beginner

COLLABORATORS

	TITLE : beginner	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		April 17, 2022	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

beginner

1.1 Modules

```
                Modules
*******

   A module is the E equivalent of a C header file and an Assembly
include file.  It can contain various object and constant definitions, and
also library function offsets and library base variables.  This information is
necessary for the correct use of a library.

   A code module is an extension of the module idea.  As well as object
and constant definitions, a code module can contain procedures and global
variables.  Code modules are like C and Assembly object files.
```


1.2 Using Modules

```
Using Modules
=============

   To use the definitions in a particular module you use the MODULE
statement at the beginning of your program (before the first procedure
definition).  You follow the MODULE keyword by a comma-separated list of
strings, each of which is the filename (with path if necessary) of a
module (without the .m extension--since every module file name must end
```

with .m). The filenames (and paths) are all relative to the logical volume Emodules: (which can be set-up using an assign as described in the 'Reference Manual'), unless the first character of the string is *. In this case the files are relative to the directory of the current source file. For instance, the statement:

```
MODULE 'fred', 'dir/barney', '*mymod'
```

will try to load the files Emodules:fred.m, Emodules:dir/barney.m and mymod.m. If it can't find these files or they aren't proper modules the E compiler will complain.

All the definitions in the modules included in this way are available to every procedure in the program. To see what a module contains you can use the showmodule program that comes with the Amiga E distribution.

1.3 Amiga System Modules

```
                Amiga System Modules
====================
```

Amiga E comes with the standard Amiga system include files as E modules. The AmigaDOS 2.04 modules are supplied with E version 2.1, and the AmigaDOS 3.0 modules are supplied with E version 3.0. However, modules are much more useful in E version 3.0 (see
Code Modules
). If you want to use any of the standard Amiga libraries properly you will need to investigate the modules for that library. The top-level .m files in Emodules: contain the library function offsets, and those in directories in Emodules: contain constant and object definitions for the appropriate library. For instance, the module asl (i.e., the file Emodules:asl.m) contains the ASL library function offsets and libraries/asl contains the ASL library constants and objects.

If you are going to use, say, the ASL library then you need to open the library using the OpenLibrary function (an Amiga system function) before you can use any of the library functions. You also need to define the library function offsets by using the MODULE statement. However, the DOS, Exec, Graphics and Intuition libraries don't need to be opened and their function offsets are built in to E. That's why you won't find, for example, a dos.m file in Emodules:. The constants and objects for these libraries still need to be included via modules (they are not built in to E).

1.4 Non-Standard Modules

```
Non-Standard Modules
====================
```

Several non-standard library modules are also supplied with Amiga E. To make your own modules you need the pragma2module and iconvert programs. These convert standard format C header files and Assembly include files to modules. The C header file should contain pragmas for function offsets, and the Assembly include file should contain constant and structure definitions (the Assembly structures will be converted to objects). However, unless you're trying to do really advanced things you probably don't need to worry about any of this!

1.5 Example Module Use

Example Module Use
==========

The gadget example program in Part Three shows how to use constants from the module intuition/intuition (see Gadgets), and the IDCMP example program shows the object gadget from that module being used (see IDCMP Messages). The following program uses the modules for the Reqtools library, which is not a standard Amiga system library but a commonly used one, and the appropriate modules are supplied with Amiga E. To run this program, you will, of course, need the reqtools.library in Libs:.

```
MODULE 'reqtools'

PROC main()
  DEF col
  IF (reqtoolsbase:=OpenLibrary('reqtools.library',37))<>NIL
    IF (col:=RtPaletteRequestA('Select a colour', 0,0))<>-1
      RtEZRequestA('You picked colour \d',
                   'I did|I can\at remember',0,[col],0)
    ENDIF
    CloseLibrary(reqtoolsbase)
  ELSE
    WriteF('Could not open reqtools.library, version 37+\n')
  ENDIF
ENDPROC
```

The reqtoolsbase variable is the library base variable for the Reqtools library. This is defined in the module reqtools and you must store the result of the OpenLibrary call in this variable if you are going to use any of the functions from the Reqtools library. (You can find out which variable to use for other libraries by running the showmodule program on the library module.) The two functions the program uses are RtPaletteRequestA and RtEZRequestA. Without the inclusion of the reqtools module and the setting up of the reqtoolsbase variable you would not be able to use these functions. In fact, if you didn't have the MODULE line you wouldn't even be able to compile the program because the compiler wouldn't know where the functions came from and would complain bitterly.

Notice that the Reqtools library is closed before the program terminates (if it had been successfully opened). This is always necessary: if you succeed in opening a library you must close it when you're finished with it.

1.6 Code Modules

```
Code Modules
============
```

You can also make modules containing procedure definitions and some global variables. These are called code modules and can be extremely useful. This section briefly outlines their construction and use. For in-depth details see the 'Reference Manual'.

Code modules can be made by using the E compiler as you would to make an executable, except you put the statement OPT MODULE at the start of the code. Also, any definitions that are to be accessed from outside the module need to be marked with the EXPORT keyword. Alternatively, all definitions can be exported using OPT EXPORT at the start of the code. You include the definitions from this module in your program using MODULE in the normal way.

The following code is an example of a small module:

```
OPT MODULE

EXPORT CONST MAX_LEN=20

EXPORT OBJECT fullname
  firstname, surname
ENDOBJECT

EXPORT PROC printname(p:PTR TO fullname)
  IF short(p.surname)
    WriteF('Hello, \s \s\n', p.firstname, p.surname)
  ELSE
    WriteF('Gosh, you have a long name\n')
  ENDIF
ENDPROC

PROC short(s)
  RETURN StrLen(s)<MAX_LEN
ENDPROC
```

Everything is exported except the short procedure. Therefore, this can be accessed only in the module. In fact, the printname procedure uses it (rather artificially) to check the length of the surname. It's not of much use or interest apart from in the module, so that's why it isn't exported. In effect, we've hidden the fact that printname uses short from the user of the module.

Assuming the above code was compiled to module mymods/name, here's how it could be used:

```
MODULE 'mymods/name'

PROC main()
```

```
  DEF fred:PTR TO fullname, bigname
  fred.firstname:='Fred'
  fred.surname:='Flintstone'
  printname(fred)
  bigname:=['Peter', 'Extremelybiglongprehistoricname']:fullname
  printname(bigname)
ENDPROC
```

Global variables in a module are a bit more problematic than the other kinds of definitions. You cannot initialise them in the declaration or make them reserve chunks memory. So you can't have ARRAY, OBJECT, STRING or LIST declarations. However, you can have pointers so this isn't a big problem. The reason for this limitation is that exported global variables with the same name in a module and the main program are taken to be the same variable, and the values are shared. So you can have an array declaration in the main program:

```
DEF a[80]:ARRAY OF INT
```

and the appropriate pointer declaration in the module:

```
EXPORT DEF a:PTR TO INT
```

The array from the main program can then be accessed in the module! For this reason you also need to be pretty careful about the names of your exported variables so you don't get unwanted sharing. Global variables which are not exported are private to the module, so will not clash with variables in the main program or other modules.