Oberon-A.doc

Contents

Chapter 1

Oberon-A.doc

1.1 Oberon-A

```
                   $RCSfile: Oberon-A.doc $
Description: Overall documentation for Oberon-A, release 1.5

 Created by: fjc (Frank Copeland)
  $Revision: 1.9 $
    $Author: fjc $
      $Date: 1995/07/02 21:44:21 $

Copyright © 1994-1995, Frank Copeland.
```

```
*** IMPORTANT NOTE FOR USERS OF OBERON-A 1.5 OR EARLIER VERSIONS ***

You are strongly advised to read the
              Updating
               section before proceeding
any further.
```

```
New links are marked with '+'.


              ~Oberon~~~~~~~~~~~~~
                What is Oberon?

              ~Oberon-A~~~~~~~~~~~
                What is Oberon-A?

              ~Distribution~~~~~~~
                Distribution and Copyright


              ~Requirements~~~~~~~
                What do I need to run Oberon-A?

              ~Installation~~~~~~~
                How do I install Oberon-A?
```

1.2 What is Oberon?

```
                The following are taken from  the  FAQ  file  for  the  comp. ↩
                    lang.oberon
Usenet newsgroup, Copyright © 1994 Michael  Gallo  and  reproduced  with
permission.
```

1.3 WHAT IS OBERON?

From "The Oberon Guide"

Oberon is simultaneously the name of a project and of its outcome. The project was started by Niklaus Wirth and [Jrg Gutknecht] late in 1985 with the goal of developing a modern and portable operating system for personal workstations. Its results are an implementation of the system for the Ceres computer and a programming language.

The development of the language Oberon needs perhaps a short justification. It became quite inevitable because the type-system of available languages turned out to be too restrictive to express the desired data model in a natural and safe way.

Easy introductions to both aspects of the Oberon project can be found in back issues of BYTE magazine. The operating system is overviewed in "Oberon: A Glimpse at the Future", volume 18 number 6 (May 1993). The Oberon language is examined in "Oberon", volume 16 number 3 (March 1991). Both articles are by BYTE's European correspondant, Dick Pountain.

1.4 THE PROGRAMMING LANGUAGE OBERON

From "From Modula to Oberon"

The programming language Oberon is the result of a concentrated effort to increase the power of Modula-2 and simultaneously to reduce its complexity. Several features were eliminated, and a few were added in order to increase the expressive power and flexibility of the language. This paper describes and motivates the changes. The language is defined in a concise report.

Whereas modern languages, such as Modula, support the notion of extensibility in the procedural realm, the notion is less well established in the domain of data types. In particular, Modula does not allow the definition of new data types as extensions of other, programmer-defined types in an adequate manner. An additional feature was called for, thereby giving rise to an extension of Modula.

. . . .

The evolution of a new language that is smaller, yet more powerful than its ancestor is contrary to common practices and trends, but has inestimable advantages. Apart from simpler compilers, it results in a concise defining document, an indispensable prerequisite for any tool that must serve in the construction of sophisticated and reliable systems.

Among the eliminations in the move from Modula-2 to Oberon are variant records, opaque types, enumeration types, subrange types, the basic type CARDINAL, local modules, and Modula's WITH statement. The major addition to Oberon is the concept of type extension (i.e., single inheritance) for records.

1.5 Bibliography

THE PROGRAMMING LANGUAGE

"Type Extensions" by N. Wirth; ACM Transactions on Programming Languages and Systems; 10, 2 (April 1988) 204-214.

"From Modula to Oberon" by N. Wirth; Software: Practice and Experience; 18,7 (July 1988) 661-670.

"The Programming Language Oberon" by N. Wirth; Software: Practice and Experience; 18,7 (July 1988) 671-690.

"Variations on the Role of Module Interfaces" by J. Gutknecht; Structured Programming; 10,1 (January 1989) 40-46.

"Object Oberon -- A Modest Object-Oriented Language" by H. Mssenbck and J. Templ; Structured Programming; 10,4 (April 1989) 199-207.

A New Approach to Formal Language Definition and Its Application to Oberon by M. Odersky; Verlag der Fachvereine Zrich; 1989.

"Oberon" by Dick Pountain; BYTE; March 1991.

"The Programming Language Oberon-2" by H. Mssenbck and N. Wirth; Structured Programming; 12,4 (April 1991).

Programming in Oberon: Steps Beyond Pascal and Modula-2 by M. Reiser and N. Wirth; ACM Press; 1992.

"A Systematic Approach to Multiple Inheritance Implementation" by J. Templ; ACM SIGPLAN Notices; Volume 28, Number 4 (April 1993).

Object Oriented Programming in Oberon-2 by H. Mssenbck; Springer-Verlag; 1993.

A Programming Language for Vector Computers by R. Griesemer; Swiss Federal Institute of Technology (ETH Zurich); Dissertation Number 10277, 1993.

THE OPERATING SYSTEM

"Designing a System from Scratch" by N. Wirth; Structured Programming; 10,1 (January 1989) 10-18.

"The Oberon System" by N. Wirth and J. Gutknecht; Software:

Practice and Experience; 19,9 (September 1989) 857-893.

The Oberon System: User Guide and Programmer's Manual by M. Reiser; ACM Press; 1992. This was reviewed in Computing Reviews articles 9109-0679, 9209-0651, 9209-0652, and 9207-0443.

Project Oberon: The Design of an Operating System and Compiler by N. Wirth and J. Gutknecht; ACM Press 1992.

"Oberon: A Glimpse at the Future" by Dick Pountain; BYTE; May 1993.

"Implementing an Operating System on Top of Another" by M. Franz; Software: Practice and Experience; 23,6 (June 1993) 677-692.

Distributed Object-Oriented Programming in a Network of Personal Workstations by Spiros Lalis; Swiss Federal Institute of Technology (ETH Zurich); 1994 (in preparation).

1.6 THE PROGRAMMING LANGUAGE OBERON-2

From "Differences between Oberon and Oberon-2"

> Oberon-2 is a true extension of Oberon. . . .
>
> One important goal for Oberon-2 was to make object-oriented programming easier without sacrificing the conceptual simplicity of Oberon. After three years of using Oberon and its experimental offspring Object Oberon we merged our experiences into a single refined version of Oberon.
>
> The new features of Oberon-2 are type-bound procedures [i.e., virtual methods], read-only export of variables and record fields, open arrays as pointer base types, and a with statement with variants. The for statement is reintroduced after having been eliminated in the step from Modula-2 to Oberon.

1.7 THE 'OBERON FAMILY' OF LANGUAGES

Object Oberon is a now defunct, experimental extension of Oberon featurifg "classes", structures somewhere between modules and records. It evolved into Oberon-2.

Seneca was also an experimental extension of Oberon. It focused on numerical programming on vector computer architectures. It evolved into Oberon-V.

Oberon-V is an experimental dialect (but not a superset) of Oberon. It is concerned with issues of numerical computing, array processing, and code verification. Since it was originally aimed at vector architectures in general and the Cray Y-MP in particular, no Oberon-V compiler has yet been implemented for the Oberon System.

1.8 What is Oberon-A?

Oberon-A is an Oberon-2 compiler and associated utilities for the Commodore Amiga personal computer. The Oberon-A compiler translates programs written in Oberon or Oberon-2. It also supports a number of language extensions that assist programming in the Amiga's unique environment. It produces MC68000 machine code directly without an intermediate assembly language stage. The object files it creates are in standard AmigaDOS format.

The Oberon-A Library is a collection of library modules that can be linked with programs created with Oberon-A. There are a number of useful modules, including the beginnings of an object-oriented application Framework.

The Oberon-A Interface is a collection of library modules that provides a complete interface to the Amiga's operating system. This is based on Commodore's 40.15 interfaces, for release 3.1 of the operating system.

The Oberon-A archive contains a programming environment utility (FPE), the compiler (OC), an error lister (OEL, by Johan Ferreira), a link utility (OL), preferences editors (OCPrefs and OLPrefs), a recompilation utility (ORU), and the source code for the Library and Interface modules. Full source code is included for all modules and programs where available. The archive also contains other software needed to use Oberon-A, most importantly the BLink linker.

1.9 Distribution and Copyright

Copyright © 1994-1995, Frank Copeland

The Oberon-A Interface is also free software, but it is not subject to the GNU licence. Instead, it may be freely distributed, but only as part of the Oberon-A archive for use in creating software using the Oberon-A compiler.

Programs created with Oberon-A may be distributed under any terms their creator desires. However, as all such programs must be linked with one or more Oberon-A Library modules, the programmer should be aware of the requirements of clause~6 of the GNU Library General Public License. Oberon-A is intended mainly for personal use, and for the creation of other free software. Commercial programmers may prefer to use a commercial Oberon compiler.

Parts of the Oberon-A compiler and Library are based on source code developed at ETH Zuerich. Permission to use, copy, modify and distribute this software is granted by ETH (see the file ETH-Copyright.txt).

Oberon-A Error Lister (OEL) and Oberon-A Bump Revision (OBumpRev) are:

Copyright (C) 1994 Johan Ferreira

OEL and OBumpRev are free software and is distributed under the GNU General Public Licence. See OEL.guide and OBumpRev.guide for details.

The archive file containing Oberon-A also includes a number of other freely-distributable programs that are used by Oberon-A. These programs are copyrighted by their authors and distributed under conditions set by those authors. These programs include:

BLink, Copyright © 1986, The Software Distillery.
intuisup.library, Copyright © 1992, Torsten Jürgeleit.

This document and others in the archive are in Commodore's AmigaGuide hypertext format. A shared code library and reader program are included in the archive. AmigaGuide, AmigaGuide.info and amigaguide.library are:

(c) Copyright 1992 Commodore-Amiga, Inc. All Rights Reserved.
Reproduced and distributed under license from Commodore.

AMIGAGUIDE SOFTWARE IS PROVIDED "AS-IS" AND SUBJECT TO CHANGE; NO WARRANTIES ARE MADE. ALL USE IS AT YOUR OWN RISK. NO LIABILITY OR RESPONSIBILITY IS ASSUMED.

1.10 What do I need to use Oberon-A?

Oberon-A requires a Commodore Amiga personal computer with at ↩ least 1MB
of memory (more is recommended), running Release 2.04 (Kickstart 37+) or later of the Amiga operating system. All the Oberon-A programs now REQUIRE 2.04+, and will *not* run under AmigaOS 1.3. In addition, all programs created using Oberon-A also require AmigaOS 2.04. These restrictions may be relaxed in a future release.

Oberon-A can be run from floppies, if sufficient RAM is available for use as a RAM disk and for making the main programs resident. See

Floppy~Installation

. However, a hard disk is highly recommended. The contents of the archive when decompressed will take up 4-5 MB of disk space. Further disk space will be required to develop programs.

A linker is needed to create executable programs from the object files generated by the compiler. A freely-distributable linker, BLink, is included in the Oberon-A archive. Unfortunately, this linker can produce numerous Enforcer hits in some circumstances (specifically, when it is run *without* the NODEBUG option. It may be replaced in time. Commodore's ALink linker (distributed with the Native Developer's Kit) and AmigaOberon's OLink have also been shown to work with Oberon-A.

The FPE program requires intuisup.library. OCPrefs and OLPrefs require EAGUI.library. These are included in the Oberon-A archive.

The documentation files are written in Commodore's AmigaGuide hypertext format. Users of AmigaOS 2.1 or greater already have the software needed to view these files. A minimal AmigaGuide installation is included in the Oberon-A archive for users of earlier Kickstart versions. The complete AmigaGuide distribution can be found on AmiNet in the text/hyper directory, or in the Fred Fish collection.

There is no editor provided with Oberon-A. Many suitable editors are available as freely distributable software. The Memacs editor distributed with the operating system can also be used.

1.11 How do I install Oberon-A?

Oberon-A can be installed temporarily while you evaluate it. ↩ This mainly involves assigning logical device names and possibly copying shared libraries to your libs: directory. The Oberon-A library modules must also be compiled the first time Oberon-A is installed. The temporary installation process must be repeated each time you re-boot your Amiga. Permanent installation involves modifying your startup sequence.

```
~Temporary~installation~
  Temporarily installing Oberon-A

~Library~modules~~~~~~~~
  Compiling the library modules

~Permanent~Installation~
  Permanently installing Oberon-A

~Uninstalling~~~~~~~~~~~
```

```
                  Removing Oberon-A from your system


                ~Floppy~installation~~~~
                  Installing Oberon-A on floppies
```

1.12 Temporarily installing Oberon-A under Kickstart 2.0+

Run the script in the file Install, either by typing "Execute Oberon-A/Install/Install" at the Shell prompt or by double-Clicking its icon. The script sets up a few logical assignments and makes the libraries used by Oberon-A available with the ADD option of the AmigaDOS Assign command. It also creates file links to aliases for the compiler and link utility that are useful for Workbench users.

1.13 Compiling the library modules

To save space, the symbol and object files for the Oberon-A library modules are not included in the distribution. A script file is provided which will compile the modules and place the symbol and object files in the OLIB directory. To run the script, double-click on the CompileLibs icon. This script will take a considerable time to complete, depending on your hardware. You will be given a chance to back out if you want.

Starting with release 1.6, the Oberon-A compiler is able to generate a number of different code and data models. The default settings are to use the SMALLCODE and SMALLDATA models. If you wish to use alternative settings, you will need to edit the OC.prefs settings file, using the OCPrefs editor.

1.14 Permanently installing Oberon-A

Permanent installation requires the addition of two assigns and two paths to your S:User-Startup. They are:

```
  Assign OBERON-A: Oberon-A
  Assign OLIB:     OBERON-A:OLIB
  Path   OBERON-A: OBERON-A:C add
```

OC-Lib and OL-Small are actually links to OC and OL respectively. The links are created for you when you run the Install script for the first time. If you have not run Install, do so, or execute the following commands from a Shell:

```
  MakeLink OBERON-A:OC-Lib OBERON-A:OC
  MakeLink OBERON-A:OL-Small OBERON-A:OL
```

The environment settings for Oberon-A can be handled in two ways:

- They can be left where they are in OBERON-A:env-archive, and the following command added to your S:User-Startup

```
    Copy OBERON-A:env-archive TO ENV: ALL NOREQ QUIET
```

- They can be moved to your own ENVARC: directory.

As BLink reportedly produces Enforcer hits, you may wish to replace it with another linker. Suitable linkers include Commodore's alink and AmigaOberon's OLink. This will involve installing the alternative linker and changing the preferences settings for the OL program. See OL.doc and OLPrefs.doc for details.

If you use it, FPE must be able to locate it's setup files. When extracted from the archive, these are placed in the directory Oberon-A/S. FPE looks for its setup files in the directory "FPE:S", so the temporary installation assigns "FPE" to the Oberon-A directory. If you keep the same directory organisation, simply place the line

```
  Assign FPE: Oberon-A
```

in your S:User-Startup file. If you move the Oberon-A/S directory you will need to adjust this accordingly.

FPE also requires intuisup.library. This must be copied to the SYS:libs directory, or to another directory to which LIBS: has been assigned. The simplest way to do this is to leave it in OBERON-A:libs, and place this line in your S:User-Startup:

```
  Assign LIBS: OBERON-A:libs add
```

1.15 Removing Oberon-A from your system

THIS SPACE INTENTIONALLY LEFT BLANK (just kidding)

If you decide not to permanently install Oberon-A, it is a simple matter to remove all traces of it from your system. First, delete the Oberon-A directory and its contents. Next, if you copied arp.library and/or intuisup.library to your LIBS: directory, delete them. Finally, remove any Assign statements you put into your startup sequence.

Even if you are not impressed by Oberon-A, I would like to hear your comments and assessment of it. See
Bug~reports~and~suggestions
.

1.16 Installing Oberon-A on floppies

Edmondo Tommasina reports that he has successfully installed Oberon-A on a twin-floppy Amiga 2000 with 3MB of RAM. The following setup is as Edmondo described it to me, with modifications to account for the

```
differences between Release 1.4 and Release 1.5.

Two disks are required. The first is a program disk. The second holds
the symbol and object files for the libraries.

Disk 1
------
OBERON:
   c (dir)
     Amigaguide                        BLink
     ORU
   Catalogs (dir)
     ...
   libs (dir)
     guienv.library                    intuisup.library
   macros (dir)
     DoLink.rexx                       DoOC.rexx
     DoOL.rexx                         DoRun.rexx
     ReadErr.aed
   olib (dir)
     ClassFace.o                       LMath.o
     Mark.o                            rexxvars.o
   s (dir)
     Alternate.fpe                     AltORU.with
     Default.fpe                       Oberon-Shell
     Oberon-Startup                    ORU.with
FPE                               FPE.info
Index.doc                         Index.doc.info
Oberon-A.doc                      Oberon-A.doc.info
OBumpRev                          OEL
OC                                OC.info
OC.prefs                          OC.prefs.info
OCLib.prefs                       OCLib.prefs.info
OCPrefs                           OCPrefs.info
OD                                OD.info
OL                                OL.info
OL.prefs                          OL.prefs.info
OLPrefs                           OLPrefs.info
OLSmall.prefs                     OLSmall.prefs.info
Setup                             Setup.info


Disk 2
------
OberonLib:
All the .sym and .obj files


Scripts
-------

The script OBERON:Setup is executed to set up the system.

Script name: setup
==================

;       Edmondo Tommasina 25.12.94
;       Oberon installation for floppy drives
;       Modified for Oberon-A release 1.5 by Frank Copeland

FailAt 5

Echo "Making logical directory assignments for:"
Echo "    OBERON-A:"
Echo "    FPE:"
Echo "    OLIB:"

resident C:Assign PURE
Assign OBERON-A:        OBERON:
Assign FPE:             OBERON-A:
Assign OLIB:            OBERON-A:OLIB

;-----------------------------------------------------------------------
;       Make shared libraries available

Assign LIBS:            OBERON-A:LIBS ADD

;-----------------------------------------------------------------------
;       Make resident usefull commands or copy to ram:

echo "Copy to Ram: usefull commands"

copy to ram: OBERON-A:c/Blink
copy to ram: OBERON-A:#?.prefs

echo "Make resident usefull commands\n"

Resident C:dir  PURE
Resident C:list PURE
Resident C:info PURE
Resident C:delete PURE

; Oberon-A programs can now be made resident

Resident OBERON-A:OC PURE
Resident OBERON-A:OL PURE
Resident OBERON-A:OD PURE
Resident OBERON-A:c/ORU PURE
Resident OBERON-A:c/OEL PURE

newshell FROM OBERON-A:s/oberon-shell

; end of setup

Script name: Oberon-Shell
=========================

Prompt "*E[32m%N.%S>*E[31m "
cd ram:
stack 15000
assign OLIB: OberonLib: ADD


Libraries
```

If there is enough RAM the quickest way to compile the library modules is to copy the source code to RAM:, compile, then copy the symbol and object files to the OberonLib: disk.

1.17 Updating from an earlier version

If you are still using version 1.4 of Oberon-A, see Updating~from~1.4 before reading any further.

The OCPrefs and OLPrefs programs have been considerably enhanced and now have fully font-adaptive GUIs. As a result of this, the command-line and Workbench arguments for these programs, as well as OC and OL, have been greatly simplified. In particular, arguments that allowed the user to over-ride preferences settings have now been eliminated. If you are using scripts or Shell aliases to run these programs you may need to modify them.

The preferences files produced by OCPrefs and OLPrefs have been changed, but should still be backward-compatible with previous versions. Old versions of preferences files should be loaded into the appropriate editor and saved in the new format. OCPrefs now has a seperate dialog for changing the defaults for compiler options and pragmas, and these should now be deleted from the SET and CLEAR settings.

OC now has the ability to generate a number of code and data models. The default settings are for the small code and small data models. Please note that these changes severely limit the choice of linker. See Code~Models for details.

The support for Matt Dillon’s dlink linker has been removed from OL for the moment.

Unlike earlier versions, all library modules are now compiled with run-time checks enabled by default. As a result of this, you may find software that previously compiled and ran without problems now generates compiler and run-time errors, especially range check errors. I know it happened to me. This is a GOOD THING(tm) - hidden bugs are still bugs.

An implementation of the Oberon0 System described in Mössenböck’s "Object Oriented Programming in Oberon-2" is now included as an extended example. See Oberon0.doc for details.

See Changes.doc for a more complete listing of the changes made in this release.

Updating an existing Oberon-A installation should be done as follows:

1. If you have modules of your own that will need to be recompiled,

run the ORU utility to create lists of modules that can be used with the compiler's BATCH option to automate the process. See ORU.doc for details.

2. Unpack the sub-archives in the Oberon-A archive into the Oberon-A directory.

3. Recompile all the library modules, using the
```
         CompileLibs
          script
```
in the Install directory.

4. Delete any files that have been renamed or removed. See Changes.doc for a list of the files affected. A script file called DeleteOld can be found in the Install directory. This will delete old versions of files that have been renamed or removed. Please study this script carefully before using it. If you are in any doubt, don't use it, and make any necessary changes by hand.

1.18 Updating from version 1.4 of Oberon-A

The settings files for the FPE program now use a new, compact format. Settings files from earlier versions are now obsolete, and must be deleted.

The compiler switches used in Release 1.4 and earlier have now been replaced by more verbose and readable pragmas. A utility program called ConvertSwitches has been written to automatically convert the old-style switches to the corresponding pragmas.

From the Shell:

- CD to the directory containing the source code to be converted.
- Call 'ConvertSwitches #?.mod'.

From the Workbench:

- Make sure the following tooltype has been created in the ConvertSwitches icon: 'FILES=#?.mod'.
- Click on the directory containing the source code to be converted.
- Hold down the shift key and double-click ConvertSwitches.

In most cases this will be sufficient, however modules that contained the old $P- switch will need to be edited. The $P- switch will be replaced with the <*STANDARD-*> option, which must be moved to a position *before* the MODULE keyword. If it is encountered after the MODULE keyword, the compiler will report an error. Switches that used the '=' character to return to the default value are converted to use the '+' character.

The syntax for making Amiga library function calls has also been changed. It is no longer necessary to specify the name of the library base variable in the call. Any existing source code must be edited to remove the base variable name from all library calls. Most if not all library interface modules follow the convention of calling the base variable 'base'. This means that in almost all cases a simple search and replace to delete all occurrences of the string 'base.' should be sufficient. However, care should be taken to avoid changing statements that are actually referencing fields in the library base variable.

See OC.doc for details.

Extensive modifications have been made to the Oberon-A Interface modules to bring them as close as possible to the interfaces used with the AmigaOberon compiler. The main changes are:

- Constant names have been changed to conform with the same naming conventions. This usually means that any prefix is deleted and the remaining identifier changed so that it starts with a lower-case letter. For example, 'idcmpActivate' becomes 'activate'. Unfortunately, the usage is not consistent, and each case has to be handled individually. As a last resort, the compiler will report as errors any use of the old identifiers.

- The way in which record types are extended has been changed. For an example, see the Node and Message types in module Exec. The main difference is that the base type is now the first field in the extended type. This means that any access to the fields of the base type must be qualified with the name of the first field. For example, 'name := message.name' becomes 'name := message.node.name'.

- Interface modules for shared code libraries no longer export an OpenLib() procedure. All such modules now attempt to open the library automatically. In cases where the library should always be available, failure to open it causes the program to halt with an error code of 100. In other cases the programmer should check that the library base variable is not NIL before calling any library functions. A 'Lib.OpenLib(TRUE)' statement can be replaced with 'ASSERT(Lib.base#NIL,100)'.

- All references to the type Exec.STRPTR have been changed to Exec.LSTRPTR. The same change must be made in your own source code.

The interface to module Strings has been changed. The most important changes are in the order of parameters in the procedures. Some procedures have been renamed, and others moved to a seperate Strings2 module.

Module Errors must now be explicitly initialised by a call to Errors.Init().

Module Kernel is now fully integrated with the compiler. Kernel.New() has been renamed to Kernel.Allocate(), and Kernel.NewFromTag() has been renamed to Kernel.New().

A number of files have been renamed, and some have been removed from the archive. Renamed files are replaced with a dummy file containing message pointing to the new file.

1.19 Tutorial

```
                This tutorial will take you through the steps needed to create ←
                   the
classic 'Hello World' program using Oberon-A. This will involve creating
the file HelloWorld.mod in a text editor, and compiling, linking and
running it under the Shell, Workbench and/or FPE environments.

Before you start, you need to select, install and configure a text
editor. MEmacs on the Extras disk is suitable if you have no other
editor. If you are going to use FPE, you must also configure it to
correctly call your choice of editor. See Getting~started~with~FPE.

If you have not already done so, install Oberon-A either temporarily or
permanently. See
              Installation
              . Then select a directory to work in,
creating one if necessary. Create a sub-directory called 'Code' in your
work directory. Using a text editor, create a file called
HelloWorld.mod, containing the following source text:

    MODULE HelloWorld;

    IMPORT Out;

    BEGIN
      Out.String ("Hello world, Oberon-A is calling"); Out.Ln
    END HelloWorld.

Once you have done this, proceed with one of the following tutorials:


              ~Shell~~~~~

              ~Workbench~

              ~FPE~~~~~~~
```

1.20 Workbench Tutorial

```
                Make sure you can access the work directory and the 'HelloWorld. ←
                   mod'
file from the Workbench. Create icons if necessary. Then follow the
following steps:

- Compile the module:

  o Open the Oberon-A drawer and the drawer for the work directory.
  o Select 'HelloWorld.mod', hold down the shift key, and double-click
    on the 'OC' icon.
  o If any errors are reported, make sure that the source text in
    'HelloWorld.mod' is *exactly* as described in the
              Tutorial
                    introduction, then repeat the process.
```

```
- Link the module:

  o Close the drawer for the work directory.
  o Select the 'OL' icon, then use the Workbench Icons-Information
    menu item to edit the icon's tooltypes. Create a new tooltype, and
    enter 'PROG=HelloWorld'. Click on the 'Save' button.
  o Select (don't open) the drawer for the work directory. Hold down
    the shift key, and double-click on the 'OL' icon.
  o Open the drawer for the work directory, and confirm that two new
    icons have appeared: one for 'HelloWorld' and one for
    'HelloWorld.with'.
  o Edit the 'OL' icon and delete the 'PROG=' tooltype you added
    earlier.

- Run the program:

  o Double-click the 'HelloWorld' icon.

If you don't like the console window that gets opened for you, edit the
icon for 'HelloWorld' and add or edit a "WINDOW=" tooltype describing a
console window that suits you. See the AmigaDOS manual for details.

The instructions for linking 'HelloWorld' given above assume that it
has not been linked before, and has no icon. If there is already a copy
of 'HelloWorld' in the work directory, follow these steps instead:

- Link the module:

  o Open the drawer for the work directory.
  o Select the 'OL' icon, hold down the shift key, and double-click on
    the 'HelloWorld' icon. NOTE: don't do it the other way around, that
    will simply run 'HelloWorld'.
```

1.21 Shell Tutorial

```
                Make sure the work directory is the current directory, then ↩
                    execute the
following steps:

- Compile the module:

  o Type 'OC HelloWorld.mod'
  o If any errors are reported, make sure that the source text in
    'HelloWorld.mod' is *exactly* as described in the
              Tutorial
                    introduction, then repeat the process.

- Link the module:

  o Type 'OL HelloWorld SCAN LINK SETTINGS=OL.prefs'

- Run the program:

  o Type 'HelloWorld'
```

```
The instructions for linking 'HelloWorld' given above assume that it
has not been linked before. If there is already a copy of 'HelloWorld'
in the work directory, you can leave out the SCAN argument for OL.
```

1.22 FPE Tutorial

```
                Run the FPE program from the Workbench or the Shell. Then  ↩
                   execute the
following steps

- Select the project:

  o Select the 'Select Project' item from the 'Project' menu.
  o Use the file requester to move to the work directory.
  o Type 'HelloWorld' into the file name gadget and hit 'Ok'.

- Compile the module:

  o Make sure that 'HelloWorld' appears in the 'Modules' list box, and
    is selected.
  o Click on the 'Compile' button.
  o If any errors are reported, make sure that the source text in
    'HelloWorld.mod' is *exactly* as described in the
              Tutorial
                   introduction, then repeat the process.

- Link the program:

  o Make sure that 'HelloWorld' appears in the 'Project' box in the FPE
    window.
  o Click on the 'Link' button.

- Run the program:

  o Click on the 'Run' button.
```

1.23 Programming with Oberon-A

```
                Apart from an overview of the Oberon-A programming cycle, this  ↩
                   section
discusses a number of the more advanced issues that arise when writing
Oberon-A programs. Any suggestions for additional areas to be covered or
expanded on are very welcome.
```

```
              ~Overview~~~~~
                 The Oberon-A programming cycle.

              ~Cleanup~~~~~~
                 Automatic cleanup of Oberon-A programs
```

1.24 Overview of the programming cycle

An Oberon program consists of one or more modules, each of which is compiled seperately. Many modules are "library" modules, which can be re-used many times in different programs. Each program has a "main program" module which acts as the entry point to the program. Any module may be used for this: there is no specific language construct to indicate that a module is the main program module.

Each module is contained in a single text file. Modules may be edited by any standard text editor that produces plain ASCII files. The Oberon-A compiler (OC) is then used to check the module for syntax errors and to translate it into an object file containing machine code. If the compiler detects any errors, it produces an error file indicating their location and nature. The error lister (OEL) can then produce a description of the error and the context in which it occurs.

A program is created out of its component modules by linking it. This combines all the object files into a single file and resolves any references between modules. This function is performed by a linker program, such as BLink. The linker must be told which module is the main program module and which other modules are to be included in the program. The Oberon-A pre-link utility (OL) is used to generate the information the linker requires.

The entire process can be summarised as follows:

```
REPEAT
  FOR each module in the program needing work DO
    REPEAT
      Edit the module source code
      Run OC to compile the module
      IF there are errors THEN
        Run OEL to generate an error report
      END
    UNTIL no more syntax errors are reported
  END
  Run OL to generate information for the linker [*]
  Run the linker to link the program
  Test and evaluate the program
UNTIL satisfied with the result

* This is only necessary the first time the program is linked and
whenever modules are added to or removed from the program.
```

1.25 Handling run-time errors

The Oberon-A compiler inserts many run-time checks into the code it
generates. These checks trap a number of errors, such as arithmetic
overflows and de-referencing NIL pointers. When such an error is
detected, a processor trap instruction (TRAP, TRAPV or CHK) is executed.

If the program does not contain code to handle such traps, it will crash
and will almost certainly crash the entire system with it. In order to
avoid this, the program should install a trap handler which deals with
such errors in a graceful and system-friendly way. A default trap
handler is provided in module~Kernel. It simply halts the program, and
executes the list of procedures installed with Kernel.SetCleanup(). See

Automatic~cleanup
. This allows most programs to fail gracefully, and
gives them the opportunity to clean up after themselves. Installing the
handler is simple: just include a call to Kernel.InstallTrapHandler() in
your program. This should be placed at the very beginning of the body
of the main program module. The handler must be removed before the
program exits, otherwise it may cause problems if the program is run
from the Shell. This is achieved by calling Kernel.RemoveTrapHandler()
from inside a cleanup procedure installed with Kernel.SetCleanup(). See

Listing~1
.

Using the default trap handler will not provide you with any information
about the cause of the error, apart from a cryptic error code if the
program is run from a Shell. Importing module~Errors into your program
will remedy this. Module Errors installs a cleanup procedure that looks
at the return code generated by the run-time system and interprets it to
produce a requester containing a meaningful error message. The error
message will identify the type of error and in most cases will indicate
the module and the position in the source text at which the error
occurs. To activate this service, simply include a statement in your
program that calls Errors.Init(). This should be placed at the very
beginning of the body of the main program module. In this case there is
no need to call Kernel.InstallTrapHandler() or RemoveTrapHandler(); this
is handled automatically for you by module Errors. See
Listing~2
.

Run-time error checking imposes a significant amount of overhead, and
adds considerably to the size of object files. This is especially true

of NIL checking, although planned changes to the code generation will improve this. If you do not wish to incur these costs, you can switch off the generation of run-time error checks either in the preferences settings for the compiler, or within your source code. See OC.doc and OCPrefs.doc for details. However, before you do so, you might want to reflect on the wisdom of Hoare (I think) who wrote something like this: developing a program with error checking enabled, but turning it off for production code, is like giving life-jackets to sailors training on dry land, and taking them away when the sailors go to sea.

1.26 Debugging Oberon-A programs

Debugging can take two forms:

- Static debugging, which relies on run-time error checking, assertions and debugging print statements; and
- Interactive debugging, where the program is run under the control of a debugger.

Static debugging

Static debugging relies on the run-time~checks generated by the compiler to detect and report as many errors as possible, as close to their source as possible. For this to work, the compiler must have all run-time checks enabled. Edit the compiler's preferences settings to ensure that all run-time checks are enabled. Remove any pragmas in the source code that disable run-time checks. Make sure the INITIALISE option is on. See OC.doc and OCPrefs.doc for details.

Run-time checks are only useful if the type and location of the error can be identified. Module~Errors is designed to provide exactly this information, and should be imported by the main program module. See Listing~2 . For most run-time errors, a requester will report the module it occurred in, the line and column in the module's source text at which it occured, and the nature of the error.

Assertions are also useful for static debugging. Assertions can be used to test the parameters passed to a procedure to ensure that they meet formal pre-conditions imposed by the programmer. They can also be used to test that the results of function procedures meet similarly imposed post-conditions. Assertions are implemented with the standard procedure ASSERT, which may include an optional return code.

Module Errors supports assertions by implementing the following conventions:

- If the program exits with a return code of 97, it reports that a pre-condition has been violated. For example, ASSERT (...,97) can be used to test a pre-condition.

- If the program exits with a return code of 98, it reports that a post-condition has been violated. For example, ASSERT (...,98) can be used to test a post-condition.

In both cases the module and location of the assertion are reported in a requester.

Another useful convention recognised by module Errors is to place the statement HALT (99) in the body of an otherwise empty procedure. If the procedure is called, module Errors will report that an attempt has been made to call an un-implemented procedure.

Debugging print statements can be used to report on the state of a computation, with or without the support of assertions. For example, they can be used to print out the actual value of a parameter that is causing a pre-condition check to fail. This is often sufficient to indicate the cause of the problem. See
listing~4
for an example.

Debugging print statements can easily be enabled and disabled by using the compiler's conditional compilation feature. Listing 4 also demonstrates this.

Interactive debugging

The Oberon-A package does not include an interactive debugger, and there are no plans to produce one. However, there are a number of third-party debuggers available which can be used with Oberon-A. These include MykesBug (which is full of bugs itself) and PowerVisor.

In order to use an interactive debugger with an Oberon-A program, it is necessary to tell the compiler to output symbol hunks in the object code. This is achieved by compiling all modules with the DEBUG argument set. The program must then be linked in such a way that the symbol hunks are included in the final executable. With BLink this happens by default, unless the NODEBUG argument is set. Other linkers, such as dlink, must be specifically instructed to include the symbol hunks. See the documentation for the linker in use for details.

The symbols should be displayed by the debugger when it lists the code of the Oberon-A program. See
Symbol~format
for a description of how the
compiler constructs the symbols.

The default trap handler in module Kernel may interfere with the trap handler used by the debugger. In this case the program must be recompiled and re-linked without the trap handler. Comment out or delete any calls to Kernel.InstallTrapHandler() or Errors.Init(). It may also be necessary to remove any other trap handlers installed by the program.

1.27 Profiling Oberon-A programs

Oberon-A programs can be profiled using the freely distributable profiler AProf written by Michael Binz. This is available on AmiNet in the dev/misc directory.

In order to use AProf to profile an Oberon-A program, it is necessary to tell the compiler to output symbol hunks in the object code. This is achieved by compiling all modules with the DEBUG argument set. The program must then be linked in such a way that the symbol hunks are included in the final executable. With BLink this happens by default, unless the NODEBUG argument is set. Other linkers, such as dlink, must be specifically instructed to include the symbol hunks. See the documentation for the linker in use for details.

The symbols will be displayed by AProf in its main window, along with timing information. See
Symbol~format
for a description of how the
compiler constructs the symbols. The symbol pattern in the AProfs preferences dialog should be cleared; it is safe to display all symbols generated by the compiler.

The default trap handler in module Kernel interferes with the trap handler used by AProf. The program must be compiled and linked without the trap handler. Comment out or delete any calls to Kernel.InstallTrapHandler() or Errors.Init(). It may also be necessary to remove any other trap handlers installed by the program. You should therefore be confident that your program will run without causing any run-time errors before trying to profile it.

1.28 Automatic cleanup of Oberon-A programs

A program that allocates and uses system resources must usually explicitly de-allocate and return those resources before it exits. Failure to do so is a common bug, especially in regard to memory. Oberon-A programs that allocate memory using the NEW standard procedure virtually eliminate the possibility of memory leaks. However, resources other than memory must still be explicitly de-allocated. To deal with this problem, module~Kernel provides a facility for installing cleanup procedures, which are automatically executed when a program exits.

A cleanup procedure must be assignment~compatible with a procedure variable with the type

```
    PROCEDURE (VAR rc : LONGINT)
```

It must be either exported (not recommended) or assignable (preferred).

When called, the procedure's parameter will be the return code that is to be passed back to AmigaDOS. The procedure will usually ignore the return code, but it may use it to determine how to behave, or even change it.

The body of the procedure should contain statements that return resources to the system. These resources must be accessed through global variables. It is a good idea to initialise all resource variables to NIL, and check if they contain non-NIL values before attempting to release them. The procedure should not call the standard procedures HALT or ASSERT, and should not allocate any resources. Great care should be taken to ensure that no run-time errors occur during the execution of a cleanup procedure.

A cleanup procedure is installed by calling the Kernel.SetCleanup() procedure with the name of the cleanup procedure as a parameter. See

Listing~3
. Any number of cleanup procedures may be installed at any given time.

The cleanup procedure will be called when the program exits, either normally, or as the result of a HALT or ASSERT statement. If the default trap handler is installed, it will also be called if the program halts as the result of a
run-time~error
. Cleanup procedures are executed in
the reverse of the order in which they were installed. The list of cleanup procedures will only be executed once, even if an error occurs while they are running.

1.29 Making Oberon-A programs resident

Programs compiled with the Large or Small data model are re-executable. This means that they can be made resident, but they may only be run by one process at a time. If such a program is run simultaneously by two or more processes, the second and subsequent instances will fail silently, hopefully without bringing the entire system down around your ears.

The PURE argument of the AmigaDOS Resident command must be used when making a re-executable program resident. For example:

```
Resident OBERON-A:OC PURE
```

It is not a good idea to set the pure bit for the program's executable file to get around this requirement. Re-executable programs are not pure, and should not claim to be pure.

It is possible to write an Oberon-A program that is not re-executable. This may happen if a module with global variables is compiled without the INITIALISE option set. To ensure that a module can be used in a re-executable program, do one of the following:

- Compile it with the INITIALISE option ON.
- Compile it with the ClearVars pragma ON for the module's main body.
- Include statements to make sure that all global variables are initialised to some safe value. In particular, all _traced_ pointer variables MUST be initialised to NIL.

Programs compiled with the Resident data model are re-entrant. They can be made resident and their code may be executed by any number of processes simultaneously. They are pure, and may have the pure bit set so that the PURE argument to the Resident command is not necessary.

1.30 Writing call-back hooks

A call-back is a procedure that is passed as a parameter to ↩ another procedure. This technique allows the procedure to be written in a generic fashion, leaving any data- or situation-specific processing up to the call-back procedure. It is used in a number of places in the Amiga operating system, notably in the animation system, the BOOPSI object system and in utility.library Hooks. Call-backs are also used in some third-party libraries such as MUI.

Call-back procedures need to be written with care, as they may be operating in an environment very different to the normal Oberon environment. Unfortunately there is no standardisation in the way call-backs are handled, although the Hook system is a step in that direction. This section will describe the different kinds of call-backs you may encounter and describe how to program each kind.

One characteristic shared by all call-back procedures is that they must be either exported, or marked as assignable. Another is that stack checking *must* be turned off, using the <*$StackChk-*> pragma.

There are four other main areas to consider when writing a call-back procedure:

* Execution~context
* Parameter-passing~convention
* Saving~registers
* Stack~checking

Call-backs are used in the following cases:

* Exec.RawDoFmt
* Graphics~library~collision~detection
* Utility~library~Hooks

1.31 Code Models

Oberon-A can generate code in Large and Small code models, and in Large, Small and Resident data models. This is controlled through preferences settings (see OCPrefs).

The Large code model is the same as that used by earlier versions of the compiler. The Small code model will produce executables that are much smaller on disk, due to a combination of more compact code and smaller relocation tables. The main benefit of the Large model is that the executable will be scatter-loaded. That is, the program will be loaded as many small hunks instead of one large one, and will make better use of fragmented memory. With the Small code model the program will load faster, because the operating system has less relocation to do. In general the program will also run faster, because procedure calls are more efficient. However, in large programs, *some* procedure calls will actually be less efficient, as they will be outside the 32K offset allowed by the BSR instruction, requiring the linker to insert a JMP instruction to compensate.

The Large data model is also the same as that used by earlier versions of the compiler. The permitted size for variables and constants is effectively unlimited, but there is a penalty in code size, in access times for constants and variables in other modules, and procedure calls are less efficient. The Small data model combines all the program's variables and constants into a single block, which may not exceed 32K bytes. Access to all constants and variables is equally efficient, and there is no procedure call penalty. The Resident data model dynamically allocates space for variables and so allows programs to be made fully resident. Otherwise it is the same as the Small data model.

The new code and data models restrict the use of linkers other than BLink. Commodore's ALink will only work with the Large code and Large data models. AmigaOberon's OLink and Dice's dlink cannot be used with either the Small or Resident data models. The LK linker can be used in BLink emulation mode, but it can take an excessive amount of time to link programs using the Small code model.

In most cases, the best option is to use the Small code and Small data models, with the BLink linker.

1.32 Debugger symbols produced by the compiler

When the compiler is run with the DEBUG argument set, it outputs symbol hunks in the object file. These symbol hunks are read by a debugger or profiler to identify particular objects in the program. Symbols are created for the following types of objects:

Exported procedures (including assignable procedures)

```
  Format: <module name>_<procedure name>
```

Local procedures

```
  Format: <module name>_<procedure #>P_<procedure name>
```

Type descriptors

```
  Format: <module name>_<type name>
```

```
Type-bound procedures

  Format: <module name>_<type name>_<procedure name>

Initialisation code

  Format: <module name>_INIT-CODE

Cleanup code

  Format: <module name>_END

Constants

  Format: <module name>_CONST

Global variables

  Format: <module name>_VAR

  Note: symbols are not produced for individual variables.

Table of root pointers for the garbage collector

  Format: <module name>_GC-OFFSETS
```

1.33 Example of using the default trap handler

```
<* STANDARD- *> (* This is necessary if the cleanup procedure is made
                ** assignable rather than exported.
                *)

MODULE Example1;

  IMPORT Kernel ...;

...

PROCEDURE* Cleanup (VAR rc : LONGINT);
BEGIN
  ...
  Kernel.RemoveTrapHandler
END Cleanup;

...

BEGIN
  Kernel.InstallTrapHandler; (* Put this *first* *)
  ...
  Kernel.SetCleanup (Cleanup);
  ...
END Example1.
```

1.34 Example of using module Errors

```
MODULE Example2;

  IMPORT Errors ...;

...

BEGIN
  Errors.Init; (* Put this *first* *)
  ...
END Example1.
```

1.35 Example of installing a cleanup procedure

```
<* STANDARD- *> (* This is necessary if the cleanup procedure is made
                ** assignable rather than exported.
                *)

MODULE Example3;

  IMPORT Kernel ...;

...

VAR
  (* Declare global variables to hold resources. *)

...

PROCEDURE* Cleanup (VAR rc : LONGINT);
BEGIN
  (* De-allocate resources here *)
END Cleanup;

...

BEGIN
  ...
  Kernel.SetCleanup (Cleanup);
  ...
  (* Allocate resources here *)
  ...
END Example3.
```

1.36 Example of the use of debugging print statements

```
<* NEW DEBUG *> (* This creates a selector which is used to control the
                ** compilation of debugging statements.
                *)
<* DEBUG+ *>    (* Turn debugging statements ON. *)
```

```
(* Another option would be to turn this selector on and off from the
** command line using the SET and CLEAR arguments. In that case, the
** above compiler commands are not needed.
*)

MODULE Example4;

IMPORT
  ...
  <* IF DEBUG THEN *> (* Assuming Out isn't normally imported *)
  Out,
  <* END *>
  ...;

...

PROCEDURE foo (x : INTEGER);
(* Pre-conditon : 0 <= x < 32 *)
BEGIN
  <* IF DEBUG THEN *>
  Out.String ("x = "); Out.Int (x, 0); Out.Ln;
  <* END *>
  ASSERT ((x >= 0) & (x < 32), 97); (* Assert pre-condition *)
  ...
END foo.

...

END Example4.
```

1.37 Execution Context

By execution context, I mean the Process or Task that will actually execute the code in the call-back procedure. In some cases the context will be that of the procedure's own program; for example, a call-back passed to Exec.RawDoFmt(). In other cases the call-back will execute as part of some other context; this is true of BOOPSI call-backs, which execute in the input device's context.

The main problem to be solved is allowing the call-back to access the global data segment containing string constants and global variables (note that calling an Amiga library function involves an *implied* access to a global variable, the library base variable). This is only a concern for modules compiled under the small or resident data models; the large data model automatically provides access to global data.

When the call-back is executing in the context of its own program, access to global data is provided by calling the GetDataSegment() procedure in module Kernel. It is safe to call this procedure (that is, it has no effect) when the large data model is being used, but if you are concerned about counting cycles, you can wrap it in a conditional compilation block:

```
    <* IF SMALLDATA OR RESIDENT THEN *>
    Kernel.GetDataSegment;
    <* END *>
```

When the call-back is executing in another context, access to global data must be provided in some other way. The preferred method is to store the address of the global data as part of a data block that is passed as a parameter to the call-back procedure. In the case of utility.library Hooks, this is the Hook data structure itself. The address of the global data segment is stored in the A4 register, so the contents of that register must first be stored in the call-back's data block, then recovered when the call-back is executed. This can be achieved using the SYSTEM.GETREG and SYSTEM.PUTREG procedures. For example, this will store the data segment's address:

```
    <* IF SMALLDATA OR RESIDENT THEN *>
    SYSTEM.GETREG (12, callBackData.dataSegment);
    <* END *>
```

Note that this call must be made by the call-back's own program. To restore the data segment address, place this code at the start of the call-back procedure, before any constants or global variables are accessed:

```
    <* IF SMALLDATA OR RESIDENT THEN *>
    SYSTEM.PUTREG (12, callBackData.dataSegment);
    <* END *>
```

Note that the conditional compilation commands are *required*; this code should not be executed under the large data model.

1.38 Parameter-passing conventions

In most cases, call-backs receive their parameters in CPU registers, in line with the normal Amiga calling conventions. At the present time Oberon-A does not allow register parameters, but the SYSTEM.GETREG procedure may be used to get around this limitation. Instead of declaring the parameters in the procedure heading, declare them as local variables. Then use SYSTEM.GETREG copy each parameter from a register into a local variable. For example:

```
    PROCEDURE* CallBack;
    (* parameter foo is passed in register A0.
    ** parameter bar is passed in register D0.
    *)

      VAR foo: FooPtr; bar : LONGINT;

    BEGIN
      SYSTEM.GETREG (8, foo);
      SYSTEM.GETREG (0, bar);
      ...
    END CallBack;
```

The copying of parameters must be the very first code executed in the procedure.

Some call-backs, notably those used by the graphics.library animation collision detection system, receive their parameters on the stack, using the C language’s calling conventions. This is of course extremely rude, but the code involved is so ancient and well-entrenched that nothing can be done about it now. As far as Oberon is concerned, such call-back’s will receive their parameters *backwards*. That is, if the C prototype is

```
    void CallBack (long a, long b);
```

then the Oberon equivalent is:

```
    PROCEDURE* CallBack (b, a : LONGINT);
```

The C calling convention requires the calling code to remove any parameters from the stack after the called procedure returns. The Oberon calling convention requires the called procedure to clean up after itself. In order to avoid a visit from the Guru, the call-back procedure must use the C convention. This is achieved by placing the <*$DeallocPars-*> pragma immediately before the procedure’s body. For example:

```
    PROCEDURE* CallBack (b, a : LONGINT);

    <*$DeallocPars-*>
    BEGIN
      ...
    END CallBack;
```

1.39 Saving Registers

The Amiga programming guidelines specify that a procedure must preserve the contents of all registers except those designated as ’scratch’ registers. For efficiency reasons Oberon-A does not normally implement this convention, and this is acceptable as long as Oberon code is called only by other Oberon code. When Oberon code is used as a call-back, however, the convention *must* be followed. This is achieved by placing a <*$SaveRegs+*> pragma before the main body of the call-back procedure. For example:

```
    PROCEDURE* CallBack;

    <*$SaveRegs+*>
    BEGIN
      ...
    END CallBack;
```

1.40 Exec.RawDoFmt

The RawDoFmt() function requires a call-back procedure (PutChProc) that is called to process each character in the formatted string. The character to be output is passed to the call-back in the lower 8 bits of the D0 register, while the value passed in RawDoFmt’s PutChData parameter is passed in the A3 register. The normal behaviour of this call-back is to copy the character into a buffer passed to RawDoFmt() as the PutChData parameter. This can be achieved by passing the following procedure to PutChProc:

```
    PROCEDURE* PutCh ();

    <*$EntryExitCode-*>
    BEGIN (* PutCh *)
      SYS.INLINE (16C0H,   (* MOVE.B D0,(A3)+ *)
                  4E75H)   (* RTS             *)
    END PutCh;
```

This procedure represents a special case in which the convention requiring non-scratch registers to be saved is not followed.

A more complex example might involve an implementation of a PrintF-like procedure that outputs the character direct to a file. Assuming an open AmigaDOS filehandle is passed in PutChData, the call-back might look something like this:

```
    PROCEDURE* PutCh ();

      VAR ch : CHAR; fh : Dos.FileHandlePtr; result : LONGINT;

    <*$SaveRegs+*> (* Save all non-scratch registers *)
    BEGIN (* PutCh *)
      (* Copy the parameters from registers to local variables *)
      SYSTEM.GETREG (0, ch); SYSTEM.GETREG (11, fh);

      (* Get the address of the global data segment. *)
      Kernel.GetDataSegment;

      (* Process the character *)
      result := Dos.FPutC (fh, ch)
    END PutCh;
```

1.41 Graphics library collison detection

While the Graphics library GELs routines actually do the work to detect collisions with other GELs and the display border, the action to be taken in the event of a collision is determined by a call-back procedure. Call-backs are installed by the Graphics.SetCollision() function.

A collision call-back receives its parameters on the stack, using C calling conventions. The calling code will remove the parameters from the stack, so the <*$DeallocPars-*> pragma must be used. Finally, the call-back will be executing in its own program’s context, so Kernel.GetDataSegment must be called to set up the global data segment.

Two parameters are passed, but their types depend on what has collided

```
with what. For a collision with the display border, the parameters are:

    ( borderflags : s.SET32; hitVSprite : Graphics.VSpritePtr )

For a collision between two GELs, the parameters are:

    ( vSprite2, vSprite1 : Graphics.VSpritePtr )

Note that the order of parameters is the reverse of that described in
the ROM Kernel Manuals.

Here are the general outlines of the two kinds of collision procedures:

  PROCEDURE* BorderCollision
    ( borderflags : s.SET32; hitVSprite : Graphics.VSpritePtr );

  <*$ < StackChk- DeallocPars- SaveRegs+ *>
  BEGIN (* BorderCollision *)
    Kernel.GetDataSegment;
    ...
    (* Process the collision *)
    ...
  END BorderCollision;
  <*$ > *>

  PROCEDURE* GELCollision
    ( vSprite2, vSprite1 : Graphics.VSpritePtr );

  <*$ < StackChk- DeallocPars- SaveRegs+ *>
  BEGIN (* GELCollision *)
    Kernel.GetDataSegment;
    ...
    (* Process the collision *)
    ...
  END GELCollision;
  <*$ > *>
```

1.42 Utility library Hooks

```
Beginning with AmigaOS 2.0, the preferred mechanism for handling
call-backs is the Utility library Hook data structure and its related
functions. The Hook mechanism greatly simplifies the task of writing
call-backs by allowing them to be written in close to the normal style
of the language in use. All the messy details of parameter conventions,
saving registers, etc. is handled transparently and the programmer need
not be bothered with them.

A Hook call-back procedure is written as a normal Oberon procedure.
There are only two special requirements. The first is that the
parameter list must match the following declaration:

    PROCEDURE* HookFunc
      ( hook    : Utility.HookPtr;
        object  : Exec.APTR;
        message : Exec.APTR )
      : Exec.APTR;
```

The second requirement is that stack checking should be disabled using the <*$StackChk-*> pragma.

The call-back must then be installed in a Hook data structure by passing it as a parameter to the InitHook procedure in module Utility. InitHook ensures that the call-back will be called with the correct parameters and a valid global data segment. It will also ensure that non-scratch registers are properly saved and restored.

That’s all there is to it.

1.43 Stack checking

The stack checking code generated by the compiler can cause serious problems for call-back procedures. The simplest (and best) solution is to simply disable it, using the <*$StackChk-*> pragma. This can be limited to the call-back procedure by saving and restoring the pragma state before and after the procedure. For example:

```
    <*$ < *> (* Save pragma state *)
    PROCEDURE* CallBack;

    <*$ StackChk- *> (* Turn off stack checking *)
    BEGIN
    ...
    END CallBack;
    <*$ > *> (* Restore pragma state *)
```

If you still want to use stack checking, there are a couple of things you need to know. Firstly, stack checking *must* not be enabled if the call-back will be executing in any context other than the call-back’s own program. Secondly, if parameters are being passed in registers you must use the <*$SaveAllRegs+*> pragma instead of <*$SaveRegs+*>.

1.44 Other documents you should read

The following documents will be the most immediately useful to you:

~Oberon-2~Report~	The Oberon-2 language report
~FPE~~~~~~~~~~~~~	Using the programmer’s environment
~OC~~~~~~~~~~~~~~	Using the compiler
~OCPrefs~~~~~~~~~	Using the preferences tool for OC
~OEL~~~~~~~~~~~~~	Using the error lister
~Error~codes~~~~~	Error codes output by the compiler
~OL~~~~~~~~~~~~~~	Using the pre-link utility
~OLPrefs~~~~~~~~~	Using the preferences tool for OL
~BLink~~~~~~~~~~~	Using the linker

The AmigaGuide file Index.doc in the main Oberon-A directory contains an index of all the documentation and source code files distributed with Oberon-A.

1.45 Useful resources for Oberon-A Programmers

```
                  Apart from the documentation provided with Oberon-A, there are a ↩
                      number
of other resources that you may find useful.

Books

  See the
              bibliography
               in the comp.lang.oberon FAQ for a list of books
  on the programming language and the Oberon System. Reiser and Wirth's
  "Programming in Oberon" is a good introduction to the language,
  pitched at a level suitable for students as well as more advanced
  programmers. Mössenböck's "Object-Oriented Programming in Oberon-2"
  is exactly what the title says :-).

UseNet newsgroups

  comp.lang.oberon

    From the comp.lang.oberon FAQ:

    "The Comp.lang.oberon newsgroup is a forum for discussing
    Oberon, both the programming language and the operating system,
    and any related issues.  Although not strictly accurate, this
    newsgroup is part of the Comp.lang.* hierarchy because it began as
    a spin-off of Comp.lang.modula2."

    The FAQ is posted in the newsgroup periodically, and can be
    obtained by anonymous ftp from rtfm.mit.edu in the /pub/usenet
    directory.

  comp.sys.amiga.programmer

    This group is for discussing Amiga programming in general, and
    there is occasional mention of Oberon and Oberon-A.

Internet ftp sites

  ETH, where Oberon was developed, maintains an ftp site for
  distributing Oberon-related material, including the various versions
  of the Oberon System. Connect by anonymous ftp to
  neptune.inf.ethz.ch, and look in the /pub/Oberon directory.

  The Swiss Oberon Users Group also maintains an archive of Oberon
  software. Connect by anonymous ftp to hades.ethz.ch, and look in the
  /pub/Oberon directory.

  Oberon-A is primarily distributed through AmiNet, a network of ftp
  sites covering most of the Western world. Almost anything you could
  wish for as an Amiga programmer is available somewhere in the
  archive. The following sites are currently part of AmiNet:
```

USA (MO)	ftp.wustl.edu	pub/aminet/
USA (TX)	ftp.etsu.edu	pub/aminet/
USA	ftp.netnet.net	pub/aminet/
Scandinavia	ftp.luth.se	pub/aminet/
Switzerland	ftp.eunet.ch	pub/aminet/
Switzerland	litamiga.epfl.ch	pub/aminet/
Germany	ftp.uni-erlangen.de	pub/aminet/
Germany	ftp.uni-paderborn.de	pub/aminet/
Germany	ftp.uni-kl.de	pub/aminet/
Germany	ftp.cs.tu-berlin.de	pub/aminet/
Germany	ftp.uni-oldenburg.de	pub/aminet/
Germany	ftp.coli.uni-sb.de	pub/aminet/
Germany	ftp.uni-stuttgart.de	cd aminet
UK	ftp.doc.ic.ac.uk	pub/aminet/
Australia	archie.au	pub/aminet/

The Amiga Modula-2 and Oberon Club Stuttgart produces a collection of over 100 disks containing freely-distributable code written in, oddly enough, Modula-2 and Oberon. The Oberon code is naturally for the AmigaOberon compiler, but there may be stuff that can be adapted for Oberon-A. The collection is currently available by anonymous ftp from ftp.rz.uni-wuerzburg.de. This site also carries the Fred Fish and Meeting Pearls CDs at times.

World Wide Web

Oberon-related pages can be found at these URLs:

http://www.inf.ethz.ch/department/CS/Oberon.html
-- ETH Zürich Oberon Home Page

http://oberon.ssw.uni-linz.ac.at/home.html
-- University of Linz Oberon Home Page

http://hades.ethz.ch/
-- Swiss Oberon User Group Home Page

http://zorro.ruca.ua.ac.be/Memex/Oberon/
-- The PC-Oberon Project Home Page

Amiga-related pages can be found at these URLs:

http://www.omnipresence.com/Amiga/WWWResources.html
-- Links to Commodore Amiga Information Resources

http://www.omnipresence.com/Amiga/News/index.html
-- Commodore Amiga Information Resources -- News

1.46 The Author

Oberon-A was mostly written by Frank Copeland.

All bug reports, suggestions and comments can be directed to:

```
  Email : fjc@wossname.apana.org.au

  Snail Mail :

    Frank J Copeland
    PO BOX 236
    RESERVOIR  VIC  3073
    AUSTRALIA

    Remember the J.  It saves a lot of confusion at my end :-).
```

Note that the e-mail address has been changed to my private mailbox, to avoid confusion with the mailing list server.

I also regularly read the comp.sys.amiga.programmer newsgroup, and will respond to any Oberon-A related posts I see there.

1.47 The Oberon-A mailing list

A mailing list has been set up to provide support for users of Oberon-A and allow discussion of the compiler and the Oberon language in general. To find out more, send e-mail to:

```
  oberon-a-request@wossname.apana.org.au
```

The Subject: line may be empty, and the body of the message should contain only the following lines, starting in the left-hand column:

```
  HELP
  HELP oberon-a
```

The listserver software will reply with information about its commands, how to subscribe, and a description of the list.

1.48 Reporting bugs and suggestions

You are encouraged to report any bugs you find, as well as any comments or suggestions for improvements you may have. I am also happy to answer any questions about the language itself. For information about Amiga programming in general you should consult the relevant Commodore and third-party documentation first. I can help if you have trouble translating examples written in C into Oberon.

Before reporting a suspected bug, check the file ToDo.doc to see if it has already been noted. If it is a new insect, clearly describe its behaviour including the actions necessary to make it repeatable. Indicate in your report which release of Oberon-A you are using. Include an example of a program or short fragment of code that demonstrates the bug. If you discover a bug in BLink, please report it but there is nothing that can be done except find a workaround. The original authors no longer support BLink.

I would like to hear your opinion of Oberon-A, even if you decide not to use it. Suggestions for improvements and additions are also most welcome. I am especially interested in the following areas:

* Compatibility with different versions of the Amiga hardware and operating system.

* How good/useful/helpful/complete the documentation is.

* How suitable it is for use by programmers with varying levels of experience, from beginners to hackers.

* How correct and useable the operating system interface modules are. These modules were translated from the C header files provided by Commodore. The translation was done quickly and only a fraction of the modules have been tested in any way.

* How useful the library modules provided are, and suggestions for additional modules.

* Departures from the language specification.

* Extensions to the language supported by the compiler.

* Memory management.

1.49 Acknowledgements

The Oberon-A compiler is a port of a compiler written for the Ceres workstation by Niklaus Wirth. The book "Project Oberon" written by Wirth and Jürg Gutknecht contains a description of this compiler and the full source code for it. The original source can also be obtained by anonymous ftp from neptune.inf.ethz.ch. Many thanks to Professor Wirth for making this source code available.

The machine code generator for early versions of the compiler was a port of the corresponding parts of Charlie Gibb's A68K assembler. This code is no longer part of the compiler, but it was extremely useful in the early stages of development and debugging.

Torsten Jürgeleit's intuisup.library is used to create and manage the user interface for FPE. EAGUI.library by Marcel Offermans and Frank Groen is used by OCPrefs and OLPrefs.

The following people have contributed modules and programs to Oberon-A:

Terje Bergstr\vm has contributed the interface module for the Universal Message System.

Johan Ferreira has contributed the Oberon-A error lister, OEL.

Helmuth Ritzer has contributed interface modules for Nico Francois' ReqTools library and the TextField Boopsi gadget, and the German catalogs.

Edmondo Tommasina provided the Italian catalogs.

Albert Weinert has contributed Classface.asm to allow access to Intuition's Boopsi functions.

Carsten Ziegeler has contributed the interface module for his GuiEnv library.

Apologies to anyone I have inadvertantly overlooked.

Thanks to those who have reported bugs and made suggestions for improving Oberon-A.

1.50 Bibliography

The following works have proved useful in developing Oberon-A:

* N. Wirth. The programming language Oberon (Revised Report). Institut für Computersysteme, ETH Zürich. (See Oberon-Report.doc).

* N. Wirth. From Modula to Oberon. Institut für Computersysteme, ETH Zürich. (See ModToOberon.doc).

* N. Wirth and J. Gutnecht. Project Oberon. Addison-Wesley, 1992.

* H. Mössenböck and N. Wirth. Differences between Oberon and Oberon-2. Institut für Computersysteme, ETH Zürich.

* H. Mössenböck and N. Wirth. The Programming Language Oberon-2. Institut für Computersysteme, ETH Zürich.

* Commodore-Amiga, Inc. Amiga ROM Kernel Reference Manual: Libraries. Third Edition. Addison-Wesley, 1992.

* Commodore-Amiga, Inc. Amiga ROM Kernel Reference Manual: Libraries & Devices. Second Edition. Addison-Wesley, 1990.

* S. Kelly-Bootle. 680x0 Programming by Example. Howard W. Sams, 1988.

* Commodore-Amiga, Inc. The AmigaDOS Manual, 3rd Edition. Bantam, 1991.

* Commodore-Amiga, Inc. The AmigaDOS Manual, 2nd Edition. Bantam, 1987.

* R. Bornat. Understanding and Writing Compilers. MacMillan, 1979.

* A. V. Aho and J. D. Ullman. Principles of Compiler Design. Addison-Wesley, 1977.

* B. S. Gottfried. Programming with C. McGraw-Hill, 1990.

* S. Ballantyne and C. Heath. The Final ARP.Library Tour. Amiga Transactor, 1, 4, 44-57.

* J. Toebes. The Art of Assembly Language. Amiga Transactor, 2, 1, 38-43.

1.51 Revision Control

All the source code and document files in Oberon-A are managed using the freely-distributable HWGRCS package. This is a port of the Un*x RCS system.

Each new release of the entire package will be identified by a two-part release number. Partial releases containing bug fixes will have an additional update number. For example, the second major release of Oberon-A will be known as "Oberon-A release 2.0". The first minor release of release 2 will be known as "Oberon-A release 2.1". The second update of release 2.1 will be called "Oberon-A release 2.1 update 2".

Individual programs are identified with a two part number of the form:

```
  <version>.<revision>
```

The version number will change whenever substantial changes are made to the program. The revision number will initially start at 0 and will change whenever a bug-fix patch of the program is released.

Each module and documention file has a two-part revision number, of the form:

```
  <version>.<revision>
```

All the component modules and documentation files of a program will have the same version number as the program. The revision number will change whenever the file's text is modified. The version number of a documentation file not associated with any one program will be the same as the current overall release number.

1.52 Oberon-A Release History

```
0.0       Initial version, written in Modula-2. Never released.
0.1 - 0.3 Intermediate versions, written in Oberon. Never released.

1.0       Initial beta-test release.  Compiler upgraded to Oberon-2.

1.1       Bug fixes and some improvements to the compiler and utilities.

1.2       More bug fixes.

1.3       - New compiler with varargs and improved symbol table
            handling.
          - Heavily revised Amiga interface modules.
```

```
 1.4       - Steady improvements to the compiler and utilities.
           - Upgraded the Amiga interface modules to Release 3.1.
           - Added more third-party interface modules.

 1.5       - Complete overhaul of the external code interface.
           - New system of compiler control and pragmas.
           - New Shell and Workbench interface for all programs.
           - Amiga Interfaces brought into line with AmigaOberon's.
           - New library modules.
           - Much improved run-time error handling.

 1.6       A progressive development, including:
           - Multiple code and data models output by the compiler.
           - New GUI versions of OCPrefs and OLPrefs.
           - Many bug fixes and minor improvements.
```