nOPLPlus

About Neuon

You can learn more about our dynamic company, and expanding EPOC software portfolio, from Neuon's web site at http://www.neuon.com

If you are a visionary C++, Java or OPL32 developer, motivated by doing something different, excited by challenges, relish the prospect of working with kindred spirits, and recognise the value of a dedicated support team, Neuon would like to hear from you.

Neuon - where innovation and quality are principles, not an afterthought

nOPLPlus

nOPLPlus is an integrated development environment for OPL programmers. There are two parts to the package:

- OPL+ editor, a project based OPL programmer's editor.
- OPP back-end translator & preprocessor, which supports standard EPOC OPL and adds a number of additional extensions.

Also available separately is the OPP SDK providing the following:

- OPP for EPOC16 (runs on Series 3a/3c and Siena). This package includes OPPDBG, a runtime source level debugger.
- OPP for MSDOS (runs on a PC allowing OPP code to be translated for EPOC16 and EPOC)

nOPLPlus release notes

This is **version 1.20** of nOPLPlus. See the topic **Release history** for changes.

Please report any problems or difficulties to Neuon's nOPLPlus Development Team (nopl@neuon.com)

Thank you for your interest. Comments and suggestions are always welcome.

Release history

V1.20

- First Neuon public release

Licence Conditions & Limited Warranty

By installing nOPLPlus you are agreeing to the following terms and conditions. Please read them carefully.

This is an evaluation version. An evaluation version lets a person try out a program before buying it. While evaluation versions are copyrighted and the copyright holder retains all rights, the author specifically grants the user the right to evaluate and distribute the program with limited exceptions.

After using the evaluation version for a defined trial period, the user must purchase a licensed copy of the program or remove the evaluation version from their EPOC device.

The trial period for nOPLPlus is 30 days from first use.

You are encouraged to:

1. Upload this evaluation version to any electronic bulletin board or www site.

2. Demonstrate the evaluation version and its capabilities.

3. Give copies of the evaluation version to potential users, so that others may have the opportunity to obtain a copy for use in accordance with the licence conditions.

End-user license agreement

IMPORTANT- READ CAREFULLY: This End-User License Agreement ("EULA") is a legal agreement between you (either an individual or a single entity) and Neuon for the software accompanying this EULA, which includes EPOC device software and may include associated media, printed materials, and "online" or electronic documentation (The "SOFTWARE"). By exercising your rights to make and use copies of the SOFTWARE, you agree to be bound by the terms of this EULA. If you do not agree to the terms of this EULA, you may not use the SOFTWARE.

License

1. Grant of license

This EULA grants you the following rights:

a) You may install and use only one copy of the SOFTWARE at any given time.

b) At the end of the trial period you are required to either Register the SOFTWARE, in order to convert it to a licensed copy, or remove it from your device. Instructions on the Registration procedure are contained in the help file topic **How to Register**.

c) You may not reverse engineer, decompile, or disassemble the SOFTWARE, except and only to the extent that such activity is expressly permitted by applicable law notwithstanding this limitation.

d) The SOFTWARE is licensed as a single product. Its component parts may not be separated for use on more than one EPOC device.

e) Without prejudice to any other rights, Neuon may terminate this EULA if you fail to comply with the terms and conditions herein. In such event, you must destroy all copies of the SOFTWARE and all of its component parts.

2. Copyright

The SOFTWARE is protected by copyright laws and international copyright treaties, as well as other intellectual property laws and treaties. All title and copyrights in and to the SOFTWARE (including but not limited to any images, photographs, animations, video, audio, music and text incorporated into the SOFTWARE, the accompanying printed materials, and any copies of the SOFTWARE) are owned by Neuon.

3. Limited warranty

a) No warranties

Neuon expressly disclaims any warranty for the SOFTWARE. The SOFTWARE is provided "as is" without warranty of any kind, either express or implied, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, or non-infringement. The entire risk arising out of use or performance of the SOFTWARE remains with you.

b) No liability for consequential damages

In no event shall Neuon or its suppliers be liable for any damages whatsoever (including, without limitation, damages for loss of business profits, business interruption, loss of business information, or any other pecuniary loss) arising out of the use of or inability to use the SOFTWARE even if Neuon has been advised of the possibility of such damages. Because some states/jurisdictions do not allow the exclusion or limitation of liability for consequential or incidental damages, the above limitation may not apply to you. Any liability of Neuon will be Limited exclusively to product replacement or refund of Registration Price.

c) No liability for errors or omission

Neuon expressly disclaims any liability for errors or omissions in the content of the SOFTWARE.

4. Reservations

All rights to the SOFTWARE not expressly granted herein are reserved by Neuon.

Registration

nOPLPlus is not free software. For full details of the licence conditions, see the topic **Licence Conditions & Limited Warranty**.

In accordance with the **Licence**, once the 30 day evaluation period has expired, nOPLPlus must either be licensed by registration, or be removed from the EPOC device.

Licensing of nOPLPlus simply requires the input of a registration code (**Tools | Register**). There is no need to re-install Merlin. To register nOPLPlus and receive your personal code, you can use one of two methods:

1. **ONLINE Registration**

   For speed of use, this is the recommended registration method. Using a secure, reliable online registration contractor, NEUON will provide you with your unique registration details with minimal fuss, and minimal delay. To register online, go to **www.neuon.com/home/register/**

2. **OFFLINE Registration**

   nOPLPlus can be registered via conventional mail. A choice of registration addresses is listed below.

   Please ensure that any cheque / international money order is sent in the correct denomination according to the address you register at. We ask that you include:

- an email address, or
- a stamped addressed envelope. If you are not able to provide this, please add an additional 1USD (dollar) or 1UKP (UK pound) to the application registration cost. Failure to include either the handling charge, or a stamp, may result in a processing delay.

To maximise the efficiency of registration by post, you have three options:

- A UK sterling cheque (made payable to **Alex Wilbur**) or money order to:

  Neuon Applications Registration,

  13 Warminster Road,

  Westbury,

  Wiltshire.

  BA13 3PA

  UK

- a US dollar cheque (made payable to **Ben Vaisvil**) to:

  Neuon Applications Registration,

  632 Concord St.

  Aurora,

  IL 60505

  USA

- an International Money Order (made payable to **Gary Belcher**) drawn on a US bank:

  Neuon Applications Registration,

  11 Derwent Road,

  Marlborough,

  Harare,

  Zimbabwe

Working with projects

The nOPLPlus editor is project based. This means you create a project and then associate a group of files with it. A single OPL application may consist of various OPL source files such as:

- the main source file
- include files
- source for modules loaded into the main application

The association of source files with a project provides faster access to the files in the project. Use the buttons on the toolbars to quickly switch between the files in a project.

Files can be added and removed from a project using the following menu options:

- Create new file

- Open file
- Remove file

Any type of file may be added to a project using the **Open file** menu option.

While the editor will open any type of file, (including binary files and files created by the standard EPOC Program editor), only plain text files can be edited with the editor.

For a project, one of the files should be designated the main application source file. When selecting the **Build project** menu or toolbar option this will be the file that is translated. Use the **Project Folders** menu option to specify which source file this is.

Working with files

The nOPLPlus editor allows the editing of plain text OPL source files.

To import existing OPL source files use the **Import OPL file** menu option or copy and paste the source from the standard EPOC Program editor.

Any type of file may be viewed within the editor, including binary and EPOC Program editor files, but only text files may be edited.

The convention is to use an OPP file extension for source files, an OPH file extension for include files.

A project would normally consist of a main OPP source file with a name matching the project name. The source for the application could be split over a number of OPP files to make the source easier to manage. For example:

MainApp.opp

UserInterface.opp

Document.opp

Engine.opp

The #include statement can be used at the end of **MainApp.opp** to include the other source files in the application:

#include "UserInterface.opp"

#include "Document.opp"

#include "Engine.opp"

Project folders

The **Project Folders** menu option allows two folders to be associated with the current project. These are saved with the project.

1. OPO Output Folder

When an OPO file is created (this does not apply to APP files), it will be created in the given folder. If the **Use Output Folder** option is not ticked then OPO files will be created in the same directory as the OPP file being translated.

This option is useful if you are building an APP which uses the **LOADM** keyword to load OPO modules at runtime from the same folder as the APP file. In this case you would set the **OPO Output Folder** option to the same folder as the APP.

2. System Includes Folder

There are three distinct keywords used to include a file:

- include "filename.oxh"
- #include "filename.oph"
- #include <filename.oph>

The first keyword is read and interpreted by the standard EPOC OPL translator and is used with OPX include files or files containing CONST declarations. These files should not contain any OPP extensions since they will not be read by OPP. These include files should be in the following folder on any disk:

\System\Opl

The second keyword, with a # prefix and double quotes, is read and interpreted by OPP. These include files can contain any OPP source code. The include file should exist in the same directory as the file being translated.

The third keyword, with a # prefix and angle brackets, instructs OPP to include a system include file at that point. System include files are general files, e.g. containing non-project specific #defines. OPP will search for these system include files in the folder specified by the **System Includes** menu option.

Project settings

The **Project Settings** menu option has the following options:

- Project

  Main source file The name of the main source which is translated when the **Build** menu or toolbar button is selected.

- OPP Display

  Show input: Instructs OPP to display all source lines as they are read.

  Show output: Instructs OPP to display all source lines output from the preprocessor.

  Show PROC's: Displays the name of each PROC during preprocessing.

  Check PROC's: OPP will report all PROC's which do not appear to have been called anywhere and those PROC's which were called but not defined.

- OPP

  Keep .PRE file: During translation OPP will first preprocess the code, removing any OPP specific extensions. The file created will have a .PRE extension. Tick this option to keep this file after translation is complete.

  Preprocess only: Runs the preprocessor only on the source code, no OPO or APP file will be generated.

  Generate debug info: This is for future use and is not implemented at present.

- Defines

  Defines: Allows #defines to be set. This can be useful if you often translate code in a number of different ways, e.g. to build an APP with and without debug routines. For example, entering the following

  ```
  DEBUG, DEBUGLEVEL=1
  ```

  is equivalent to having the following at the start of the code

  ```
  #define DEBUG
  #define DEBUGLEVEL=1
  ```

Format settings

The **Format Settings** menu has the following options:

- Font

  Controls the font used by the nOPLPlus editor. Avoid using a bold or italic font if you use the syntax highlighting option.

- Indentation

  Controls the tab settings used throughout all source code.

Preferences

The **Preferences** menu option has the following options:

- Syntax highlighting

  The OPL+ editor is capable of displaying key lines in bold or italic, e.g. PROC...ENDP lines can be displayed in bold and comments displayed in italic.

  Syntax highlighting is optional, and defaults to off, since it does slow down the editor slightly, especially with very large files.

- Show tabs

  Display tab characters.

- Show spaces

  Display spaces.

- Show page breaks

  Display page breaks.

- Sort go to proc list

  If this option is ticked then the **Go to proc** menu and toolbar option will sort the list of PROC's found in the current source file rather than displaying them in the order they appear in the file.

Translation

There are two options for translating source code:

- Translate

  Translates the current open file.

- Build

  Builds the current project by translating the main source file, as set by the **Project Settings** menu option.

Finding text

The **Edit | Find** menu option contains the following options:

- Find & Find Next

  Find text within the current open file.

- Replace

  Replace text within the current file.

- Go to proc

  Displays a list of procedures in the current open file. Select a procedure in the list to jump to the source line. See the preference settings for an option which controls whether the list is sorted or not.

- Go to line

  Enter a line number to jump to that line.

- Find in files

  Find text within all files below a given folder. Select the found text to open the file and jump to the line.

- Display found

  Displays the text found using the **Find in files** menu option.

Formatting

The **Edit | Format** menu option contains the following options:

- Layout

  Select a region of source code and then use this menu option to auto-indent the code.

- Indent & Unindent

  Select a region of source code and then use this option to shift the code to the left or right.

- Uppercase & Lowercase

  Will make the current selection UPPERCASE or lowercase.

- Prefix Lines

  Will add a given prefix, e.g. "REM ", to the start of each line in the current selection.

- Remove prefix

  Will remove a given prefix, e.g. "REM ", from the start of each line in the current selection.

OPP

OPP is an OPL preprocessor which also adds a number of useful extensions to the OPL language. The nOPLPlus editor invokes OPP when you translate code.

OPP greatly enhances the capabilities of OPL, some of the facilities it provides are:

#include

#define

#ifdef, #ifndef,..., #endif

'C' style structures

Adds support for multi-dimensional arrays to OPL

LIBPROC - procedure libraries which are only included if they have been called.

Facilities to prevent reverse translation of code

C & C++ style comments

See the OPP help file for full details.

"!!" characters

is an ANSI standard token. Due to the use of # by OPL, the pre-processor uses the identifiers !! instead. Consider the following macro definition:

```
#define M(name)
module!!name!!%:
```

The !! characters indicate a delimiter for a macro function argument. OPP notes the delimiter and then discards the !! characters from the expansion.

M(a) would expand to modulea%:

Without the !! OPP would not detect the presence of the name argument since neither “e” nor % are normal delimiter characters.

"!" character

The ANSI standard defines the # character to have a special meaning within a macro definition. Due to the fact that # is used within OPL, OPP looks for the character ! rather than #. Consider:

```
#define ASSERT(expr)
if not (expr)
    :\
    print !expr,”failed”
    :\
endif
```

The presence of the ! character before the macro function variable instructs OPP to quote the expression when expanded. Thus the following line...

```
ASSERT(a%>0)
```

would expand to

```
if not (a%>0)
    print “a%>0”,”failed”
endif
```

#define

The #define pre-processor directive is used to create a simple macro definition. For example the following line is added to the OPP source file:

```
#define MAX_ARRAY    10
```

From this point onwards any occurrence of MAX_ARRAY in the OPP code will be replaced by 10 when the pre-processor is run.

Macros may be defined from other macros, e.g.

```
    #define WIDTH         5
    #define HEIGHT 10
    #define SIZE (WIDTH*HEIGHT)
```

#if, #ifdef, #ifndef, #else, #elif, #endif

OPL includes the conditions if, else, elseif and endif. OPP has similar directives #if, #ifdef, #ifndef, #else, #elif and #endif which are analogous to OPL commands.

Using conditional pre-processor directives provides control over the sections of OPP code which are to be translated. For example:

```
#define DEBUG
#ifdef DEBUG
    print “Translated with brief debug enabled”
    #ifdef MOREDEBUG
        print “Translated with additional debug enabled”
    #endif
#else
    print “Debug code not translated”
#endif
```

The #ifdef directive tests for the presence of a macro, if it exists then the remaining code up to a matching #endif or #else is passed to the OPL translator. #ifndef has the opposite affect, i.e. the condition is true if the macro does not exist.

The condition “#if expression” is true if the expression evaluates to true or non-zero. Examples of valid expressions are as follows:

```
    #if DEBUG_LEVEL > 2
    #if OsVersion >= $300
    #if LcdType = 11
    #if (PsuType = 2) or (PsuType = 3)
```

The “#elif expression” may be used as follows:

```
#if DEBUG_LEVEL = 1
    print “debug level 1”
#elif DEBUG_LEVEL = 2
    print “debug level 2”
#elif DEBUG_LEVEL = 3
    print “debug level 3”
#else
print “no debug” #endif
```

- The current release of OPP does not support use of “defined()” in #if or #elif expressions.

#include

Other OPP files may be included into the file being translated using the #include directive. This takes one of two forms as shown in the following examples:

```
#include <os\calls>

#include "my_procs.oph"
```

The first #include will cause the pre-processor to search for a file in the system include directory. The default system include directory is:

```
C:\System\Apps\OPLPlus\Include\
```

This location can be altered from the OPL+ menu item **Project | Folders**.

If the file is found it will be included into the translation at that point. This form of include is used for system include files. These include files are generic files which are not written for any one specific OPP program.

- By default the OPH extension is assumed for system include files, although this may be overridden by explicitly stating the extension in the include filename.

The second form of include, which uses the delimiters "" rather than <>, is for specifying include files which are specific to the program being translated. The processor will default to looking in the same directory as the program being translated and will look for a file with the same extension. A full filename which includes a path and file extension may be given to override this.

Usually include files only contain pre-processor directives such as macro definitions, in which case the file extension OPH should be used rather than OPP. The OPH editor is supplied to allow include files to be listed on the system screen separate from any OPP and OPL files.

- On EPOC32 the OPL translator supports the include keyword mainly for including OPX headers. It is important to understand that the OPL include statement will be read and handled by the OPL translator, and the #include keyword read and handled by OPP. This has two important consequences:

  1. Always use the OPL include for OPX header files, failure to do so will confuse the translator.

  2. Do not add any OPP extensions to an include file which is included using the OPL include keyword.

#pragma check_procs

When OPP processes OPL source it records the procedures which have been defined and those which have been called. The **check_procs pragma** instructs OPP to check the list of procedures called against the list of defined procedures. If OPP finds any procedures which have been called but not defined then it will list them. OPP will also list any procedures which are defined but never explicitly called. Note that in this later case OPP cannot detect procedure calls which are made using a string variable (refer to the EPOC Programming Manual, Advanced Topics section).

- The best place for this pragma is at the end of the OPL source.

#pragma info, warn, error

These pragma's instruct OPP to display a information, warning or error message. In the case of the warning and error messages OPP may abort the translation depending upon the **stop_on pragma** setting.

#pragma no_revtran

Revtran is a program which allows OPO files to be reverse translated back into OPL source code. The **no_revtran pragma** may be used to prevent the reverse translation of the **no_revtran** line. The way it works is to insert some OPL code into the translation which causes the Revtran program to fall over.

- It cannot be guarantee that the no_revtran option will stop future versions of Revtran from working, or that it will keep sufficiently determined people from reverse translating your programs. The no_revtran option has been tested against Revtran 3.3a.

The "#pragma no_revtran" line should appear within a procedure at the top of the OPP file. Revtran will be able to reverse translate up to this line but no further.

#pragma pack

🖷 The use of this pragma statement is strongly discouraged.

#pragma pack 2 will set the structure packing size to 2 bytes.

The packing size controls how OPP packs fields into structures and the alignment of fields within a structure.

The default pack size is 1 which means that OPP packs structures so that there are no spaces between fields.

#pragma pause

#pragma pause" will pause the pre-processor until a key is pressed. This could be used at the end of the OPP source file or immediately after a **show_macros pragma**.

#pragma show_input

Use #pragma show_input on (the on part is optional) to instruct the pre-processor to display lines as they are read from the program editor from that point onwards.

Use #pragma show_input off to switch the display off.

#pragma show_output

Use #pragma show_output on (the on is optional) to instruct the pre-processor to display lines as they are sent to the OPL translator, i.e. after pre-processing.

Use #pragma show_output off to switch output off.

#pragma show_macros

#pragma show_macros will instruct the pre-processor to print out all defined macros at that point in the translation.

#pragma show_procs

When this pragma is encountered OPP will print out the name of each procedure as it is defined. This is useful if you want to see the progress of the translation and which procedures have been included.

#pragma stop_on

Use #pragma stop_on never to instruct the pre-processor to display any pre-processor error or warning messages and to continue processing the OPP code.

Use #pragma stop_on error to stop whenever an error occurs but continue if there is a warning.

#pragma stop_on warn will cause the pre-processor to stop on both warnings and errors.

The default mode is "error", in which case as soon as a pre-processor error is detected control will return to the OPL+ editor.

🗇 The "never" switch could be used whenever you want to find all errors within a newly written include file in one go, rather than fixing each one individually. It may also be useful if any error occurs within an include file and you need to pinpoint exactly which line the error was on.

#pragma to_file

#pragma to_file preproc.opl would instruct the pre-processor to send pre-processed lines both to the translator and the file **preproc.opl**. This may be used to extract pre-processor directives and macros from an OPP source file.

🗏 The filename does not include quotes.

#undef

A macro definition may be removed using the #undef directive followed by the macro name, for example:

```
#define DEBUG
#define MOD(a,b)
    (a-(a/b)*b)
#undef DEBUG
#undef MOD
```

'C' style operators

The OPL pre-processor will look for the characters '|' and '0x' within the OPP code and will convert them into something which the OPL translator will recognise. This feature is provided for greater compatibility with standard C include files.

The '|' character (entered into the OPP editor using the key combination **CTRL+124**) is used in C code, and is equivalent to the bit-wise OR operator in OPL, i.e. the following line...

```
#define FLAG (&1 | &4 | &32)
```

is equivalent to

```
#define FLAG (&1 OR &4 OR &32)
```

When using bit-wise operations which make use of the OPPEVAL() macro or in #if statements it is vital that the values are explicitly forced to integer values by preceding them with & or $. If this is not done then it can result in a logical OR operation rather than a bit-wise OR, consider:

```
OPPEVAL(1 | 2)
OPPEVAL(&1 | &2)
```

The first will use a logical OR since 1 and 2 are treated as floating point numbers and will give a result of -1, whereas the second will use a bit-wise OR and give the result 3. Logical and bit-wise OR's are discussed at the back of the Psion Programming Manual.

OPL uses the $ and & characters as prefixes for short and long hexadecimal numbers respectively. In 'C' 0x is used as the prefix. For convenience the pre-processor will convert any 0x to $ or &, so the following line

```
#define FLAGS (0x100|0x00000400)
```

is equivalent to

```
#define FLAGS ($100 OR &00000400)
```

If there are more than 4 digits following the 0x characters a long hexadecimal will be generated (using the & prefix), otherwise a short will be used (using the $ prefix).

A number of other C style operators which are found in normal C code are also supported by OPP:

Operator	Purpose	[Example]
++	set variable=variable+1	[i%++]
-=	set variable=variable-1	[i%--]
+=	set variable=variable+value	[i%+=2]
-=	set variable=variable-value	[i%-=2]
*=	set variable=variable*value	[i%*=2]
/=	set variable=variable/value	[i%/=2]

Building a single OPO/APP from N files

Using the #include directive it is possible to build a large OPO or APP from a number of separate OPP source code files. Splitting the code in this way makes it easier to manage.

At the end of your main source file add include statements to include each additional source file, e.g.

```
#include "UserInterface.opp"
```

```
#include "Engine.opp"
#include "Documents.opp"
```

EPOC32 & EPOC16 pointers

OPP allows you to write one set of OPP code that can be used with both EPOC16 and EPOC32 machines.

The basic problem with writing EPOC16 and EPOC32 code is that on EPOC16 pointers are 16bits wide, whereas on EPOC32 pointers should be 32bits wide.

Pointers are used when allocating memory dynamically and using OPP's C style structure facility.

To solve the problem OPP introduces a new pointer variable type identified with an @ character. Depending upon whether OPP is running on an EPOC16 or EPOC32 machine any variables declared with the @ symbol will be converted by OPP automatically into either 16bit or 32bit integers.

The following OPP code demonstrates this:

```
#ifdef EPOC32
    #pragma epoc32
#endif
STRUCT s
    int%
    ptr@
ENDS
PROC main:
local <s*>p@
    p@=alloc(SIZEOF(s))
    p@->int%=1
    p@->ptr@=p@
ENDP
```

With EPOC32 defined you get the following out of OPP:

```
PROC main:
local p&
    p&=alloc(6)
    pokew p&,1
    pokel p&+2,p&
ENDP
```

With EPOC32 not defined you get the following out of OPP:

```
PROC main:
local p%
    p%=alloc(4)
    pokew p%, 1
    pokew uadd(p%,2),p%
ENDP
```

LIBPROC

OPP supports a special type of procedure which is designed to enable procedure source libraries to be built. A library procedure is defined using **LIBPROC** rather than **PROC**:

```
LIBPROC test:
    print “test”
```

ENDP

A library procedure is identical to a normal procedure with one exception. When OPP processes an OPL source file it records which procedures have been called. When OPP finds a library procedure it checks the list of procedures which have been used and only includes the procedure if it has been called.

The library procedure allows a number of useful procedures to be grouped into a single OPL source file and then the file included as a whole in a number of separate programs. Only the procedures actually used within a program will be translated and included in the final OPO or OPA module.

- OPP processes source serially so that if **a LIBPROC** appears in the source code but is not referenced until later in the file then it will not be included.

Line continuation '\'

The line continuation character may be used in long or multi-statement macro definitions (as above) or in normal OPL code:

```
dchoice “Choice”, “Value1,
    \   Value2,
    \   Value3”,
```

Macro functions

Macro functions are an extension of the simple macros described previously. They are analogous to OPL procedures in that they take arguments. For example:

```
#define GT?(a,b)
    if (a<=b)  :\
        print a,”>“,b,”failed” :\
    endif
```

With the above definition the following OPP code

```
    GT?(x%,0)
```

would expand to

```
    if (x%<=0) :print x%,”>”,0,”failed” :endif
```

Macros - built-in

OPP includes a number of pre-defined built-in macros:

__FILE__

Name of file being translated, “C:\OPP\TEST.OPP”

__LINE__

Line number being translated, 23

__DATE__

Date of translation, “May 10 1998”

__TIME__

Time of translation, “23:53:06”

__PROC__

Name of current procedure, “main”

OPP

OPP version number, $19F

Psion* Indicates translating on a machine

DOS

Indicates translating on a PC

XTran*

Set if translating for S3 on a S3a

OsVersion*

SIBO/EPOC OS version number, 3.18F=$318F

RomVersion*

SIBO/EPOC ROM version number, 3.20F=$320F

PsuType*

Power supply type 0=old MC, 1=MC, 2=Series 3, 3=Series 3a

LcdType*

LCD type, 11=S3a

OPPEVAL()

Evaluate expression, OPPEVAL(SIZE+2)

SIZEOF()

See topic **STRUCT**, SIZEOF(my_struct)

OFFSETOF()

See topic **STRUCT**, OFFSETOF(my_struct,field)

*These macros are only available when using the EPOC16 version of OPP.

Multi-dimensional arrays

With standard OPL only one dimensional arrays are supported:

local a%(5)

REM 1-d array of integers

local a$(5,8)

REM 1-d array of 5x8 character strings

a%(1)=1

a%(2)=2

a$(1)="abcdefgh"

When using OPP multi-dimensional arrays may be used, for example:

local a%(5,3)

REM 2-d array of integers

local a$(2,5,8)

REM 2-d array of 8 character stringsa%(1,1)=1

a%(1,2)=2

a$(1,1)="abcdefgh"

OPP supports arrays with up to 20 dimensions!

The memory structure for a multi-dimensional array is such that the if you add one to the last subscript then this will be located in memory adjacent to the previous array element. For example a%(2,3) is stored as:

a%(1,1) a%(1,2) a%(1,3) a%(2,1) a%(2,2) a%(2,3)

Multi-dimensional arrays may be declared as local or global variables. However in the case of global variables an additional syntax is required in order to be able to access any variables which are outside the scope of the current OPL file. Suppose for example you have the following in one OPL module or source file:

global a%, b%(2,3)

In a separate OPL module which may be loaded into memory using the standard LOADM OPL function you can reference the global variables from the first module:

```
a%=1
b%(2,1)=2
```

In the case of the variable a% there is no problem with the above code, however in the case of the 2d array b%() OPP needs to know what the definition of the array was in order to work out where the element (2,1) is in memory. This is accomplished using an external variable definition:

```
extern b%(2,3)
```

The extern declaration syntax behaves exactly like a local or global OPL definition but is used merely to declare to OPP the dimensions of global multi-dimension arrays which are outside the scope of the current file.

opp_init.oph

Whenever OPP translates a file it will automatically include the following file before reading any OPP code:

```
OPP_INIT.OPH
```

With the EPOC32 version of OPP it will look for this file in the system include directory

```
\System\Apps\OPLPlus\Include\OPP_INIT.OPH
```

- If there are any OPP directives which are used in all OPP code then the OPP_INIT.OPH file is a good place to put them without having to explicitly use a #include <opp_init>.

STRUCT

The following sample code shows how structures are declared and used:

```
STRUCT user_data
    id%
    forename$(40)
    surname$(40)
    char_data#
    int_data%
    long_data&
    float_data
    string_data$(50)
ENDS
PROC main:
local <user_data*>p@
    p@ = make@:
    show:(p@)
    destroy:(p@)
ENDP
PROC make@:
local <user_data*>ptr@
    ptr@=alloc(SIZEOF(user_data))
    if ptr@=0
        stop
    endif
    ptr@->id%=1
    ptr@->forename$="Neuon”
    ptr@->surname$="Software”
```

```
    ptr@->char_data#=%a
    ptr@->int_data%=1
    ptr@->long_data&=123
    ptr@->float_data=1.23
    ptr@->string_data$="Some data"
    return ptr@
ENDP
PROC show:(<user_data*>ptr@)
    print ptr@->id%
    print ptr@->forename$
    print ptr@->surname$
    print ptr@->char_data#
    print ptr@->int_data%
    print ptr@->long_data&
    print ptr@->float_data
    print ptr@->string_data$
ENDP
PROC destroy%:(<user_data*>ptr@)     freealloc ptr@
ENDP
```

A structure is used to access a block of memory. The structure defines a number of fields which represent locations within the block of memory into which data may be written and from which may be read.

The STRUCT line starts a structure definition and names the structure. The structure name is used later when defining pointers or variables which point to a location in memory which contains data in the given format. The structure name must be unique amongst all structure definitions.

The structure and field names may contain any characters used in a normal OPL variable plus the underscore character. Unlike OPL variables the names may be of any length.

The STRUCT line is followed by the names of the fields within the structure, with one line per field. These field names are used when referencing the memory within a structure. The **ENDS** line marks the end of the structure.

SIZEOF() is a built-in macro which gives the size of a named structure. This is required when allocating memory, as shown above.

OFFSETOF(struct,field) will give the offset of the field called "field" within the structure called "struct".

```
    OFFSETOF(user_data, int_data%)
```

gives 85 since the int_data% field is offset 85 bytes into the user_data structure.

The OFFSETOF() macro is typically used when you need to access the memory address of a given field directly, e.g. you could write:

```
    peekw Uadd(ptr@,OFFSETOF(user_data,int_data%))
```

which would be equivalent to

```
    ptr@->int_data%","
```