CFB Software

Astrobe

Oberon for Cortex-M7 Microcontrollers

Astrobe can be used to write software using Oberon for a variety of ARM microcontrollers. This document describes features that are specific to the use of Astrobe when developing software for the Cortex-M7 family of ARM microcontrollers.

Astrobe
Oberon for Cortex-M7 Microcontrollers

Table of Contents

1 Introduction

Astrobe for Cortex-M7 is a fast and responsive integrated development environment for Windows. It is used to write software to run on the powerful ARM Cortex-M7 microcontrollers.

Starter, Personal and Professional editions of Astrobe for Cortex-M7 are available. Refer to the *Astrobe for Cortex-M7 Features Matrix* on the Astrobe website at http://www.astrobe.com/ for details of what is included in each edition.

2 File Descriptions

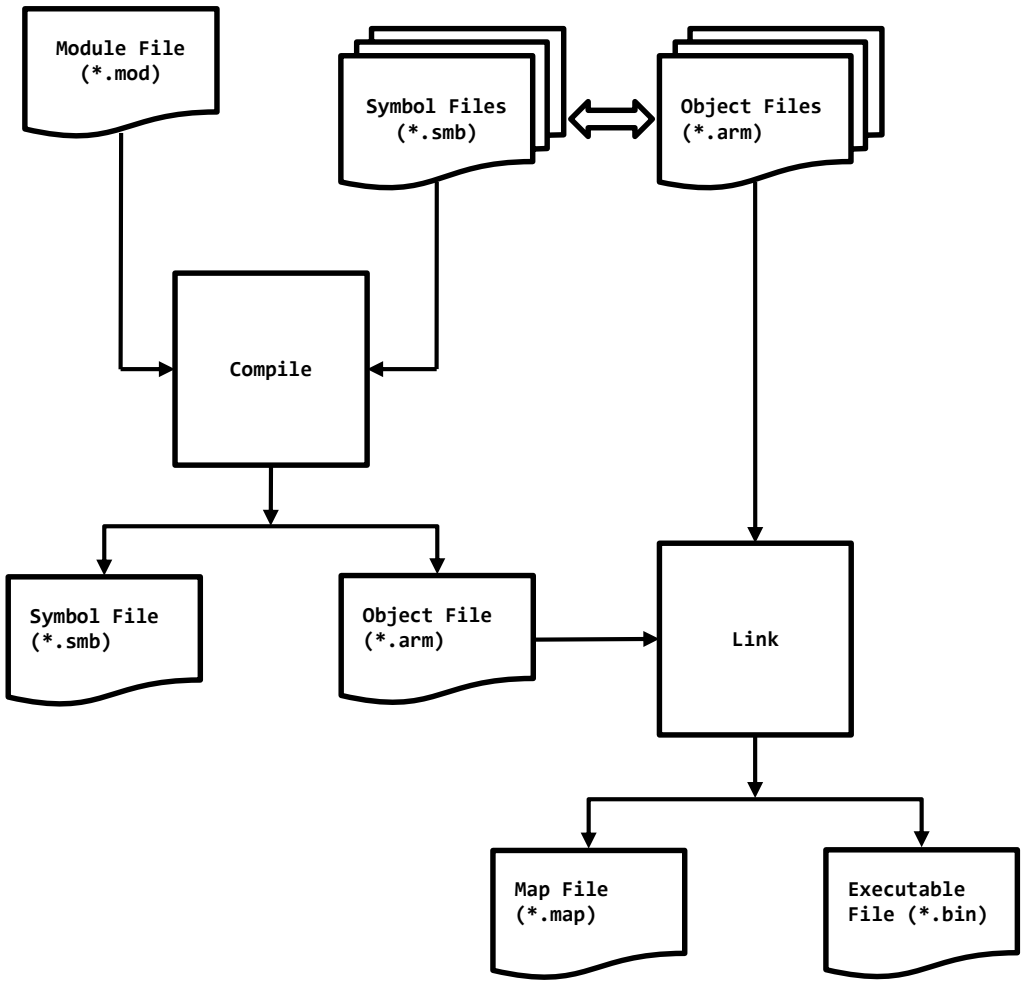
The Astrobe compiler and linker expect there to be a correspondence between the names of modules in the source code and the associated filenames.

When you are creating a new source code file you should give the file the same name as its module name with a *.mod* extension.

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

Extension	Type	Created by	Used by	Scope	Description
.arm	Binary	Compile	Link	Module	Linkable object file
.bin	Binary	Link	Upload	Application	Linked binary executable file
.def	Text			Module	Definitions of exported system items e.g. SYSTEM
.hex	Text	Link		Application	Linked hex executable code
.ini	Text	Configuration	Compile Link Upload	Application	Compile, link, build and upload options
.lst	Text	Disassemble		Module	* Disassembler listing
.map	Text	Link		Application	Code and data memory usage
.mod	Text	Edit	Compile	Module	Source code
.ref	Binary	Link		Application	Trap reference resource data
.res	Any		Link	Module	Resource data
.smb	Binary	Compile	Compile	Module	Symbol file of exported items

*** Professional Edition only**



2.1 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 you are editing *DigiClock.mod* in the Astrobe editor you can automatically open the source code of *LcdDisplay* by clicking on its name in the IDE's Import navigation pane.

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 from the current configuration 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 bin file *DigiClock.bin* and hex file *DigiClock.hex*.

When *DigiClock* is uploaded the flash memory of the target processor is programmed with the contents of the bin file. The hex file is made available for optional use with 3rd-party upload software e.g. FlashMagic.

2.2 Linking and Loading

An application created with Astrobe is made up from a selection of the following modules:

System Modules
Startup code module
Astrobe MCU-specific library modules
Astrobe general library modules

User-developed Modules
Common user 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 binary format suitable to be uploaded by Astrobe and executed on the target processor. The executable is also created as a standard HEX format file suitable for use with 3rd-party uploaders.

A feature of the Oberon language is that all of the information regarding dependencies between the various modules is 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 Astrobe 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 of the target Cortex-M7 microcontroller
- The start and end addresses of the data and code areas

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

The details of microcontroller type, the crystal frequency and the physical locations of the library files are as specified the current *configuration*. See *Library Organisation* below for details.

If you are using the built-in function NEW to allocate memory from the 'heap' to dynamic POINTER variables you can also use the configuration feature to specify:

- The address of the start of the heap
- The limit of the heap

If you keep the default values the CPU RAM is shared between global variables, the stack (local variables) and the heap (POINTER variables). This is suitable for typical applications.

However, if your system has non-CPU RAM that is directly addressable in the same way as CPU RAM then you can change these values so the non-CPU RAM is used by the heap. More memory is then available for global and local variables.

The values entered are listed in the linker progress report and linker map file.

2.3 Startup Code

The stack pointer, 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.

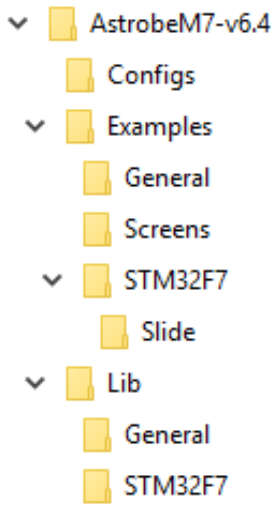
Memory mapping control and phase-locked loop (PLL) options of the microcontroller and the floating point unit are configured in the process of initialising the Astrobe library module *Main*. The module *Main* should be included in the IMPORT list of the main module of every Astrobe application to ensure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

The source code of the *Main*, *MCU* and *Traps* modules can be modified to allow different configurations of memory mapping and PLL features and to customise the output of runtime error messages.

2.4 Library Folders

Groups of common files that are shared between several applications developed using Astrobe may be conveniently organised in a system of *library folders* avoiding the need to duplicate copies of common / shared files.

The Configurations, Library and Examples folder structure supplied with Astrobe is:



The folder *Astrobe\Lib\General* contains the generic system library files (e.g. *Out.*, Reals.** etc.) that are common to all the Cortex-M7 microcontrollers.

The folder *Astrobe\Lib\STM32F7* contains the MCU-specific versions of the library files (e.g. *Main.*, MCU.** etc.) for the STM32F746 and STM32F767 microcontrollers.

The library folders are standard Windows folders containing collections of symbol (**.smb*) and object files (**.arm*).

The list of library folders is stored in a *configuration* file. The commands on the Astrobe *Configuration* menu are used to maintain the configuration files. The name of each folder is stored on a separate line in the configuration's *Library Pathnames* textbox e.g.

C:\AstrobeM7-v6.4\Lib\STM32F7
C:\AstrobeM7-v6.4\Lib\General

or

%astrobe%\Lib\STM32F7
%astrobe%\Lib\General

where %astrobe% is substituted with the location of the library and example files that you specified when you installed or last upgraded Astrobe.

The editor, compiler, linker and builder first search the *<current folder>* when trying to locate imported symbol and object files. They then search each of the library folders in the list. The search continues until the file is found or the last folder in the list has been searched.

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

2.5 Resource Data

The usual way to process constant data in an Oberon 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 Astrobe, 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 Modules* section of this document.

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 Astrobe 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.

2.6 Target Processor Types

The supported target processor types are STMicroelectronics microcontrollers:

- STM32F767, STM32F746

2.7 STM32 Targets - Memory Map

The memory usage of an Oberon program executing on an STM32 device depends on the FLASH ROM / STATIC RAM capacity of the type of target microcontroller.

	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	02000 0200H	INC	End of stack	See below
Mapped Interrupt Vectors	02000 0000H	INC	02000 01FFH	512
User Code	00000 0240H		See below	
System Vars	00000 0200H		00000 023FH	64
Interrupt Vectors and Startup	00000 0000H		00000 01FFH	512

2.8 STM32 Targets - User Code (FLASH)

MCU	Start Address	End Address	Size
STM32F746	0800 0000H	0810 0000H	1 MB
STM32F767	0800 0000H	0820 0000H	2 MB

2.9 STM32 Targets – Global + Heap + Stack Data (CPU SRAM)

MCU	Start Address	End Address	Size
STM32F746	2000 0000H	2005 0000H	320 KB
STM32F767	2000 0000H	2008 0000H	512 KB

2.10 Non-supported Targets

The configuration files can be used to target microcontrollers other than the supported STM32 targets e.g. other STMicroelectronics Cortex-M7 MCUs or MCUs from other manufacturers e.g. NXP. The *Data Range* and *Code Range* entries allow you to specify the Code and Data Flash and RAM address ranges to use when the Astrobe linker produces the binary executable file.

Developers targeting non-supported MCUs will need to develop their own versions of the STM32-specific library modules supplied with Astrobe. If the MCUs are not mbed-enabled they would also need to use 3rd-party tools to upload the executables to their development boards. The latter are usually available from the designers of the MCU.

3 Library Modules

The following library modules are included with all editions of Astrobe for Cortex-M7:

Module name	Description
Clock	Real time clock
Convert	Conversion of integers to / from strings
GPIO	Configure and control General Purpose IO pins
Graphics	Draw lines, circles and ellipses on display devices
I2C	Reading from and writing to the I2C1/2/3 bus in Master mode
In	Formatted ASCII text input
LinkOptions	Values of options supplied by the user at link time
Main	Initialisation code required by an STM32F7 application
Math	Basic mathematical and trigonometrical functions
MAU	Memory allocation unit
Out	Formatted ASCII text output
Put	String-handling helper functions used by *Convert* and *Reals*
Random	Pseudo-random number generator
Reals	Real number support and conversion to / from strings
ResData	Access constant user data attached to the program by the linker
Serial	Basic polled USART serial IO
STM32F7\MCU	STM32F7-family processor and peripheral addresses
SPI	Reading from and writing to the Serial Peripheral Interface bus
Storage	User-definable memory allocation / deallocation procedures
Strings	General string-handling functions
SYSTEM	Implementation-specific low level functions
Timers	Microsecond and millisecond time measurement and delays
Traps	Runtime error trapping

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 built-in functions *NEW* or *DISPOSE* then module *MAU* is called and the *MAU* module is 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.

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 Astrobe 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 the section titled *Runtime Error Codes* below.

3.1 Clock

Definition

```
DEFINITION MODULE Clock;

IMPORT MCU, SYSTEM;

PROCEDURE* Pack*(hh, mm, ss: INTEGER; VAR ctime: INTEGER);

PROCEDURE* Unpack*(ctime: INTEGER; VAR hh, mm, ss: INTEGER);

PROCEDURE* Time*(): INTEGER;

PROCEDURE* GetHMS*(VAR hh, mm, ss: INTEGER);

PROCEDURE* SetHMS*(hh, mm, ss: INTEGER);

PROCEDURE* Hours*(): INTEGER;

PROCEDURE* Minutes*(): INTEGER;

PROCEDURE* Seconds*(): INTEGER;

PROCEDURE* Init*();

END Clock.
```

Description

The module Clock contains functions for accessing the time components of the Real Time Clock (RTC).

Init initialises the RTC hardware and should be called before any other Clock procedures are called.

SetHMS stops the clock, updates the value in the Time register using the hours, minutes and seconds parameters. The clock is then re-started.

Time returns the value read from the Time Register. The individual hours, minutes and seconds fields can be extracted from this value using the *Unpack* function.

GetHMS is equivalent to:

```
Unpack(Time(), hh, mm, ss)
```

Hours, *Minutes* and *Seconds* can be used to conveniently access the individual components of the time if they are not all required.

3.2 Convert

Definition

```
DEFINITION MODULE Convert;

IMPORT Put;

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

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

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

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

END Convert.
```

Description

The module Convert contains functions for converting integers to strings and vice versa.

3.3 GPIO

Definition

```
DEFINITION MODULE GPIO;

TYPE
  PortConfiguration* = RECORD
    mode*: INTEGER;
    outputType*: INTEGER;
    speed*: INTEGER;
    resistors*: INTEGER;
    alternateFunction*: INTEGER
  END;

  Pin* = RECORD
    base*, no*: INTEGER
  END;

PROCEDURE* Map*(base, no: INTEGER; VAR pin: Pin);

PROCEDURE* Configure*(VAR pin: Pin; config: PortConfiguration);

PROCEDURE* Put*(pin: Pin; state: BOOLEAN);

PROCEDURE* Get*(pin: Pin; VAR state: BOOLEAN);

END GPIO.
```

Description

The General Purpose Input/Output (GPIO) module contains functions to configure the hardware characteristics, and control the behaviour, of each of the GPIO pins.

Map associates a variable with a port name and number.

Configure converts the symbolic configuration options in the PortConfiguration record to the numeric values required to set the configuration registers.

Put sets the logic value of the GPIO pin to high (TRUE) or clears it (FALSE).

Get returns the current state of the pin. TRUE is high and FALSE is low.

Example

The following extracts from the Astrobe Blinker example illustrates their use:

```
CONST
  PORTB = MCU.GPIOBBase;

VAR
  LED: GPIO.Pin;
  config: GPIO.PortConfiguration;
  …
  …
  config.mode := GPIO.Mode_Out;
  config.speed := GPIO.Speed_High;
  config.outputType := GPIO.Push_Pull;
  config.resistors := GPIO.Floating;
  config.alternateFunction := GPIO.AF0
  …
  …
  (* Green LED is connected to GPIO port B0 *)
  GPIO.Map(PORTB, 0, LED);
  GPIO.Configure(LED, config);
  …
  …
  (* Switch the LED on *)
  GPIO.Put(LED, TRUE);
```

Refer to the STM Reference manual for your microcontroller for an explanation of the configuration options.

3.4 Graphics

Definition

```
DEFINITION MODULE Graphics;

TYPE
  DrawDotProc* = PROCEDURE (colour, x, y: INTEGER);

PROCEDURE Line*(colour, x0, y0, x1, y1: INTEGER);

PROCEDURE Circle*(colour, x0, y0, r: INTEGER);

PROCEDURE Ellipse*(colour, x0, y0, a, b: INTEGER);

PROCEDURE Init*(maxX, maxY: INTEGER; dd: DrawDotProc);

END Graphics.
```

Description

The module Graphics contains functions to draw lines, circles and ellipses on display devices that allow a dot to be drawn at specified x and y co-ordinates.

Init must be called before any other Graphics function. The *maxX and maxY* parameters specify the largest x and y co-ordinates at which a dot can be drawn. dd is a user-supplied procedure which draws a dot at the x and y co-ordinates e.g.

```
PROCEDURE DrawDot(colour, x, y: INTEGER);
BEGIN
   …
END DrawDot;

Init(132, 132, DrawDot);
```

References

Ref: Project Oberon - Wirth & Gutknecht, ACM Press 1992

3.5 I2C Bus

Definition

```
DEFINITION MODULE I2C;

IMPORT MCU, SYSTEM;

CONST
  I2C1* = 1;
  I2C2* = 2;
  I2C3* = 3;


TYPE
  ConfigurePinsProc* = PROCEDURE;

PROCEDURE WriteBytes*(addr: BYTE; data: ARRAY OF BYTE; offset, count: INTEGER);

PROCEDURE Write*(addr: BYTE; data: ARRAY OF BYTE);

PROCEDURE ReadBytes*(addr: BYTE; VAR data: ARRAY OF BYTE; offset, count: INTEGER);

PROCEDURE Read*(addr: BYTE; VAR data: ARRAY OF BYTE);

PROCEDURE Init*(bus: INTEGER; ConfigurePins: ConfigurePinsProc; freq: INTEGER);

END I2C.
```

Description

The module I2C contains functions to communicate with slave devices that are connected to the microcontrollers I2C1, I2C2 or I2C3 bus. This enables you to write applications which can use a wide variety of external I2C standard parts, such as serial RAMs and EEPROMS, LCDs, and digital sensors such as accelerometers, compasses, temperature and pressure gauges etc.

Init must be called before any other I2C function.

- The *bus* parameter can be *I2C1, I2C2 or I2C3*. It is used to specify which I2C bus subsequent calls will operate on.
- The *ConfigurePins* parameter is a user-supplied procedure which should configure the I2C function on the pins on the target microcontroller that the I2C device is connected to.
- The *freq* parameter is the I2C clock rate. Valid values are either 100000 or 400000.

The *addr* parameter is the 7-bit value that identifies the slave I2C device as specified in its datasheet. The *Read / Write* procedures shift the address value by one to allow for the read / write bit which is automatically set when reading, and cleared when writing.

The *data* parameter is declared as *ARRAY OF BYTE* so the actual parameter used can be of *any* data type. The *Read / Write* calls provide a simplified interface and are implemented with default values for *offset* and *count* as follows:

```
ReadBytes(addr, data, 0, LEN(data))
WriteBytes(addr, data, 0, LEN(data))
```

Use *ReadBytes / WriteBytes* instead of *Read / Write* in special cases when you need to specify how many bytes are transferred e.g. if the data is to be only partially read or written (*offset* # 0).

See the I2C examples included with Astrobe for typical uses, example slave addresses and processor-specific configuration and initialisation code.

3.6 In

Definition

```
DEFINITION MODULE In;

IMPORT Convert;

CONST
  (* Possible result values *)
  noError* = Convert.noError;
  overflow* = Convert.overflow;
  syntaxError* = Convert.syntaxError;

TYPE
  GetCharProc* = PROCEDURE (VAR ch: CHAR);

VAR
  result*: INTEGER;

PROCEDURE Init*(g: GetCharProc);

PROCEDURE Char*(VAR ch: CHAR);

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

PROCEDURE Int*(VAR n: INTEGER);

END In.
```

Description

The module In contains basic text formatting input functions.

Input can be redirected from any device that accepts ASCII character data by calling *Init* with the name of a procedure that inputs a single character to that device. When the system module Main is initialised, *Serial.GetCh* is the procedure passed to *In.Init* and input is directed from the serial port USART3.

In.Char reads the next character from the input device.

In.String skips all characters with a value less than an ASCII space (020X) and then reads all characters until LEN(s) characters have been read or a character with a value less than an ASCII space has been read. A terminating null character is appended to the array if it is not full.

In.Int reads the next string and attempts to convert it into an integer. The global value *result* is set to one of the possible result values depending on the success of the conversion.

3.7 LinkOptions

Definition

```
MODULE LinkOptions;

IMPORT SYSTEM;

VAR
  (* Startup parameters *)
  Fosc*: INTEGER;
  Target*: INTEGER;
  HeapStart*: INTEGER;
  HeapLimit*: INTEGER;
  StackStart*: INTEGER;
  ResourceStart*: INTEGER;

END LinkOptions.
```

Description

The module LinkOptions contains variables which are initialised with data taken from the current configuration when the application was linked. Consequently, this data can be accessed by the application when it is running on the target MCU. Target is always zero for the Cortex-M7 MCUs.

3.8 STM32F7\MCU

Definition

```
DEFINITION MODULE MCU;

IMPORT SYSTEM;

CONST
  (* see below *)

PROCEDURE* Configure*();

PROCEDURE* NVIC_EnableIRQ*(irqNo: INTEGER);

END MCU.
```

Constants are included for the following peripheral and control registers:

System Clocks (CLK)
Reset and Clock Control (RCC)
GPIO Ports (GPIOA-K)
16 & 32-bit Timers (TIM2/3/4/5)
Power Controller (PWR)
Real Time Clock (RTC)
Nested Vectored Interrupt Controller (NVIC)
USART2/3
Serial peripheral interface (SPI1/2/3)
I2C1/2/3

See the source code module for more details e.g.:

```
AstrobeM7\Lib\STM32F7\MCU.mod
```

Description

Defines the constants representing the hardware-specific peripheral addresses and control codes for the STM32F7 MCU.

Configure is called by Main. It initialises the RCC_CFGR clock configuration values used by Main:

HSECLK, SW, HPRE, PPRE1, PPRE2 and MCO2PRE.

NVIC_EnableIRQ is used in the initialisation of modules that implement interrupt handlers. It enables interrupts for the peripheral assigned to the IRQ position *irqNo*. These positions are enumerated in the NVIC chapter of the STM32F76xxx and STM32F77xxx Reference manual (RM0410) e.g. the irqNo value for timer TIM3 is 29.

The source code of *MCU* can be modified to enable customisation of the startup process.

3.9 Main

Definition

```
DEFINITION MODULE Main;

IMPORT SYSTEM, MCU, Traps, In, Out, Serial, Caches, LinkOptions (* must be last in the
import list *);

END Main.
```

Description

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

Input/output is initialised to operate via USART3 by calling *In.Init* and *Out.Init* with procedures *Serial.GetCh* and *Serial.PutCh* respectively.

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 can be modified to enable user-customisation of the startup process.

3.10 Math - Mathematical Functions

Definition

```
DEFINITION MODULE Math;

CONST
  pi*        = 3.14159266;
  twoDivPi*  = 0.63661977;
  fourDivPi* = 1.27323954;
  piDivTwo*  = 1.57079633;
  piDivFour* = 0.78539816;
  ln2* = 0.69314718;

PROCEDURE* Sqrt*(x: REAL): REAL;

PROCEDURE Ln*(x: REAL): REAL;

PROCEDURE Exp*(x: REAL): REAL;

PROCEDURE Sin*(x: REAL): REAL;

PROCEDURE Cos*(x: REAL): REAL;

PROCEDURE ArcTan*(x: REAL): REAL;

END Math.
```

Description

The module 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°).

3.11 MAU - Memory Allocation Unit

Definition

```
DEFINITION MODULE MAU;

IMPORT LinkOptions, SYSTEM;

TYPE
  Proc* = PROCEDURE (VAR p: INTEGER; T: INTEGER);

PROCEDURE New*(VAR p: INTEGER; T: INTEGER);

PROCEDURE Dispose*(VAR p: INTEGER; T: INTEGER);

PROCEDURE SetNew*(p: Proc);

PROCEDURE SetDispose*(p: Proc);

PROCEDURE Allocate*(VAR p: INTEGER; typeDesc: INTEGER);

PROCEDURE Deallocate*(VAR p: INTEGER; typeDesc: INTEGER);

END MAU.
```

Description

The module 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 replaced with a user-defined module.

If a user module calls the Oberon NEW function to allocate dynamic memory to a pointer variable then *MAU.New* is automatically called and the MAU module is automatically imported as if you had added it to your import list. You should not call *MAU.New* directly.

MAU also contains some functions that can be called explicitly by a user module. If so, you must then explicitly include MAU in the module's IMPORT list.

MAU.New calls *Allocate* which assigns the required number of bytes of memory from the heap to the pointer variable.

MAU.Dispose calls *Deallocate* which can potentially be used to return dynamic memory that is no longer needed to the heap.

The standard versions of *Allocate* and *Deallocate* only make the memory available for later reuse if the block being deallocated is the most recent block to be allocated.

The standard versions of *Allocate* and *Deallocate* are included in the *Storage* library module so that you can modify them. *SetNew* can be used to replace the standard version of *Allocat*e, and *SetDispose* can be used to replace the standard version of *Deallocat*e with ones that you have written. See the documentation of the module Storage for more information.

3.12 Out

Definition

```
DEFINITION MODULE Out;

IMPORT Convert;

TYPE
  PutCharProc* = PROCEDURE(ch: CHAR);

PROCEDURE Init*(p: PutCharProc);

PROCEDURE Char*(ch: CHAR);

PROCEDURE String*(s: ARRAY  OF CHAR);

PROCEDURE Ln*();

PROCEDURE Int*(n, width: INTEGER);

PROCEDURE Hex*(n, width: INTEGER);

END Out.
```

Description

The module 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* and subsequent output is directed to the serial port UART0.

Procedure *Out.Ln* writes the carriage return / line feed pair of characters (*0DX, 0AX*)

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

3.13 Put

Definition

```
DEFINITION MODULE Put;

PROCEDURE* Init*();

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

PROCEDURE* Ch*(ch: CHAR);

PROCEDURE* Str*(s: ARRAY OF CHAR);

PROCEDURE Int*(n: INTEGER);

END Put.
```

Description

The module Put contains string-handling helper functions used internally by the modules Convert and Reals.

3.14 Random

Definition

```
DEFINITION MODULE Random;

PROCEDURE Next*(range: INTEGER): INTEGER;

PROCEDURE* Seed*(value: INTEGER);

END Random.
```

Description

The module 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.

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

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

3.15 Reals

Definition

```
DEFINITION MODULE Reals;

IMPORT Put;

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

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

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

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

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

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

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

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

END Reals.
```

Description

The module Reals contains functions to support operations using real numbers.

Exponent(r) returns the value of *exp,* and *Mantissa(r)* returns the normalised value of *r,* resulting from a call to the Oberon 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

3.16 ResData

Definition

```
MODULE ResData;

IMPORT LinkOptions, SYSTEM;

TYPE
  Resource* = POINTER TO RECORD
  END;

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

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

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

PROCEDURE* GetByte*(r: Resource; index: INTEGER; VAR data: BYTE);

PROCEDURE* GetChar*(r: Resource; index: INTEGER; VAR ch: CHAR);

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

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

PROCEDURE* GetRealArray*(r: Resource; 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; name: ARRAY OF CHAR);

END ResData.
```

Description

The module ResData contains functions to access constant resource data items (e.g. fonts, bitmaps, input data etc.) that were appended to the executable by the Astrobe linker. See the section titled *Resource Data* in this document 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.

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*.

```
PROCEDURE GetData();
VAR
  r: ResData.Resource;
  count: INTEGER;
  weights: ARRAY OF REAL;
BEGIN
  ResData.Open(r, "MyData");
  IF ResData.Size(r) = 0 THEN
    (* Report error *)
  ELSE
    ResData.GetInt(r, 0, count);
    NEW(weights, count);
    ResData.GetRealArray(r, 1, count, weights)
    ...
```

3.17 Serial

Definition

```
DEFINITION MODULE Serial;

IMPORT MCU, SYSTEM;

CONST
  USART2* = 2;
  USART3* = 3;

PROCEDURE* TxReady*(): BOOLEAN;

PROCEDURE* PutCh*(ch: CHAR);

PROCEDURE* RxReady*(): BOOLEAN;

PROCEDURE* GetCh*(VAR ch: CHAR);

PROCEDURE Init*(usartNo, baudRate: INTEGER);

END Serial.
```

Description

The module Serial contains functions for sending and receiving single ASCII characters via a USART serial port using polling.

Init is automatically called by *Main* to initialise USART3 with the baud rate set to 38,400 baud. The format is fixed at 8 bits, 1 stop bit, no parity.

The USART is selected by passing USART2 or USART3 as the usartNo parameter.

GetCh waits until a character is present in the UART receive buffer before returning it as a parameter. It should be used if a response is required before the program can proceed. Otherwise *RxReady* can be used to test if a character is available to be read before optionally calling *GetCh*. If *RxReady* returns FALSE other processing can be performed instead.

Similarly, *PutCh* waits until the UART transmit buffer is empty before sending the character. *TxReady* returns TRUE if the transmit buffer is empty and can be called before optionally calling *PutCh*. If *TxReady* returns FALSE other processing can be performed instead.

3.18 SPI (Serial Peripheral Interface)

Definition

```
DEFINITION MODULE SPI;

IMPORT GPIO, MCU, SYSTEM;

CONST
  SPI1* = 0;
  SPI3* = 1;

  CLKDIV2*   = 0;
  CLKDIV4*   = 1;
  CLKDIV8*   = 2;
  CLKDIV16*  = 3;
  CLKDIV32*  = 4;
  CLKDIV64*  = 5;
  CLKDIV128* = 6;
  CLKDIV256* = 7;

TYPE
  ConfigurePinsProc* = PROCEDURE;

PROCEDURE* Response*(): BYTE;

PROCEDURE* SendByte*(data: BYTE);

PROCEDURE* SendChar*(ch: CHAR);

PROCEDURE* SendData*(data: INTEGER);

PROCEDURE* Send*(buf: ARRAY OF BYTE);

PROCEDURE* ReceiveByte*(): BYTE;

PROCEDURE* ReceiveChar*(): CHAR;

PROCEDURE* ReceiveData*(): INTEGER;

PROCEDURE* Receive*(VAR buf: ARRAY OF BYTE);

PROCEDURE Init*(bus, clkDiv, nBits: INTEGER; ConfigurePins: ConfigurePinsProc);

END SPI.
```

Description

Functions used to communicate with slave devices that are connected to the microcontroller's SPI1 or SPI3 bus. This enables you to write applications which can use a wide variety of external SPI standard parts, such as LCD displays, SD cards, 7-segment LEDs, and digital sensors such as accelerometers, magnetometers, etc.

Init must be called before any other SPI function.

- The *bus* parameter must be SPI1 or SPI3. It is used to specify which SPI bus subsequent calls will operate on.
- The *clkDiv* parameter is the PCLK divisor which is used for the SPI baud rate control.
- The *nBits* parameter is the frame length (from 4 to 16) used for data transfers e.g. 8 when transferring bytes.
- The *ConfigurePins* parameter is a user-supplied procedure which should configure the SPI function on the pins on the target microcontroller that the SPI device is connected to.

See the SPI examples included with Astrobe for typical uses, processor-specific configuration and initialisation code.

The *buf* parameters are declared as *ARRAY OF BYTE* so the actual parameters used can be of *any* data type.

3.19 Storage

Definition

```
DEFINITION MODULE Storage;

IMPORT LinkOptions, MAU, SYSTEM;

PROCEDURE Allocate*(VAR p: INTEGER; typeDesc: INTEGER);

PROCEDURE Deallocate*(VAR p: INTEGER; typeDesc: INTEGER);

PROCEDURE* HeapStart*(): INTEGER;

PROCEDURE* HeapPtr*(): INTEGER;

PROCEDURE* HeapUsed*(): INTEGER;

PROCEDURE* HeapAvailable*(): INTEGER;

PROCEDURE* StackStart*(): INTEGER;

PROCEDURE* StackPtr*(): INTEGER;

PROCEDURE* StackUsed*(): INTEGER;

PROCEDURE* StackAvailable*(): INTEGER;

END Storage.
```

Description

The module Storage contains a copy of the default dynamic memory allocation / deallocation procedures from the library module MAU that are invoked by the standard functions NEW / DISPOSE.

You can modify the source code of these functions to tailor their performance and efficiency to suit the requirements of your particular projects. There are additional benefits associated with being able to substitute your own functions e.g. you could add memory usage tracing features.

When the module Storage is initialised it calls

```
MAU.SetNew(Allocate);
MAU.SetDispose(Deallocate);
```

which replace MAU.Allocate and MAU.Deallocate with the functions Storage.Allocate and Storage.Deallocate respectively.

Storage also contains stack- and heap-monitoring procedures.

HeapStart returns the address of the start of the heap.

HeapPtr returns the address of the start of the block of memory that will be used for the next allocation from the heap.

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

HeapAvailable returns the number of bytes of RAM currently free to be used for additional dynamic variables. By default this value is shared with the stack.

StackStart returns the address of the start of the stack.

StackPtr returns the address of the start of the block of memory that will be used for the next allocation from the stack.

StackUsed returns the number of bytes or RAM currently allocated to the stack.

StackAvailable returns the number of bytes of RAM currently free to be used for additional storage. By default this value is shared with the heap.

3.20 Strings

Definition

```
DEFINITION MODULE Strings;

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

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

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

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

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

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

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

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

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

END Strings.
```

Description

Contains general functions for processing strings i.e. string constants and arrays of characters terminated with the null character *0X*.

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

Extract(src, pos, n, dest) extracts a substring *dest* with n characters from position *pos (0 <= pos < Length(src))* in *src*.

Copy(src, dest) copies *Length(src)* characters to *dest*.

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 string pattern *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 <= pos < Length(s)).*

Insert(src, pos, dest) inserts the string *src* into the string *dest* at position pos (0 <= pos <= Length(dst)). If *pos = Length(dest), src* is appended to *dest*.

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.

3.21 SYSTEM

Definition

```
DEFINITION MODULE SYSTEM;

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

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

PROCEDURE BFI*(VAR x: INTEGER; msb, lsb, y: INTEGER);

PROCEDURE BFI*(VAR x: INTEGER; bitNo, y: INTEGER);

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

PROCEDURE CLZ*(data: INTEGER): INTEGER;

PROCEDURE COPY*(src, dest, nWords: INTEGER);

PROCEDURE EMIT*(instruction: INTEGER);

PROCEDURE EOR*(x, y: INTEGER): INTEGER;

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

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

PROCEDURE LDREG*(Rn, v: INTEGER);

PROCEDURE NULL*(x: <numeric type>): BOOLEAN;

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

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

PROCEDURE RBIT*(data: INTEGER): INTEGER;

PROCEDURE REG*(Rn: INTEGER): INTEGER;

PROCEDURE REV*(data: INTEGER): INTEGER;

PROCEDURE REV16*(data: INTEGER): INTEGER;

PROCEDURE REVSH*(data: INTEGER): INTEGER;

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

PROCEDURE UBFX*(VAR x: INTEGER; y, msb, lsb: INTEGER);

PROCEDURE UBFX*(VAR x: INTEGER; y, bitNo: INTEGER);

PROCEDURE VAL*(typeName: <any basic type>; x: <any basic TYPE OR pointer>): typeName;

END 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 implementation dependent and possibly non-portable.

ADR returns the absolute address of the given variable.

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

COPY copies *nWords* consecutive words from *src* to *dest*, where *src*, *dest* and *nWords* are all INTEGERs. *src* and *dest* are absolute memory addresses and can be obtained from variable names using the SYSTEM.ADR function.

GET loads the value of the word or byte at absolute memory location *address* into variable *v*. The size of the variable *v* determines whether a *load register byte* (LDRB) or *word* (LDR) instruction is used to perform the transfer.

PUT stores the value of *x* into the word or byte at absolute memory location *address*. The size of *x* determines whether a *store register byte* (STRB) or *word* (STR) instruction is used to perform the transfer.

An optional integer constant parameter *inc* can be used with *GET* and *PUT* to automatically increment / decrement the value of *address*. If *inc* is in the range 1..255 the address is incremented after the value is stored. If *inc* is in the range -1..-255 the address is decremented before the value is stored. Typical values for *inc* are 1 for byte accesses and 4 for word accesses.

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

General Astrobe Extensions:

Extensions to the standard Oberon SYSTEM features are provided to give low-level access to MCU features that might otherwise require the use of assembly language.

PC, LNK, SP and *FP* are the current values of the program counter, link register, stack pointer and frame pointer i.e. the registers *r15..r12*.

EMIT inserts the 32-bit value of any ARM Thumb-2 instruction at the current code location. Examples of its use can be seen in the source code of the *Traps* and *IAP* library modules.

EOR (x, y) performs a bitwise Exclusive OR of the two integer values.

LDREG (Rn, v) stores the value v in CPU register Rn where Rn is a constant in the range 0..15.

NULL returns true if x is negative or positive INTEGER or REAL zero.

Astrobe for Cortex-M7 Extensions:

BFI updates a bitfield, i.e. just a portion of 32-bit word, with an INTEGER value. *x* is the target variable and *y* is the source data. *msb* is the most-significant bit and *lsb* is the least-significant bit of the bitfield. If msb = lsb (i.e. only a single-bit is accessed) then the two parameters can be replaced by the single *bitNo.*

Examples of its use can be seen in the realtime clock (RTC) library module e.g. the following statement updates just the *minutes* value stored in *time*; the *hours* and *seconds* values remain unchanged:

```
SYSTEM.BFI(time, 11, 8, mm);      (* time:11:8 := minutes units *)
SYSTEM.BFI(time, 14, 12, mmt);    (* time:14:12 := minutes tens *)
```

where *minutes = mmt * 10 + mm.*

CLZ counts the number of leading zeroes in an integer value.

RBIT, REV, REV16 and REVSH are bit and byte-ordering functions. Each generates the single Thumb-2 instruction with the same name:

- *RBIT* reverses the bit order of the integer value.
- *REV* reverses the byte order of the integer value.
- *REV16* reverses the byte order in each 16-bit halfword of the integer value.
- *REVSH* reverses the byte order in the lower 16-bit halfword of the integer value and sign extends the result to 32 bits.

REG (Rn) returns the current value of CPU register Rn where Rn is a constant in the range 0..15. e.g. SYSTEM.REG(15) is equivalent to SYSTEM.PC. Use a local variable in a non-leaf procedure to store the returned result to avoid this register value being overwritten when it is assigned. Refer to the Disassembler listing to check that the code generated is what you expected.

UBFX extracts a *bitfield*, i.e. just a portion of 32-bit word, from an INTEGER value. *x* is the target variable and *y* is the source data. *msb* is the most-significant bit and *lsb* is the least-significant bit of the bitfield. If msb = lsb (i.e. only a single-bit is accessed) then the two parameters can be replaced by the single *bitNo.*

Examples of its use can be seen in the realtime clock (RTC) library module e.g. the following statement extracts just the minutes values from time:

```
SYSTEM.UBFX(mm, time, 11, 8);      (* mm := time:11:8 *)
SYSTEM.UBFX(mmt, time, 14, 12);    (* mmt := time:14:12 *)
```

where *minutes = mmt * 10 + mm.*

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.

3.22 Timers

Definition

```
DEFINITION MODULE Timer;

IMPORT MCU, SYSTEM;

CONST
  uSecs* = 1000000;
  MSecs* = 1000;

  TIM2* = 2;
  TIM3* = 3;
  TIM4* = 4;
  TIM5* = 5;

TYPE
  Timer* = RECORD END;

PROCEDURE* Delay*(timer: Timer; delay: INTEGER);

PROCEDURE* Init*(VAR timer: Timer; timerNo, units: INTEGER);

PROCEDURE* Start*(timer: Timer);

PROCEDURE* Stop*(timer: Timer);

PROCEDURE* Elapsed*(timer: Timer): INTEGER;

END Timer.
```

Description

The module Timers contains functions for timed delays and for measuring elapsed execution time. Several different timers can be used in the same application. The *timer* parameter determines which one is used.

Init must be called before the first call to each different timer. *timerNo* is used to associate one of the timers (e.g. *TIM4*) to the *timer* variable*. units* can be either *Timers.MSecs* or *Timers.uSec.* However, microsecond *units (uSecs)* can only be used for the 32-bit timers e.g. TIM2*.* The units specified in *Init* are used for that timer for all subsequent calls. If it is a 32-bit timer the units can be changed by calling *Init* again.

Examples are:

```
Init(usecTimer, Timers.TIM2, Timers.uSecs);
Init(msecTimer, Timers.TIM3, Timers.MSecs);
```

Consequently, both of these calls would both result in a 1-second delay：

```
Delay(usecTimer, 1000000);
Delay(msecTimer, 1000);
```

The delay functions reset the timer and then execute a continuous loop until the measured elapsed time exceeds the given *delay*.

Elapsed returns the time that passed between the two most recent calls of *Start* and *Stop*.

Because the delay functions reset the timer they should not be called in sections of code that are being timed by the same timer.

NOTE: The timer functions use the prescale register facility of the STM32F7 Timers. The prescale value is calculated using the formula:

```
(2 * MCU.PCLK + (divisor DIV 2)) DIV divisor
```

where divisor is 1000 for millisecond times and 1000000 for microsecond times. e.g. the microsecond delay is effected by setting the uSec prescale register to

```
(uSecPrescale * delay) - 1
```

3.23 Traps

Definition

```
DEFINITION MODULE Traps;

TYPE
  InterruptHandler* = PROCEDURE();

PROCEDURE* ShowRegs*(b: BOOLEAN);

PROCEDURE* Assign*(addr: INTEGER; p: InterruptHandler);

PROCEDURE Init*;

END Traps.
```

Description

The module Traps Implements default interrupt handlers for software interrupts and the standard internal Cortex-M7 exceptions: *NMI, Hard Fault, Memory Manager, Bus Fault* and *Usage Fault*. The default handlers are installed when *Traps.Init* is called from the *Main.Init* function at startup time.

The supervisor call (*SVC*) instruction handler is invoked whenever Astrobe 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.

Traps.Assign is used to assign an interrupt handler to the related interrupt vector.

Traps.ShowRegs determines whether or not the values of the registers R0 to R12 are displayed when the trap handler is executed.

See the *Interrupt Handlers* section in the ARM Cortex-M Oberon Programmers Guide (included with Astrobe) and the *Runtime Errors* section below for more details.

The source code of *Traps* can be modified to enable user-customisation of the interrupt handling process.

4 Debugging

4.1 Runtime Error Codes

The error codes assigned to runtime errors and assertions detected by Oberon are:

Code	Reason
1	Index out of bounds
2	Type test failure
3	Source and destination arrays are not the same length
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	Attempt to dispose a NIL pointer
13..19	Reserved
20..99	Library assertions
100..255	User-defined assertions

4.2 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
Graphics	DrawDot	20	DrawDot is defined
In	UndefinedGetCh	20	In.Init has already been called
I2C	Init	32	bus IN {I2C1, I2C2, I2C3}
I2C	Init	33	(freq = 100000) OR (freq = 400000)
Math	Sqrt	20	x >= 0.0
Math	Ln	22	x > 0.0
Out	PutCh	20	Out.Init has already been called
Put	EOS	20	pos < LEN(s)
Put	Ch	21	pos < LEN(buffer) - 1
Put	Str	22	pos + Length(s) < LEN(buffer)
Put	Int	23	(pos + nDigits) < LEN(buffer)
Put	Int	24	pos < LEN(buffer)
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	RealToStrF	24	digits IN {0..7}
ResData	Open	20	ID = "OB7R"
ResData	Open	21	Version number
ResData	GetInt	22	index <= r.nItems
ResData	GetByte	23	index <= r.nBytes
ResData	GetRealArray	29	count <= LEN(items)
SPI	Init	20	nBits in {3..15}
SPI	Init	21	bus IN {1, 3}
SPI	Init	22	clkDiv IN {CLKDIV2 .. CLKDIV256}
Storage	Allocate	11	Heap does not overflow
Storage	Deallocate	12	Pointer is not NIL

Module	Procedure	Code	Assertion
Strings	Extract	20	n > 0
Strings	Extract	21	n < LEN(dest)
Strings	Extract	22	pos >= 0
Strings	Extract	23	pos + n <= Length(src)
Strings	Append	24	Length(src) + Length(dest) <= LEN(dest)
Strings	Pos	25	Length(pattern) > 0
Strings	Pos	26	(pos + Length(pattern)) <= Length(src)
Strings	Delete	27	pos >= 0
Strings	Delete	28	n > 0
Strings	Delete	29	Length(s) - pos - n >= 0
Strings	Insert	30	Length(dest) - pos >= 0
Strings	Insert	31	pos >= 0
Strings	Insert	32	(Length(src) + Length(dest)) < LEN(dest)
Strings	Copy	33	Length(src) <= Length(dest)
Strings	Copy	34	src[i] = 0X where i < LEN(src)
Strings	Length	35	s[i] = 0X where i < LEN(s)
Strings	Cap	36	s[i] = 0X where i < LEN(s)
Timers	Init	20	Units = MSecs or Timer is TIM2 or TIM5
Timers	Delay	21	delay >= 0
Timers	Delay	22	Units = uSecs or delay is a 16-bit value
Traps	IntArrayToChars	20	LEN(chars) >= LEN(array)

4.3 Programmer-defined Assertions

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

```
ASSERT(Storage.StackAvailable < minRequired, 130)
```

where `minRequired` is a user-defined value.

4.4 Reporting Runtime Errors

The above runtime, library and programmer-defined error conditions and assertions result in the execution of a Cortex-M7 supervisor call instruction (SVC) which calls a default trap handler in the Astrobe 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 values of registers R0 to R12 at the time of the runtime error or assertion failure

If the procedure call `Traps.ShowRegs(FALSE)` is made before the trap occurs the display of register values is suppressed. This is useful if the display only has a few lines and cannot show all of the information without scrolling.

The information is reported using the standard IO functions exported by the Astrobe *Out* module. By default the messages will appear on a serial terminal connected to UART3. The trap handler then processes an infinite loop until the system is reset.

The source code of *Traps* can be modified to allow user-customisation of the trap-handling process.

Professional Edition: When debugging your program, you can use the register values in conjunction with the assembly listing of the module to identify the values of variables at the time of failure.

4.5 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.

- 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.

4.6 Diagnosing System Exceptions

Traps caused by runtime errors or assertion failures which result in Supervisor Calls (SVC) are easy to locate as they give you the module name and line number of the offending line of source code. Hardware-related and other system exceptions are more difficult to locate as they only give you the module name and the address of the instruction that failed. Fortunately they are much rarer than runtime errors.

The following types of Cortex-M7 hardware system exceptions are handled by the Astrobe *Traps* module:

- NMI
- Hard Fault
- Memory Manager
- Bus Fault
- Usage Fault

Refer to the *ARM v7-M Architecture Reference Manual* which can be downloaded from the ARM website for information on the possible causes of these exceptions.

If the exception is not caused by a secondary effect and you have the Professional Edition of Astrobe it is often possible to identify the line of code in your application which generated the offending instruction. To do this you need to have:

- The runtime error message displayed when your application terminated. This will give you the module name and exception address.
- The map file for the main module (*<ModuleName>.map*) which was created when you linked / built the application. The start address of the module is listed in the *Code Address* column of the map file.
- A disassembler listing (*Project > Disassemble*) of the problem module.

You can then calculate the offset and find the corresponding line of code in the disassembly listing using the following formula:

offset = exception address – start address - 8

The 8 is subtracted because the Cortex-M7 program counter is 8 bytes ahead of the current instruction. If you look in the disassembler listing for the instruction with the same offset you will see the accompanying Oberon source line which generated that instruction.

5 Compile, Link and Build Commands

The Professional Edition of Astrobe includes separate command-line programs for the Oberon Cortex-M7 Compiler, Builder and Linker which correspond to the built-in compile, build and link commands in the IDE.

The separate compiler, builder 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 or upgrading to a newer version of Astrobe.

All of the commands have two required parameters.

```
AstrobeCompile <configfile>.ini [<path>]<ModuleName>.mod

AstrobeBuild <configfile>.ini [<path>]<MainModuleName>.mod

AstrobeLink <configfile>.ini [<path>]<MainModuleName>.mod
```

MainModuleName is the filename of the main module being compiled or linked.

configfile is the name of the configuration file containing the options to use.

5.1 Examples

```
AstrobeCompile  D:\AstrobeM7-v6.4\Configs\STM32F767.ini  Lists.mod

AstrobeBuild  D:\AstrobeM7-v6.4\Configs\STM32F767.ini  Blinker.mod

AstrobeLink  D:\AstrobeM7-v6.4\Configs\STM32F767.ini  Blinker.mod
```

5.2 Command Return Codes

If the command executes without any compiler or linker errors it returns zero otherwise it returns 1. Examples of DOS batch scripts which use these return values are:

```
REM
REM Rebuild General Library
REM
SET astrobe=D:\AstrobeM7-v6.4
SET cfg=%astrobe%\configs\STM32F767.ini
SET compile="C:\Program Files (x86)\AstrobeM7 Professional Edition\AstrobeCompile.exe"
REM
cd %astrobe%\Lib\General
del *.arm
del *.smb
%compile% %cfg% Math.Mod
if errorlevel 1 goto ErrorExit
%compile% %cfg% Random.Mod
if errorlevel 1 goto ErrorExit
%compile% %cfg% Put.Mod
if errorlevel 1 goto ErrorExit
%compile% %cfg% Convert.Mod
if errorlevel 1 goto ErrorExit
%compile% %cfg% In.Mod
if errorlevel 1 goto ErrorExit
%compile% %cfg% Out.Mod
if errorlevel 1 goto ErrorExit
%compile% %cfg% LinkOptions.Mod
if errorlevel 1 goto ErrorExit
%compile% %cfg% ResData.Mod
if errorlevel 1 goto ErrorExit
%compile% %cfg% Traps.Mod
if errorlevel 1 goto ErrorExit
%compile% %cfg% Reals.Mod
if errorlevel 1 goto ErrorExit
%compile% %cfg% Strings.Mod
if errorlevel 1 goto ErrorExit
echo No errors detected
goto OK
:ErrorExit
echo Errors detected
:OK
pause
```

```
REM
REM Rebuild General Library
REM
SET astrobe=D:\AstrobeM7-v6.4
SET cfg=%astrobe%\configs\STM32F767.ini
SET build="C:\Program Files (x86)\AstrobeM7 Professional Edition\AstrobeBuild.exe"
REM
cd %astrobe%\Lib\General
del *.arm
del *.smb
%build% %cfg% Build.Mod
if errorlevel 1 goto ErrorExit
echo No errors detected
goto OK
:ErrorExit
echo Errors detected
:OK
pause
```