

# Armaide Oberon for LPC2000 Microcontrollers

## Table of Contents

1 Introduction.....	3
2 Oberon-07 Language Extensions.....	3
2.1 Leaf Procedures.....	3
2.2 Interrupt Handlers.....	3
2.3 Local Dynamic Arrays.....	5
3 Oberon-07 Language Differences and Restrictions.....	6
4 Implementation Limits.....	7
5 File Descriptions.....	8
6 Linking and Loading.....	9
6.1 Startup Code.....	10
6.2 Library Organisation.....	11
6.3 Library Module Definitions.....	13
6.3.1 Convert.....	15
6.3.2 * IAP.....	16
6.3.3 LPC.....	17
6.3.4 * LPC2103.....	18
6.3.5 * LPC2106.....	19
6.3.6 * LPC2148.....	20
6.3.7 * LPC2214.....	21
6.3.8 * LPC2378.....	22
6.3.9 Main.....	23
6.3.10 * Math - Mathematical Functions.....	24
6.3.11 MAU - Memory Allocation Unit.....	25
6.3.12 Out.....	26
6.3.13 Put.....	27
6.3.14 * Random.....	28
6.3.15 Reals.....	29
6.3.16 * ResData.....	30
6.3.17 Serial.....	32
6.3.18 * Strings.....	33
6.3.19 SYSTEM.....	35
6.3.20 Timer.....	36
6.3.21 Traps.....	37
6.4 * Resource Data.....	38
6.5 Link Options.....	39
6.6 Memory Map.....	40
7 Runtime Errors.....	42
7.1 Runtime Error Codes.....	42
7.2 Library Assertion Error Codes.....	43
7.3 Programmer-defined Assertions.....	43
7.4 Reporting Runtime Errors.....	43

7.5 Diagnosing Runtime Errors.....	44
8 * Command-line Tools.....	45
8.1 Compiler and Linker.....	45
9 Programming Conventions and Guidelines.....	47
9.1 Essentials.....	47
9.2 Indentation.....	48
9.3 Semicolons.....	48
9.4 Dereferencing.....	48
9.5 Letter case.....	49
9.6 Names.....	49
9.7 White space.....	49
9.8 Alignment.....	50
9.9 Boolean Expressions.....	51
9.10 Acknowledgements.....	51

# 1 Introduction

Armaide is a fast and responsive integrated development environment for Windows. It is used to write software to run on the powerful NXP / Philips LPC2000 family of ARM-based microcontrollers.

**NOTE:** Sections marked \* are only applicable to the Standard and / or Professional Editions of Armaide.

## 2 Oberon-07 Language Extensions

The ARM compiler included in Armaide implements the Oberon-07 language as defined in the included report titled *The Programming Language Oberon (Revision 1.11.2008)* by Niklaus Wirth. This implementation of the language has the following extensions.

### 2.1 Leaf Procedures

The code that is generated by the Oberon compiler for procedure calls is efficient for most normal purposes. On occasions where faster execution speed is required (e.g. for fast interrupts) *Leaf* procedures can be used. These are identified by an asterisk in the procedure header.

```
PROCEDURE* Speedy(n: INTEGER);
```

Features of leaf procedures that result in faster execution speed are:

- Parameters are stored in registers
- Registers do not need to be saved or restored
- INTEGER and SET local variables are stored in registers
- Array index checks are suppressed

Limitations of leaf procedures are:

- Procedures (other than standard procedures) cannot be called from a leaf procedure
- REAL operations are restricted
- Array index out of range errors are not detected

The following examples illustrate the difference between an asterisk used to indicate that a procedure is a leaf procedure and an asterisk used to indicate that the procedure is exported:

```
PROCEDURE GetValue(VAR n: INTEGER);    (* Private non-leaf procedure *)
PROCEDURE GetValue*(VAR n: INTEGER);  (* Exported non-leaf procedure *)
PROCEDURE* GetValue(VAR n: INTEGER);  (* Private leaf procedure *)
PROCEDURE* GetValue*(VAR n: INTEGER); (* Exported leaf procedure *)
```

### 2.2 Interrupt Handlers

An Oberon-07 interrupt handler is a normal procedure which has an *offset specification* (a numeric constant) in square brackets instead of a list of parameters. e.g.

```
PROCEDURE TimerHandler[4];
```

The value of offset is either 0, 4 or 8 depending on the type of interrupt being handled. The value is used in the actual ARM instruction needed to effect the return from the handler and resume execution at the location where the interrupt occurred. For example, for an IRQ handler the value is 4. Thus, when the IRQ handler is compiled the last instruction generated is:

```
SUB PC, R14, 4
```

i.e. the offset, 4, is subtracted from the value of the link register (R14) and stored into the program counter. Execution then continues from that address.

The value of the offset for each type of interrupt is:

Interrupt Type	Offset
FIQ	4
IRQ	4
Abort Data	8
Abort Instruction	4
SWI	0
Undefined Instruction	0

An example of an IRQ procedure used to handle interrupts from the LPC2xxx timer *Timer1* is:

```
PROCEDURE TimerHandler[4]; (* IRQ *)
BEGIN
  INC(timeVal);
  (* Clear the MR0 interrupt *)
  SYSTEM.PUT(LPC.T1IR, {0});
  (* Update the VIC priority hardware *)
  SYSTEM.PUT(LPC.VICVectAddr, 0)
END TimerHandler;
```

The handler increments the global variable *timeVal*. It then resets the timer and interrupt priority hardware ready to handle the next interrupt.

An example of the code required to install this handler is:

```
SYSTEM.PUT(LPC.VICVectAddr0, SYSTEM.ADR(TimerHandler));
```

Two source code files *IRQTimer.mod* and *IRQBlinker.mod* are included in the Armaide *Examples* folder. These form a complete working example of an interrupt-driven blinking LED. This example was developed using the information in the NXP Application Note AN10254: "*Handling Interrupts Using FIQ and IRQ for LPC2000 Microcontrollers*".

## 2.3 Local Dynamic Arrays

A local dynamic array is an array of any type declared within a procedure. The number of elements (i.e. length) of the array is determined at runtime. The syntax is:

```
ArrayType = "ARRAY OF" type.
```

For example:

```
TYPE
  Event = RECORD
    date: Date;
    time: Time;
    location: Location
  END;

PROCEDURE P();
VAR
  counts: ARRAY OF INTEGER;
  weights: ARRAY OF REAL;
  alarms: ARRAY OF Event;
  ...
  ...
```

The length of a local dynamic array is established by a call to `NEW` with a second parameter indicating the desired number of array elements. For example:

```
NEW(data, len)
```

where *len* is a non-negative INTEGER expression.

The great advantage of local dynamic arrays is that they can be safely allocated on the stack. Thus no garbage collector or special treatment is required to free their memory when the procedure terminates. This allows the most efficient use of memory without the associated problems of memory leaks and fragmentation.

For example, given the following procedure, the call `DisplayHighest(20, 200)` would read the next 200 integer data items and display the top 20 values read. `DisplayHighest(20, 20)` would just display the next 20 values in ascending order.

```
PROCEDURE DisplayHighest(CONST selection, total: INTEGER);
VAR
  data: ARRAY OF INTEGER;
BEGIN
  ASSERT(selection <= total, 100);
  NEW(data, total);
  FOR i := 0 TO total - 1 DO ReadInteger(data[i]) END;
  Sort.Integers(data);
  FOR i := 0 TO selection - 1 DO Out.Int(data[i], 6); Out.Ln END;
END DisplayHighest;
```

When the procedure terminates, the 800 bytes (or 80 bytes in the second case) of RAM allocated to the data array is automatically returned to the system to be reused.

**NOTE:** This implementation of dynamically allocated arrays is subject to the restrictions that they are one-dimensional and cannot be elements of other data structures.

### 3 Oberon-07 Language Differences and Restrictions

The ARM compiler included in Armaide implements the Oberon-07 language as defined in the included report titled *The Programming Language Oberon (Revision 1.11.2008)* by Niklaus Wirth. This implementation of the language is subject to the following differences and restrictions.

#### LONGREAL

The 64-bit data type LONGREAL is not supported.

#### UNPK(x, e)

Prerequisite:  $x \geq 0.0$

#### PACK(x, e)

Prerequisite:  $1.0 \leq x < 2.0$

#### Negative divisors

The DIV and MOD operators do not support negative divisors. An attempt to use a negative divisor will result in a compilation error or runtime exception.

If a divisor  $j$  can potentially be negative, then:

```
i := i DIV j;
```

can be implemented as

```
IF j < 0 THEN i := -i DIV ABS(j) ELSE i := i DIV j END;
```

#### Smallest negative INTEGER

The smallest negative INTEGER allowed is  $-2^{31} + 1$  instead of  $-2^{31}$

#### Procedure Variables

It is an error if a procedure variable is assigned a procedure which is declared in a different scope. However, the compiler does not report the error.

#### Extending imported types

Anonymous pointer types declared in an external module cannot be extended in a client module. Pointer types in an external module can be modified if they point to a named record.

i.e. the type *Item* declared as:

```
Item* = POINTER TO RECORD value: INTEGER END;
```

can be extended in a client module if the declaration is changed to the equivalent form:

```
ItemDesc* = RECORD value: INTEGER END;  
Item* = POINTER TO ItemDesc;
```

## 4 Implementation Limits

Items	Maximum
Modules in a single application	128
Type extension levels	4
Parameters to a procedure	11
Value of a CASE label	255
Labels in a CASE statement	256
Maximum step size in a FOR statement	255

### CASE Statements

In the ARM Oberon-07 compiler, the CASE statement has been implemented in a way that provides maximum speed and predictability of code-generation at the expense of memory consumption.

Oberon-07 CASE statements are less versatile than they were in Pascal, Modula-2 or earlier versions of Oberon. Consequently, for best results, restrict the use of CASE statements to situations where:

- The case labels are naturally integers
- The smallest case label is 0 or close to zero.
- The case labels are relatively contiguous
- There are a large (e.g. more than half-a-dozen) number of cases
- All cases have similar probabilities of occurrence

Otherwise consider using an IF-ELSIF...ELSIF-ELSE series of statements instead.

In some cases a hybrid combination of CASE and IF statements can result in a good compromise between readability, efficiency and memory usage.

## 5 File Descriptions

The Armaide compiler and linker expect there to be a correspondence between module names and the associated filenames.

Unless you have a good reason to do otherwise you should give the file the same name as its module name with a *.mod* extension.

The filenames of module-related files created by Armaide are made from the name of the module and one of the following file extensions:

Extension	Type	Created by	Used by	Scope	Description
.mod	Text	Edit	Compile	Module	Source code
.smb	Binary	Compile	Compile	Module	Symbol file of exported items
.def	Text		Edit	Module	Readable file of exported items
.arm	Binary	Compile	Link	Module	Linkable object file
.res	Any		Link	Module	Resource data
.opt	Text	Link	Link	Application	Linker options
.ref	Binary	Link		Application	Trap reference resource data
.bin	Binary	Link		Application	* Linked binary executable file
.hex	Text	Link		Application	Uploadable hex executable code
.map	Text	Link		Application	Code and data memory usage

### \* Professional Edition only

#### Example

A module named *LcdDisplay* is saved as the file *LcdDisplay.mod*. When it is compiled the compiler generates a symbol file *LcdDisplay.smb* and an object file *LcdDisplay.arm*.

The main module of the application called *DigiClock* is saved as *DigiClock.mod*. *DigiClock* imports *LcdDisplay*.

When *DigiClock.mod* is being edited double-clicking on the name *LcdDisplay* in the list of imports opens the file *LcdDisplay.mod* (or *LcdDisplay.def* if the source code is not available).

When *DigiClock* is compiled the compiler uses the information in the symbol file *LcdDisplay.smb* to ensure that the use of all of the variables, procedures etc. from *LcdDisplay* conform with the declarations of those items in *LcdDisplay*. It is not necessary to have the source code of *LcdDisplay* available to validate the use of its exported items.

When *DigiClock* is linked the linker uses the Link Options data in *DigiClock.opt* and combines the object files *Main.arm*, *DigiClock.arm*, *LcdDisplay.arm* and all other imported modules. The linker creates the memory usage map file *DigiClock.map*, the trap reference resource file *DigiClock.ref* and executable hex file *DigiClock.hex*.

## 6 Linking and Loading

An application created with Armaide is made up from the following modules:

*System Modules*

Startup code module  
Armaide library modules

*User-developed Modules*

Common library modules  
Application-specific modules  
Main module

The simplest application consists of a single Main module accessing the System Modules.

The Linker / Loader combines all of the components needed by an application into a single file in HEX format suitable to be uploaded and executed on the target processor.

A feature of the Oberon language is that all of the information regarding dependencies between the various modules are defined in the source code. There is no need to create and maintain separate 'make files' as commonly used in other systems.

The only details the Armaide Linker / Loader needs to know to be able to build an application are:

- The name of the main module
- The physical locations of the folders containing the library files
- The crystal frequency and type of the target LPC2000 microcontroller

When the Armaide *Project > Link* command is selected the current module whose source code is in view is taken to be the main module.

The physical locations of the library files are relative to the folder that contains the main module source code. See *Library Organisation* below for details.

The details of microcontroller type and crystal frequency are entered in a dialog box in response to the *Project > Link Options* command.

## 6.1 Startup Code

Stack pointers, interrupt vectors etc. are initialised by startup code generated by the linker. The startup code is the first part of the application to execute when the microcontroller is reset.

The initialisation code of each module of the application is then executed in turn starting with the lowest module in the dependency chain. Execution continues all the way up until the initialisation code of the main module is started and the application proceeds.

The memory mapping (MAM) control and phase-locked loop (PLL) options of the microcontroller are configured in the process of initialising the Armaide library module *Main*. The module *Main* should be included in the IMPORT list of the main module of every Armaide application to ensure that the application is correctly initialised.

### \* Professional Edition:

A user-supplied Startup binary file can be substituted for the startup code generated by the linker. A sample assembler source file is included (an assembler is not provided). The source code of the *Main*, *LPC* and *Traps* modules is also provided to allow different configurations of the MAM and PLL features and to customise the output of runtime error messages.

## 6.2 Library Organisation

Libraries are groups of common files that are shared between several applications developed using Armaide. They allow applications to be conveniently organised without having to duplicate copies of common / shared files.

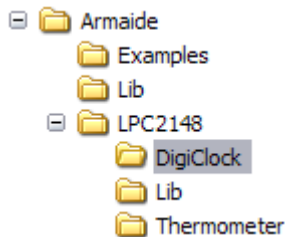
Armaide libraries are standard Windows folders containing collections of definition (\*.def), symbol (\*.smb) and object files (\*.arm).

The editor, compiler and linker search folders in the following sequence when looking for symbol and object files:

```
<current folder>
<current folder>\Lib
<parent folder>
<parent folder>\Lib
<grandparent folder>
<grandparent folder>\Lib
...
etc. (to the search depth limit).
```

<current folder> is the folder which contains the source file (\*.mod) currently being compiled or the main object file (\*.arm) currently being linked.

An example folder structure is:



Folder *Armaide\Lib* contains the system library files (i.e. *LPC.\**, *Convert.\** etc.) that are common to most applications developed using Armaide.

Folder *LPC2148\Lib* contains files which are common to more than one application targeting the LPC2148 microcontroller and its peripherals.

Folder *DigiClock* contains files only required for that application.

Hence when the compiler needs to find the symbol file for the imported module *Timer.smb*, while compiling *LPC2148\DigiClock\DigiClock.Mod*, it will search the folders in the following order:

1. *Armaide\LPC2148\DigiClock*
2. *Armaide\LPC2148\DigiClock\Lib*
3. *Armaide\LPC2148*
4. *Armaide\LPC2148\Lib*
5. *Armaide\*
6. *Armaide\Lib*
7. *etc. etc.*

and will stop searching as soon as the file *Timer.smb* has been found or the *search depth* limit has been reached. The default limit is 2 and can be increased to a maximum of 9 folder levels by selecting the *Compiler / Linker options* from the Armaide *Tools* menu.

The linker will search the folders in a similar way when looking for the object file *Timer.arm*

The folder *Documents\Armaide\Lib* contains the system library files (i.e. *LPC.\**, *Convert.\** etc.) that can be used by applications developed using Armaide.

## 6.3 Library Module Definitions

The following library modules are included with Armaide:

Module name	All Armaide Editions
<i>Convert</i>	Conversion of integers to / from strings
<i>FPU</i>	Support of mathematical operations on floating point numbers
<i>LPC</i>	Common definitions for all LPC2100 / LPC2200-family processors
<i>Main</i>	Initialisation code required by an LPC2000-family application
<i>MAU</i>	Memory Allocation Unit
<i>Out</i>	Formatted ASCII text output
<i>Put</i>	String-handling helper functions used by <i>Convert</i> and <i>Reals</i>
<i>Reals</i>	Real number support and conversion to / from strings
<i>Serial</i>	Basic polled UART serial IO
<i>SYSTEM</i>	Implementation-specific low level functions
<i>Timer</i>	Microsecond and millisecond time measurement and delays
<i>Traps</i>	Runtime error trapping

Module name	* Standard and Professional and Editions only
* <i>IAP</i>	In-Application (IAP) Programming for flash ROM reading and writing
* <i>LPC2103</i>	LPC2101 / 2102 / 2103 processor definitions
* <i>LPC2106</i>	LPC2104 / 2105 / 2106 processor definitions
* <i>LPC2148</i>	LPC2142 / 2144 / 2146 / 2148 processor definitions
* <i>LPC2214</i>	LPC2212 / 2214 processor definitions
* <i>LPC2378</i>	LPC2364 / 2366 / 2368 / 2378 processor definitions
* <i>Math</i>	Basic mathematical and trigonometrical functions
* <i>Random</i>	Pseudo-random number generator
* <i>ResData</i>	Access constant user data attached to the program by the linker
* <i>Strings</i>	General string-handling functions

*FPU*, *MAU* and *SYSTEM* are special i.e. they are dependent on the version of the compiler and must follow some specific conventions. See the library module descriptions below for further details.

If a user module calls the Oberon NEW function then module *MAU* is called and the *MAU* module automatically imported. *MAU* also contains some functions that can be called explicitly by a user module. If so, the programmer must then include *MAU* in the module's IMPORT list.

If a user module uses mathematical operations (e.g. divide, multiply etc.) on floating point numbers then an *FPU* function is called and the *FPU* module automatically imported. Normally a user module would not explicitly call an *FPU* function.

All other library modules are normal i.e.

- they must be explicitly imported by modules which access their exported items

- they could be replaced with alternative versions developed by an Armaide user.

Some library procedures use assertions to check that the values of input parameters are within a valid range. Invalid values result in a runtime assertion error. The error codes and reason for the error are listed in Section 6. *Runtime Errors* below.

## 6.3.1 Convert

### Description

Functions for converting integers to strings and vice versa.

### Definition

```
DEFINITION MODULE Convert;

IMPORT Put;

CONST
  (* possible values for result *)
  noError* = 0;
  overflow* = 1;
  syntaxError* = 2;

PROCEDURE StrToInt*(CONST str: ARRAY OF CHAR; VAR n: INTEGER; VAR result: INTEGER);

PROCEDURE IntToStr*(CONST n: INTEGER; VAR s: ARRAY OF CHAR);

PROCEDURE IntToHex*(CONST n: INTEGER; VAR s: ARRAY OF CHAR);

END Convert.
```

## 6.3.2\* IAP

### Description

Functions for using the In-Application (IAP) programming functions of the target processor. These allow the reading and writing of the on-chip flash memory as well as obtaining the part identification number and boot code version number.

Each function and its parameters has a direct correspondence to one of the IAP commands. Refer to the chapter titled *Flash memory system and programming* in the NXP target processor User Manual (UM10xxx) for details of their use.

NOTE: The IAP functions use 128 bytes of RAM at the top of memory. This is declared in module LPC (the first module with global variables to be loaded) to prevent global variables in your program from being overwritten.

### Definition

```
DEFINITION MODULE IAP;

IMPORT SYSTEM, LPC;

CONST
  (* ISP / IAP return codes *)
  OK* = 0;
  InvalidCommand* = 1;
  SrcAddrError* = 2;
  DstAddrError* = 3;
  SrcAddrNotMapped* = 4;
  DstAddrNotMapped* = 5;
  CountError* = 6;
  InvalidSector* = 7;
  SectorNotBlank* = 8;
  SectorNotPrepared* = 9;
  CompareError* = 10;
  Busy* = 11;
  ParamError* = 12;
  AddrError* = 13;
  AddrNotMapped* = 14;
  CmdLocked* = 15;
  InvalidCode* = 16;
  (* 17, 18 only used by ISP *)
  ReadProtected* = 19;

PROCEDURE PrepareSectors*(CONST first, last: INTEGER): INTEGER;

PROCEDURE WriteFlash*(CONST src, dest, nBytes: INTEGER): INTEGER;

PROCEDURE EraseSectors*(CONST first, last: INTEGER): INTEGER;

PROCEDURE BlankCheckSectors*(CONST first, last: INTEGER; VAR offset, contents: INTEGER):
INTEGER;

PROCEDURE ReadPartID*(VAR partID: INTEGER): INTEGER;

PROCEDURE ReadBootVersion*(VAR version: INTEGER): INTEGER;

PROCEDURE CompareBytes*(CONST addr1, addr2, nBytes: INTEGER; VAR offset: INTEGER): INTEGER;

PROCEDURE ReinvokeISP*();

END IAP.
```

### 6.3.3 LPC

#### Description

Defines the constants representing the hardware-specific addresses and control codes which are common to the NXP LPC2000-family of microcontrollers.

If the application uses features that are specific to the target processor, one of the LPC2xxx modules described below should be imported instead of this. It is not necessary to import both modules as each of the LPC2xxx modules includes the general LPC definitions.

The sole procedure, *Configure*, sets the read-only exported variables *Fosc* and *Target* to the value of the crystal frequency and the integer part of the target name supplied to the Armaide *Link Options* dialog.

e.g. if frequency 14745600 was entered and LPC2106 was selected, then *Fosc* = 14745600 and *Target* = 2106.

Depending on the target, *Configure* also initialises the exported global variables:

*MSEL*, *PSEL*, *CCLK* and *PCLK* (LPC21xx & LPC22xx targets)  
*MSEL*, *NSEL*, *CCLK*, *Fcco* and *CCLKSEL* (LPC23xx targets)

Source code of *LPC* is supplied with the Professional Edition to enable customisation of the startup process.

#### Definition

Constants are included for the following functions:

- Vectored Interrupt Controller (VIC)
- Pin Connect Block
- General Purpose Input/Output (GPIO)
- Memory Accelerator Module (MAM)
- Phase Locked Loop (PLL)
- APB / VPB Divider
- Power Control
- External Interrupts
- Timer 0
- Timer 1
- Universal Asynchronous Receiver Transmitter 0 (UART0)
- Serial Peripheral Interface 0 (SPI0)
- Real Time Clock (RTC)
- Watchdog timer

See the definition module for details:

See Armaide\Lib\LPC.def

### 6.3.4 \* LPC2103

#### Description

Contains definitions for the configuration registers and peripheral addresses of the LPC2103 and compatible targets LPC2101 and LPC2102.

Refer to the relevant NXP User Manual UM10161 for detailed descriptions of the function of each of the items contained in this module.

#### Definitions

Constants include those from module LPC and the following additional functions:

- Fast General Purpose Input/Output (GPIO)
- Universal Asynchronous Receiver Transmitter 1 (UART1)
- Inter-Integrated Circuit interface 0 (I2C0)
- Analog/Digital Converter (ADC)
- Inter-Integrated Circuit interface 1 (I2C1)
- Synchronous Serial Port interface (SSP)
- Timer 2
- Timer 3
- Reset Source Identification
- Code Security Protection
- Syscon Miscellaneous

See the corresponding source code file for the details:

Armaide\Lib\LPC2103.mod

## 6.3.5\* LPC2106

### Description

Contains definitions for the configuration registers and peripheral addresses of the LPC2106 and compatible targets LPC2104 and LPC2105.

Refer to the relevant NXP User Manual UM10275 for detailed descriptions of the function of each of the items contained in this module.

### Definitions

Constants include those from module LPC and the following additional functions:

- Pulse Width Modulator (PWM)
- Universal Asynchronous Receiver Transmitter 1 (UART1)
- Inter-Integrated Circuit interface 0 (I2C0)

See the corresponding source code file for the details:

`Armaide\Lib\LPC2106.mod`

## 6.3.6\* LPC2148

### Description

Contains definitions for the configuration registers and peripheral addresses of the LPC2148 and compatible targets LPC2142, LPC2144 and LPC2146.

Refer to the relevant NXP User Manual UM10139 for detailed descriptions of the function of each of the items contained in this module.

### Definitions

Constants include those from module LPC and the following additional functions:

- Phase Locked Loop 1 (PLL1)
- Universal Asynchronous Receiver Transmitter 1 (UART1)
- Analog/Digital Converter (ADC)
- Inter-Integrated Circuit interface 0 (I2C0)
- Inter-Integrated Circuit interface 1 (I2C1)
- Synchronous Serial Port interface (SSP)
- Reset Source Identification
- Code Security Protection
- System Control Miscellaneous
- Pulse Width Modulator (PWM)
- USB Controller

See the corresponding source code file for the details:

Armaide\Lib\LPC2148.mod

## 6.3.7\* LPC2214

### Description

Contains definitions for the configuration registers and peripheral addresses of the LPC2214 and the compatible target LPC2212.

Refer to the relevant NXP User Manual UM10114 for detailed descriptions of the function of each of the items contained in this module.

### Definitions

Constants include those from module LPC and the following additional functions:

- Fast General Purpose Input/Output (GPIO)
- Pulse Width Modulator (PWM)
- Universal Asynchronous Receiver Transmitter 1 (UART1)
- Inter-Integrated Circuit interface 0 (I2C0)
- Inter-Integrated Circuit interface 1 (I2C1)
- Analog/Digital Converter (ADC)
- Serial Peripheral Interface 1 (SPI1)
- Synchronous Serial Port interface (SSP)
- Controller Area Network (CAN1 - CAN4)
- Reset Source Identification
- Code Security Protection
- System Control Miscellaneous

See the corresponding source code file for the details:

Armaide\Lib\LPC2214.mod

## 6.3.8 \* LPC2378

### Description

Contains definitions for the configuration registers and peripheral addresses of the LPC2378 and compatible targets LPC2364, LPC2366 and LPC2368.

Refer to the relevant NXP User Manual UM10211 for detailed descriptions of the function of each of the items contained in this module.

### Definitions

Constants include those from module LPC and the following additional functions:

- Fast General Purpose Input/Output (GPIO)
- Clock Dividers
- External Memory Controller (EMC)
- Static RAM access registers
- Timer 2
- Timer 3
- Pulse Width Modulator (PWM) 1
- Universal Asynchronous Receiver Transmitter 1 (UART1)
- Universal Asynchronous Receiver Transmitter 2 (UART2)
- Universal Asynchronous Receiver Transmitter 3 (UART3)
- I2C Interface 0
- I2C Interface 1
- I2C Interface 2
- SSP0 Controller
- SSP1 Controller
- A/D Converter 0 (AD0)
- D/A Converter
- CAN Controllers and Acceptance Filter
- Multimedia Card Interface (MCI) Controller
- I2S Interface Controller (I2S)
- General-purpose DMA Controller
- USB Controller
- USB Host Controller
- USB OTG and I2C Registers
- Ethernet MAC (32 bit data bus)

See the corresponding source code file for the details:

Armaide\Lib\LPC2378.mod

## 6.3.9 Main

### Description

When Main is initialised it calls LPC.Configure and then executes the second-level startup code - memory mapping (MAM) control, runtime error-trapping initialisation, phase-locked loop (PLL) setup etc.

Main must be included in the list of imports in the main module of the application. The linker will report an error if Main was not loaded during the linking process.

The source code of *Main* is supplied with the Professional Edition to enable user-customisation of the startup process.

### Definition

```
DEFINITION MODULE Main;  
  
IMPORT SYSTEM, LPC, Serial, Out, Traps;  
  
END Main.
```

## 6.3.10 \* Math - Mathematical Functions

### Description

Math contains basic mathematical and trigonometrical functions.

*Sqrt* returns the square root of  $x$ .

*Ln* is the natural logarithm (log to base  $e$ ) of  $x$ .

*Exp* returns the value of the mathematical constant  $e$  to the power of  $x$ .

The parameter  $x$  for *Sin*, *Cos* and *ArcTan* is in radians ( $2 * \pi$  radians =  $360^\circ$ ).

### Definition

```
DEFINITION MODULE Math;
  PROCEDURE Sqrt*(x: REAL): REAL;
  PROCEDURE Ln*(x: REAL): REAL;
  PROCEDURE Exp*(CONST x: REAL): REAL;
  PROCEDURE Sin*(CONST x: REAL): REAL;
  PROCEDURE Cos*(CONST x: REAL): REAL;
  PROCEDURE ArcTan*(x: REAL): REAL;
END Math.
```

## 6.3.11 MAU - Memory Allocation Unit

### Description

MAU contains the functions used by the system for dynamic variable memory allocation. MAU is dependent on the version of the compiler and must follow some specific conventions. It should not be substituted by a user-defined module.

If a user module calls the Oberon NEW function then *MAU.New* is automatically called and the MAU module automatically imported. *MAU.New* should not be called directly by a user module.

*HeapUsed* returns the number of bytes of RAM currently used by all dynamic variables.

*MemAvailable* returns the number of bytes of RAM currently free to be used for the user stack and additional dynamic variables.

### Definition

```
DEFINITION MODULE MAU;
IMPORT SYSTEM;
PROCEDURE New*(VAR p: INTEGER; CONST T: INTEGER);
PROCEDURE HeapTop*(): INTEGER;
PROCEDURE HeapUsed*(): INTEGER;
PROCEDURE MemAvailable*(): INTEGER;
END MAU.
```

## 6.3.12 Out

### Description

Out contains basic formatted text output functions. Output can be directed to any device that accepts ASCII character data by calling *Init* with the name of a procedure that outputs a single character to that device. When the system module Main is initialised, *Serial.PutCh* is the procedure passed to *Out.Init*. Hence, by default, output is directed to the serial port UART0.

Procedure *Out.Ln* writes the carriage return / line feed pair of characters (*ODX*, *OAX*)

Procedures with a *width* parameter output text which is left-justified. If the number of characters that are output is less than *width*, a sufficient number of blanks are output to make up the difference.

### Definition

```
DEFINITION MODULE Out;

IMPORT Convert;

TYPE
  PutCharProc* = PROCEDURE(CONST ch: CHAR);

  PROCEDURE Init*(CONST p: PutCharProc);

  PROCEDURE Char*(CONST ch: CHAR);

  PROCEDURE String*(CONST s: ARRAY OF CHAR);

  PROCEDURE Ln*();

  PROCEDURE Int*(CONST n, width: INTEGER);

  PROCEDURE Hex*(CONST n, width: INTEGER);

  PROCEDURE Real*(CONST r: REAL; CONST width: INTEGER);

END Out.
```

### 6.3.13 Put

#### Description

Contains string-handling helper functions used internally by Convert and Reals.

#### Definition

```
DEFINITION MODULE Put;  
  PROCEDURE Init*();  
  PROCEDURE* EOS*(VAR s: ARRAY OF CHAR);  
  PROCEDURE* Ch*(CONST ch: CHAR);  
  PROCEDURE Str*(CONST s: ARRAY OF CHAR);  
  PROCEDURE Int*(n: INTEGER);  
END Put.
```

### 6.3.14 \* Random

#### Description

Random is a pseudo-random number generator based on the example in *Programming in Oberon - Reiser & Wirth, ACM Press 1992*.

*Next* returns the next random number in a reproducible sequence.

When Random is initialised by the system *Seed* is called with the value 314159.

Call *Random.Seed* with a different value to initiate a different sequence of numbers.

#### Definition

```
DEFINITION MODULE Random;  
  
  PROCEDURE Next*(CONST range: INTEGER): INTEGER;  
  
  PROCEDURE Seed*(CONST value: INTEGER);  
  
END Random.
```

## 6.3.15 Reals

### Description

Real number support functions.

*Exponent(r)* returns the value of *exp*, and *Mantissa(r)* returns the normalised value of *r*, resulting from a call to the Oberon-07 function *UNPK(r, exp)*. *Real* constructs a real number from a normalised mantissa and exponent.

*Ten* returns the value  $10.0^e$

*RealToStrE* displays the real value in exponential notation, *RealToStrF* uses fixed point notation. *digits* is the number (1..7) of significant digits to use.

### Examples

x	digits	RealToStrE	RealToStrF
0.0	1	0.0E+00	0.0
0.0	7	0.000000E+00	0.0
0.70710698	5	7.0711E-01	0.70711
0.70710698	6	7.07107E-01	0.707107
0.70710698	7	7.071070E-01	0.707107
0.999999	5	1.0000E+00	1.0
0.999999	6	9.99999E-01	0.999999
Reals.Ten(7)	7	1.000000E+07	10000000.0
Reals.Ten(20)	7	1.000000E+20	1.000000E+20

### Definition

```
DEFINITION MODULE Reals;

IMPORT Put;

(* Possible values for result *)
CONST
  noError* = 0;
  overflow* = 1;
  syntaxError* = 2;

PROCEDURE Exponent*(CONST x: REAL): INTEGER;

PROCEDURE Mantissa*(CONST x: REAL): REAL;

PROCEDURE Real*(CONST m: REAL; CONST e: INTEGER): REAL;

PROCEDURE Ten*(CONST e: INTEGER): REAL;

PROCEDURE StrToReal*(CONST s: ARRAY OF CHAR; VAR x: REAL; VAR result: INTEGER);

PROCEDURE RealToStrE*(x: REAL; CONST digits: INTEGER; VAR s: ARRAY OF CHAR);

PROCEDURE RealToStrF*(x: REAL; CONST digits: INTEGER; VAR s: ARRAY OF CHAR);

END Reals.
```

## 6.3.16 \* ResData

### Description

Contains functions to access constant resource data (e.g. fonts, bitmaps, input data etc.) that was appended to the executable by the Armaide linker. See the section titled *Resource Data* below for more information.

*name* is the (case-sensitive) name of the module that had the same name as the linked resource file. If the name is longer than eight characters it is truncated to eight characters.

*index* is a zero-based 32-bit offset into the resource data.

*Size* returns the size of the resource data in bytes.

*Count* returns the number of named resources attached to the current application.

### Definition

```
MODULE ResData;

IMPORT LPC, SYSTEM, Out;

TYPE
  Resource* = POINTER TO RECORD
  END;

  DirEntry* = RECORD
    name*: Name;
    size*: INTEGER
  END;

  PROCEDURE Size*(CONST r: Resource): INTEGER;

  PROCEDURE GetInt*(CONST r: Resource; CONST index: INTEGER; VAR data: INTEGER);

  PROCEDURE GetIntArray*(CONST r: Resource; CONST index: INTEGER; count: INTEGER;
    VAR items: ARRAY OF INTEGER): INTEGER;

  PROCEDURE GetReal*(CONST r: Resource; CONST index: INTEGER; VAR data: REAL);

  PROCEDURE GetRealArray*(CONST r: Resource; CONST index: INTEGER; count: INTEGER;
    VAR items: ARRAY OF REAL): INTEGER;

  PROCEDURE Count*(): INTEGER;

  PROCEDURE GetDirectory*(VAR list: ARRAY OF DirEntry);

  PROCEDURE Open*(VAR r: Resource; CONST name: ARRAY OF CHAR);

END ResData.
```

### Example

If the first word in the binary data file *MyData.res* is a 32-bit integer with the value 10, the following code assigns 10 to the INTEGER variable *count*. The next ten words of resource data are then interpreted as REAL data and stored into the dynamically allocated array *weights*.

```
VAR
  r: ResData.Resource;
  count: INTEGER;
  weights: ARRAY OF REAL;
BEGIN
```

```
ResData.Open(r, "MyData");  
ResData.GetInt(r, 0, count);  
NEw(weights, count);  
ResData.GetRealArray(r, 1, count, weights)  
...
```

## 6.3.17 Serial

### Description

Contains basic functions for sending and receiving single ASCII characters via the UART0 serial port using polling. The communications parameters used are 8 bits, no parity, 1 stop bit.

The baudrate is initially set to 38,400 baud but can be changed using the *Init* function.

### Definition

```
DEFINITION MODULE Serial;

IMPORT SYSTEM, LPC, Timer;

PROCEDURE Init*(CONST baudRate: INTEGER);

PROCEDURE PutCh*(CONST ch: CHAR);

PROCEDURE GetCh*(VAR ch: CHAR);

END Serial.
```

## 6.3.18 \* Strings

### Description

Contains general functions for processing strings i.e. string constants and arrays of characters terminated with the null character 0X. The functions and following descriptions are based on the corresponding definitions in the Oakwood Guidelines for Oberon-2 Compiler Developers.

*Length(s)* returns the number of characters in s up to and excluding the first 0X.

*Extract(src, pos, n, dest)* extracts a substring dest with n characters from position pos ( $0 \leq \text{pos} < \text{Length}(\text{src})$ ) in src. If  $n > \text{Length}(\text{src}) - \text{pos}$ , dest is only the part of src from pos to the end of src, i.e.  $\text{Length}(\text{src}) - 1$ . If the size of dest is not large enough to hold the result of the operation, the result is truncated so that dest is always terminated with a 0X.

*Copy(s, dest)* copies all characters up to and including 0X to dest. If the size of dest is not large enough to hold the result of the operation, the result is truncated so that dest is always terminated with a 0X.

*Append(s, dest)* has the same effect as *Insert(s, Length(dest), dest)*.

*Pos(pat, s, pos)* returns the position of the first occurrence of pat in s. Searching starts at position pos. If pat is not found, -1 is returned.

*Delete(s, pos, n)* deletes n characters from s starting at position pos ( $0 \leq \text{pos} < \text{Length}(s)$ ). If  $n > \text{Length}(s) - \text{pos}$ , the new length of s is pos.

*Insert(src, pos, dest)* inserts the string src into the string dest at position pos ( $0 \leq \text{pos} \leq \text{Length}(\text{dst})$ ). If  $\text{pos} = \text{Length}(\text{dest})$ , src is appended to dest. If the size of dest is not large enough to hold the result of the operation, the result is truncated so that dest is always terminated with a 0X.

*Replace(src, pos, dest)* has the same effect as *Delete(dest, pos, Length(src))* followed by an *Insert(src, pos, dest)*.

*Cap(s)* replaces each lower case letter within s by its upper case equivalent.

## Definition

```
DEFINITION MODULE Strings;

  PROCEDURE Length*(CONST s: ARRAY OF CHAR): INTEGER;

  PROCEDURE Extract*(CONST src: ARRAY OF CHAR; CONST pos: INTEGER; n: INTEGER;
    VAR dest: ARRAY OF CHAR);

  PROCEDURE Copy*(CONST src: ARRAY OF CHAR; VAR dest: ARRAY OF CHAR);

  PROCEDURE Append*(CONST src: ARRAY OF CHAR; VAR dest: ARRAY OF CHAR);

  PROCEDURE Pos*(CONST pattern, s: ARRAY OF CHAR; pos: INTEGER): INTEGER;

  PROCEDURE Delete*(VAR s: ARRAY OF CHAR; pos, n: INTEGER);

  PROCEDURE Insert*(CONST src: ARRAY OF CHAR; CONST pos: INTEGER; VAR dest: ARRAY OF CHAR);

  PROCEDURE Replace*(CONST src: ARRAY OF CHAR; pos: INTEGER; VAR dest: ARRAY OF CHAR);

  PROCEDURE Cap*(VAR s: ARRAY OF CHAR);

END Serial.
```

## 6.3.19 SYSTEM

### Description

SYSTEM is a pseudo-module i.e. it contains no source code. Its functionality is implemented entirely within the compiler. Some of the functions allow parameters of any *basic* type i.e. INTEGER, SET, BOOLEAN etc. to be passed. Others allow parameters of *any* type. Generic functions of this type are normally not possible to write using the Oberon language.

The presence of SYSTEM in the IMPORT list of a module indicates that the module is very implementation dependent and hence, inherently non-portable.

*ADR* returns the absolute address of the given variable.

*SIZE* returns the number of bytes used by a variable of the given type.

*BIT* returns TRUE if the specified bit number of the contents of the given address is equal to 1, otherwise FALSE.

*GET* stores the value of the word at memory address *addr* into variable *v*.

*PUT* stores the value of *x* into the word at memory address *addr*.

*VAL* is a *type-transfer / typecast* mechanism. It should be used with extreme care as it effectively bypasses any type-safety checks. It allows the value of *x* (which can be of any type) to be interpreted as if it were declared as type *typeName*. No value conversion takes place – the bit pattern of the result is identical to the bit pattern of the original.

### Definition

```
DEFINITION MODULE SYSTEM;

VAR
  PC*: INTEGER;
  LNK*: INTEGER;
  SP*: INTEGER;
  FP*: INTEGER;

PROCEDURE ADR*(CONST variableName: <any type>): INTEGER;

PROCEDURE SIZE*(CONST typeName: <any type>): INTEGER;

PROCEDURE BIT*(CONST address, bitNo: INTEGER): BOOLEAN;

PROCEDURE GET*(CONST address: INTEGER; VAR v: <any basic type>);

PROCEDURE PUT*(CONST address: INTEGER; CONST x: <any basic type>);

PROCEDURE VAL*(CONST typeName: <any type>; CONST x: <any type>): typeName;

END SYSTEM.
```

## 6.3.20 Timer

### Description

Contains functions for timed delays and for measuring elapsed execution time.

The delay functions execute a continuous loop until the measured elapsed time exceeds the given *delay*. The units of time are microseconds or milliseconds depending on which procedure is used.

*Elapsed* returns the time that passed between the two most recent calls of *Start* and *Stop*. The units of time of the value returned by *Elapsed* depend on the value of units passed to the *Init* function. *Init* must be called before the first call to *Start*.

### Definition

```
DEFINITION MODULE Timer;

IMPORT LPC, SYSTEM;

CONST
  (* Possible values for units *)
  uSecs* = TRUE;
  MSecs* = FALSE;

PROCEDURE* uSecDelay*(CONST delay: INTEGER);

PROCEDURE MSecDelay*(CONST delay: INTEGER);

PROCEDURE* Init*(CONST units: BOOLEAN);

PROCEDURE* Start*;

PROCEDURE* Stop*;

PROCEDURE* Elapsed*(): INTEGER;

END Timer.
```

## 6.3.21 Traps

### Description

Implements default interrupt handlers for software interrupts and the standard internal ARM interrupts *Abort Data*, *Abort Instruction* and *Undefined Instruction*. The default handlers are installed when *Traps.Init* is called from the *Main.Init* function at startup time.

The software interrupt instruction (SWI) handler is invoked whenever Armaide executes a statement which results in a runtime error (e.g. index out of range, integer overflow etc.) Control also passes to the handler if an ASSERT statement is executed with a parameter which equates to FALSE.

See the *Interrupt Handlers* section above and the *Runtime Errors* section below for more details.

The source code of *Traps* is supplied with the Professional Edition of Armaide to enable user-customisation of the interrupt handling process.

### Definition

```
DEFINITION MODULE Traps;  
  
    PROCEDURE Init*;  
  
END Traps.
```

## 6.4 \* Resource Data

The usual way to process constant data in an Oberon-07 program is to declare the values in a CONST list or store them in a global array in the initialisation section of a module. Neither of these methods is practical when dealing with large amounts of constant data (e.g. the definition of a font, a bitmap image etc.).

Typically on a PC system, this sort of data would be stored in a file to be read at runtime. As a file system is often not available on the smaller embedded systems targeted by Armaide, a different approach is required. The solution used is to gather together all of the relevant data files at link time and append them to the linked executable to be stored in Flash ROM when the program is uploaded.

A library module *ResData* is provided to allow the programmer to conveniently access the data from Flash ROM within the program as if it were data stored in a random-access disk file. A description of the available functions is included in the *Library Module Definitions* section above.

Several resources can be attached to the one program; each is identified by its module name. Typically, the steps involved in making a resource file are:

- Make a copy of the original data file
- Rename the copy to match the associated module name with the extension *.res*
- Move the renamed copy to the folder which contains the source code of the module

At link time, after the Armaide linker has linked all of the object files *<module>.arm* into the executable program, it looks for the corresponding resource files named *<module>.res* and appends them to the executable.

If you need to associate several different resource files with one module you could create an empty resource module for each separate resource e.g.

```
MODULE MyData;  
END MyData.
```

and then include the names of those resource modules in the IMPORT list of the associated module.

The resource file can contain any type of data. How that data is interpreted is determined by the programmer. The only requirement is that the size of the file is a multiple of four bytes.

## 6.5 Link Options

The Link Options (<module>.opt) file is created or updated when the *Project > Link Options* menu command is executed. It contains the necessary details to produce an executable file from the main object file (<module>.arm) for a specific target processor. It is a text file which is located in the same folder as the main module.

The format of the file is:

```
ID=<value>
ID=<value>
etc.
```

The IDs used are:

ID	Description	Units	Default
Target	Processor type		None
CrystalFreq	Crystal Frequency (Fosc) of target	Hz	None

The supported target processor types are

```
LPC2000 (Generic)
LPC2101, LPC2102, LPC2103
LPC2104, LPC2105, LPC2106
LPC2142, LPC2144, LPC2146, LPC2148
LPC2212, LPC2214
LPC2364, LPC2366, LPC2368, LPC2378
```

**NOTE:** The standard startup files supplied with Armaide are only suitable for use with Revision 'B' parts of the LPC2378. Refer to the NXP LPC2378 Errata Sheet Version 1.7 for details of the limitations of earlier revisions if you intend to customise the startup files.

## 6.6 Memory Map

The memory usage of an Oberon program executing on an LPC2000 device depends on the FLASH ROM / STATIC RAM capacity of the type of target microcontroller.

LPC2xxx

	Start Address	Direction	End Address	Size (bytes)
Global Data	See below	DEC	Start - Size	Global Data Size
Stacks	End of Global Data	DEC	End of heap	See below
Heap	04000 0100H	INC	End of stack	See below
Unused	04000 0040H	INC	04000 00FFH	256 - 64
Mapped Interrupt Vectors	04000 0000H		04000 003FH	64
User Code	00000 0400H		See below	
Module Table	00000 0200H		00000 03FFH	512
System Vars	00000 0100H		00000 02FFH	256
Startup	00000 0000H		00000 00FFH	256

User Code (FLASH)

	End Address	Size
LPC2101	00000 1FFFH	7K
Evaluation Edition LPC2102	00000 3FFFH	15K
LPC2103	00000 7FFFH	31K
LPC2142	00000 FFFFH	63K
LPC2104 / 05 / 06 LPC2144 LPC2212 LPC2364	00001 FFFFH	127K
LPC2146 LPC2214 LPC2366	00003 FFFFH	255K
LPC2148 LPC2368 / 78	00007 DFFFH	511K

\* Global Data / Stacks / Heap (RAM)

	Global Data Start Address	Stacks Size	Heap Size
Evaluation Edition LPC2101	04000 07FFH	~2K - Global - Heap	~2K - Global - Stack
LPC2102	04000 0FFFH	~4K - Global - Heap	~4K - Global - Stack
LPC2103 LPC2364	04000 1FFFH	~8K - Global - Heap	~8K - Global - Stack
LPC2104 LPC2142 / 44 LPC2212 / 14	04000 3FFFH	~16K - Global - Heap	~16K - Global - Stack
LPC2105 LPC2146 / 48 LPC2366 / 68 / 78	04000 7FFFH	~32K - Global - Heap	~32K - Global - Stack
LPC2106	04000 FFFFH	~64K - Global - Heap	~64K - Global - Stack

NOTE: The top 128 bytes of RAM is declared in LPC to reserve it for use by the IAP functions so they do not overwrite your global data.

## 7 Runtime Errors

### 7.1 Runtime Error Codes

The error codes assigned to runtime errors and assertions detected by Oberon-07 for ARM are:

Code	Reason
1	Index out of bounds
2	Type test failure
3	Destination array shorter than source array
4	Invalid value in case statement
5	Reserved
6	String too long or destination string too short
7	Integer division by zero or negative divisor
8, 9, 10	FPU assertions
11	Heap overflow
12..19	Reserved
20..99	Library assertions
100..255	User-defined assertions

## 7.2 Library Assertion Error Codes

The values of the input parameters of some library procedures are checked using *ASSERT* to ensure that they are within a valid range. The following is a list of the error codes which are reported if the assertions fail:

Module	Procedure	Code	Assertion
Convert	IntToHex	20	LEN(s) >= 10
Math	Sqrt	20	x >= 0.0
Math	Ln	22	x > 0.0
Out	PutCh	20	Out.Init has not been called
Reals	Real	20	(1.0 <= ABS(m)) & (ABS(m) < 2.0)
Reals	Ten	21	(e >= 0) & (e <= 38)
Reals	RealToStrE	22	LEN(s) > digits + 6
Reals	RealToStrE	23	digits IN {1..7}
Reals	RealToStrF	24	digits IN {1..7}
ResData	Open	20	ID = "OB7R"
ResData	Open	21	Version number
ResData	GetInt	22	index <= r.nItems
ResData	GetIntArray	23	index <= r.nItems
ResData	GetReal	24	index <= r.nItems
ResData	GetRealArray	25	index <= r.nItems

## 7.3 Programmer-defined Assertions

The programmer can use the Oberon-07 *ASSERT* function to trap an application-specific error e.g. to detect impending stack overflow:

```
ASSERT(MAU.MemAvailable < minRequired, 130)
```

where *minRequired* is a user-defined value.

## 7.4 Reporting Runtime Errors

The above runtime, library and programmer-defined error conditions and assertions result in the execution of an ARM software interrupt instruction (SWI) which calls a default trap handler in the Armaide library module *Traps*.

The trap handler reports:

- the address of the instruction which caused the error
- the name of the module that was being executed
- the line number of the corresponding statement in the source code
- the error code identifying what type of error it is

The information is reported using the standard IO functions exported by the Armaide *Out* module. By default the messages will appear on a serial terminal connected to UART0. The trap handler then processes an infinite loop until the system is reset.

The source code of *Traps* is included in the Professional Edition of Armaide to allow user-customisation of the trap-handling process.

## 7.5 Diagnosing Runtime Errors

When a runtime error occurs or an assertion fails use the module name and line number information reported by the trap handler to identify the source of the error.

If you have the source code of the module:

- Open the source code of the module in the editor
- Use the *Search > Goto* command to locate the actual source line by its line number.

If the address is located in one of the library modules and you do not have the source code:

- Refer to the *Library Assertion Codes* table above to identify the function and assertion associated with the error code.

## 8 \* Command-line Tools

### 8.1 Compiler and Linker

The Professional Edition includes separate command-line programs for the Oberon-07 ARM compiler and LPC2000 linker in addition to the corresponding compile and link commands in the IDE.

The separate compiler and linker can be used with automatic 'build' tools, DOS-batch commands etc. These are useful for handling a regular series of compilations and links when building multiple configurations, multiple targets etc. They can also be useful when recompiling a number of modules after changing the interface of a low-level imported module.

Both commands require one parameter. The compiler needs the filename of the module being compiled. The linker needs the filename of the link options file associated with the main module of the application.

The syntax of the commands is:

```
ArmaideCompile [<path>]<moduleName>.mod
```

```
ArmaideLink [<path>]<mainModule>.opt
```

e.g.

```
ArmaideCompile Main.mod
```

```
ArmaideLink c:\ArmProjects\LPC2106\Blinker.opt
```

If the path is included in the filename, it is used as the working folder for the command. The search folders used are relative to the working folder.

## Command Return Codes

If the command executes without any compiler or linker errors it returns zero otherwise it returns 1. An example of a DOS batch script which uses these return values is:

```
@echo off
REM
REM Rebuild Library
REM
del Lib\*.arm
del Lib\*.smb
ArmaideCompile Lib\Math.Mod
if errorlevel 1 goto ErrorExit
ArmaideCompile Lib\Random.Mod
if errorlevel 1 goto ErrorExit
ArmaideCompile Lib\FPU.Mod
if errorlevel 1 goto ErrorExit
ArmaideCompile Lib\Put.Mod
if errorlevel 1 goto ErrorExit
ArmaideCompile Lib\LPC.Mod
if errorlevel 1 goto ErrorExit
ArmaideCompile Lib\LPC214x.Mod
if errorlevel 1 goto ErrorExit
ArmaideCompile Lib\LPC221x.Mod
if errorlevel 1 goto ErrorExit
ArmaideCompile Lib\MAU.Mod
if errorlevel 1 goto ErrorExit
ArmaideCompile Lib\Timer.Mod
if errorlevel 1 goto ErrorExit
ArmaideCompile Lib\Convert.Mod
if errorlevel 1 goto ErrorExit
ArmaideCompile Lib\Serial.Mod
if errorlevel 1 goto ErrorExit
ArmaideCompile Lib\Out.Mod
if errorlevel 1 goto ErrorExit
ArmaideCompile Lib\Traps.Mod
if errorlevel 1 goto ErrorExit
ArmaideCompile Lib\Main.Mod
if errorlevel 1 goto ErrorExit
ArmaideCompile Lib\Reals.Mod
if errorlevel 1 goto ErrorExit
echo No errors detected
goto OK
:ErrorExit
echo Errors detected
:OK
pause
```

## 9 Programming Conventions and Guidelines

This chapter describes the programming guidelines and source code formatting conventions which have been used in software developed using Armaide.

Some programming guidelines are more important than others. In the first section, the more important ones are described. The remaining sections contain more cosmetic rules which describe the look-and-feel of Oberon-07 programs published by CFB Software. If you like them, feel free to use them for your programs as well. It may make your programs easier to understand for someone who is used to the design, documentation, and coding patterns used in applications developed using Armaide.

### 9.1 Essentials

The most important programming conventions all centre around the aspect of evolvability. It should be made as easy as possible to change existing programs in a reliable way, even if the program has been written a long time ago or by someone else. Evolvability can often be improved by increasing the locality of program pieces: if a piece of program may only have an effect on a clearly locatable stretch of program text, it is easier to know where a program modification may necessitate further changes. Basically, it's all a matter of keeping "ripple effects" under control.

#### Precondition Checks

Preconditions are one of the most useful tools to detect unaccounted ripple effects. Precondition checks allow to pinpoint semantic errors as early as possible, i.e. as closely to their true source as possible. After larger design changes, properly used assertions can help to dramatically reduce debugging time.

Whenever possible, use static means to express what you know about a program's design. In particular, use the type and module systems of Oberon-07 for this purpose; so the compiler can help you to find inconsistencies, and thus can become an effective refactoring tool.

Precondition assertions should be used consistently. Don't allow client code to "enter" your module if it doesn't fulfill the preconditions of your module's procedures. In this way, you avoid propagation of foreign errors into your own code.

```
PROCEDURE Ten*(CONST e: INTEGER): REAL;  
BEGIN  
  ASSERT((e >= 0) & (e <= 38), 21)  
  ...  
END
```

Assertion codes should be in the range 100 to 255 to avoid being confused with those used in the Armaide runtime system and libraries..

#### Global Variables

There should be as few global variables as possible. Global variables can be accessed from many places in a program, at different times. This makes it difficult to keep track of all possible interactions ("side effects") with such variables. This in turn increases the likelihood of introducing errors when changing the use of them.

#### Function Procedures

Procedures which return a result should not modify global variables or VAR parameters as side effects. It is easier to deal with function procedures if they are true functions in the mathematical sense, i.e., if they don't have side effects. Returning function results is ok.

Procedures should be kept as small as is practicable. It is preferable if the whole function is visible on the screen without having to scroll.

## 9.2 Indentation

A new indentation level is realised by pressing the tab key. The number of spaces inserted depends on the editor option *Indent width*

A monotype font (e.g. Times New Roman, Consolas) should be used to assist consistent indentation.

Do not use more than three levels of nesting (IF, WHILE etc.). Aim to limit the scope of each block statement so that it is completely visible on one screen.

Combine nested IFs into single boolean expressions where appropriate:

```
IF (p # NIL) THEN
  IF (p.val # 0) THEN
```

should be written as:

```
IF (p # NIL) & (p.val # 0) THEN
```

Oberon-07 uses short-circuit evaluation of such expressions. i.e. if the first expression is FALSE, the second expression is not evaluated.

## 9.3 Semicolons

Semicolons are used to separate statements, not to terminate statements. This means that there should be no superfluous semicolons.

Good

```
IF done THEN
  Print(result)
END
```

Bad

```
IF done THEN
  Print(result);
END
```

## 9.4 Dereferencing

The optional dereferencing operator  $\wedge$  should be left out wherever possible.

Good

```
h.next := p.prev.next
```

Bad

```
h $\wedge$ .next := p $\wedge$ .prev $\wedge$ .next
```

## 9.5 Letter case

In general, each identifier starts with a small letter, except:

- A module name always starts with a capital letter
- A type name always starts with a capital letter
- A procedure always starts with a capital letter, this is true for procedure constants, types, variables, parameters, and record fields.

Good

```
null = 0X;  
DrawDot = PROCEDURE (x, y: INTEGER);  
PROCEDURE Proc (i, j: INTEGER; Draw: DrawDot);
```

Bad

```
NULL = 0X;  
PROCEDURE isEmpty (q: Queue): BOOLEAN;  
R = RECORD  
  draw: DrawDot  
END;
```

Don't capitalise identifiers with more than one character. They should be reserved for the language.

## 9.6 Names

- A proper procedure has a verb as name, e.g. *DrawDot*
- A function procedure has a noun or a predicate as name, e.g. *Exponent(r)*, *IsEmpty(q)*
- Procedure names which start with the prefix *Init* are snappy, i.e., they have an effect only when called for the first time. If called a second time, a snappy procedure either does nothing, or it halts. In contrast, a procedure which sets some state and may be called several times starts with the prefix *Set*.
- *CamelCaps* should be used to identify each word in an identifier, e.g. *startAddress* not *startaddress*
- Names should not be unnecessarily long nor unnecessarily abbreviated, e.g. *maxStep* not *maximumForLoopStep*, *nextPage* not *nxtpg* etc.

## 9.7 White space

A single space should be inserted between lists of symbols, between actual parameters, and between operators:

Good

```
VAR a, b, c: INTEGER;  
DrawRect(1, t, r, b);  
a := i * 8 + j - m[i, j];
```

Bad

```
VAR a,b,c: INTEGER;  
DrawRect(1,t,r,b);  
a:=b;  
a := i*8 + j - m[i,j];
```

## 9.8 Alignment

- Opening and closing keywords are either aligned or on the same line
- IMPORT, CONST, TYPE, VAR, PROCEDURE sections are one level further indented than the outer level.
- PROCEDURE X and END X are always aligned
- If the whole construct does not fit on one line, there is never a statement or a type declaration after a keyword
- The contents of IF, WHILE, REPEAT, FOR, CASE constructs are one level further indented if they do not fit on one line.

### Good

```
IF expr THEN S0 ELSE S1 END;
REPEAT S0 UNTIL expr;
WHILE expr DO S0 END;

IF expr THEN
  S0
ELSE
  S1
END;

REPEAT
  S0
UNTIL expr;

i := 0; WHILE i # 15 DO DrawDot(a, i); INC(i) END;

TYPE Square = POINTER TO RECORD(Rectangle) END;

IMPORT Lists, Out,
  Reals, Main;

VAR
  proc: Lists.Proc;
```

### Bad

```
IF expr THEN S0
ELSE S1 END;

PROCEDURE P;
BEGIN ... END P;

BEGIN i := 0;
  j := a + 2;
  ...

REPEAT i := 0;
  j := a + 2;
  ...
```

## 9.9 Boolean Expressions

Boolean expressions are often misused. Complex logical expressions can often be reduced to a simpler form. Use truth tables to confirm that the simpler form is equivalent.

```
IF (reptype # summary) OR ((reptype = summary) & ~printing)
```

can be simplified to:

```
IF (reptype # summary) OR ~printing
```

Some transformations reveal that two booleans are essentially equivalent and one can be removed altogether.

```
IF continue THEN finished := FALSE ELSE finished := TRUE END;
```

should just be:

```
finished := ~continue;
```

**NOTE:** DO NOT be tempted to make the same transformation to the statement:

```
IF continue THEN finished := FALSE END;
```

Finally,

```
IF continue = TRUE THEN
```

should just be:

```
IF continue THEN
```

## 9.10 Acknowledgements

The guidelines in this chapter have been adapted from the original *BlackBox Component Builder Programming Conventions* with the kind permission of Oberon microsystems AG. ([www.oberon.ch](http://www.oberon.ch))