CFB Software

Astrobe

Oberon for Arm Microcontrollers

Astrobe
Oberon for Arm Microcontrollers

Table of Contents

1 Introduction

Astrobe is a fast and responsive integrated development environment for Windows. It is used to write software to run on Arm Cortex-M0, M3, M4 and M7 microcontrollers and Raspberry Pi RP2040 (Cortex-M0+) and RP2350 (Cortex-M33) microcontrollers. In the following when we refer to Astrobe it applies to all of these versions.

Refer to the Astrobe website at https://www.astrobe.com/ for the latest information on the availability of the different versions of Astrobe.

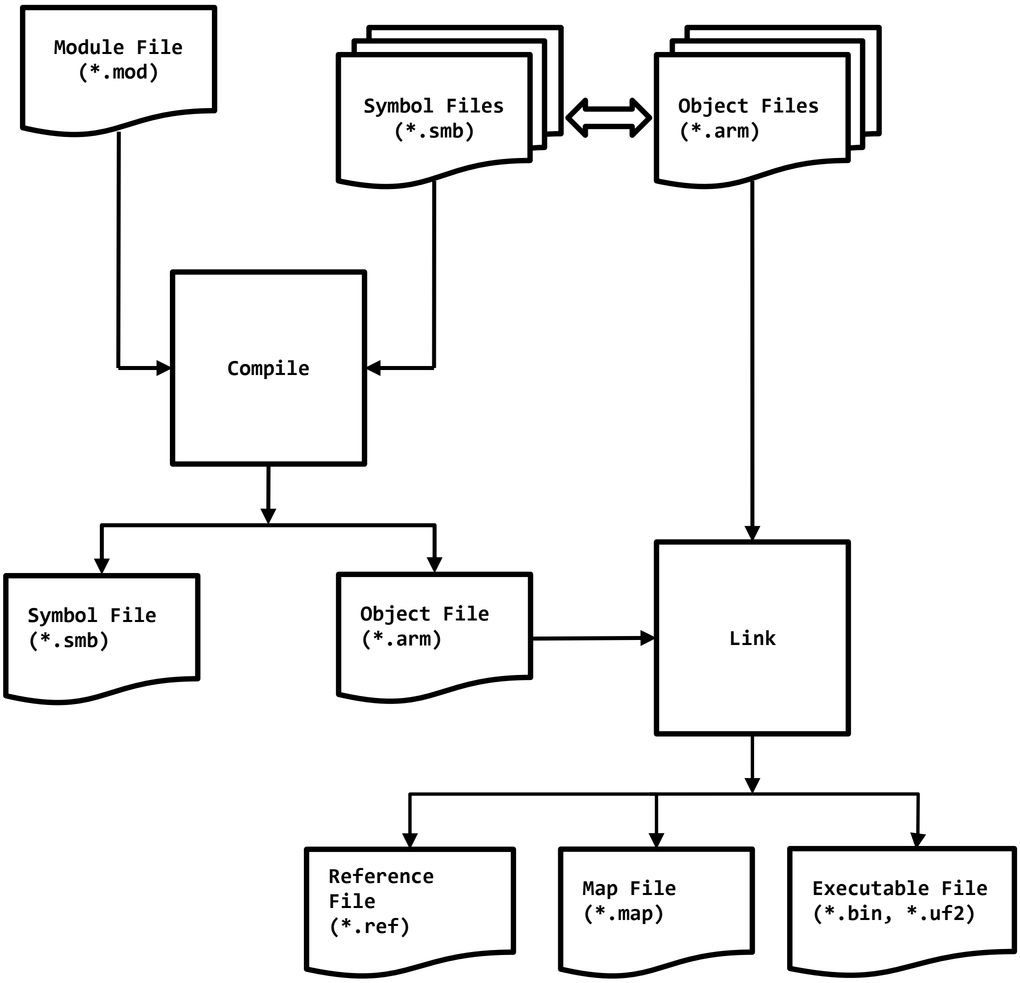
2 File Descriptions

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:

Ext	Type	Created by	Used by	Scope	Description
.arm	Binary	Compile	Link	Module	Linkable object file
.asm	Text	Disassemble		Application	Disassembler listing
.bin	Binary	Link	Upload M0, M3, M4, M7	Application	Linked binary executable file
.def	Text		Edit	Module	SYSTEM interface
.drf	Binary	Link	Disassemble Application	Application	Reference information
.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	Traps	Application	Trap reference resource data
.res	Any		Link	Module	Resource data
.s	Text	Disassemble		Application	Assembler source
.smb	Binary	Compile	Compile	Module	Symbol file of exported items
.uf2	Binary	Link	Upload RP2040, RP2350	Application	Linked UF2-format executable file



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* conforms to 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 the executable file *DigiClock.bin.*

When *DigiClock* is uploaded the flash memory of the target processor is programmed with the contents of the executable file.

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.

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 modules
- 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 the code and data address ranges and the physical locations of the library files are as specified for 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 are configured in the process of initialising the Astrobe library module *Main*. The module *Main* must be included in the IMPORT list of the main module of every Astrobe application to ensure that the application is correctly initialised.

You can modify the source code of the *Main*, *MCU* and *Traps* modules 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 library folders are standard Windows folders containing collections of source (**.mod),* symbol (**.smb*) and object files (**.arm*).

The folder *Lib\General* contains generic system library files that are common to all microcontrollers targeted by Astrobe e.g. *Out.*, Reals.** etc.

The remaining folders in *Lib* contain microcontroller-specific versions of the library files e.g. *Main.*, MCU.** etc.

2.5 Configuration Files

The Compile, Link, Build and Upload options for the target microcontrollers are stored in *Configuration* (**.ini*) files. Examples of these are included with Astrobe for the target microcontrollers used on the supported development boards.

The commands on the Astrobe *Configuration* menu are used to maintain and access the configuration files. See the *Configuration Files* section of the Astrobe Help file for more information.

Configuration entries include the locations of the library folders and the code and data address ranges to be used when linking.

2.5.1 Library Folders

The list of library folders to be searched is stored in the configuration file. The name of each library folder is stored on a separate line in the configuration's *Library Pathnames* textbox. Examples are:

> *D:\AstrobeM0-v10.0\Lib\STM32F091*
> *D:\AstrobeM0-v10.0\Lib\General*
>
> *%AstrobeM4%\Lib\STM32F303ZE*
> *%AstrobeM4%\Lib\General*
>
> Where, for example, *%AstrobeM4%* is substituted with the location of the library and example files that you specified when you installed or last upgraded Astrobe for Cortex-M4.

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.

2.5.2 Data Addresses

The configuration files have entries, *Data Range* and *Code Range* to 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 other MCUs can create new configuration files and develop their own hardware-specific library modules using the files and source code supplied with Astrobe as examples.

2.6 Uploading Executable Files

Development boards supported with Astrobe allow executable files (**.bin / *.uf2*) which were created by the Astrobe *Link* or *Build* commands to be uploaded via a USB connection from the PC to the development board. This is done using the Astrobe *Upload* command.

2.7 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.

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.

Study the source code of the *Traps* library module for an example of how to use resource files.

3 Library Modules

The following library modules are included with Astrobe:

Module name	Description
Bits	Bitwise operations on integers
Convert	Conversion of integers to / from strings
DateTime	Date and time string conversions
Error	Error messages referenced by Traps
FPU (M0, M3, RP2040)	Support of mathematical operations on floating point numbers
GPIO	General Purpose IO pin configuration and control
Graphics	Device-independent drawing of lines, circles and ellipses
I2C	Reading from and writing to the I2C bus in Master mode
In	Formatted ASCII text input
LinkOptions	Values of options supplied by the user at link time
MAU	Memory allocation unit
MCU	Microcontroller-specific definitions and peripheral addresses
Main	Initialisation code required by an application
Math	Basic mathematical and trigonometric functions
Out	Formatted ASCII text output
Put	String-handling helper functions used by *Convert*, *Reals* etc.
RTC	Real-time Clock time and date
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
SPI	Reading from and writing to the Serial Peripheral Interface bus
SYSTEM	Implementation-specific low level functions
Serial	Basic polled UART serial IO
Storage	User-definable memory allocation / deallocation procedures
Strings	General string-handling functions
Timers	Microsecond and millisecond time measurement and delays
Traps	Runtime error trapping

3.1 Special Library Modules

The modules *FPU, MAU* and *SYSTEM* are special i.e. they are dependent on the version of the compiler and must follow some specific conventions.

3.1.1 FPU – Floating-point Unit

FPU is only needed for Astrobe for Cortex-M0, RP2040 and Cortex-M3. If a user module uses mathematical operations (e.g. divide, multiply etc.) on variables that are declared as REALs then an *FPU* function is called and the *FPU* module is automatically imported. It should not be replaced with a user-defined module and its interface definition must not be changed. User modules should not explicitly call an *FPU* function.

3.1.2 MAU - Memory Allocation Unit

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 and its interface definition must not be changed.

If a user module calls the Oberon NEW function to allocate dynamic memory to a pointer variable then *MAU.Ne*w 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.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 source code of *Allocate* and *Deallocate* is included in the Storage library module so that you can modify them if you need to. SetNew can be used to replace the standard version of Allocate, and SetDispose can be used to replace the standard version of Deallocate with ones that you have written.

3.1.3 SYSTEM

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.

The procedure declarations and comments describing each function are included in the definition file *SYSTEM.def* which is located in the *Lib\General* folder.

3.2 General Library Modules

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.

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	Attempt to call a NIL procedure variable
6	String too long or destination string too short
7	Integer division by zero or negative divisor
8, 9, 10	FPU assertions
11	Reserved
12	Attempt to dispose a NIL pointer
13..19	Reserved
20..25	Library assertions – see the Error module for definitions
26..99	Reserved
100..199	User-defined assertions
200..255	User-defined assertions with customisable trap handlers

4.2 User-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.

User-defined assertions should use error codes in the range 100 – 255 to distinguish them from Runtime and Library errors.

Error codes 100 – 199 will display error information in the same way as the Library errors.

Error codes 200 – 255 can be used if you want to handle the error in a different way. An example application, called *UserTraps*, is supplied with Astrobe to demonstrate how this can be done.

4.3 Reporting Runtime Errors

The above runtime, library and programmer-defined error conditions and assertions result in the execution of an Arm supervisor call instruction (SVC) which calls a default trap handler in the Astrobe library module *Traps*.

The trap handler reports:

- an error code or message describing what type of error it is
- the name of the module and procedure that was being executed
- the address of the instruction which caused the error
- the line number of the corresponding statement in the source code
- the values of the registers which are automatically saved at the time of the runtime error or assertion failure

If the *Stack Trace* option on the Astrobe Configuration dialog was enabled when the module was compiled, the details of the sequence of procedure calls that led to the error are included:

```
TestTraps
integer divided by zero or negative divisor
TestTraps.DivByZero @0800250CH, Line: 22
TestTraps.Run       @08002556H, Line: 28
TestTraps..init     @08002562H, Line: 32
  r0 = 0000000BH,         11
  r1 = 00000000H,          0
  r2 = 00000341H,        833
  r3 = 00003200H,      12800
 r12 = FFFFFFFFH,         -1
  lr = 0800255BH,  134227291
  pc = 0800250EH,  134227214
 psr = 61000000H, 1627389952
```

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

The error messages that are displayed are defined in the module *Error*. If there is no message corresponding to the error code, the error code is displayed instead.
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 the UART device defined in the *Main* module. The trap handler then processes an infinite loop until the system is reset.

You can modify the source code of *Traps* to customise the trap-handling process.

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

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

4.5 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 type of hardware system exceptions handled by the Astrobe *Traps* module can include the following:

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

Refer to the relevant *Armv6-M (M0, RP2040, M3), ARM v7-M (M4, M7) or Armv8-M (RP2350) Architecture Reference Manual* which can be downloaded from the Arm website, for details of which exceptions may occur and the possible causes of these exceptions.

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

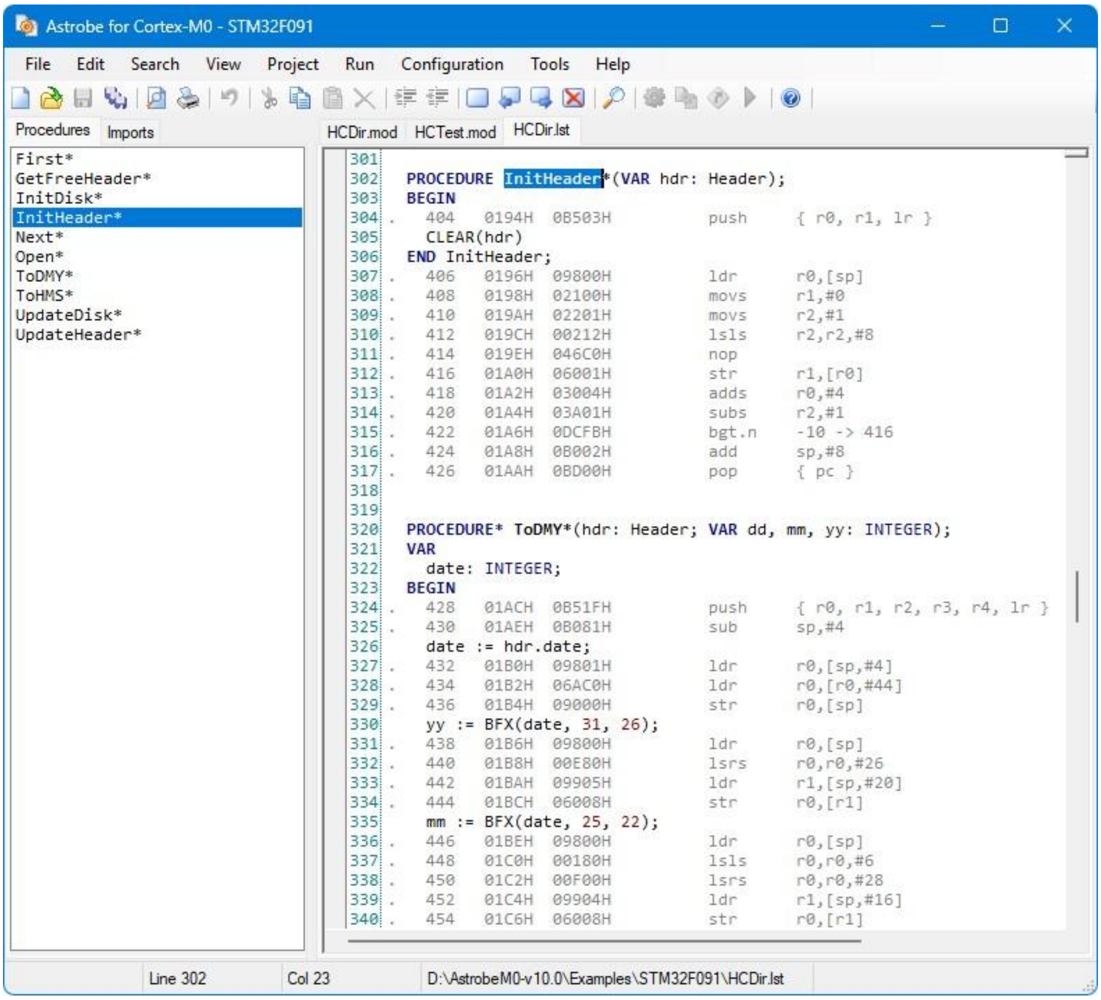
- The runtime error message displayed when your application terminated. This will give you the module name and exception address.
- The map file for the main module (*<ModuleName>.map*) which was created when you linked / built the application. The start address of the module is listed in the *Code Address*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

4.5.1 Using the Module Disassembler Listing:

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

offset = exception address – start address – 4

Look in the disassembler listing of the module where the exception occurred for the instruction with the same offset in column 2. You will see the accompanying Oberon source line which generated that instruction.



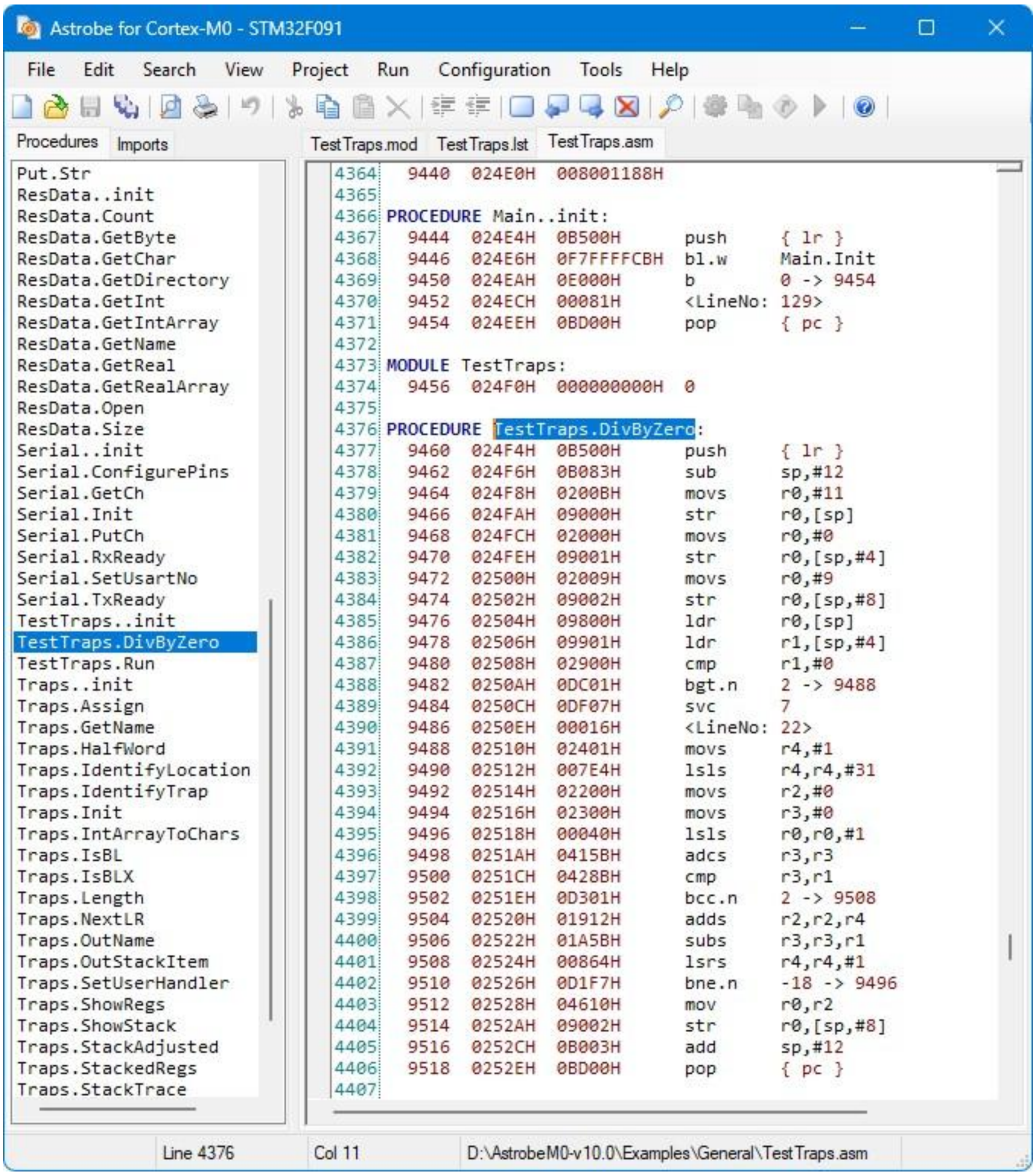
4.5.2 Using the Application Disassembler Listing:

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

offset = exception address – code start address

where the addresses are hexadecimal numbers and *code start address* is the first *Code Range* entry on the Astrobe Configuration dialog.

The heading of that block of assembly instructions will show the name of the module and procedure where the instruction is located.



5 Compile, Link, Build and Disassemble Commands

Separate command-line programs for the Astrobe Oberon Compiler, Builder, Linker and Disassembler which correspond to the built-in Compile, Build, Link and Disassemble Application commands in the IDE are included.

The separate programs 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.

The commands have the following parameters:

```
AstrobeCompile [astrobeFolder] configFilename sourceFilename
AstrobeBuild   [astrobeFolder] configFilename sourceFilename
AstrobeLink    [astrobeFolder] configFilename objectFilename
AstrobeDecode  configFilename executableFilename
```

Where *astrobeFolder* is the (optional) name of the folder that is substituted for the *%Astrobe..%* parameter in the configuration file search paths an *configFileName* is the name of the configuration file containing the options to use.

5.1 Examples

```
AstrobeCompile  D:\AstrobeM3\Configs\STM32L152.ini  Lists.mod

AstrobeBuild  D:\AstrobeM3\Configs\STM32L152.ini  Blinker.mod

AstrobeLink  D:\Astrobe D:\Astrobe\Configs\STM32L152.ini  Blinker.arm

AstrobeDecode  D:\AstrobeM3\Configs\STM32L152.ini  Blinker.bin
```

5.2 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 for use with Astrobe for RP2040, which uses these return values is:

```
REM
REM Rebuild Libraries
REM
SET rootdir=C:\AstrobeRP2040
SET configs=%rootdir%\configs
SET lib=%rootdir%\lib
SET build="C:\Program Files\Astrobe for RP2040\AstrobeBuild.exe"
REM
cd %lib%
del /s *.arm
del /s *.smb
%build% %configs%\PiPico.ini %lib%\General\Build.mod
if errorlevel 1 goto ErrorExit
REM
%build% %configs%\PiPico.ini %lib%\PiPico\Build.mod
if errorlevel 1 goto ErrorExit
REM
echo No errors detected
goto OK
:ErrorExit
echo Errors detected
:OK
```