DTNEWDATETIME&:.

DTMINUTESDIFF&:

Usage: `diff&=DTMINUTESDIFF&:(start&,end&)`

Returns the difference `diff&` in whole minutes between the two date/time objects with handles `start&` and `end&`.

See DTNEWDATETIME&:.

DTSECONDSDIFF&:

Usage: `diff&=DTSECONDSDIFF&:(start&,end&)`

Returns the difference `diff&` in whole seconds between the two date/time objects with handles `start&` and `end&`.

See DTNEWDATETIME&:.

DTMICROSDIFF&:

Usage: `diff&=DTMICROSDIFF&:(start&,end&)`

Returns the difference `diff&` in whole microseconds between the two date/time objects with handles `start&` and `end&`.

See DTNEWDATETIME&:, DTNOW&:.

DTWEEKNOINYEAR&:

Usage: `w&=DTWEEKNOINYEAR&:(id&,yearstart&,rule&)`

Returns the week number in the year of the date/time object with handle `id&`. The first day of the year is specified by the date/time object with handle `yearstart&`. This would usually be set to 1 January in the appropriate year, but also allows you to set the start of the year to the beginning of the financial year or the academic year, for example.

`rule&` can take three values (0,1,2), allowing the week number to be calculated by one of three different rules:

value	*meaning*
0	the first day of the year is always in week one,
1	requires the first week of the year to have at least four days in it,
2	requires the first week of the year to have the full seven days.

The Agenda application and other Series 5 applications use the rule with value 1 by default.

DTDAYNOINYEAR&:

Usage: `DTDAYNOINYEAR&:(id&,yearstart&)`

Returns the day number in the year of the date/time object with handle `id&`. The first day of the year is specified by the date/time object with handle `yearstart&`.

DTDAYNOINWEEK&:

Usage: `n&=DTDAYNOINWEEK&:(id&)`

Returns the day number in the week (1-7) of the date/time object with handle `id&`.

DTDAYSINMONTH&:

Usage: `n&=DTDAYSINMONTH&:(id&)`

Returns the number of days in the month of the month specified by the month component of the date/time object with handle `id&`.

DTSETHOMETIME:

Usage: `DTSETHOMETIME:(id&)`

Sets the system time to the time specified in the date/time object with handle `id&`.

LCCOUNTRYCODE&:

Usage: `cc&=LCCOUNTRYCODE&:`

Returns the country code for the current system home country (LC stands for 'Locale'), which may be used to select country-specific data. The country code for any given country is the international dialling prefix for that country

LCDECIMALSEPARATOR$:

Usage: `decSep$=LCDECIMALSEPARATOR$:`

Returns the decimal separator (the character used in decimal numbers to separate whole part from fractional part) according to the local system setting.

LCSETCLOCKFORMAT:

Usage: `LCSETCLOCKFORMAT:(format&)`

Sets the system clock format to either digital or analog. If `format&=0` then the clock is set to analog or if it is 1 then the clock is set to digital.

LCCLOCKFORMAT&:

Usage: `format&=LCCLOCKFORMAT&:`

Returns the current system clock format The procedure returns 0 if the system clock is analog and 1 if it is digital.

LCSTARTOFWEEK&:

Usage: `start&=LCSTARTOFWEEK&:`

Returns the day of the week which set as the first day of the week in the system setting. A return value of 1 indicates Monday, 2 Tuesday, and so on.

LCTHOUSANDSSEPARATOR$:

Usage: `thouSep$=LCTHOUSANDSSEPARATOR$:`

Returns the thousands separator (the character used to separate every three digits of a large number) according to the local system setting.

SYSTEM OPX

To use this OPX, you must included the header file System.oxh, which contains the following declaration:

```
DECLARE OPX SYSTEM,KUidOpxSystem&,KOpxSystemVersion%
  BackLightOn&: : 1
  SetBackLightOn:(state&) : 2
  SetBackLightOnTime:(seconds&) : 3
  SetBacklightBehavior:(behaviour&) : 4
  IsBacklightPresent&: : 5
  SetAutoSwitchOffBehavior:(behaviour&) : 6
  SetAutoSwitchOffTime:(seconds&) : 7
  SetActive:(state&) : 8
  ResetAutoSwitchOffTimer: : 9
  SwitchOff: : 10
  SetSoundEnabled:(state&) : 11
  SetSoundDriverEnabled:(state&) : 12
  SetKeyClickEnabled:(state&) : 13
  SetPointerClickEnabled:(state&) : 14
  SetDisplayContrast:(value&) : 15
  MaxDisplayContrast&: : 16
  IsReadOnly&:(file$) : 17
  IsHidden&:(file$) : 18
  IsSystem&:(file$) : 19
  SetReadOnly:(file$,state&) : 20
  SetHiddenFile:(file$,state&) : 21
  SetSystemFile:(file$,state&) : 22
  VolumeSize&:(drive&) : 23
  VolumeSpaceFree&:(drive&) : 24
  VolumeUniqueID&:(drive&) : 25
  MediaType&:(drive&) : 26
  GetFileTime:(file$,DateTimeId&) : 27
  SetFileTime:(file$,DateTimeId&) : 28
  DisplayTaskList: : 29
  SetComputeMode:(State&) : 30
  RunApp&:(lib$,doc$,tail$,cmd&) : 31
  RunExe&:(name$) : 32
  LogonToThread:(threadId&,BYREF statusWord&) : 33
  TerminateCurrentProcess:(reason&) : 34
  TerminateProcess:(proc$,reason&) : 35
  KillCurrentProcess:(reason&) : 36
  KillProcess:(proc$,reason&) : 37
  PlaySound:(file$,volume&) : 38
  PlaySoundA:(file$,volume&,BYREF statusWord&) : 39
  StopSound&: : 40
  Mod&:(left&,right&) : 41
```

```
  XOR&:(left&,right&) : 42
  LoadRsc&:(file$) : 43
  UnLoadRsc:(id&) : 44
  ReadRsc$:(id&) : 45
  ReadRscLong&:(id&) : 46
  CheckUid$:(Uid1&,Uid2&,Uid3&) : 47
  SetPointerGrabOn:(WinId&,state&) : 48
  MachineName$: : 49
  MachineUniqueId:(BYREF high&,BYREF low&) : 50
  EndTask&:(threadId&,previous&)  : 51
  KillTask&:(threadId&,previous&) : 52
  GetThreadIdFromOpenDoc&:(doc$,BYREF previous&) : 53
  GetThreadIdFromAppUid&:(uid&,BYREF previous&) : 54
  SetForeground: : 55
  SetBackground: : 56
  SetForegroundByThread&:(threadId&,previous&) : 57
  SetBackgroundByThread&:(threadId&,previous&) : 58
  GetNextWindowGroupName$:(threadId&,BYREF previous&) : 59
  GetNextWindowId&:(threadId&,previous&) : 60
  SendKeyEventToApp&:(threadId&,previous&,code&,scanCode&,
  modifiers&,repeats&) : 61
  IrDAConnectToSend&:(protocol$,port&) : 62
  IrDAConnectToReceive:(protocol$,port&,BYREF statusW&) : 63
  IrDAWrite:(chunk$,BYREF statusW&) : 64
  IrDARead$: : 65
  IrDAReadA:(stringAddr&,BYREF statusW&): 66
  IrDAWaitForDisconnect: : 67
  IrDADisconnect: : 68
  MainBatteryStatus&: :69
  BackupBatteryStatus&: :70
  CaptureKey&:(keyCode&,mask&,modifier&) :71
  CancelCaptureKey:(handle&) :72
  SetPointerCapture:(winId&,flags&) :73
  ClaimPointerGrab:(winId&,state&) :74
  OpenFileDialog$:(seedFile$,uid1&,uid2&,uid3&) : 75
  CreateFileDialog$:(seedPath$) : 76
  SaveAsFileDialog$:(seedPath$,BYREF useNewFile%) : 77
END DECLARE
```

BACKLIGHTON&:

Usage: `backLight&=BACKLIGHTON&:`

Returns -1 if the backlight is switched on or 0 if it is switched off.

SETBACKLIGHTON:

Usage: `SETBACKLIGHTON:(state&)`

Switches the backlight on if `state&=1` or off if `state&=0`.

SETBACKLIGHTONTIME:

Usage: `SETBACKLIGHTONTIME:(seconds&)`

Sets the time in seconds (`seconds&`) that the backlight should remain on after it has been switched on.

SETBACKLIGHTBEHAVIOR:

Usage: `SETBACKLIGHTBEHAVIOR:(behavior&)`

Sets the backlight's turning off behaviour.

`behavior&=0` sets the turning off of the backlight to be on a timer,

`behavior&=1` sets the backlight not to be on a timer.

ISBACKLIGHTPRESENT&:

Usage: `ret&=ISBACKLIGHTPRESENT&:`

Returns -1 if there is a backlight present and 0 if there is not.

SETAUTOSWITCHOFFBEHAVIOR:

Usage: `SETAUTOSWITCHOFFBEHAVIOR:(behavior&)`

Sets the machine's auto switch off behaviour.

`behavior&=0`	disables the machine's auto switch off mechanism.
`behavior&=1`	sets the machine auto switch off to occur only when its batteries are low.
`behavior&=2`	sets the machines auto switch off to occur always.

SETAUTOSWITCHOFFTIME:

Usage: `SETAUTOSWITCHOFFTIME:(seconds&)`

Sets the time in seconds (`seconds&`) for which the machine may remain switched on when it is not being used.

SETACTIVE:

Usage: `SETACTIVE:(state&)`

Sets the current process active if `state&=1` or not active if `state&=0`. This will determine whether or not the machine can automatically turn off when the user is not using the machine: if the process is active then the machine will not automatically turn off.

RESETAUTOSWITCHOFFTIMER:

Usage: `RESETAUTOSWITCHOFFTIMER:(seconds&)`

'Tickles' the machine's auto switch off timer, restarting its count down to switching off.

SWITCHOFF:

Usage: `SWITCHOFF:`

Switches off the machine.

As with the keyword OFF, you should be careful not to use SWITCHOFF: in a loop. If you do, it may be impossible to switch the Series 5 back on, and you may then have to reset it.

SETSOUNDENABLED:

Usage: `SETSOUNDENABLED:(state&)`

Enables the machine's sound if `state&=1` or disables it if `state&=0`.

SETSOUNDDRIVERENABLED:

Usage: `SETSOUNDDRIVERENABLED:(state&)`

Switches the machines sound driver on if `state&=1` or off if `state&=0`.

SETKEYCLICKENABLED:

Usage: `SETKEYCLICKENABLED:(state&)`

Determines whether or not a keypress makes a click. `state&=1` enables the click and `state&=0` disables it.

SETPOINTERCLICKENABLED:

Usage: `SETPOINTERCLICKENABLED:(state&)`

Determines whether or not a pointer event makes a click. (A pointer event occurs whenever the machine's screen is pressed.) `state&=1` enables the click and `state&=0` disables it.

SETDISPLAYCONTRAST:

Usage: `SETDISPLAYCONTRAST:(value&)`

Sets the contrast on the machine's screen. `value&` specifies the contrast value, which can be between zero and the maximum display contrast.

See MAXDISPLAYCONTRAST&:.

MAXDISPLAYCONTRAST&:

Usage: `maxContrast&=MAXDISPLAYCONTRAST&:`

Returns the maximum value to which the machines display contrast can be set.

See SETDISPLAYCONTRAST:.

ISREADONLY&:

Usage: `readOnly&=ISREADONLY&:(file$)`

Returns -1 if the file, `file$`, is a read-only file and 0 if it is not.

ISHIDDEN&:

Usage: `hidden&=ISHIDDEN&:(file$)`

Returns -1 if the file, `file$`, is hidden by the system and 0 if it is not.

ISSYTEM&:

Usage: `system&=ISSYTEM&:(file$)`

Returns -1 if the file, `file$`, is a system file and 0 if it is not.

SETREADONLY:

Usage: `SETREADONLY:(file$,state&)`

Sets the file, `file$`, to be read only if `state&=1` or not read-only if `state&=0`.

SETHIDDENFILE:

Usage: `SETHIDDENFILE:(file$,state&)`

Sets the file, `file$`, to be hidden by the system if `state&=1` or not to be hidden if `state&=0`.

SETSYSTEMFILE:

Usage: `SETSYSTEMFILE:(file$,state&)`

Sets the file, `file$`, to be a system file if `state&=1` or not to be a system file if `state&=0`.

VOLUMESIZE&:

Usage: `VOLUMESIZE&:(drive&)`

Returns the size in bytes of the storage space on the drive specified by `drive&`. `drive&` can take values 0 to 25. 0 specifies the `A:`, 1 specifies `B:`, 2 specifies `C:` and so on.

VOLUMESPACEFREE&:

Usage: `free&=VOLUMESPACEFREE&:(drive&)`

Returns the size in bytes of the storage space that is available for use on the drive specified by `drive&`. `drive&` can take values 0 to 25. 0 specifies the `A:`, 1 specifies `B:`, 2 specifies `C:` and so on.

VOLUMEUNIQUEID&:

Usage: `volUid&=VOLUMEUNIQUEID&:(drive&)`

Returns the unique identification number of the media on the drive specified by `drive&`. `drive&` can take values 0 to 25. 0 specifies the `A:`, 1 specifies `B:`, 2 specifies `C:` and so on.

See MEDIATYPE&:.

MEDIATYPE&:

Usage: `media&=MEDIATYPE&:(drive&)`

Returns the media type present on the drive specified by `drive&`. `drive&` can take values 0 to 25. 0 specifies the `A:`, 1 specifies `B:`, 2 specifies `C:` and so on. The value returned may be any of the following,

value	*meaning*	*constant declaration in System.oxh*
0	no media present	`CONST KMediaNotPresent&=0`
1	media unknown	`CONST KMediaUnknown&=1`
2	floppy disk	`CONST KMediaFloppy&=2`
3	hard disk	`CONST KMediaHardDisk&=3`
4	CD-ROM	`CONST KMediaCdRom&=4`
5	RAM	`CONST KMediaRam&=5`
6	flash	`CONST KMediaFlash&=6`
7	ROM	`CONST KMediaRom&=7`
8	remote	`CONST KMediaRemote&=8`

GETFILETIME:

Usage: `GETFILETIME:(file$,dateTimeId&)`

Returns the time that the file, `file$`, was last modified into `dateTimeId&`. It is necessary to pass this procedure the ID of a date/time object which the procedure will then set to the required time. To obtain and read a date/time object, see the ‘Date OPX’ section.

SETFILETIME:

Usage: `SETFILETIME:(file$,dateTimeId&)`

Sets the time that the file, `file$`, was last modified to `dateTimeId&`. It is necessary to pass this procedure the ID of a date/time object which the procedure will use to set the time. To get a date/time object, which provides microsecond accuracy, see the ‘Date OPX’ section.

DISPLAYTASKLIST:

Usage: `DISPLAYTASKLIST:`

Displays the system-wide task list. The user may then close a file, go to another task or close the dialog.

SETCOMPUTEMODE:

Usage: `SETCOMPUTEMODE:(state&)`

Changes the priority control for the current program.

The following values are available for `state&`:

value	*meaning*	*constant declaration in System.oxh*
0	compute mode disabled	`CONST KComputeModeDisabled&=0`
1	compute mode on	`CONST KComputeModeOn&=1`
2	compute mode off	`CONST KComputeModeOff&=2`

This puts the current process into compute mode (`state&=KComputeModeOn&`) or takes it out of compute mode (`state&=KComputeModeOff&`). In compute mode, a process runs at the lower background priority even when it is the foreground process. Disabling compute mode control (`state&=KComputeModeDisabled&`) prevents the window server from changing the program's priority when it moves between background and foreground.

OPL runs in compute mode by default to support simple OPOs which cannot always be assumed to be well-behaved programs.

This default behaviour is necessary because OPL programs would not otherwise give any background programs a chance to run, simply by running in a tight loop. If your program doesn't run in a tight loop, i.e. if it calls GETEVENT32, GET, etc. regularly, you can use `SETCOMPUTEMODE:(KComputeModeOff&)`.

Note that TBarInit: in Toolbar.opo sets compute mode off, since any program that has a toolbar shouldn't be running in a tight loop at any time. (See the 'Friendlier Interaction' section of the 'GUI.pdf' document for more details.)

RUNAPP&:

Usage: `thread&=RUNAPP&:(lib$,doc$,tail$,cmd&)`

Runs an application and returns a thread ID for the application. The thread ID can be used to logon to the application, to find out when and why it finished. This ID can also be used to locate the window group, end the task etc.

`lib$` is the C++ application name.

`doc$` is the document name, if any, to pass to the application.

`tail$` is the tail end, needed only by certain applications such as OPL.

`cmd&` can take values:

value	*meaning*
0	open
1	create
2	run
3	background

The values 0, 1 and 2 are the same as for `CMD$(3)` (see the 'Alphabetic Listing' section of the 'Glossary.pdf' document). The value 3 will run the application in the background.

For example, to run an OPL program:

```
k&=RUNAPP&:("OPL","","RZ:\System\Opl\Toolbar.opo",2)
```

Note that `tail$` contains a leading `R`: this is required by OPL.

To open the Data help file:

```
k&=RUNAPP&:("Data","Z:\System\Data\Help","",0)
```

See also the 'OPL Applications' section in the 'Advanced.pdf' document.

See LOGONTOTHREAD:.

RUNEXE&:

Usage: `RUNEXE&:(file$)`

Runs an executable EXE file `file$` and returns its thread ID. The thread ID can then be used to logon to the thread and find out when and why it finished.

See LOGONTOTHREAD:.

LOGONTOTHREAD:

Usage: `LOGONTOTHREAD:(threadId&,statusWord&)`

Logs on to the thread with ID `threadId&` and sets the status word `statusWord&` when the thread has completed. As for other 32-bit status words, `statusW&` will be set to &80000001 for pending and 0 for successful completion.

TERMINATECURRENTPROCESS:

Usage: `TERMINATEPROCESS:(reason&)`

Terminates the current process giving it the reason `reason&`. This causes the process to receive a shutdown message. `reason&` may take any value, with zero used to mean no errors.

TERMINATEPROCESS:

Usage: `TERMINATEPROCESS:(proc$,reason&)`

Terminates the process `proc$` giving it the reason `reason&`. This causes the process to receive a shutdown message. `reason&` may take any value, with zero used to mean no errors.

KILLCURRENTPROCESS:

Usage: `KILLCURRENTPROCESS:(reason&)`

Kills the current process giving it the reason `reason&`. The process is killed **without** receiving a shutdown message. `reason&` may take any value, with zero used to mean no errors.

KILLPROCESS:

Usage: `KILLPROCESS:(proc$,reason&)`

Kills the process `proc$` giving it the reason `reason&`. The process is killed **without** receiving a shutdown message. `reason&` may take any value, with zero used to mean no errors.

PLAYSOUND:

Usage: `PLAYSOUND:(file$,volume&)`

Plays the file `file$`, which may be a sound file or an alarm file. Does not return until the sound has completed.

`volume&` specifies the volume at which the file should be played, 0 specifies no volume and 3 specifies maximum volume. The play will be synchronous (i.e. the OPL program will stop running until the file has finished playing). For asynchronous sound playing see PLAYSOUNDA:.

PLAYSOUNDA:

Usage: `PLAYSOUNDA:(file$,volume&,statusWord&)`

Plays the file `file$`, which may be a sound file or an alarm file, asynchronously. It returns immediately, with the status word being set on completion or cancellation of the sound.

`volume&` specifies the volume at which the file should be played, 0 specifies no volume and 3 specifies maximum volume. The play will be asynchronous (i.e. the OPL program will continue running while the sound file is playing). `statusWord&` is the variable which will contain information about the state of the sounds file play. For synchronous sound playing see PLAYSOUND:.

STOPSOUND&:

Usage: `STOPSOUND&:`

Stops the sound file that is currently playing. The return value is 1 if there was a sound file playing and 0 if not.

See PLAYSOUND:, PLAYSOUNDA:.

MOD&:

Usage: `rem&=MOD&:(left&,right&)`

Returns `left&` modulo `right&` (i.e. the remainder of `left&` divided by `right&`).

XOR&:

Usage: `res&=XOR&:(left&,right&)`

Returns the exclusive OR of `left&` and `right&`. A bit in the result has value 1 if the corresponding bit is set in **either** `left&` or in `right&`, but **not in both**, otherwise it is 0.

E.g. with `left&=5` (binary 00000101) and `right&=3` (binary 00000011), `XOR&:(left&,right&)` returns 6 (binary 00000110).

`left&`	00000101
`right&`	00000011
`XOR&:(left&,right&)`	00000110

This is often used to invert particular bits in an integer. So `left&=XOR&:(left&,3)` inverts bits 0 and 1 in `left&`, where bit 0 is the rightmost bit, and leaves the other bits alone.

LOADRSC&:

Usage: `id&=LOADRSC&:(file$)`

Loads the resource file `file$` and returns a handle for it.

See UNLOADRSC:, READRSC$:.

UNLOADRSC:

Usage: `UNLOADRSC:(id&)`

Unloads the resource file whose handle is `id&`.

See LOADRSC&:.

READRSC$:

Usage: `string$=READRSC$:(id&)`

Reads the resource in a resource file specified by `id&` which must already be loaded.

See LOADRSC&:.

READRSCLONG&:

Usage: `long&=READRSCLONG&:(id&)`

Reads a 32-bit value from a resource file specified by the `id&` which must already be loaded.

See LOADRSC&:.

CHECKUID$:

Usage: `CHECKUID$:(Uid1&,Uid2&,Uid3&)`

Returns a string which is a unique product of the three supplied UIDs.

SETPOINTERGRABON:

Usage: `SETPOINTERGRABON:(WinId&,state&)`

Enables (or disables) pointer grabs in a window. After this function has been called with `state&=1`, any "down" event in this window will cause a pointer grab, terminated by the next corresponding "up" event. The terminating "up" event is also sent to the window with the grab.

This function would typically be used for "drag-and-drop", and would typically be called after window creation so that pointer grab is enabled for the lifetime of the window.

`state&=0` disables the grab.

See CLAIMPOINTERGRAB:.

MACHINENAME$:

Usage: `name$=MACHINENAME$:`

Returns the machine name.

MACHINEUNIQUEID:

Usage: `MACHINEUNIQUEID:(BYREF high&,BYREF low&)`

Sets `high&` and `low&` (which are passed BYREF) to high and low parts of the machine's unique ID.

ENDTASK&:

Usage: `ret&=ENDTASK&:(threadId&,previous&)`

Sends a *shutdown* message to a window group in the thread, `threadId&`.

`previous&` specifies the window group in the thread that is previous to the one required. Hence, setting `previous&` to 0 will specify the first window group in the thread.

The value returned by this procedure is the value of the window group on which action was taken, or -1 if a window group following that specified by `previous&` was not present. This return value could be passed back to this procedure to end the next window group in the thread.

An error will be raised if the window group is busy, does not respond to such requests or is in a system thread.

See KILLTASK&:.

KILLTASK&:

Usage: `ret&=KILLTASK&:(threadId&,previous&)`

Shuts down the window group that follows the window group specified by `previous&` in the thread, `threadId&`. It will ignore any wishes of the window group not to be shutdown.

The value returned by this procedure is the value of the window group on which action was taken, or -1 if a window group following that specified by `previous&` was not present. This return value could be passed back to this procedure to end the next window group in the thread.

See ENDTASK&:.

GETTHREADIDFROMOPENDOC&:

Usage: `threadId&=GETTHREADIDFROMOPENDOC&:(doc$,BYREF previous&)`

Returns the thread ID `threadId&` of the thread that contains the open document `doc$`.

`doc$` should contain the full path of the open document and could for example be `c:\opl\prog.opo`.

`previous&`, passed by reference, is useful for situations where the document may be open more than once. Setting `previous&` to zero will cause this procedure to find the first window group that contains `doc$`. If one is found, `previous&` will be set to the window group value of that found document. This value could then passed back to this procedure as `previous&`, so the next window group instance of `doc$` will be found, and so on. If this procedure sets `previous&` to -1 then `doc$` could not be found after `previous&`. An error will also be raised if the open document is not found.

GETTHREADIDFROMAPPUID&:

Usage: `threadId&=GETTHREADIDFROMAPPUID&:(uid&,BYREF previous&)`

Returns the thread ID `threadId&` of the thread that contains a running instance of the application with UID `uid&`.

`previous&`, passed by reference, is useful for situations where the application may be running more than once. Setting `previous&` to zero will cause this procedure to find the first window group that is an instance of this application. If one is found, `previous&` will be set to the value of that window group. This value could then passed back to this procedure as `previous&`, so the next instance of the application can be found, and so on. If no instance of this application can be found following `previous&` then `previous&` will be set to -1. An error will also be raised if a running instance of the application is not found.

SETFOREGROUND:

Usage: `SETFOREGROUND:`

Moves the calling process to foreground.

See SETBACKGROUND:.

SETBACKGROUND:

Usage: `SETBACKGROUND:`

Moves the calling process to background.

See SETFOREGROUND:.

SETFOREGROUNDBYTHREAD&:

Usage: `ret&=SETFOREGROUNDBYTHREAD&:(threadId&,previous&)`

Moves the window group after `previous&` in the thread `threadId&` to foreground. Using `previous&=0` specifies the first window group in the thread.

The value returned will be the window group on which action was taken, or -1 if the following window group was not found. This return value could be passed back to this procedure as `previous&` to put the next window group in `threadId&` to foreground.

See SETBACKGROUNDBYTHREAD&:.

SETBACKGROUNDBYTHREAD&:

Usage: `ret&=SETBACKGROUNDBYTHREAD&:(threadId&,previous&)`

Moves the window group after `previous&` in the thread `threadId&` to background. Using `previous&=0` specifies the first window group in the thread.

The value returned will be the window group on which action was taken, or -1 if the following window group was not found. This return value could be passed back to this procedure again as a value for `previous&` to put the next window group in `threadId&` to background.

See SETFOREGROUNDBYTHREAD&:.

GETNEXTWINDOWGROUPNAME$:

Usage: `name$=GETNEXTWINDOWGROUPNAME$:(threadId&,BYREF previous&)`

Returns the name of the window group that follows the window group `previous&` in the thread `threadId&`. If `previous&=0` the procedure will return the name of the first window group found.

See GETNEXTWINDOWID&:.

GETNEXTWINDOWID&:

Usage: `winId&=GETNEXTWINDOWID&:(threadId&,BYREF previous&)`

Returns the ID of the window group that follows the window group `previous&` in the thread `threadId&`. If `previous&` is 0 the procedure will return the ID of the first window group found.

See GETNEXTWINDOWGROUPNAME$:.

SENDKEYEVENTTOAPP&:

Usage: `SENDKEYEVENTTOAPP&:(threadId&,previous&,code&,scanCode&,modifiers&,repeats&)`

Sends a key event to the window group (application) that follows the window group with ID `previous&` (or the first window group if `previous&=0`), in the thread with ID `threadId&`. The key event is specified by `code&`, `scanCode&`, `modifiers&` and `repeats&`.

The value returned is the window group to which the key was sent.

IRDACONNECTTOSEND&:

Usage: `IRDACONNECTTOSEND&:(protocol$,port&)`

Synchronously sends out an infrared (IR) beam which looks for another infrared device. If another infrared device is looking to receive a beam then the devices will connect, otherwise this procedure will raise an error after a couple of seconds.

`protocol$` can take either of the constants defined in System.oxh. These are `KIrTinyTP$`, used for communicating with other EPOC32 devices and `KIrmux$` which is used for IR printing.

The value 8 is recommended for `port&` when communicating with other EPOC32 devices. Both devices will need to be using the same value for `port&`. `port&` must be 2 for connecting to IR printers.

If connecting to another EPOC32 device with this procedure the other device must connect by using IRDACONNECTTORECEIVE&:.

After connecting to another EPOC32 device you can both send and receive information.

After connecting to an IR printer use IRDAWRITE: to send text. When IRDADISCONNECT: is called the printer will begin printing.

See IRDACONNECTTORECEIVE:, IRDAWRITE:, IRDAREAD:, IRDADISCONNECT:, IRDAWAITFORDISCONNECT:.

IRDACONNECTTORECEIVE:

Usage: `IRDACONNECTTORECEIVE:(protocol$,port&,BYREF statusW&)`

Asynchronously waits, without timing-out, for another EPOC32 device to attempt to connect via infrared (IR). This procedure takes the status word `statusW&`. When connection occurs the status word is set to zero.

Both EPOC32 devices must be using the same port which is specified by `port&`. The recommended port is 8.

One of the devices must connect with this procedure and the other with IRDACONNECTTOSEND&.

`protocol$` specifies the IR protocol and should take the constant `KIrTinyTP$`, defined in System.oxh, when communicating with other EPOC32 devices.

After connecting to another EPOC32 device you can both send and receive information.

See IRDAWRITE:, IRDAREAD:, IRDADISCONNECT:, IRDAWAITFORDISCONNECT:.

IRDAWRITE:

Usage: `IRDAWRITE:(chunk$, BYREF statusW&)`

Sends a string `chunk$` via infrared to another device after connection has been made. The procedure is asynchronous and the success of sending the string is reported in `statusW&`.

See IRDACONNECTTOSEND&:, IRDACONNECTTORECEIVE:, IRDAREAD;, IRDADISCONNECT:, IRDAWAITFORDISCONNECT:.

IRDAREAD$:

Usage: `chunk$=IRDAREAD$:`

Reads a text string via infrared from another device after connection has been made. The procedure is synchronous.

See IRDACONNECTTOSEND&:, IRDACONNECTTORECEIVE:, IRDWRITE:, IRDADISCONNECT:, IRDAWAITFORDISCONNECT:.

IRDAREADA:

Usage: `IRDAREADA:(stringAddr&, BYREF statusW&)`

Reads a text string via infrared from another device after connection has been made. The procedure is asynchronous with status word `statusW&`. The string passed by address should have a maximum length of 255 bytes. To pass a string by address use, `ADDR(string$)`.

See IRDACONNECTTOSEND&:, IRDACONNECTTORECEIVE:, IRDWRITE:, IRDADISCONNECT:, IRDAWAITFORDISCONNECT:.

IRDAWAITFORDISCONNECT:

Usage: `IRDAWAITFORDISCONNECT:`

Verifies, when disconnecting from another IR device, that the other device has received everything sent. It should therefore be called by the device that last sends some information.

This procedure should not be used with printers: use IRDADISCONNECT: instead.

See IRDACONNECTTOSEND&:, IRDACONNECTTORECEIVE:, IRDADISCONNECT:, IRDAWAITFORDISCONNECT:.

IRDADISCONNECT:

Usage: `IRDADISCONNECT:`

Disconnects from another IR device. It should be called by the device that is the last to receive some information. IRDAWAITFORDISCONNECT: is used by the last device to send some information, except when printing via IR when IRDADISCONNECT: should be used to disconnect from the printer and commence printing.

See IRDACONNECTTOSEND&:, IRDACONNECTTORECEIVE:, IRDADISCONNECT:, IRDAWAITFORDISCONNECT:.

MAINBATTERYSTATUS&:

Usage: `MAINBATTERYSTATUS&:`

Returns the current power of the main batteries. The following possible return values are supplied as constants in System.oxh:

```
CONST KBatteryZero&          = 0
CONST KBatteryVeryLow&       = 1
CONST KBatteryLow&           = 2
CONST KBatteryGood&          = 3
```

BACKUPBATTERYSTATUS&:

Usage: `BACKUPBATTERYSTATUS&:`

Returns the current power of the backup batteries. The following possible return values are supplied as constants in System.oxh:

```
CONST KBatteryZero&          = 0
CONST KBatteryVeryLow&       = 1
CONST KBatteryLow&           = 2
CONST KBatteryGood&          = 3
```

CAPTUREKEY&:

Usage: `CAPTUREKEY&:(keyCode&,mask&,modifier&)`

Allows the program that calls it to capture keys when it is not in foreground. The keys to be captured are specified using `keycode&`, `mask&` and `modifier&`. The value returned is a handle which can be passed to CANCELCAPTUREKEY: to cancel the capture.

The following constants are supplied in System.oxh to be used in conjunction with this procedure:

```
CONST KModifierAutorepeatable&  = &00000001
CONST KModifierKeypad&          = &00000002
CONST KModifierLeftAlt&         = &00000004
CONST KModifierRightAlt&        = &00000008
CONST KModifierAlt&             = &00000010
CONST KModifierLeftCtrl&        = &00000020
CONST KModifierRightCtrl&       = &00000040
CONST KModifierCtrl&            = &00000080
CONST KModifierLeftShift&       = &00000100
CONST KModifierRightShift&      = &00000200
CONST KModifierShift&           = &00000400
CONST KModifierLeftFunc&        = &00000800
CONST KModifierRightFunc&       = &00001000
CONST KModifierFunc&            = &00002000
CONST KModifierCapsLock&        = &00004000
CONST KModifierNumLock&         = &00008000
CONST KModifierScrollLock&      = &00010000
CONST KModifierKeyUp&           = &00020000
CONST KModifierSpecial&         = &00040000
CONST KModifierDoubleClick&     = &00080000
CONST KModifierPureKeycode&     = &00100000
CONST KAllModifiers&            = &001fffff
```

See CANCELCAPTUREKEY&:.

CANCELCAPTUREKEY:

Usage: `CANCELCAPTUREKEY:(handle&)`

Cancels an outstanding key capture which is specified by the handle `handle&`.

See CAPTUREKEY&:.

SETPOINTERCAPTURE:

Usage: `SETPOINTERCAPTURE:(WinId&,flags&)`

Sets the (OPL) window with window ID `winId&` to capture pointer events.

`flags&` can take a summed combination of the following:

value	*meaning*
0	capture disabled
&01	capture enabled
&02	capture (not drag-and-drop)

When a window captures pointer events, the position of any events received will be relative to its origin, i.e. the top left corner of that window, i.e. the window's 0,0 co-ordinate becomes the origin for pointer events. Events don't need to be inside the window to be registered. For example, if the screen was touched 1 pixel above and 1 pixel to the left of a window that was capturing pointer events, the pointer event would be returned with the position of the event being -1,-1 .

"Capture (not drag-and-drop)" will cause events of the "drag-and-drop" nature to be sent to the window that would have received them had the pointer not been captured.

CLAIMPOINTERGRAB:

Usage: `CLAIMPOINTERGRAB:(WinId&,state&)`

Claims the pointer grab for the window with ID `winId&` from another window, if a pointer grab is already in effect in the other window. All subsequent events will be delivered to this window, instead of the window that originally had the pointer grab. The next "up" event terminates the pointer grab, and is also delivered to the window that claimed the grab, if `state&` is set to 1.

This function would typically be used where clicking in a window pops up another window, and where the popped-up window wishes to grab the pointer as though the original click had been in that window.

To summarise the values of `state&` for the OPL window,

value	*meaning*
0	don't deliver the following "up" event to this window
1	deliver the following "up" event to this window

See SETPOINTERGRABON:

OPENFILEDIALOG$:

Usage: `file$=OPENFILEDIALOG$:(seedFile$,uid1&,uid2&,uid3&)`

Displays the standard 'Open file' dialog. `seedFile$` provides a starting file which is displayed when the dialog is opened. So, for example, if `seedFile$="C:\Documents\Myfile"` then the file `MyFile` will be initially displayed in the 'Name' selector box, the `Documents` folder will be initially displayed in the 'Folder' selector box and `C` will be displayed in the disk selector. You must always seed the dialog: you cannot pass a null `seedFile$` string to OPENFILEDIALOG$:. If `seedFile$` does not include a drive name, then an error is raised.

The selection of files may be restricted by UID by specifying appropriate values for `uid1&`, `uid2&` and `uid3&` in exactly the same way as when using the dFILE keyword. See the 'Alphabetic Listing' section for details of this.

The return value is the filename, including the full path of the file selected to be opened.

CREATEFILEDIALOG$:

Usage: `file$=CREATEFILEDIALOG$:(seedPath$)`

Displays the standard 'Create new file' dialog. `seedPath$` provides a starting path which is displayed when the dialog is opened. So, for example, if `seedPath$="C:\Documents\"` then the `Documents` folder will be intially displayed in the 'Folder' selector box and `C` in the disk selector. If you supply a filename in `seedPath$`, then this will be displayed as a suggested filename in the 'Name' selector box. You must always seed the dialog: you cannot pass a null `seedPath$` string to CREATEFILEDIALOG$:. If `seedPath$` does not include a drive name, then an error is raised.

The return value is the filename, including the full path of the file to be created.

SAVEASFILEDIALOG$:

Usage: `path$=SAVEASFILEDIALOG$:(seedPath$,BYREF useNewFile%)`

Displays the standard 'Save as' dialog. `seedPath$` provides a starting path which is displayed when the dialog is opened. So, for example, if `seedPath$="C:\Documents\"` then the `Documents` folder will be initially displayed in the 'Folder' selector box and `C` in the disk selector. If you supply a filename in `seedPath$`, then this will be displayed as a suggested filename in the 'Name' selector box. You must always seed the dialog: you cannot pass a null `seedPath$` string to SAVEASFILEDIALOG$:. If `seedPath$` does not include a drive name, then an error is raised.

`UseNewFile%` specifies the initial setting of the checkbox which determines whether a new file should be used when saving. This value is non-zero then the tick will be set initially, if it is zero, then it will not be. The procedure writes back a value to this variable which will be `KTrue%` if the symbol is set on exiting the dialog and `KFalse%` if not (see the listing of Const.oph in Appendix E in the 'Appends.pdf' document for definitions of these constants). If you pass #0 as the value of this argument, then this item will not be displayed in the dialog at all (as in many Series 5 applications) so that whether a new file is used on not is not decided by the user, but by the programmer.

The return value is the filename to save as, including the full path.

SPRITE AND BITMAP OPX

In general sprite procedures behave in a similar way to the sprite keywords of the Series 3a, 3c and Siena.

To use this OPX, you must included the header file Bmp.oxh, which contains the following declaration:

```
DECLARE OPX BMP,&10000258,$100
   BITMAPLOAD&:(name$,index&) : 1
   BITMAPUNLOAD:(id&) : 2
   BITMAPDISPLAYMODE&:(id&) : 3
   SPRITECREATE&:(winId%,x&,y&,flags&) : 4
   SPRITEAPPEND:(time&,bitmap&,maskBitmap&,invertMask&,dx&,dy&) : 5
   SPRITECHANGE:(index&,time&,bitmap&,maskBitmap&,invertMask&,dx&, dy&) : 6
   SPRITEDRAW:() : 7
   SPRITEPOS:(x&,y&) : 8
   SPRITEDELETE:(id&) : 9
   SPRITEUSE:(id&) : 10
END DECLARE
```

BITMAPLOAD&:

Usage: `id&=BITMAPLOAD&:(name$,index&)`

Loads a bitmap from the file `name$` (in the current directory if no other is specified). If the file contains more than one bitmap, then `index&` specifies which bitmap to load. The first (or only) bitmap is specified by `index&` having the value 0. The value returned is the ID of the open bitmap, which may be used to refer to it when calling sprite procedures or when using gBUTTON, for example.

See BITMAPUNLOAD:.

BITMAPUNLOAD:

Usage: `BITMAPUNLOAD:(id&)`

Unloads the bitmap whose ID is `id&`. You can call BITMAPUNLOAD: immediately after passing the bitmap ID to gBUTTON or, if used with the Sprite OPX procedures, **after** drawing the sprite, as the access count on a bitmap is incremented to ensure it is not unloaded prematurely.

See BITMAPLOAD&:.

BITMAPDISPLAYMODE&:

Usage: `mode&=BITMAPDISPLAYMODE&:(id&)`

Returns the graphics mode of the bitmap with ID `id&`. The graphics mode is that specified at its creation (see gCREATEBIT), i.e. 0 for 2-colour mode, 1 for 4-colour mode and 2 for 16-colour mode.

SPRITECREATE&:

Usage: `id&=SPRITECREATE&:(winId%,x&,y&,flags&)`

Initialises a sprite in the window given by `winId%` with its top left-hand corner at `x&,y&`. The value of `flags&` determines whether or not the sprite's members are flashing. If `(flags& AND 1)=1` then they are flashing and if the value is 0 then they are not. The value returned is the ID of the sprite which should be used when referring to it in other sprite procedures.

See SPRITEAPPEND:, SPRITECHANGE:, SPRITEDRAW:, SPRITEPOS:, SPRITEDELETE:, SPRITEUSE:.

SPRITEAPPEND:

Usage: `SPRITEAPPEND:(time&,bitmap&,maskBitmap&,invertMask&,dx&,dy&)`

Appends *members* or *frames* to the current sprite, which must have been created using CREATESPRITE&: before attempting to append to it. `bitmap&` is the ID of the bitmap to be used for this member of the sprite, and `maskBitmap&` is the ID of the bitmap to be used as a mask. The bitmap and mask must be of identical sizes. `invertMask&` takes the value of 1 or 0 to determine whether the mask is to be inverted or not respectively. For example, if you are using identical bitmap and mask and `invertMask&=1` then the member will appear as the bitmap, whereas if `invertMask&=0`, the member will be blank. `time&` is the period of time in microseconds for which the member appears in the sprite and `dx&,dy&` is the offset position of the top left-hand of the bitmap from the top left-hand corner of the sprite.

See SPRITECHANGE:.

SPRITECHANGE:

Usage: `SPRITECHANGE:(id&,time&,bitmap&,maskBitmap&,invertMask&,dx&,dy&)`

Changes a member of a sprite originally set up with SPRITEAPPEND.

`id&` specifies the number of the sprite member which is to be changed. This number is determined by the order in which the members were originally appended, the first member to be appended being labelled 0. `bitmap&` is the ID of the bitmap to be used for this member of the sprite, and `maskBitmap&` is the ID of the bitmap to be used as a mask. The bitmap and mask must be of identical sizes. `invertMask&` takes the value of 1 or 0 to determine whether the mask is to be inverted or not respectively. For example, if you are using identical bitmap and mask and `invertMask&=1` then the member will appear as the bitmap, whereas if `invertMask&=0`, the member will be blank. `time&` is the period of time in microseconds for which the member appears in the sprite and `dx&,dy&` is the offset position of the top left-hand of the bitmap from the top left-hand corner of the sprite.

A sprite may not be changed unless it has already been drawn.

See SPRITEAPPEND:, SPRITEDRAW:.

SPRITEDRAW:

Usage: `SPRITEDRAW:`

Draws the sprite once it has been set up. It need only be called once, and any changes made to the sprite are made as soon as the procedure making the change is called. The sprite will be drawn in the window and at the position specified by the arguments passed to SPRITECREATE.

See SPRITECREATE&:, SPRITEAPPEND:, SPRITECHANGE:, SPRITEPOS:.

SPRITEPOS:

Usage: `SPRITEPOS:(x&,y&)`

Repositions the whole of the current sprite to the point `x&,y&`.

See SPRITECREATE&:,SPRITEDRAW:.

SPRITEDELETE:

Usage: `SPRITEDELETE:(id&)`

Deletes the sprite with ID `id&`.

SPRITEUSE:

Usage: `SPRITEUSE:(id&)`

Sets the current sprite to be that with ID `id&` in order that it may be changed.

See SPRITECHANGE:, SPRITEPOS:.

Note that using many sprites at one time may cause undesirable effects. Firstly, sprites maybe animated at a slower speed than requested, and secondly the response time to any key or pointer event can be lengthened considerably. The number of sprites which can be drawn at one time without these effects occurring depends mainly on the animation speed: the faster it is the less sprites may be used successfully at one time. As a general rule, it is advisable not to use more than around 20 sprites at any one time when the animation speed is 0.2 sec, but you should experiment to find out what is suitable for your particular program.

DATABASE OPX

To use this OPX, you must included the header file Dbase.oxh, which contains the following declaration:

```
DECLARE OPX DBASE,KUidOpxDBase&,KOpxDBaseVersion%
   DbAddField:(keyPtr&,fieldName$,order&) : 1
   DbAddFieldTrunc:(keyPtr&,fieldName$,order&,trunc&) : 2
   DbCreateIndex:(index$,keyPtr&,dbase$,table$) : 3
   DbDeleteKey:(keyPtr&) : 4
   DbDropIndex:(index$,dbase$,table$) : 5
   DbGetFieldCount&:(dbase$,table$) : 6
   DbGetFieldName$:(dbase$,table$,fieldNum&) : 7
   DbGetFieldType&:(dbase$,table$,fieldNum&) : 8
   DbIsDamaged&:(dbase$) : 9
   DbIsUnique&:(keyPtr&) : 10
   DbMakeUnique:(keyPtr&) : 11
   DbNewKey&: : 12
   DbRecover:(dbase$) : 13
   DbSetComparison:(KeyPtr&,comp&) : 14
END DECLARE
```

DBADDFIELD:

Usage: `DBADDFIELD:(key&,field$,order&)`

Adds the field, `field$`, to the key, `key&` (see DBNEWKEY&: for creating a new key). This new key can then be used to create an index on a table. `order&` specifies how this field should be ordered:

value	*meaning*	*constant declaration in Dbase.oxh*
1	ascending	`CONST KDbAscending&  = 1`
0	descending	`CONST KDbDescending&  = 1`

You can use this procedure repeatedly to add more than one field to the same key. The order in which the fields are added dictates the significance of these fields in building up the index. The first field added to a key is the primary field; if there were any identical values in this field then the secondary field would be used and so on. For example, if the primary field in an ‘Address’ table was ‘Surname’ and there were two people with the surname Brown, then the secondary field or tertiary fields like ‘Forename’ and ‘Age’ might be used to define how the index will be ordered.

When using string fields in an index they cannot be longer than 240 bytes. To limit the size of a string field see CREATE in the 'Alphabetic Listing' section of the 'Glossary.pdf' document. If a string field is the last field in a key (or first and only) then it may be truncated to a specified length - see DBADDFIELDTRUNC:.

Note that the size of an index can become huge if the key is long. The key length *k* is the length in bytes of the sum of all the fields in the key. An integer or a long integer field is 4 bytes, a floating-point field 8 bytes and a text field depends on the length assigned to it (see also the previous paragraph). The size of the index depends at least linearly on *k* and hence may be large if the key is long.

When the key has been built as required, it can be used to create an index.

See DBCREATEINDEX:.

DBADDFIELDTRUNC:

Usage: `DBADDFIELDTRUNC:(key&,field$,order&,trunc&)`

Adds a truncated string field `field$` to the key `key&`. Only the last field added to a key (or first and only) can be truncated. `order&` specifies how this field should be ordered and may take the values:

value	*meaning*	*constant declaration in Dbase.oxh*
1	ascending	`CONST KDbAscending&  = 1`
0	descending	`CONST KDbDescending&  = 1`

`trunc&` is the truncation length and determines how many bytes of the string field will be used in building up the index (up to 240).

This procedure is useful for building up indexes on OPL16 databases whose string fields have a set length of 255 bytes. When the key has been built up as required it can be used to create an index.

See DBADDFIELD:, DBCREATEINDEX:.

DBCREATEINDEX:

Usage: `DBCREATEINDEX:(index$,key&,file$,table$)`

Creates an index with the name `index$` on the table `table$` in the database `file$`. The index is used for sorting and vastly increases the speed in table lookup. The index is based on `key&`, the supplied key. The database must be closed when this procedure is used.

The opening of an ordered view on a table must be requested in the SQL query in the OPEN command. See Appendix F in the 'Appends.pdf' document for SQL specification. If a suitable index has been created then it will be used.

See DBNEWKEY:, DBADDFIELD:, DBADDFIELDTRUNC:, DBMAKEUNIQUE:, DBSETCOMPARISON:, DBDROPINDEX:, OPEN.

DBDELETEKEY:

Usage: `DBDELETEKEY:(key&)`

Deletes the key `key&`. This should be called as soon as the key is no longer required. Any keys left undeleted when all modules that declare or include a declaration of the Dbase OPX have been unloaded will be automatically deleted

See DBNEWKEY&.

DBDROPINDEX:

Usage: `DBDROPINDEX:(index$,file$,table$)`

Drops the index `index$` of the table `table$` of the database `file$`. The database must be closed to use this procedure.

See DBCREATEINDEX:.

DBGETFIELDCOUNT&:

Usage: `n&=DBGETFIELDCOUNT&:(dbase$,table$)`

Returns the number of fields in one of the records in the table `table$` of the database `dbase$`. This number can then be used to analyse the contents of the record.

See DBGETFIELDNAME$:, DBGETFIELDTYPE&:.

DBGETFIELDNAME$:

Usage: `name$=DBGETFIELDNAME$:(dbase$,table$,fieldNum&)`

Returns the name of the field whose number is `fieldNum&` in the table `table$` of the database `dbase$`. This is useful for analysing the records of a table.

See DBGETFIELDCOUNT&:, DBGETFIELDTYPE&:.

DBGETFIELDTYPE&:

Usage: `type&=DBGETFIELDTYPE&:(dbase$,table$,fieldNum&)`

Returns the type of the field whose number is `fieldNum&` in the table `table$` of the database `dbase$`. This is useful for analysing the records of a table.

The values that may be returned (many of which aren't supported by OPL databases), are as follows:

value	*type*
0	bit
1	signed byte (8 bits)
1	unsigned byte (8 bits)
2	integer (16 bits)
3	unsigned integer (16 bits)
4	long integer (32 bits)
5	unsigned long integer (32 bits)
6	64-bit integer
7	single precision floating-point number (32 bits)
8	double precision floating-point number (64 bits)
9	date/time object
10	ASCII text
11	Unicode text
12	Binary
13	LongText8
14	LongText16
15	LongBinary

Types 2,4,8,10 correspond to OPL's %,&, floating point and $ types respectively.

See DBGETFIELDCOUNT&, DBGETFIELDNAME&

DBISDAMAGED&:

Usage: `i&=DbIsDamaged&:(dbase$)`

Returns 1 if the database `dbase$` considers that it may have damaged indexes and 0 if not. If the database is damaged, then this does not make it unusable, but attempting to use any damaged index will result in an error. Recovering the database will restore any damaged indexes.

See DBRECOVER:.

DBISUNIQUE&:

Usage: `i&=DBISUNIQUE&:(key&)`

Returns -1 if the key `key&` is unique or 0 if it is not.

See DBMAKEUNIQUE:.

DBMAKEUNIQUE:

Usage: `DBMAKEUNIQUE:(key&)`

Sets the key `key&` to be unique. The index created will then not allow any records to be added if they exactly match the indexed fields of another record.

See DBNEWKEY:, DBADDFIELD:, DBCREATEINDEX:.

DBNEWKEY&:

Usage: `k&=NEWKEY&:`

Returns a handle to a key which can be used for creating indexes.

See DBDELETEKEY:, DBCREATEINDEX:, DBADDFIELD:, DBADDFIELDTRUNC:.

DBRECOVER:

Usage: `DBRECOVER:(dbase$)`

Recovers a damaged database `dbase$`, restoring any damaged indices.

See DBISDAMAGED&:.

DBSETCOMPARISON:

Usage: `DBSETCOMPARISON:(key&,comp&)`

The text comparison is used to determine the order of an index. The argument `comp&` sets the text comparison mode of a key `key&`, and takes one of the constant values supplied in the header file Dbase.oxh:

```
CONST KDbCompareNormal&       = 0
CONST KDbCompareFolded &      = 1
CONST KDbCompareCollated&     = 2
```

PRINTER OPX

Printer OPX provides access to print and text handling. An object to be printed is first created dynamically by sending text, bitmaps and formatting information. Printing itself is then handled by calling standard print dialogs. One procedure is provided for each of the four dialogs usually called from standard applications.

```
DECLARE OPX PRINTER,KUidOpxPrinter&,KOpxPrinterVersion%
  SendStringToPrinter:(string$) : 1
  InsertString:(string$,pos&) : 2
  SendNewParaToPrinter: : 3
  InsertNewPara:(pos&) : 4
  SendSpecialCharToPrinter:(character%) : 5
  InsertSpecialChar:(character%,pos&) : 6
  SetAlignment:(alignment%) : 7
  InitialiseParaFormat:(Red&, Green&, Blue&, LeftMarginInTwips&,
               RightMarginInTwips&, IndentInTwips&,
               HorizontalAlignment%, VerticalAlignment%,
               LineSpacingInTwips&, LineSpacingControl%,
               SpaceBeforeInTwips&, SpaceAfterInTwips&, KeepTogether%,
               KeepWithNext%, StartNewPage%, WidowOrphan%, Wrap%,
               BorderMarginInTwips&, DefaultTabWidthInTwips&) : 8
  SetLocalParaFormat: : 9
  SetGlobalParaFormat: : 10
```

```
  RemoveSpecificParaFormat: : 11
  SetFontName:(name$) : 12
  SetFontHeight:(height%) : 13
  SetFontPosition:(pos%) : 14
  SetFontWeight:(weight%) : 15
  SetFontPosture:(posture%) : 16
  SetFontStrikethrough:(strikethrough%) : 17
  SetFontUnderline:(underline%) : 18
  SetGlobalCharFormat: : 19
  RemoveSpecificCharFormat: : 20
  SendBitmapToPrinter:(bitmapHandle&) : 21
  InsertBitmap:(bitmapHandle&,pos&) : 22
  SendScaledBitmapToPrinter:(bitmapHandle&,xScale&,yScale&) : 23
  InsertScaledBitmap:(bitmapHandle&,pos&,xScale&,yScale&) : 24
  PrinterDocLength&: : 25
  SendRichTextToPrinter:(richTextAddress&) : 26
  ResetPrinting: : 27
  PageSetupDialog: : 28
  PrintPreviewDialog: : 29
  PrintRangeDialog: : 30
  PrintDialog: : 31
  SendBufferToPrinter:(Addr&) : 32
END DECLARE
```

SENDSTRINGTOPRINTER:

Usage: `SENDSTRINGTOPRINTER:(string$)`

Append a string to whatever has already been sent to the printer.

See INSERTSTRING:, SENDNEWPARATOPRINTER:.

INSERTSTRING:

Usage: `INSERTSTRING:(string$,pos&)`

Insert `string$` at position `pos&` in the buffer. Inserting at position zero puts the string ahead of anything already sent or inserted.

See SENDSTRINGTOPRINTER:, INSERTNEWPARA:.

SENDNEWPARATOPRINTER:

Usage: `SENDNEWPARATOPRINTER:`

Paragraphs delimit paragraph formatting, such as centring. This procedure is equivalent to

`SENDSPECIALCHARTOPRINTER:(KParagraphDelimiter%).`

See SENDSPECIALCHARTOPRINTER:, INSERTNEWPARA:, SENDSTRINGTOPRINTER:.

INSERTNEWPARA:

Usage: `INSERTNEWPARA:(pos&)`

Inserts a new paragraph at position `pos&` in the buffer.

See SENDNEWPARATOPRINTER:, INSERTSTRING:.

SENDSPECIALCHARTOPRINTER:

Usage: `SENDSPECIALCHARTOPRINTER:(character%)`

Sends a special character, for example line and page breaks etc., to the printer. Constants for special characters are defined in Const.oph (see the 'Calling Procedures' section of the 'Basics.pdf' document for details of how to use this file and Appendix E in the 'Appends.pdf' document for a listing of it) as follows:

```
CONST KParagraphDelimiter%    = $06
CONST KLineBreak%             = $07
CONST KPageBreak%             = $08
CONST KTabCharacter%          = $09
CONST KNonBreakingTab%        = $0a
CONST KNonBreakingHyphen%     = $0b
CONST KPotentialHyphen%       = $0c
CONST KNonBreakingSpace%      = $10
CONST KVisibleSpaceCharacter%= $0f
```

See SENDNEWPARATOPRINTER:, INSERTSPECIALCHAR:.

INSERTSPECIALCHAR:

Usage: `INSERTSPECIALCHAR:(character%,pos&)`

Inserts a special character at `pos&` in the buffer.

See SENDSPECIALCHARTOPRINTER:.

SETALIGNMENT:

Usage: `SETALIGNMENT:(alignment%)`

Sets alignment state of a paragraph. Default is `KPrintLeftAlign%`.

The setting does not take effect until either SETLOCALPARAFORMAT: or SETGLOBALPARAFORMAT: is called. The allowable alignments, defined in Printer.oxh, are:

```
CONST KPrintLeftAlign%        = 0
CONST KPrintCenterAlign%      = 1
CONST KPrintRightAlign%       = 2
CONST KPrintJustifiedAlign%   = 3
CONST KPrintUnspecifiedAlign%= 4
```

See INITIALISEPARAFORMAT:.

INITIALISEPARAFORMAT:

Usage: `INITIALISEPARAFORMAT:(Red%, Green%, Blue%, LeftMarginInTwips&, RightMarginInTwips&, IndentInTwips&, HorizontalAlignment%, VerticalAlignment%, LineSpacingInTwips&, LineSpacingControl%, SpaceBeforeInTwips&, SpaceAfterInTwips&, KeepTogether%, KeepWithNext%, StartNewPage%, WidowOrphan%, Wrap%, BorderMarginInTwips&, DefaultTabWidthInTwips&)`

Sets a state for formatting. The setting does not take effect until either `SETLOCALPARAFORMAT` or `SETGLOBALPARAFORMAT` is called.

`Red%,Green%,Blue%` set the background colour. Each value can be in the range 0 to 255. Default colour is white, with all three arguments equal to 255.

`LeftMarginInTwips&` sets left text margin relative to left page margin. Default is zero.

`RightMarginInTwips&` sets right text margin, relative to right page margin. Default is zero.

`IndentInTwips&` sets left, right and first line indent. Default is zero.

`HorizontalAlignment%` sets horizontal alignment of paragraph. Default is left alignment. See `SETALIGNMENT` for values.

`VerticalAlignment%` sets vertical alignment of paragraph (for use by spreadsheet applications). Default is top alignment. Allowable values, supplied in Printer.oxh, are:

```
CONST KPrintTopAlign%          = 0
CONST KPrintBottomAlign%       = 2
CONST KPrintUnspecifiedAlign%  = 4
```

`LineSpacingInTwips&` sets inter-line spacing in twips. Default is 200 (10 point).

`LineSpacingControl%` sets the control for `LineSpacingInTwips&` value. Default is `KLineSpacingAtLeastInTwips&`. Allowable values are:

```
CONST KLineSpacingAtLeastInTwips%= 0
CONST KLineSpacingExactlyInTwips%= 1
```

`SpaceBeforeInTwips&` sets the space above a paragraph. Default is zero.

`SpaceAfterInTwips&` sets the space below a paragraph. Default is zero.

`KeepTogether%` prevents a page break within paragraph when `KTrue%`. Default is `KFalse%`. (Constants are defined in Const.oph.)

`KeepWithNext%` prevents a page break between this paragraph and the following paragraph when `KTrue%`. Default is `KFalse%`.

`StartNewPage%` inserts a page break immediately before this paragraph when `KTrue%`. Default is `KFalse%`.

`WidowOrphan%` prevents the printing of the last line of a paragraph by itself on the top of a new page (widow) or the first line of a paragraph by itself on the bottom of the page (orphan). Default is `KFalse%`.

`Wrap%` forces the paragraph to line wrap at the right margin when `KTrue%`. `KFalse%` disables line wrap. Default is `KTrue%`.

`BorderMarginInTwips&` sets distance in twips between paragraph border and enclosed text (must be non-negative). Default is zero.

`DefaultTabWidthInTwips&` specifies the spacing between the default tab stops. Default is 360.

SETLOCALPARAFORMAT:

Usage: `SETLOCALPARAFORMAT:`

Sets the initialised global paragraph formatting as local.

See SETGLOBALPARAFORMAT:.

SETGLOBALPARAFORMAT:

Usage: `SETGLOBALPARAFORMAT:`

Sets the local paragraph format as global (after initialising formatting).

See SETLOCALPARAFORMAT:, REMOVESPECIFICPARAFORMAT:.

REMOVESPECIFICPARAFORMAT:

Usage: `REMOVESPECIFICPARAFORMAT:`

Unsets local paragraph formatting, using global formatting instead

See SETGLOBALPARAFORMAT:, SETLOCALPARAFORMAT:.

SETFONTNAME:

Usage: `SETFONTNAME:(name$)`

Gives a new font name. Takes immediate effect locally, but requires the use of SETGLOBALCHARFORMAT: to set as the global default.

SETFONTHEIGHT:

Usage: `SETFONTHEIGHT:(height%)`

Gives a new font height in twips

(1 twip = 1/20 points or 1/1440 inches)

Takes immediate effect locally, but requires the use of SETGLOBALCHARFORMAT: to set as the global default.

SETFONTPOSITION:

Usage: `SETFONTPOSITION:(POS%)`

Sets font position as normal, superscript or subscript. The following constants can be used for this:

```
CONST KPrintPosNormal%        = 0
CONST KPrintPosSuperscript%   = 1
CONST KPrintPosSubscript%     = 2
```

Takes immediate effect locally, but requires the use of SETGLOBALCHARFORMAT: to set as the global default.

SETFONTWEIGHT:

Usage: `SETFONTWEIGHT:(weight%)`

Sets the font weight (normal or bold). Allowable values are:

```
CONST KStrokeWeightNormal%    = 0
CONST KStrokeWeightBold%      = 1
```

Takes immediate effect locally, but requires the use of SETGLOBALCHARFORMAT: to set as the global default.

SETFONTPOSTURE:

Usage: `SETFONTPOSTURE:(posture%)`

Sets font posture using the following constants:

```
CONST KPostureUpright%        = 0
CONST KPostureItalic%         = 1
```

Takes immediate effect locally, but requires the use of SETGLOBALCHARFORMAT: to set as the global default.

SETFONTSTRIKETHROUGH:

Usage: `SETFONTSTRIKETHROUGH:(strikethrough%)`

Set strike-through on or off using:

```
CONST KStrikethroughOff%      = 0
CONST KStrikethroughOn%       = 1
```

Takes immediate effect locally, but requires the use of SETGLOBALCHARFORMAT: to set as the global default.

SETFONTUNDERLINE:

Usage: `SETFONTUNDERLINE:(underline%)`

Set underline on or off using:

```
CONST KUnderlineOff%          = 0
CONST KUnderlineOn%           = 1
```

Takes immediate effect locally, but requires the use of SETGLOBALCHARFORMAT: to set as the global default.

SETGLOBALCHARFORMAT:

Usage: `SETGLOBALCHARFORMAT:`

Sets the currently active character format to be the global default.

See REMOVESPECIFICCHARFORMAT:.

REMOVESPECIFICCHARFORMAT:

Usage: `REMOVESPECIFICCHARFORMAT:`

Unsets local character formatting, replacing it with the global formatting instead.

See SETGLOBALCHARFORMAT:.

SENDBITMAPTOPRINTER:

Usage: `SENDBITMAPTOPRINTER:(bitmapHandle&)`

Appends a bitmap from its handle `bitmapHandle&`.

See INSERTBITMAP:, SENDSCALEDBITMAPTOPRINTER:.

INSERTBITMAP:

Usage: `INSERTBITMAP:(bitmapHandle&,pos&)`

Inserts a bitmap specified by `bitmapHandle&` at position `pos&` in the buffer to be sent to the printer. The handle is the same as that returned by gLOADBIT.

See SENDBITMAPTOPRINTER:, INSERTSCALEDBITMAP:.

SENDSCALEDBITMAPTOPRINTER:

Usage: `SENDSCALEDBITMAPTOPRINTER:(bitmapHandle&,xScale&,yScale&)`

Appends a scaled bitmap specified by `bitmapHandle&` and scaled according to `xScale&` and `yScale&` to the buffer sent to the printer. The handle is the same as that returned by gLOADBIT. Default scaling is `xScale& = yScale& = 1000`.

See SENDBITMAPTOPRINTER:, INSERTSCALEDBITMAP:.

INSERTSCALEDBITMAP:

Usage: `INSERTSCALEDBITMAP:(bitmapHandle&,pos&,xScale&,yScale&)`

Inserts a scaled bitmap specified by `bitmapHandle&` and scaled according to `xScale&` and `yScale&` at position `pos&` in the buffer sent to the printer. The handle is the same as that returned by gLOADBIT.

See INSERTBITMAP:, SENDSCALEDBITMAPTOPRINTER:.

PRINTERDOCLENGTH&:

Usage: `PRINTERDOCLENGTH&:`

Returns the count of characters currently held in the buffer. It is not possible to insert characters beyond the length of the document buffer. Bitmaps and special characters each count as one character.

SENDRICHTEXTTOPRINTER:

Usage: `SENDRICHTEXTTOPRINTER:(richTextAddress&)`

Sends the address of a rich text object `richTextAddress&` to the printer. This is intended to be used on a pointer returned by another OPX. This new Rich Text will **replace** all content currently stored.

RESETPRINTING:

Usage: `RESETPRINTING:`

Deletes **all** text, bitmaps and formatting in the printing buffer.

PAGESETUPDIALOG:

Usage: `PAGESETUPDIALOG:`

Calls the standard page setup dialog.

See PRINTPREVIEWDIALOG:, PRINTRANGEDIALOG:, PRINTDIALOG:.

PRINTPREVIEWDIALOG:

Usage: `PRINTPREVIEWDIALOG:`

Calls the standard print preview dialog. This allows the data sent to the printer to be viewed. The other three dialogs can all be called from this dialog.

See PAGESETUPDIALOG:, PRINTRANGEDIALOG:, PRINTDIALOG:.

PRINTRANGEDIALOG:

Usage: `PRINTRANGEDIALOG:`

Calls a standard print range dialog.

See PAGESETUPDIALOG:, PRINTPREVIEWDIALOG:, PRINTDIALOG:.

PRINTDIALOG:

Usage: `PRINTDIALOG:`

Calls a standard print dialog.

See PAGESETUPDIALOG:, PRINTPREVIEWDIALOG:, PRINTRANGEDIALOG:.

SENDBUFFERTOPRINTER:

Usage: `SENDBUFFERTOPRINTER:(addr&)`

Appends the contents of a buffer to whatever has already been sent to the printer. The `addr&` parameter should be the address of a buffer into which text has been inserted by dEDITMULTI.

INDEX

S

T

U

V

X