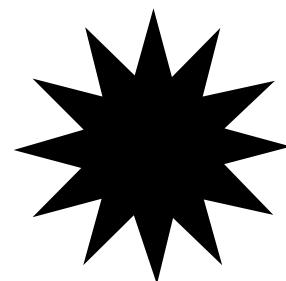
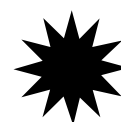


Esp



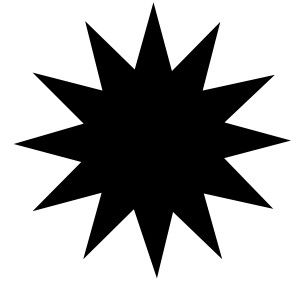
Version 2.0



GEOS Software Development Kit Library

Version 2.0

Esp



Initial Edition

Geoworks, Inc.
Alameda, CA



Geoworks provides this publication "as is" without warranty of any kind, either express or implied, including but not limited to the implied warranties of merchantability or fitness for a particular purpose. Geoworks may revise this publication from time to time without notice. Geoworks does not promise support of this documentation. Some states or jurisdictions do not allow disclaimer of express or implied warranties in certain transactions; therefore, this statement may not apply to you.

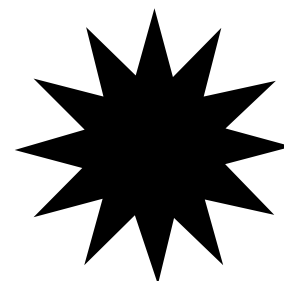
Copyright © 1994 by Geoworks, Incorporated.
All rights reserved. Published 1994
Printed in the United States of America

Geoworks®, Geoworks Ensemble®, Ensemble®, GEOS®, PC/GEOS®, GeoDraw®, GeoManager®, GeoPlanner®, GeoFile®, GeoDex® and GeoComm® are registered trademarks of Geoworks in the United States and Other countries.

Geoworks® Pro, PEN./GEOS, Quick Start, GeoWrite, GeoBanner, GeoCrypt, GeoCalc, GeoDOS, Geoworks® Writer, Geoworks® Desktop, Geoworks® Designer, Geoworks® Font Library, Geoworks® Art Library, Geoworks® Escape, Lights Out, and Simply Better Software are trademarks of Geoworks in the United States and other countries.

Trademarks and service marks not listed here are the property of companies other than Geoworks. Every effort has been made to treat trademarks and service marks in accordance with the United States Trademark Association's guidelines. Any omissions are unintentional and should not be regarded as affecting the validity of any trademark or service mark.

Contents



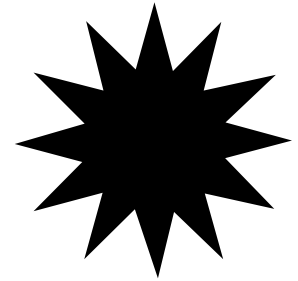
	<i>List of Code Examples</i>	7
1	Introduction to Esp	9
1.1	What is Esp?	11
1.2	Should I Use Esp?	12
1.3	Roadmap to the Esp Book	13
2	Esp Basics	15
2.1	The Purpose of Esp	17
2.2	Esp Ground Rules	17
	2.2.1 GEOS is a Multitasking Environment	18
	2.2.2 Upward and Downward Compatibility	19
	2.2.3 Flags	20
2.3	Differences from MASM	21
	2.3.1 Data Types	21
	2.3.2 Symbols and Labels	32
	2.3.3 Segments and dgroup	32
	2.3.4 Miscellaneous Enhancements	39
2.4	Defining Classes	51
	2.4.1 Defining a Class	51
	2.4.2 Creating a Class's Class Structure	53
	2.4.3 Defining your Process Class	55
2.5	Error-Checking Code	55
3	Routine Writing	59
3.1	GEOS Conventions	61
	3.1.1 Parameters and Local Variables	63
	3.1.2 The "uses" Directive	66



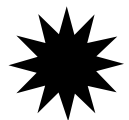
	3.1.3 .enter and .leave	66
	3.1.4 Inheriting a Stack Frame.....	69
3.2	LMem Heaps and Chunks.....	70
3.3	Objects and Classes	73
	3.3.1 Object Structure	74
	3.3.2 Messages	77
4	The UI Compiler	93
4.1	UIC Overview	95
4.2	Declaring Classes.....	97
	4.2.1 Declaring Fields.....	98
	4.2.2 Changing a Default Value	100
4.3	Creating Objects and Chunks	102
	4.3.1 Setting Up a Resource	102
	4.3.2 Creating Objects	103
	4.3.3 Creating Chunks	105
	4.3.4 Creating VisMonikers	107



Code Examples

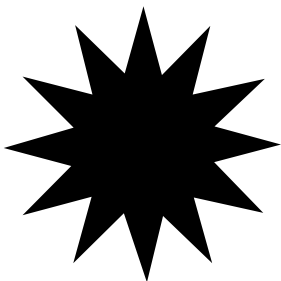


Code Display 2-1	Declaring an Enumerated Type	24
Code Display 2-2	Data Structure Declaration Examples	31
Code Display 2-3	Declaring Global Variables	34
Code Display 2-4	Creating a Class	54
Code Display 2-5	Writing a Procedure	67
Code Display 2-6	Using an LMem Heap	72
Code Display 2-7	Finding a Class's Master Section	75
Code Display 3-1	Modifying a Superclass	101



 **Esp Programming**

Introduction to Esp



1.1	What is Esp?.....	11
1.2	Should I Use Esp?	12
1.3	Roadmap to the Esp Book	13

1

This software development kit is geared towards C programmers. The SDK is based around the GEOS-specific language Goc, which is based on C. Very few programmers will ever need to use anything else. However, a few programmers will want to write the most efficient code possible; for their benefit, this SDK lets you program in Esp, the GEOS assembly language.

Only experienced GEOS programmers should attempt to program in assembly. Furthermore, the SDK is not intended to teach assembly language; you should be familiar with 80x86 assembly before you attempt to program in Esp.

1.1

1.1 What is Esp?

Esp is an 80x86 assembly language specially designed for GEOS programming. In many ways, it is very similar to other common 80x86 assemblers (such as MASM). On a low level, inside a routine, Esp code looks very much like MASM code (though it provides some extra features). However, on a higher level, Esp resembles MASM much less, and resembles Goc much more.

Esp incorporates support for the GEOS object-oriented programming environment. It provides routines to let you send messages and manipulate objects. Furthermore, all GEOS kernel routines and system libraries may be called from either Goc or Esp code. Thus, programming in Esp is very much like programming in Goc. The routines are written in a different language, but they fit together into a program in much the same way that Goc routines do.

In addition to Esp, this SDK provides a special User-Interface Compiler (UIC). The UIC generates object-blocks for assembly applications. It reads source files written in the GEOS-specific language Espire, and creates special resources that Glue can incorporate into GEOS applications. If you write applications in assembly language, you will use the UIC to create objects and object blocks at compile time.



Many of the sample applications provided with this SDK have been written twice: once in Goc (in the APPL/SDK_C directory), and once in Esp (in the APPL/SDK_ASM directory). These sample applications are an excellent way to familiarize yourself with Esp; you can see how the same task is accomplished in Goc and Esp. In particular, they will let you compare Esp and Goc techniques for creating structures and classes, and compare Goc and Espire techniques for creating objects and classes.

1.2

1.2 Should I Use Esp?

Not everyone will want to use Esp. For many programmers, assembly language is much harder to use than high-level languages like Goc. You may not find the gains in program efficiency worth this extra effort.

There are some cases when you should consider using Esp. Firstly, if you are very familiar with 80x86 assembly language, and feel as comfortable with it as with C, you may prefer to write your programs in Esp. Once you've written a few programs, you may find Esp as easy a language to work with as Goc.

Secondly, if you are writing computation-intensive programs, you may find that they are running too slowly (especially if you are writing them for GEOS platforms with less processing power). In these cases, you may wish to rewrite the programs in Esp. Alternatively, you may choose to put the most memory-intensive routines in special "Esp resources"; you can write those routines in Esp, while writing the rest of the application in Goc.

Thirdly, if you are writing a library that will be used by many different applications, you may wish to write that library in Esp. If it is a routine library, you will merely need to write Esp routines that conform to C pass-and-return conventions; then any application, whether written in Goc or in Esp, will be able to call those routines. Similarly, if the library defines new object classes, you can write the message handlers in Esp; any application that uses those objects will be able to take advantage of the added efficiency of Esp, even if the application is written in Goc.



1.3 Roadmap to the Esp Book

The Esp book is fairly short. This is because its main role is to help you integrate knowledge you already have. You should already know how to program for GEOS, and how to program in non-GEOS assembly language for the 80x86; this book tells you how to combine this knowledge, and describes the special features of Esp.

The book has the following chapters:

1.3

1. Introduction to Esp (this chapter)

This chapter gives a brief overview of Esp, the GEOS assembly language, and of this book.

2. Esp Basics

This chapter describes the basic syntactic differences between Esp and other 80x86 assembly languages (such as MASM). It also describes the basic ground rules for programming in assembly in the GEOS environment.

3. Routine Writing

This chapter describes how to write routines in Esp. It discusses Esp's special features that make routine-writing simpler and more uniform, by taking care of such chores as creating local variables, managing stack frames, etc. It also discusses how to write handlers and send messages in Esp.

4. The UI Compiler

This chapter describes the GEOS User-Interface Compiler (UIC), a special utility that compiles Esp object blocks. You will need to use UIC if you are writing an Esp application that has any objects created at compile-time.

The book also contains an appendix which describes, briefly, how to incorporate Esp resources into a Goc application.

When you are ready to start programming, you will want to consult the Assembly Reference for Esp API of GEOS routines and structures and to consult the PCGEOS\INCLUDE*.DEF and *.UIH files for assembly information about GEOS object classes. To find out a message's pass and return parameters, see the class's **.def** file; for more generic information about a message, you can see its (Goc) reference entry in the Objects book.



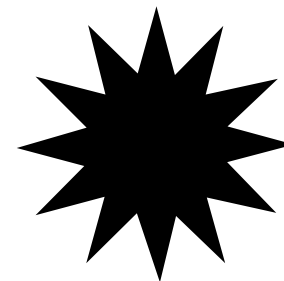
Introduction to Esp

* 14

1.3



Esp Basics



2

2.1	The Purpose of Esp	17
2.2	Esp Ground Rules	17
2.2.1	GEOS is a Multitasking Environment	18
2.2.2	Upward and Downward Compatibility	19
2.2.3	Flags.....	20
2.3	Differences from MASM	21
2.3.1	Data Types.....	21
2.3.1.1	Constants	21
2.3.1.2	Simple Types	22
2.3.1.3	Enumerated Types.....	23
2.3.1.4	Structures.....	25
2.3.1.5	Unions	27
2.3.1.6	Records	28
2.3.1.7	Creating New Types.....	30
2.3.2	Symbols and Labels.....	32
2.3.3	Segments and dgroup	32
2.3.3.1	The dgroup Segment.....	33
2.3.3.2	Accessing Segments	34
2.3.3.3	Declaring Static Variables.....	36
2.3.3.4	Strings	38
2.3.4	Miscellaneous Enhancements	39
2.3.4.1	Pseudo-Ops and Directives.....	40
2.3.4.2	Miscellaneous Macros.....	43
2.3.4.3	Useful Miscellaneous Macros	46
2.3.4.4	dword Macros	50
2.4	Defining Classes	51
2.4.1	Defining a Class.....	51
2.4.1.1	Defining a Class's Messages	52
2.4.1.2	Defining a Class's Instance Data Fields	53
2.4.1.3	Defining a Class's Vardata.....	53

2.4.2	Creating a Class's Class Structure.....	53
2.4.3	Defining your Process Class.....	55
2.5	Error-Checking Code	55

Esp is an assembly language for 80x86 microprocessors. It is designed for creating applications, libraries, and drivers that will run under GEOS. As such, it is very similar to other common 80x86 assembly languages (such as MASM), but has special features and functionality to make it easier to write GEOS code.

To experienced MASM programmers, Esp code will be easy to read. Variable declarations will be slightly different, and there will be a few instructions (actually pseudo-ops) which look new. Nevertheless, Esp code will look very familiar. This chapter describes the stylistic differences between Esp and MASM.

2.1

This book assumes that you already know how to program in 80x86 assembly language.

2.1 The Purpose of Esp

Esp is mainly a superset of MASM. With the exception of a few special cases (which are noted in this chapter), MASM code can be ported intact to Esp programs without requiring major modifications. You should not find it hard to reuse your existing code.

Esp is, however, philosophically different from other assembly languages. It is designed for an object-oriented, multitasking environment. This means that it works with different assumptions from other assembly languages.

2.2 Esp Ground Rules

There are certain rules you must follow when programming in Esp. These rules are imposed by the nature of GEOS.

If you violate these rules, the results are unpredictable. Error-checking code may find violations of these rules; however, this is not guaranteed. Therefore, you must be sure to follow the rules under all circumstances.



2.2.1 GEOS is a Multitasking Environment

2.2

GEOS uses preemptive multitasking. It uses interrupts to halt each thread's execution when its allotted time slice ends. This has two major consequences for assembly programs. First, there are more interrupts being sent than you might expect. Second, if interrupts are disabled, this drastically degrades GEOS's performance; under almost no circumstances should a geode disable interrupts. In fact, under the GEOS API, only the kernel and drivers are permitted to disable interrupts; libraries and applications must leave interrupts enabled at all times.

Experienced MASM programmers will already know how to cope with interrupts, but it's worth reiterating the basic rules. The main thing to remember is that the interrupt handlers use the stack to save the state. The interrupt can occur after any instruction. Whenever you use an instruction which alters the stack, or makes assumptions about the stack, you should ask yourself what would happen if a context switch occurred right before or right after that instruction.

For example, suppose you want to read the top word on the stack into `cx`, but you want it to stay on the stack. The canonical way to do this would be the following:

```
pop    cx      ; Read the value . . .
push   cx      ; . . . and push it back.
```

This takes only two bytes and 27 cycles; this is fairly good for a memory reference. An overzealous optimizer might think, "Well, that word's still right there above the stack, so we don't have to push it back, do we?" He might write the following bad code:

```
pop     cx
sub     sp, 2 ; THIS IS VERY BAD
```

"Aha," he might think, "This takes only 16 cycles!" Unfortunately, the code is extremely fragile. If a context switch occurs right after the `pop`, the interrupt handler will push all the registers onto the stack; this will overwrite the data at that location. The worst part is that this bug is intermittent; the code will work fine, as long as the context switch doesn't occur at that precise location. That means the bug can easily sneak through testing.



It bears repeating: Whenever you perform an unusual operation on the stack, ask yourself what would happen if a context switch occurred immediately before or immediately after the instruction.

2.2.2 Upward and Downward Compatibility

GEOS is intended to run on a wide range of platforms, from 8088-based machines up through powerful desktop computers (80486s and beyond). Because of this, you should avoid writing code that makes assumptions about which processor is being used. Even if you know you're writing for a GEOS platform which uses a particular processor, you should write more flexible code; this will make it much easier to port the code across several GEOS platforms.

2.2

The main thing is to use only those instructions that are available on all 80x86 machines. (This also means not using instruction variants which are available on only some machines; for example, you may not pass an immediate argument to `shl`). Do not try to access the 80386's 32-bit registers.

A future version of GEOS may take advantage of protected mode in more powerful chips. Again, if you take certain precautions, your code will run unchanged under this version of GEOS. You must follow these rules:

- ◆ Perform no arithmetic on segment register values. In protected mode, a segment register doesn't contain a physical segment address; rather, it contains an index (or *selector*) into a hardware segment descriptor table. Adding one to a segment register doesn't advance it 16 bytes, as in real mode; it changes it to a completely different selector.
- ◆ Do not attempt to access interrupt vectors except through GEOS. Very few applications will need to do anything like this, anyway.
- ◆ Do not try to use the `sti`, `cli`, `in` or `out` instructions in applications or libraries; they are privileged instructions under protected mode. Only drivers and the kernel may use these instructions.
- ◆ Do not try to write self-modifying code. It is extremely difficult to do this under protected mode.

- ◆ Do not try to use segment registers for temporary data storage. Under protected mode, the processor will complain if you load anything but a valid selector into a segment register.

There is currently no way for geodes to use a floating-point coprocessor directly. However, all GEOS floating-point routines will automatically use a floating-point coprocessor if one is present.

2.2

2.2.3 Flags

GEOS makes certain assumptions about the flags. Your application must follow these if it is to work with GEOS properly.

The 80x86 flags are divided into two groups: *status flags* and *control flags*. There are five status flags: the *overflow* flag, the *sign* flag, the *zero* flag, the *auxiliary-carry* flag, and the *carry* flag (abbreviated as OF, SF, ZF, AF, and CF, respectively). Status flags provide information about the results of recent operations. For example, if the result of a subtraction operation is zero, the zero flag (ZF) is set. Routines are allowed to set and change these status flags at will. Even if a routine says that it destroys nothing, it is presumed to destroy all the status flags unless it specifically says “flags preserved.” Some routines will return flags with meaningful settings; for example, many routines set CF to indicate an error, and clear it otherwise. In these cases, the routine’s reference will describe all values returned in flags. By the same token, you may call routines with any settings you wish for the status flags, unless the routine specifically requires that the status flags have certain settings.

There are three *control flags*: the *direction* flag, the *interrupt* flag, and the *trap* flag (abbreviated as DF, IF, and TF, respectively). These flags change how the processor operates. Routines have much less leeway about how and whether to use these flags.

Most routines should leave interrupts enabled. In practice, only drivers will need to disable interrupts. Most kernel routines require that they be called with interrupts enabled. If a routine doesn’t specifically say that it can be called with interrupts disabled, then it cannot be. This is not an issue for most programmers, since the GEOS API permits only drivers to disable interrupts.



All GEOS routines assume that the direction flag (DF) is cleared. Feel free to set this flag before using string instructions; however, you should make sure to clear DF before calling any kernel routine. Again, some routines may specifically permit you to call them with DF set; you should not assume this is the case unless the routine reference says so.

You should never change TF; this is used by the debugger.

2.3

2.3 Differences from MASM

Esp has a number of differences with other 80x86 assemblers. Some of these are entirely transparent to the programmer; these differences will not be detailed here.

The main algorithmic difference between Esp and MASM is that Esp only reads the source code once. As a result, MASM directives that rely on multiple passes are treated differently; for example, the IF1 and IF2 directives are both synonymous with IF.

In all cases where an algorithmic break is not involved, you can force Esp to use the MASM syntax and directives by passing the flag “-m”.

2.3.1 Data Types

Esp makes it easy to declare and define structures, records, enumerated types, and similar constructs. Its conventions are, however, slightly different from those of MASM; you should be aware of these differences.

2.3.1.1 Constants

Esp's rules for constants are almost the same as MASM's. Esp is slightly more versatile. For example, hexadecimal constants may be specified with either the MASM convention or the C convention; that is, “123h” is exactly the same as “0x123”.

A single character surrounded by double quotes is parsed as the ASCII value of that character; for example,



```
LETTER_A    = "a"
```

is identical to

```
LETTER_A    = 61h
```

You may use any of the standard C character escapes; these are listed in Table 2-2 on page * 39. Since “\” is the escape character, you have to use a doubled backslash to put a backslash in the character string; that is, “\\” specifies the single character 5Ch.

2.3

2.3.1.2 Simple Types

Esp defines many standard data types beyond those provided by MASM. These types can be used alone, or they can serve as building blocks for structures. The types are listed in Table 2-1 on page * 22.

Swat can read symbolic information about the data types to display their values in the most useful format. For example, if you have an array of **bytes** containing the values 46h, 6Fh, 6Fh, 21h, Swat will display the hex values; but if the same values are **sbytes**, Swat will display them as signed, decimal

Table 2-1 *Major Esp Data Types*

Type	Size	Description
byte	1	Unsigned 8-bit integer (0—255); synonym is “db”
sbyte	1	Signed 8-bit integer (-128—127); synonym is “sb”
char	1	GEOS character; synonym is “dc”
word	2	Unsigned 16-bit integer (0—65,535)
sword	2	Signed 16-bit integer (-32,768—32,767)
dword	4	Unsigned 32-bit integer (0—4,294,967,296)
sdword	2	Signed 32-bit integer (-2,147,483,648—2,147,483,647)
nptr	2	Near pointer (i.e. offset address)
fpnr	4	Far pointer (i.e. segment + offset)
hptr	2	Global handle
lptr	2	Chunk handle (i.e. near pointer to a near pointer)
optr	4	object descriptor; high word is hptr, low word is lptr
sptr	2	Segment address (or descriptor).

These are the main Esp data types, with their size in bytes.



integers (i.e. “70, 111, 111, 33”); and if they are declared as an array of **chars**, Swat will translate the values into ASCII characters (i.e. “Foo!”). Similarly, Swat can use the information about a pointer’s type to display its referent appropriately.

2.3.1.3 Enumerated Types

Sometimes you will have a variable that indicates one of a number of conditions by holding an arbitrarily-chosen integer. For example, you may have variables that indicate a month, using a different integer to indicate each month. In these cases, it is best to create an enumerated type. Enumerated types let you use the values by name, making the code much easier to read. Furthermore, when you use Swat to examine a variable of an enumerated type, you will (by default) be shown the value’s name instead of its numerical value; this makes debugging easier.

2.3

Esp gives you considerable control over enumerated types. You can declare how long the values will be, what the initial value should be, and by how much the values should be incremented. To define a new enumerated type, use a declaration of the format

```
<typename> etype <size> [, <first> [, <step>]]
```

typename Any arbitrary name for the type. Usually this will begin with the application’s name, to avoid conflicting with other enumerated type names. By convention, the type name is singular (i.e. “**HelloColor**”, not “**HelloColors**”).

size The size of values of this type. This may be either of the reserved words “word” or “byte”.

first The value of the first member of the type. This defaults to zero.

step The increment between members of the type. This defaults to one.

Each member of the enumerated type is declared like this:

```
<name> enum <typename> [, <value>]
```

name The name of this member of the type. As a matter of convention, the name of each member of a type begins with an abbreviation of the type name; for example, a member of **HelloColor** might be named HC_BLUE.



typename The name of the enumerated type.

value The value of this member of the type. This defaults to the previous element's value plus the `step` specified in the type declaration.

For example, to declare an enumerated type for the months of the year, you might do this:

2.3

Code Display 2-1 Declaring an Enumerated Type

```
HelloMonth        etype byte, 1     ; One byte is enough to hold the twelve months.
                                     ; We specify that the first month should have a
                                     ; value of one, as is conventional.

HM_JANUARY        enum HelloMonth
HM_FEBRUARY       enum HelloMonth
HM_MARCH           enum HelloMonth ; and so on . . .
```

Note that members of the enumerated type need not be declared all at once. You can have other declarations, or even code, intervening.

The name of the enumerated type will always evaluate to one step more than the last member of the enumerated type (that is, the last one *before* the use of the type's name; more members could be declared later). You can use this to verify that a value is in bounds for an enumerated type. For example, suppose you had the following enumerated type:

```
MyColor           etype byte 0, 2
MC_BLUE            enum MyColor ; MC_BLUE = 0
MC_RED             enum MyColor ; MC_RED = 2
MC_GREEN           enum MyColor ; MC_GREEN = 4
```

At this point, the name **MyColor** would evaluate to 6, i.e. MC_GREEN plus the step-value of two. If a routine expected to be passed a member of the **MyColor** enumerated type, it could check this by comparing the value to the value of **MyColor**.



2.3.1.4 Structures

Esp lets you define structures. Structure declarations have the following format:

```
<StructureType>    struct
    <FieldName> <FieldType> [<DefaultValue>]
                                ;any number of these
<StructureType>    ends
```

2.3

StructureType

This may be any valid, unique identifier.

FieldName

This may be any valid, unique identifier.

FieldType This may be any previously-defined type. It may be a simple type, an array, a record, another structure, or any other type you wish.

DefaultValue

This is the default value for this field of the structure.

The fields are declared from low to high. That is, the first field named is at the low end of the structure, and has the same address as the structure itself.

For example, you might declare a simple data structure like this:

```
MyDataStructure    struct
    MDS_aField      sbyte
    MDS_anotherField sword    -1
    MDS_oneLastField dword
MyDataStructure    ends
```

You can declare and initialize one of these structures much the same way as you would an array:

```
aStructure MyDataStructure    <1,2,3>
```

This format is versatile. If you leave a space blank, it will automatically be initialized to the default value (or zero, if no default value was specified). If you don't put any values between the angle-brackets, the whole structure will be initialized to its default values. Thus,

```
aStructure MyDataStructure    <>
```



is equivalent to

```
aStructure MyDataStructure <0, -1, 0>
```

One of the fields of a structure may be another structure. For example, you might make the following declaration:

```
MyOtherStructure struct
    MOS_char1      char
    MOS_char2      char
    MOS_dataStruct MyDataStructure
    MOS_signedLong sdword
MyOtherStructure ends
```

2.3

You might initialize the structure like this:

```
bigStruct MyDataStructure \
    <'a', , <1,2,3>, -0xabcd123>
```

As noted above, the *MOS_char2* field would be initialized to zero.

Esp evaluates a field name as the displacement from the start of the structure to the start of the field. For example, if **MyStructure** is defined as shown above, then *MDS_aField* would evaluate as zero, *MDS_anotherField* as one, and *MDS_oneLastField* as three. You can use these displacements to access fields by using the dot operator or the bracket operator. Both of these are addition operators for calculating effective addresses. Several displacements can be used sequentially. For example, suppose we declared **bigStruct** as shown above. We want to load the *MDS_anotherField* field from that structure into **ax**. If **es:[di]** was the address of the **bigStruct** variable, we could do the following:

```
mov ax, es:[di].MOS_dataStruct.MDS_anotherField
```

Esp would figure out the displacement from the start of a **MyOtherStructure** to the *MOS_dataStruct* field; it would add this to the displacement from the start of a **MyDataStructure** to the *MDS_int2* field, and use the combined displacement in the instruction, producing an equivalent machine instruction, e.g.

```
mov ax, es:[di].3
```

You can use the dot operator this way in any effective-address instruction.



2.3.1.5 Unions

Esp supports *unions* as well as structures. A union is a variable that might, at different times, have values of different sizes or types.

A union is declared much like a structure. The basic format is:

```
<UnionType>          union
    <FieldName> <FieldType> [<DefaultValue>]
                                ;any number of these
<UnionType>          ends
```

2.3

UnionType

This may be any valid, unique identifier.

FieldName

This may be any valid, unique identifier.

FieldType

This may be any previously-defined type. It may be a simple type, an array, a record, a structure, or any other type you wish.

DefaultValue

This is the default value for this field of the union.

Every field of the union begins at the base of the union, and the union is as large as its largest component field. For example,

```
MyUnion      union
    MU_sbyte  sbyte      -2
    MU_word   word       1234
MyUnion      ends
```

would declare a union with two fields. The union would be two bytes long.

Unions are initialized slightly differently from structures. You can initialize a union to all zeros by putting nothing between the angle brackets, e.g.

```
aVariable    MyUnion    <>
```

You can initialize the union to contain the default value for one of its components by putting the component's name between the angle brackets, e.g.

```
aVariable    MyUnion    <MU_sbyte>
```



would initialize the first byte of the union to 0xfd (i.e. -2), and clear the second byte. If you wish to override the default value, simply put the new value after the field name, like so:

```
aVariable    MyUnion    <MU_sbyte 12>
```

2.3.1.6 Records

2.3

Sometimes you will need to store several pieces of information, each of which can be represented in less than a byte. One common situation is when you need to have several flags for an object. Each one of the flags is a boolean quantity, so it can be represented with one bit; it would be inefficient to store each flag in its own byte-sized variable.

Esp allows you to declare byte- or word-sized records. Each field of the record may be one or more bits long; multi-bit fields may hold values from an appropriately-sized enumerated type. A record declaration has the following format:

```
<recordname>    record
    [<fieldname> [<type>]] :<size> [= <value>]
    ;...there may be many such lines
<recordname>    ends
```

recordname

This is the name of the record. It usually begins with the geode's name, to ensure that the name won't conflict with a name in an included header file.

fieldname

The name of the field. If a field has no name, you cannot directly access it; thus, nameless fields can be used to pad the record to byte- or word-length.

type

If the field contains a member of an enumerated type, you should specify the type here.

size

This is the size of the field in bits. The combined sizes of the fields should not be greater than sixteen.

value

You may specify a default value here. If a variable of this record is declared without initializers, the field will be initialized to this value.



The fields are declared from high to low; that is, the first field declared occupies the high end of the record. However, the last field declared always has an offset of zero; that is, it is always at the extreme low end of the record. Thus, if the fields don't add up to a full byte or word, there will be unused bits at the *high* end of the record. The size of the record is equal to the total width of the fields, rounded up to the next byte.

In order to read a field from a record, you need to know the field's position in the record, and you need to know how long the field is. Esp gives you this information with the reserved words `offset`, `mask`, and `width`.

2.3

"`offset <fieldName>`" is assembled into the field's offset from the low end of the record; that is, shifting the record to the right by this amount will bring the field to bit 0. "`mask <fieldName>`" is assembled into a byte or word, as appropriate, with all the bits in the specified field set, and all the other bits cleared. You can also take the mask of a record; "`mask <recordName>`" assembles to a mask with all bits in named fields turned on, and all other bits turned off. "`width <fieldName>`" assembles to the width of the field, in bits.

For example, suppose you define the record **HelloRecord** thus:

```
HelloRecord      record
    HR_A_FLAG:1
    HR_ZERO_TO_SEVEN:3
    HR_ANOTHER_FLAG:1
HelloRecord      ends
```

In this case, "`mask HR_ZERO_TO_SEVEN`" would assemble to `0eh`, "`offset HR_ZERO_TO_SEVEN`" would assemble to `1`, and "`width HR_ZERO_TO_SEVEN`" would assemble to `3`. "`mask HelloRecord`" would assemble to `1Fh`. If you wanted to load *HR_zeroToSeven* into **ax**, you would do the following (assuming **es:[di]** pointed to the record):

```
mov  ax, es:[di] ;load the record into ax
and  ax, mask HR_ZERO_TO_SEVEN
                                ; Clear the other fields
mov  cl, offset HR_ZERO_TO_SEVEN
shr  ax, cl
```

To test if a given flag (e.g. *HR_aFlag*) was set, you would simply do this:

```
test es:[di], mask HR_zero
```



Esp Basics

* 30

Note that in Esp, unlike MASM, you must use either `mask` or `offset` to access a field. If you use the name of the field without either of these keywords, Esp will generate an error. (You can return to default MASM behavior by assembling with the “-m” switch; in this case, “<fieldName>” will be considered equivalent to “`offset <fieldName>`”.)

2.3

You can initialize a record in much the same way that you initialize a structure, i.e. by putting the values in angle-brackets. It is important to note that the initializers only initialize *named* fields; all unnamed fields are automatically initialized to zero. For example, suppose you declared **GapRecord** thus:

```
GapRecord    record
    GR_A_BIT:1
    GR_A_NYBBLE:4
    :2
    GR_ANOTHER_BIT:3
GapRecord    ends
```

And then declared a variable thus:

```
instanceOfGR    GapRecord    <0x1,0xF,0x7>
```

instanceOfGR will be initialized to 0x03E7; the two bits between GR_A_NYBBLE and GR_ANOTHER_BIT will be initialized to zero.

You can also use the name of the record, combined with the initializer, as an immediate value. For example, the instruction

```
move    ax, GapRecord <0x1, 0xF, 0x3>
```

assembles equivalently to

```
move    ax, 0x03E7
```

2.3.1.7 Creating New Types

Esp overloads the `TYPE` operator as a type-creation directive. It is useful if you will be creating many arrays of exactly the same size. This is the format:

```
<TypeName>    TYPE    <n>    dup(<BaseType>)
```

TypeName

The name of the new type.



n The number of elements in the array.

BaseType The type of each element in the array.

Variables of this type will be initialized to all zeros, unless you specify an initial value with MASM's usual array-initializer (angle-bracket) syntax.

For example, you might store social-security numbers in arrays which are nine bytes long (with one byte per digit). In this case, you could make the following declaration:

```
SocSecNum    TYPE 9 dup(byte)
```

2.3

You could declare one of these variables and initialize it like this:

```
FranksSSN    SocSecNum <1,2,3,4,5,6,7,8,9>
```

Code Display 2-2 Data Structure Declaration Examples

```
COMMENT@-----
    This shows how you might combine various Esp types, and how you
    might use those declarations in code.
-----@

;
; Types
;

MyColor          etype byte

MC_CLEAR         enum MyColor      ; This defaults to zero
MC_BLACK         enum MyColor      ; This is MC_CLEAR + 1, or one
MC_WHITE         enum MyColor      ; 2...
MC_RED           enum MyColor
MC_BLUE          enum MyColor
MC_GREEN         enum MyColor

MyRecord         record
    MR_BIG:1
    MR_COLOR MyColor:8
    MR_POINTY:1
MyRecord         end

ShortString      TYPE      9 dup(char)
```



Esp Basics

* 32

```
MyStructure    struct
    MS_number      sword
    MS_label       ShortString
    MS_record      MyRecord
MyStructure    ends

;
; Initialized Variables
;
```

2.3 idata segment

```
AStructure      MyStructure      <-123, <"Foo!", 0>,
                                   (mask MR_BIG OR (MC_RED SHL offset MR_COLOR))>

idata    ends
```

2.3.2 Symbols and Labels

Esp improves on MASM's rules for symbols and labels.

You can declare a local label in Esp. A local label's scope is limited to the procedure that contains it. Local labels in Esp have independent namespaces; that is, you might have several routines, each of which contains the label "done: "; whenever you use the label, Esp would understand it to be the version defined locally.

All labels inside of procedures are presumed to be local. If you want to use the label outside of the procedure, you should declare it thus:

```
<myLabel> label near
```

2.3.3 Segments and dgroup

Geodes are divided into *segments*. Each segment is loaded into memory all at once, and accessed with a given segment address (hence the name).

Segments should be declared in the **.gp** file just as they are for Goc geodes. You must also mark the beginning and end of each segment in the assembly source file. At the beginning of the segment, put a line like



```
<segmentName>      segment resource
```

At the end of the resource, put a line like

```
<segmentName>      ends
```

You can enter and leave a segment multiple times. You can even do so in different code files, as long as the resource is not an LMem heap. The linker will combine the resources appropriately.

Every resource has a resource ID. This resource ID is determined at link-time; this means that a resource in a multi-launchable application will have the same ID in each copy of the application running.

2.3

2.3.3.1 The **dgroup** Segment

Every geode is assigned a fixed memory resource for its global variables (and, if the geode has a process object, for the process thread's stack). This resource is known as *dgroup*. The **dgroup** segment is fixed and non-sharable. Variables in the **dgroup** will keep the same address throughout a session of GEOS.

The **dgroup** segment contains the process object's instance data. Whenever a message is sent to the process object, the **dgroup**'s segment address will automatically be loaded into **ds**. In general, the **dgroup** segment is used for most statically allocated, global variables. Because the segment contains the process object's stack, you should not try to change the segment's size or dynamically allocate space in it.

To declare a global variable, place it in the pseudo-segment *idata* or *udata*. The assembler combines these two pseudo-segments into the fixed, non-sharable **dgroup** segment. The **idata** pseudo-segment contains variables that must be initialized to non-zero values. All variables in **udata**, on the other hand, are automatically initialized to zero. They thus take up no space in the executable file, since their initial values need not be stored.

If the geode declares any new classes, the class declarations should be put in the **idata** pseudo-segment. This is discussed at length in "Defining Classes" on page 51.

Code Display 2-3 Declaring Global Variables

```
; Note that the geode will not have segments named idata or udata; these are  
; pseudo-segments, and are combined into dgroup by the assembler.
```

```
;-----  
;      Initialized Variables  
;-----
```

```
2.3 idata segment  
      MyAppProcessClass      mask CLASSF_NEVER_SAVED  
      MyGlobalString  char    "Franklin Tiberius Poomm, Esq.",0  
idata ends
```

```
;-----  
;      Uninitialized Variables  
;-----
```

```
udata segment  
      MyEmptyArray  sword  20 dup (?)  
udata ends
```

2.3.3.2 Accessing Segments

`GetResourceHandleNS, GetResourceSegmentNS, handle, segment,
GeodeGetResourceHandle, vSegment`

Accessing a resource is slightly more complicated in GEOS than it is in traditional PC programming. A given resource may move around while it is not being accessed. For this reason, you must access non-fixed resources through handles.

All geode resources are GEOS memory blocks, as described in “Memory Management,” Chapter 15 of the Concepts Book. This means that every resource has a global handle. You will often need to get the handle of a resource. For example, whenever you send a message to an object, you need to know the handle of the object’s resource. If you want to access data in an unlocked, non-fixed resource, you will need to get the resource’s handle so you can lock it.



One problem is that there may be several copies of a given resource in memory at a time. For example, if you write a multi-launchable application, every copy of that application running at a time will have its own **dgroup** segment. For this reason, you must use a special macro to get the handle of a non-sharable resource, namely **GetResourceHandleNS**. This macro is passed the resource name of a segment; it returns the segment's global handle.

If you know that a resource is locked or fixed in memory, you can use **GetResourceSegmentNS** to get the segment address directly. This macro is passed the resource name of a locked or fixed segment; it returns the segment's base address.

2.3

If you know that there is only one copy of a resource in memory, you can use a shorter and faster syntax to get the handle or segment. There are two common situations when you can be sure that there is only one copy of a resource: The application might be single-launchable, or the resource might be sharable (for example, code or read-only data). To get the handle of such a resource, use the `Esp` directive `handle`. For example, to load the handle of the **HelloInitCode** resource into `bx`, you would use

```
mov    bx, handle HelloInitCode
```

If you know that such a segment is locked or fixed in memory, you can get its segment address with the `segment` directive. For example, to load the segment address of the **HelloInitCode** resource into `bx`, you would use

```
mov    bx, segment HelloInitCode
```

If you know the resource ID of a segment, you can find out the segment's handle by calling **GeodeGetResourceHandle**. This routine is passed the resource ID and returns the resource's global handle. The call is somewhat faster than the macro **GetResourceHandleNS**, since the macro first determines the resource ID, then calls **GeodeGetResourceHandle**. However, the call is slower than using the `handle` directive, so you should use that when appropriate.

Ordinarily, to find out the segment address of the **dgroup** segment, you would use **GetResourceSegmentNS** or the `segment` directive. However, if you are running code from the process thread, you can take advantage of the fact that the process thread's stack is kept in the **dgroup** resource. This



Esp Basics

* 36

means that the **dgroup** segment address must be in **ss**. Thus, to load to segment address of **dgroup** into **ds**, you could just use

```
push  ss    ; The segmov macro can also do this;
pop   ds    ; see "segmov" on page 49
```

Remember, this only works if the code is being run by the process thread.

■ GetResourceHandleNS

2.3 GetResourceHandleNS <resource>, <reg16>

This macro finds the handle of a resource and loads it into a register.

Pass:	<i>resource</i>	The name of the resource.
	<i>reg16</i>	A 16-bit general-purpose register (<i>not</i> a segment register).
Returns:	<i>reg16</i>	Contains handle of resource.
Destroyed:	Nothing.	

■ GetResourceSegmentNS

GetResourceSegmentNS <resource>, <segreg> [, TRASH_BX]

This routine loads the segment address of a locked or fixed resource into **ds** or **es**. The macro is somewhat faster if you use the TRASH_BX option.

Pass:	<i>resource</i>	The name of the resource.
	<i>segreg</i>	This must be ds or es .
Returns:	<i>segreg</i>	The segment address is loaded into this register.
Destroyed:	If TRASH_BX is passed, bx is destroyed; otherwise nothing is destroyed.	

■ GeodeGetResourceHandle

This routine is passed the resource ID of a resource. It returns the resource's handle.

Pass:	bx	Resource ID number.
Returns:	bx	Resource handle.
Destroyed:	Nothing.	

2.3.3.3 Declaring Static Variables

Esp has slightly different conventions for declaring variables than MASM does. In Esp, you do not need to use the "db", "dw", or "dd" reserved words



when declaring variables (though you certainly may). Instead, you can simply use one of Esp's predefined data types, or define one of your own. The Esp syntax for declaring a variable is

```
[<variableName>]  <dataType>[.<typePointedTo>] \
                                     [<initValue>]
```

variableName

This may be any suitable label; acceptable names for variables are the same as in MASM.

2.3

dataType

This may be one of the standard Esp data types (see Table 2-1 on page * 22). It may also be a structure or record, or any other geode-defined data type.

typePointedTo

If **dataType** is a pointer, you can specify what data type it points to. If you do not, the pointer is untyped (i.e. it is a "void pointer").

initValue

This may be any value appropriate for the data type.

To declare an array of any data type, simply use the following format:

```
[<variableName>]  <dataType>[.<typePointedTo>] \
                                     <n>dup(<init>)
```

variableName

This is actually the label of the first element in the array, i.e. the element at the lowest memory location.

n

The number of elements in the array.

init

The initial value of each element in the array. If you have an init value of "?", all bytes will be set to zero.

If you want to give each element a different initial value, you can use the following format:

```
[<variableName>]  <dataType> <initValue> ,
                                     <initValue>...
```

In this case, each comma can be followed by any amount of whitespace or newlines. The last element in the array is simply the one not followed by a comma.

For example, to declare an array of words, one might use



```
myByteArray      word  1, 2, 3, 4
```

Note that if the variable is in the **udata** pseudo-segment, any specified initializers will generate a link-time error.

2.3.3.4 Strings

2.3

Esp provides a special format for declaring arrays of byte-sized values (*strings*). A sequence of characters surrounded by single or double quotes is treated like a comma-separated sequence of the ASCII values. (No null terminator is added.) For example,

```
myString char    "abc"
```

is functionally equivalent to

```
myString char    61h, 62h, 63h
                  ; ASCII values of a,b,c
```

This is only valid if the data type is byte-sized (**db**, **sb**, or a synonym). If the data type is larger, all of the characters are written to one variable.

You can mix the two formats. For example, to declare a null-terminated string, you can use

```
myString char "abc", 0
```

Characters within a string are translated into their ASCII counterparts, with two exceptions, namely delimiter characters and escape sequences. The delimiter character marks the end of the string, except when it is doubled; in that case, it represents the delimiter character itself. For example, the declaration

```
myString char    "ab" "cd"
```

is equivalent to

```
myString char    61h, 62h, 22h, 63h, 64h
                  ; 22h is ASCII for "
```

If the string is bound by double-quotes, single-quote characters are treated literally. If it is bound by single-quotes, double-quote characters are treated literally. For example,

```
myString char    "ab" "cd'ef"
```



is equivalent to

```
myString char    'ab"cd''ef'
```

Both of these describe strings which contain the following characters:

```
ab"cd'ef
```

Certain character sequences (called *escape sequences*) are used to specify special characters. Esp supports the full range of C escape sequences; these are shown in Table 2-2 on page * 39.

2.3

2.3.4 Miscellaneous Enhancements

Many of Esp’s features are general enhancements of MASM. Our engineers simply felt that a given behavior was useful or preferable to the ordinary MASM behavior. In most cases these changes are backwards-compatible; Esp simply adds new directives and pseudo-ops besides those provided with MASM. In a few cases, it changes the behavior of existing directives and pseudo-operatives. In these cases, you can usually force MASM behavior by passing the “-m” flag to Esp.

Table 2-2 Esp Escape Sequences

Character Sequence	Description
\n	newline (ASCII 10)
\r	return (ASCII 13)
\b	backspace (ASCII 8)
\f	formfeed (ASCII 12)
\t	tab (ASCII 9)
\\	backslash
\'	Single-quote
\"	Double-quote
\000	ASCII code in octal
\x00	ASCII code in hexadecimal



2.3.4.1 Pseudo-Ops and Directives

Esp provides a wide range of pseudo-ops and directives. Some of these will be described in later chapters; a few of the most useful will be described here. This section also details those Esp instructions which are different from their MASM equivalents.

call

2.3

As noted earlier, Esp adds special functionality to the `call` instruction. The main change is that `call` automatically locks movable resources when necessary. This is transparent to the application.

`call` can also be used to call statically-defined methods. This is discussed at greater length in “Messages”, section 3.3.2 of chapter 3.

push and pop

As noted earlier, these instructions can take multiple operands. The operands to `push` are pushed from left to right; that is,

```
push ax, bx, [wordVariable]
```

expands to

```
push ax
push bx
push [wordVariable]
```

The operands to `pop` are popped from right to left. This means that you can pass arguments to `push` and `pop` in the same order, e.g.

```
push  ax, bx, cx, dx
call  MessyProcedure    ; this trashes ax-dx
pop   ax, bx, cx, dx    ; this restores them
```

The TYPE Operator

In Esp, `TYPE <register>` returns the size of the register (in bytes), not zero as in MASM. To find out if an operand is a register, use the `.TYPE` operator.



The .TYPE Operator

Under Esp, bit seven of the .TYPE return value is clear if the expression has local scope (i.e. it uses one or more symbols which are not available outside of the current assembly); if all symbols of the expression are of global scope, bit seven is set.

If you use .TYPE with a code-related expression, the high byte is set thus:

Table 2-3 .TYPE high-byte return values 2.3

Position	Meaning if set
8 (10h)	Procedure is near
9 (20h)	Procedure contains ON_STACK symbols
10	Procedure may not be jumped to
11	Procedure may not be called
12	Procedure is a static method
13	Procedure is a private static method
14	Procedure is a dynamic method
15	Procedure is a method

LENGTH and SIZE

The LENGTH and SIZE operators are used to find the number of elements in an array and the total size of the array in bytes, respectively. In MASM, these operators only work if a variable is declared with the dup directive. In Esp, these are more versatile. If several variables of the same class are declared on a single line after a label, they are treated as an array. For example, suppose you have the declaration

```
SomeNums    dw    1,2,3
```

MASM would not recognize that this is an array; it would therefore say that SomeNums has a LENGTH of one and a SIZE of two. Esp would treat this as an array, and would thus recognize that SomeNums has a LENGTH of three and a SIZE of six.



.assert

`.assert` is used to check assumptions about code. If the assumption is false, `.assert` prints an error message to **stderr** and halts assembly. If the assumption is true, assembly continues normally, and the object code is not affected. `.assert` has the following format:

```
.assert    <expression> [ , <errorString>]
```

2.3

expression If this expression evaluates to zero, the assertion will fail, and assembly will halt.

errorString

If the assertion fails, this string will be printed to **stderr**, along with the location of the assertion. If no string is specified, Esp will print “assertion failed”.

For example, suppose you need to check whether `al` contains a certain value, such as `MY_COLOR_WHITE`. The canonical way to do this would be

```
cmp    al, MY_COLOR_WHITE
jz     itsWhite
```

You might know, however, that `MY_COLOR_WHITE` is the first member of the enumerated type, and has the value zero. You can take advantage of this to write more efficient code, since testing a register for zero-ness is faster than comparing it with an immediate value. On the other hand, this code would be fragile, since the enumerated type could be changed in the future. The solution is to use the `.assert` macro:

```
.assert    (MY_COLOR_WHITE EQ 0), \
           <MY_COLOR_WHITE does not equal zero>

test  al, al      ; Test if al = MY_COLOR_WHITE
                     ; (i.e. zero)

jz     itsWhite
```

You can also use the macro **CheckHack**, described below, which automatically generates an appropriate error message.

ornf, andnf, xornf

Sometimes you will want to use the `and`, `or`, and `xor` macros solely for their effects on the destination operand; you won't care about the settings of the



flags. In these cases, you can use “no-flags” variants, `andnf`, `ornf`, and `xornf`. Esp can take advantage of the fact that you don’t care about the flags to optimize the instructions. For example, the instruction

```
ornf    cx, 0x0100
```

is assembled as

```
or      ch, 0x01
```

which is one byte shorter, but sets the flags differently than “`or cx, 0x0100`” would. For this reason, all the status flags have indeterminate values after a “no-flags” operation.

2.3

The “no-flags” instructions have another advantage: They document that the program doesn’t care about the flag settings after the instruction, i.e. that the code is using the instruction solely for its effect on the destination operand.

EQ, NE

Esp lets you use the EQ and NE directives to compare strings or segments, as well as immediate values. Of course, the operands must be defined at assemble-time.

2.3.4.2 Miscellaneous Macros

Esp comes with a tremendous number of predefined macros. Some of these perform common tasks in a roundabout, but more efficient, way. Others are clearer, self-documenting ways to perform common tasks. When you use Esp macros, you can take advantage of code that has been fine-tuned and checked until it’s practically bulletproof.

All macros are defined in **.def** header files. Since these files are distributed with the SDK, you can examine the source code to see exactly what the macros do and how they work. You can use these macros as starting points for writing your own macros. Some of these are defined in specific libraries; they are usually defined in the library’s **.def** file. This section contains more general-purpose macros, which are defined in **geos.def**.



Assembly-Control Macros

`PrintMessage`, `ErrMsg`, `ForceRef`, `PrintE`, `CheckHack`

Esp provides some macros which do not affect the final code at all. Instead, these macros produce useful side-effects during assembly.

2.3

One such macro is **PrintMessage**. This macro prints a message to **stderr** when it is assembled; it does not have any effect on the object code. This is useful for leaving reminders for yourself. For example, an early version of a program might use an inefficient, brute-force technique to do something. You might then put in a reminder to yourself to improve the algorithm later:

```
call  MyStupidAndSlowSearchRoutine
PrintMessage <Remember to improve this algorithm!>
```

PrintError is much like **PrintMessage**, except that it also generates an `.err` directive, halting assembly.

You may sometimes make assumptions about data structures or values in order to write more efficient code. For example, you might rely on the fact that a given constant is equal to zero. In these circumstances, you should check the assumptions with the **CheckHack** macro. This macro evaluates an expression. If the expression evaluates to *true* (i.e. non-zero), assembly will proceed normally; otherwise, assembly will halt, and an appropriate message will be printed to **stderr**. This is functionally equivalent to using the `.assert` directive, but it is clearer.

For example, the code on page 42 might be rewritten this way with the **CheckHack** macro:

```
CheckHack    <MY_COLOR_WHITE EQ 0>
test  al, al      ; Test if al = MY_COLOR_WHITE
                        ; (i.e. zero)
jz     itsWhite
```

ForceRef makes sure that there is a reference to a symbol. If you declare a symbol (such as a local variable) but never use it, Esp will generate a warning. You can suppress this warning by using the **ForceRef** macro.

PrintE prints the value of an expression when it is assembled. It does not affect the object code in any way.



■ PrintMessage

```
PrintMessage < <string> >
```

This macro prints a message to **stderr** when it is assembled. It does not affect the object code in any way.

Pass: *string* A string to print to **stderr**. The string should be surrounded by angle-brackets, not quotation marks.

Include: **geos.def**

2.3

■ PrintError

```
PrintError < <string> >
```

This macro prints a message to **stderr** when it is assembled, then generates a **.err** directive, halting assembly.

Pass: *string* A string to print to **stderr**. The string should be surrounded by angle-brackets, not quotation marks.

Include: **geos.def**

■ ForceRef

```
ForceRef <symbol>
```

This macro forces a reference to a symbol. This prevents Esp from generating a “symbol not referenced” warning.

Pass: *symbol* Any global or local symbol.

Include: **geos.def**

■ PrintE

```
PrintE < <string> > %( <expr> )
```

PrintE prints the value of an expression to **stderr**. It does not affect the assembled object code in any way.

Pass: *string* A string to print to **stderr**. The string is surrounded by angle-brackets, not by quotation marks.

expr An expression.

Include: **geos.def**

■ CheckHack

CheckHack <expr>

This macro checks to see if an expression is true. If the expression is false (i.e. evaluates to zero) at assemble-time, **CheckHack** prints an appropriate error message to **stderr** and generates a **.err** directive, halting assembly.

Pass: *expr* An expression whose value is known at assemble-time.

2.3

Include: **geos.def**

2.3.4.3 Useful Miscellaneous Macros

`clr, tst, BitSet, BitClr, segmov, segxchg, CmpStrings, XchgTopStack`

You will find that there are certain simple tasks you perform over and over again. For example, you will often find yourself clearing registers, or copying values from one segment register to another. Esp provides macros to perform many of these common tasks.

These macros are useful for two reasons. First of all, they are reliable and heavily-tested ways of performing common tasks as efficiently as possible.

Second, and more important, they are self-documenting. For example, suppose you need to clear **ax**. The fastest way to do this is

```
xor    ax, ax
```

However, this code is confusing. First of all, an inexperienced programmer would not immediately recognize that the instruction clears **ax**. Second, it's unclear what the programmer wants this instruction to do. On the one hand, perhaps the programmer is only interested in clearing **ax**; on the other, she may be relying on `xor` to set the flags appropriately. If you don't know exactly what the programmer wanted to do, it's hard to maintain the code.

On the other hand, if the programmer used the **clr** macro like this:

```
clr    ax
```

the code becomes much clearer: The programmer wanted to clear **ax**, and does not care about the flags (since `clr` is documented as destroying the flags).



clr

Suppose you need to clear a memory location or a register. There are three different ways you might do this.

If you know that a register's value is zero, you can copy that register to the location to be cleared. This is the fastest way to clear any location.

If you need to clear a location and you don't have a convenient clear register, you can `mov` an immediate value of zero into it. This is the usual way to clear a memory location. 2.3

You can also clear a location by `xor`'ing it with itself. If the location is a register, this is faster than moving a zero into it. On the other hand, if the location is in memory, it is faster to move a zero into it.

The macro `clr` automatically chooses between these three techniques. It can take any number of byte- or word-sized arguments. It proceeds down the list from left to right. If the first argument is a register, `clr` clears this register by `xor`'ing it with itself. It then copies this register to all the other arguments to `clr`. If the first argument is a memory location, it moves a zero into this location, then starts over with the next argument.

Note that the `xor` technique changes the status flags; therefore, the status flags become undefined after use of `clr`. If you need to preserve the flags, move an immediate value of zero into each location, or save the flags on the stack.

■ clr

```
clr      <location> [, <location>...]
```

This macro sets all of its arguments to zero, using the most efficient technique for each location.

Pass: *location* A byte- or word-sized memory location or general-purpose register.

Destroyed: flags

Tips & Tricks: If any of the arguments is a register, put it at the head of the list. In particular, if any of the arguments is `ax`, put it at the head of the list, ahead of any other registers.

Include: **geos.def**



tst and tst_clc

You may often need to check a value to see if it's non-zero. There are two different efficient ways to do this.

2.3

If you are testing a register, the most efficient technique is to `or` the register with itself. This does not change the operand, and it sets `ZF` appropriately. On the other hand, if you are testing a memory location, the most efficient technique is to `cmp` the location with zero. This also sets the `ZF` appropriately. The `tst` macro chooses the appropriate technique for its operand.

Note that either one of these techniques will always clear `CF`. If you are taking advantage of this, you should use the synonymous macro `tst_clc`. This macro behaves identically to `tst`, but documents that the program relies on `CF` being cleared.

■ `tst, tst_clc`

```
tst      <location>
tst_clc  <location>
```

This macro tests a byte- or word-sized location to see if it is equal to zero.

Pass: *location* A byte- or word-sized memory location or general-purpose register.

Returns: `ZF` Set according to *location*'s value.
 `CF` Cleared.
 `SF` Set according to the operand's value.

Destroyed: Other flags

Tips & Tricks: If you take advantage of the fact that this macro clears `CF`, you should document this by using the `tst_clc` version.

Include: **`geos.def`**

Moving Values Between Segment Registers

The `mov` instruction does not allow you to move values from one segment register directly to another. Esp provides the macro `segmov` to do this. This macro takes either two or three arguments. It can be called with two arguments, a source segment register and a destination segment register. In this case, `segmov` pushes the value from the source and pops it into the destination. It can also be called with a third argument, a general-purpose register. In this case, `segmov` uses the general-purpose register as an



intermediate register. This makes the operation much faster, but destroys the value in the intermediate register; the instruction is also two bytes longer.

To exchange two segment registers, use `segxchg`. This macro pushes both segment registers, then pops them in the same order, thus exchanging their contents.

■ **segmov**

2.3

```
segmov <destSeg>, <sourceSeg> [, <useReg>]
```

This macro copies a value from one segment register to another. If a general-purpose register is passed as a third argument, it will be used as an intermediate register, making the macro much faster, but two bytes longer.

Pass: *destSeg, sourceSeg* Any segment registers.

Returns: *destSeg* Set to equal *sourceSeg*.

Destroyed: *useReg* (if passed).
All flags are preserved.

Include: **geos.def**

■ **segxchg**

```
segxchg <seg1>, <seg2>
```

This routine exchanges the contents of two segment registers. It does not have any other effects.

Pass: *seg1, seg2* A segment register.

Returns: *seg1, seg2* Exchanged.

Destroyed: Nothing; all flags are preserved.

Include: **geos.def**

Setting and Clearing Bits in a Record

You will often find yourself setting and clearing bit flags in a record. `Esp` provides macros to do this for you. The macros are no more efficient than doing it by hand, but they are clearer to read.



To set a bit in a record, call **BitSet**. This macro is passed the location of the record and the name of the field to set (without the `mask` operator). It sets the bit by or'ing the two values. For example,

```
BitSet      myRecord, MR_A_FLAG
```

is equivalent to

```
orrf      myRecord, mask MR_A_FLAG
```

2.3

To clear a bit in a record, use the **BitClr** macro. This macro is passed the location of the record and the name of the field to clear (without the `mask` operator). It sets the bit by and'ing the destination with the bitwise not of the flag.

■ BitSet

```
BitSet      <location>, <fieldName>
```

This macro turns on all the bits in the specified field of a record.

Pass: *location* The location containing the record; this may be a general-purpose register, or it may be in memory.
 fieldName The name of the field to set. All bits in this field will be set.

Destroyed: Flags are destroyed.

■ BitClr

```
BitClr      <location>, <fieldName>
```

This macro turns off all the bits in the specified field of a record.

Pass: *location* The location containing the record; this may be a general-purpose register, or it may be in memory.
 fieldName The name of the field to clear. All bits in this field will be cleared.

Destroyed: Flags are destroyed.

2.3.4.4 dword Macros

```
cmpdw, jgedw, jgdw, jledw, jldw, tstdw, pushdw, popdw,  
notdw, negdw, incdw, decdw, movdw, adddw, adcdw, subdw,  
clrdw, shrdw, sardw, shldw, saldw, xchgdw
```

The 80x86 chips provide instructions for performing arithmetic on byte- and word-sized operands. You may, however, be working with dword-sized (32-bit)



values. Esp provides many macros for dealing with these values, whether they are in registers or in memory.

These macros are designed to look and behave much like their byte- and word-sized counterparts. However, there are often small differences between the macros and the instructions. For example, many dword macros set the flags slightly differently from the corresponding instructions. The reference entries detail any such differences. Remember, when in doubt, you can always look at the macro's source code.

2.4

2.4 Defining Classes

Every application defines at least one new class, its own process class. Most applications define several more classes in addition to the process class.

When you create a class, there are two things you must do. You must put the class's class structure in the application's **idata** segment; and you must define the class's messages and instance data fields. (You may also need to define special structures, enumerated types, etc., for the class.)

Note that if you wish to create instances of your class at compile time, you will have to do this in a **.ui** file, and you will have to write an Espire definition of your class (in the **.ui** file) which matches the Esp one. The "Espire" language and the User-Interface compiler are discussed in "The UI Compiler," Chapter 4.

2.4.1 Defining a Class

Every class needs to be defined. The class's definition must be included once, and only once, in the compilation, before the class name is ever actually used (e.g. before you create the class structure). You can do this by putting the class definition high in the application's **.asm** file, or (if there are several **.asm** files) by putting it in a common **.def** file.

A class's definition has this basic format:



```
<className>      class <superClassName> \  
                  [, master [, variant]]  
                  ; class's messages...  
                  ; class's instance data fields...  
                  ; class's vardata fields...  
<className>      endc
```

className This is the name of the class you are defining.

2.4

superClassName

This is the name of the class's immediate superclass.

For an example of a class definition, see Code Display 2-4 on page * 54.

2.4.1.1 Defining a Class's Messages

In Esp, you specify very little when you define a class's messages. You simply specify the message name, without arguments or other information, like this:

```
<msgName>      message
```

msgName This is the name of the message.

When you send the message, it is your responsibility to load the correct arguments into the appropriate registers, or push them on the stack, as described in section 3.3.2.2 of chapter 3; Esp will not do any type-checking.

You can export or import messages in Esp, much as you can in Goc (as described in section 5.4.1.1 on page 151 of "GEOS Programming," Chapter 5 of the Concepts Book). To export a range of message numbers, to be used by subclasses, you use this directive:

```
<messageRangeName>      export <numToExport>
```

messageRangeName

This is the name of the message range to export. A subclass which wishes to use the exported range will use this name to import it.

numToExport

This is the number of messages to export.

To "import" a message, i.e. define a message in a message range which was exported by your class's superclass, define the message like this:



```
<messageName>      message <exportedRangeName>
```

exportedRangeName

This is the name of the message range exported by your class's superclass.

2.4.1.2 Defining a Class's Instance Data Fields

To define a class's instance data fields, put lines with this format in your class definition: 2.4

```
<fieldName>      <fieldType> [<defaultValue>]
```

fieldName This is the name of the instance data field.

fieldType This is the type of the instance data field. It may be any standard or application-defined data type.

defaultValue

This is the default value of the field when an object of this class is instantiated.

2.4.1.3 Defining a Class's Vardata

To define a hint or vardata field for a class, put lines with this format in your class definition:

```
<varFieldName>      vardata      [<fieldType>]
```

varFieldName

This is the name of the hint or vardata field.

fieldType This field is optional; it is the type of data associated with the vardata field. It may be any standard or application-defined data type.

2.4.2 Creating a Class's Class Structure

Once you have defined a class, you must create its class structure. The class structure must be in fixed memory; therefore, it is generally placed in the application's **idata** "resource", which means it will be in the application's **dgroup** resource at run-time.



Esp Basics

* 54

To create a class structure, put the following line in your application's **idata**:

```
<className>          [mask <ClassFlag> \
                        [or mask <ClassFlag>]*]
```

className This is the name of the class.

ClassFlag This is a member of the **ClassFlags** record (e.g. CLASSF_NEVER_SAVED); you may have zero or more of these or'd together.

2.4

Code Display 2-4 Creating a Class

```
; Here we create a subclass of GenTriggerClass. Note that if we wanted to create
; any of these objects at startup, we would have to put a corresponding definition
; in the application's .ui file.
```

```
MyTriggerClass  class GenTriggerClass
```

```
; Here are the class's messages:
```

```
MSG_MT_DO_SOMETHING_CLEVER      message
;
;      Pass:          cx = freeble factor
;                      dx = coefficient of quux
;      Return:        ax = # of roads a man must walk down
;      Destroyed:     cx, dx
```

```
; Here are the class's new instance fields:
```

```
        MTI_fieldOne    byte
        MTI_fieldTwo    MyStruct      <0, 17, "Frank T. Poomm">
```

```
; Here are the object's vardata fields:
```

```
GT_MY_VARDATA_FIELD    vardata lptr
```

```
MyTriggerClass  endc
```

```
; We also have to create the class's class structure. We do this in the idata
; resource:
```

```
idata  segment
```

```
MyTriggerClass
```

```
idata  ends
```



2.4.3 Defining your Process Class

Every application with a process thread needs to define a new process class for its process object. This is much like defining any other class. There are a couple of differences, however.

Process objects do not have vardata, and they do not have ordinary instance data. Notionally, all the variables in the **dgroup** segment are the process object's instance data. In fact, while you must create a class structure for the process object (as described in section 2.4.1 on page 51), you do not need to define the process object (with `class... endc`) unless you are defining messages for your process class.

2.5

2.5 Error-Checking Code

`ERROR_CHECK, ERROR, ERROR_C, ERROR_NC, ERROR_Z, ERROR_NZ...`

Error-checking is as important in assembly code as in Goc. Esp provides error-checking facilities which are very much like those of Goc. It allows you to write code which will only be run by the error-checking version of your geode. It also provides many routines and macros which are useful for checking for errors.

There are two main ways to designate code “error-checking”. If you want to declare a single line as “error-checking,” you should bracket the line with “`EC<...>`”, like this:

```
EC<    call    MyECValidationRoutine>
```

In the error-checking version of the code, this line will be included as an ordinary instruction; in the non-error-checking version, the line will be stripped out. (To include a line only in the *non*-error-checking version, bracket the line with “`NEC<...>`”).

When the compiler is compiling error-checking code, it defines the flag `ERROR_CHECK` to non-zero. You can use this to designate several lines as error-checking code:



Esp Basics

* 56

2.5

```
if      ERROR_CHECK
    ; bx should be non-zero; is it?
    pushf
    tst   bx
    jnz   noError
; if we reach this, it's an error
    ERROR MY_FATAL_ERROR_CODE
noError:    ; not an error condition
    popf
endif
```

Esp also provides several macros for error-checking. There are a few macros and routines of general usefulness and they are documented here.

There are many macros which call **FatalError**, passing an error number. The most basic is `ERROR`. This macro is called with a single argument, namely an error number. It generates a fatal error; the error code is available for the debugger.

There are similar macros which call **FatalError** if the flags are set in a particular way. For example, `ERROR_C` checks to see if the carry is set. If it is (that is, if a `jc` instruction would jump), `ERROR_C` calls `ERROR` with the specified code; otherwise, it continues normally. Conversely, `ERROR_NC` calls `ERROR` if `CF` is *not* set. For example, the code sample on page 56 could be written more clearly like this:

```
if      ERROR_CHECK
    ; bx should be non-zero; is it?
    pushf
    tst   bx
    ERROR_Z MY_FATAL_ERROR_CODE
    popf
endif
```

There is an `ERROR_` macro to correspond to every conditional jump instruction except `jcxz`. For example, there is an `ERROR_GE`; this macro calls **FatalError** in those situations in which `jge` would jump.



■ ERROR

`ERROR` `<errorNumber>`

This macro generates a fatal error.

Pass: *errorNumber* This is an error code for use by the debugger.

Returns: Nothing.

Destroyed: Everything.

Include: **ec.def**

2.5

■ ERROR_C, ERROR_NC, ERROR_Z, ERROR_NZ...

`ERROR_x` `<errorNumber>`

These macros call `ERROR` if the status flags are set in a particular way. Each of these macros corresponds to a conditional jump instruction (`ERROR_x` corresponds to `jx`); the macro calls `ERROR` in those situations in which the corresponding conditional jump instruction would jump. (For example, `ERROR_C` calls `ERROR` in those situations in which `jc` would jump, i.e. when **CF** is set.) There is one such macro for every conditional jump instruction except `jcxz`.

Pass: *errorNumber* This is an error code; it is passed to `ERROR` if the error condition occurs.

Returns: Nothing.

Destroyed: Nothing (unless the error condition occurs, in which case everything is destroyed).

Include: **ec.def**

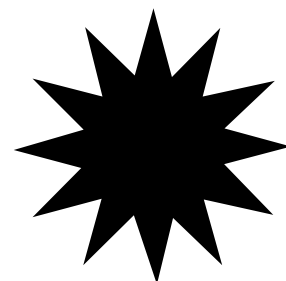
Esp Basics

* 58

2.5



Routine Writing



3

3.1	GEOS Conventions	61
3.1.1	Parameters and Local Variables	63
3.1.1.1	Initializing Local Variables	64
3.1.2	The “uses” Directive	66
3.1.3	.enter and .leave	66
3.1.4	Inheriting a Stack Frame.....	69
3.2	LMem Heaps and Chunks	70
3.3	Objects and Classes.....	73
3.3.1	Object Structure	74
3.3.2	Messages.....	77
3.3.2.1	Handling Messages.....	77
3.3.2.2	Sending Messages.....	81
3.3.2.3	ObjMessage	82
3.3.2.4	Other Ways of Sending Messages	90

This chapter deals with some of Esp's special features, and describes how to write routines (and message-handlers) in Esp.

Not all GEOS programmers will use Esp in the same way. Most will, of course, not use it at all, using Goc instead. Some will write a few heavily-used routines in Esp, or perhaps write an Esp library which will be called by one or more applications which are written in Goc. And some will write entire applications in Esp.

3.1

This chapter describes how to write Esp routines for a variety of purposes. It describes conventions for writing both Esp routines and message-handlers, as well as special Esp conventions and techniques which differ significantly from their Goc counterparts.

3.1 GEOS Conventions

GEOS has certain presumptions about how routines behaves. If you write your code to follow these conventions, Esp and Swat can work together to make writing and debugging much simpler than they are with other assemblers.

Some of these conventions have been described earlier. To recap: You should never change `ss`; the kernel does this automatically when switching between threads. You should never change `if` or `tf`. You may set `df`, but you must clear it before calling any routine or returning. You may not load an invalid segment address into `ds` or `es`.

Most routines are passed their arguments in registers. If arguments are passed on the stack, the routine being called should pop them off the stack when returning. (Esp automatically does this if the routine declares its arguments properly.) If a routine uses local arguments, it should set `bp` to point to the base of the stack frame; it can then access the arguments with a displacement from `ss:[bp]`. Again, Esp can do this automatically.



Routines can have local variables as well as arguments. The local variables are kept on the stack immediately below the return address. As with arguments, they are accessed with a displacement from **ss:[bp]**.

For example, suppose the near routine **HelloProc** is passed three arguments on the stack: **AnInt**, **AChar**, and **AnOptr**. **HelloProc** itself declares two local variables, **LocalInt** and **LocalOptr**. The calling routine pushes the three arguments on the stack, then calls **HelloProc**. The **call** pushes the return address on the stack (Figure 2-1.a).

HelloProc immediately pushes **bp** on the stack and copies the current value of **sp** to **bp**. **bp** now points to the base of **HelloProc**'s stack frame. **HelloProc** then subtracts six from **sp**, making room for its local variables (Figure 2-1.b).

When **HelloProc** is ready to return, it copies **bp** to **sp**, thus removing everything from the stack up to the saved **bp**. It then pops **bp** off the stack; **bp** now has the value it had when **HelloProc** was called. Finally, it executes

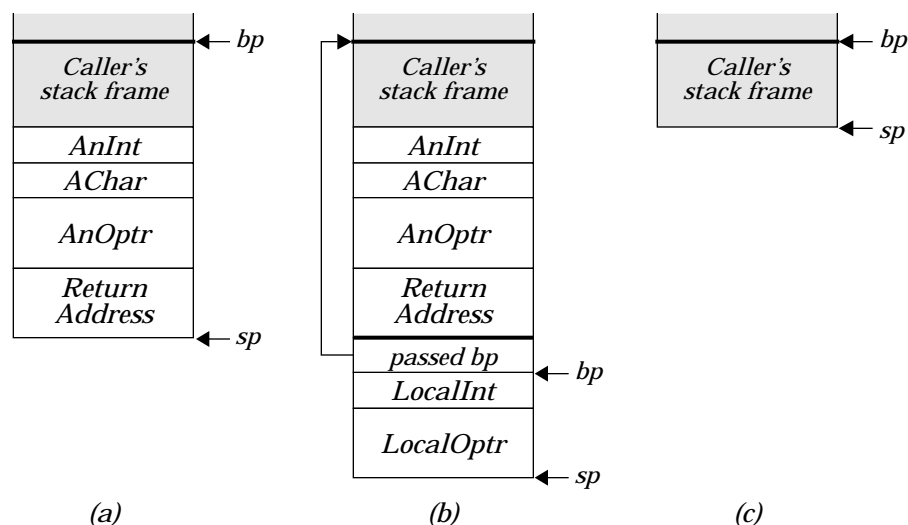


Figure 2-1 Stack conventions for GEOS routines

- (a) This is the stack's setup when **HelloProc** was called, but before **HelloProc** did anything to the stack.
- (b) This is the stack after **HelloProc** has finished setting up the stack frame.
- (c) This is the stack after the **ret** instruction is executed. The **ret** pops the return address and the parameters off the stack.



a `retf 8`, which pops the arguments off the stack and returns. (Note that you can simply use the “`ret`” pseudo-op; Esp will automatically expand it to a `retn` or `retf` which pops all local arguments off the stack.)

As noted above, Esp does most of the bookkeeping for you. You need simply declare your arguments and local variables, then use the `.enter` and `.leave` directives. Esp will automatically set up the stack as necessary.

Passed parameters are always word-aligned. This is because they are pushed on the stack by the caller, and `push` always pushes a word-sized values. Local variables, on the other hand, are not necessarily word-aligned; Esp allocates just enough space for the local variables. (It will, if necessary, add a padding byte, so the total amount of space used by local variables remains word-aligned.)

3.1

3.1.1 Parameters and Local Variables

Esp makes it easy to use local variables and parameters. You need simply declare them in the beginning of a procedure; Esp will automatically set up the stack frame.

Parameters are declared on the same line as the procedure name. The first line of a procedure therefore has the following format:

```
<ProcName> proc (near|far) \
    [<paramName>:<paramType> \
    [, <paramName>:<paramType>]*]
```

ProcName The name of the procedure. The name’s scope is the entire assembly.

near|far Either “near” or “far”. Near routines may only be called from within the same resource, but there is slightly less overhead when calling or returning.

paramName The name of a parameter. This name is a label whose scope is limited to the procedure.

paramType The type of the parameter.



You may specify the order of the parameters with the `.model` directive. If you use `".model pascal"` (the default), parameters must be declared in the order in which they appear on the stack (that is, the first argument declared is the first one to be pushed on the stack). If you use `".model cdecl"`, the *last* parameter declared is the first one to be pushed on the stack. Assuming we use the Pascal convention, **HelloProc** (described in Figure 2-1), would have the following first line:

3.1

```
HelloProc proc far      AnOptr:optr, AChar:char, \
                        AnInt:int
```

Local variables are declared immediately after this line. Each local variable has the following format:

```
<varName>    local <varType>
```

varName The name of the local variable. The scope of this variable is limited to the procedure.

varType The variable's type.

As noted above, passed parameters are word-aligned, since they must be pushed on the stack by the caller. Local variables, on the other hand, are not aligned. Local variables are not initialized, unless you explicitly specify an initial value (as described below).

You can use the name of a parameter or local variable to access it. The name is equivalent to an effective-address displacement from `ss:[bp]`. The name is therefore valid only if `bp` points to the start of the stack frame.

If you use parameters or local variables, you must use the `.enter` and `.leave` directives. `.enter` expands into the instructions necessary to create the stack frame, as well as the aliases for the variable names. `.leave` destroys the stack frame, restoring `sp` and `bp` to their values at the time `.enter` was used.

3.1.1.1 Initializing Local Variables

If you wish, you can specify that local variables be initialized. When `Esp` builds the stack frame (i.e. at the `.enter` instruction). The initialization is done with a "push" instruction; therefore, you can only initialize variables with values that can be pushed (i.e. 16-bit registers, memory locations, or 16-bit immediate values).



There are some rules for initializing local variables:

- ◆ *All* of the initialized variables must be declared before *any* of the uninitialized variables.
- ◆ Each of the initialized variables must be a whole number of words in size. You must push enough values to precisely fill the variable.
- ◆ The values are pushed after the stack frame is set up; therefore, you cannot initialize a local variable to contain `bp` (except as described below). If you want to copy the old `bp` to a local variable, initialize it to contain `ss:[bp]` (the location of the stored `bp`). If you want to copy the new `bp` (i.e. the current frame pointer) to a local variable, you must do it by hand after the `.enter` instruction.

3.1

To initialize the variable, put a `push` instruction at the end of the declaration, on the same line. Like any GEOS `push`, the instruction may have multiple arguments; they will be pushed in order, from left to right (therefore the first argument given will be the highest word of the variable).

For example, a procedure might have the following declarations:

```
HelloProc    proc near
AnOptr       local optr    push bx, di
AMemValue    local int     push ds:[GlobalVariable]
AByte        local byte    ; not initialized
AnotherInt   local int     ; not initialized
```

When `Esp` set up the stack frame, it would take steps equivalent to the following instructions:

```
push  bp           ; Set up the stack frame
mov   bp, sp       ; (ss:[bp] = base of frame)
push  bx           ; Initialize AnOptr: high word
push  di           ; low word
push  ds:[GlobalVariable] ; copy mem. location
                                ; to local variable
sub   sp, 4        ; Leave room for AByte and
                                ; AnotherInt (and add one byte, so
                                ; local vars are word-aligned)
```

There is a special case for initializing local variables. The *first* local variable may be initialized with `"push bp"`. (If it is a multi-word variable, the high



word, and only the high word, may be initialized with `bp`.) In this case, `bp` is not actually pushed on the stack. Instead, that variable's location is made to be `ss:[bp]`; that is, the variable is another name for the location containing the stored `bp`. You may read the passed `bp` from this variable; similarly, if you write a value to this variable, that value will be returned in `bp`. This is useful if you need to return a value in `bp`. If you want a local variable to contain the passed `bp`, but you do not want to change `bp` on return, you should push-initialize the variable with `ss:[bp]`, as described above.

3.1

3.1.2 The “uses” Directive

Many routines will need to preserve the state of some or all of their registers. Esp provides for this with the `uses` directive. This directive is used to specify a list of registers that should be pushed at the start of the routine, and popped at the end. Uses has the following format:

```
uses <reg> [, <reg>]*
```

`uses` must be used in conjunction with `.enter` and `.leave`. The registers will be pushed at the point where `.enter` is used; they will be popped where `.leave` is used.

3.1.3 .enter and .leave

Esp provides several conveniences for writing routines. It can automatically save registers, set up stack frames, and set up local arguments. You can signal when to do this by using the `.enter` and `.leave` directives. Ordinarily, the `.enter` directive is put right after the local variable declarations, and before any actual code. The `.leave` directive is put just before the `ret` directive.

`.enter` sets up a stack frame if necessary. It will only do this if the routine declares parameters or local arguments (see section 3.1.1 on page 63). Setting up a stack frame entails pushing `bp` on the stack and copying the current `sp` to `bp`. `.enter` also pushes any registers declared with “uses”. If you use the `.enter` directive, you must use it before you push anything on the stack, try to access any of the parameters or local variables, or change any registers which must be preserved.



If you use a `uses` directive, you must use the `.enter` and `.leave` directives. `.enter` will push all named registers after it sets up the stack frame. `.leave` will pop the registers before it destroys the stack frame.

To inherit a stack frame, put an `inherit` instruction after the `.enter` instruction. This is discussed in section 3.1.4 on page 69.

Note that if you use local variables or stack parameters, `.enter` and `.leave` will automatically preserve `bp`; that is, `bp` will have the same value after `.leave` as it had before `.enter`. If you need to return a value in `bp`, you should set up a local variable to hold the stored `bp`, as described in section 3.1.1.1 on page 64. (You can also copy the return value to `bp` *after* the `.leave` instruction destroys the stack frame, but before the `ret`.)

3.1

Code Display 2-5 Writing a Procedure

; This shows how `Esp` automatically sets up the stack frame to accommodate local variables and passed parameters, and to preserve registers. The procedure is the **HelloProc** described in section 3.1 on page 61.

```
COMMENT @%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    HelloProc
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

SYNOPSIS: This shows how `Esp` automatically sets up the stack frame to accommodate local variables and passed parameters, and to preserve registers. The procedure is the **HelloProc** described in section 3.1 on page 61.

CALLED BY: `HelloOtherProc`

PASS: stack: AnInt (pushed first)
 AChar
 AnOptr

RETURN: ax = freeble factor

DESTROYED: nothing

SIDE EFFECTS: none

PSEUDO CODE/STRATEGY:

Set up the local variables, then call `HelloInnerProc` to do the dirty work

REVISION HISTORY:



Routine Writing

* 68

Name	Date	Description
----	----	-----
Frank T. Poomm	4/1/93	Initial version

%%%

```
HelloProc proc far      anOptr:optr, aChar:char, anInt:int
```

```
uses bx, cx, dx
```

```
3.1 localInt      local int      push ax
    localOptr     local optr
```

```
.enter
```

```
        call HelloInnerProc      ; Calculate freeble factor; put it in LocalInt
                                      ; (HelloInnerProc presumably inherits the stack
                                      ; frame from HelloProc)
    mov     ax, localInt          ; Return the freeble factor
```

```
.leave
```

```
ret
```

```
HelloProc endp
```

```
; Esp would expand this to code like this:
```

```
    ; Set up stack frame
push    bp          ; preserve value of bp
mov     bp, sp      ; Set bp to point to stack frame
push    ax          ; Initialize LocalInt...
sub     sp, 4        ; ...and leave enough uninitialized space for an optr

    ; Set up names of parameters/local variables
anOptr     equ ss:[bp+6]    ; All of these names have local scope.
aChar      equ ss:[bp+6][4]
anInt      equ ss:[bp+6][6]
localInt   equ ss:[bp][-2]
localOptr  equ ss:[bp][-6]

    ; Preserve registers specified in "uses" line
push     bx          ; Esp recognizes that bp was preserved when the
push     cx          ; stack frame was set up, so it does not push bp
push     dx          ; again here.

call     HelloInnerProc

mov     ax, ss:[bp][-2] ; This copies the "localInt" variable into ax
```



```
; Restore the registers & destroy the stack frame
pop     dx
pop     cx
pop     bx
mov     sp, bp ; This pops the stack frame
pop     bp     ; This restores bp

; Return, freeing passed parameters
retf    8

HelloProc     endp
```

3.1

3.1.4 Inheriting a Stack Frame

You may write a routine which is called only by a single other routine. For example, if you were writing a sort routine, you might write several comparison routines, each of which is only called by the sort routine. In this situation, you might want to let the routine use local variables and parameters that belong to its caller. You can do this by having the routine *inherit* the stack frame of its caller.

If you inherit a stack frame, Esp assumes that **ss:[bp]** points to the caller's stack frame (i.e. that **bp** was set to point to that routine's stored **bp**). Esp declares variable names in the local namespace which indicate appropriate displacements from the frame pointer. If a routine inherits a stack frame, it may not (of course) create a stack frame of its own; that is, it may not have any of its own local variables or passed parameters.

The simplest way to inherit a stack frame is with a directive of this kind:

```
.enter inherit    <routineName>
```

routineName

This is the name of the routine whose stack frame will be inherited. This routine must be in the same assembly, and under certain circumstances it must already have been assembled before the inheritor is assembled.

Esp will automatically declare the names of the variables and parameters in the local namespace. As always, it is your responsibility to keep **ss:[bp]** pointing to the stack frame.



3.2

A routine which inherits a stack frame may specify what local variables and parameters it expects to find in that frame. It does this by declaring the local variables and parameters exactly the way other routines declare their own local variables and parameters. Esp will assume that these correspond to the variables in the stack frame, and will declare those local variables appropriately. Indeed, you need not even use the name of the routine from which the frame is inherited; you can simply use the reserved words “near” or “far”. (Esp needs to know whether the inherited frame is from a near or far procedure in order to figure out how much space the frame uses for the return address.)

For example, suppose you have the declaration

```
HeirProc    proc near    callerParamOptr:optr,\
                        callerParamInt:int

    callerLocalInt    local int
    callerLocalOptr    local optr

    .enter            inherit near
```

As noted, Esp will not change `bp`; it assumes that `ss:[bp]` already points to the base of the stack frame. Esp will declare the variables in the local scope; that is, `callerParamOptr` will refer to `ss:[bp+4]`, `callerParamInt` will refer to `ss:[bp+4][4]`, `callerLocalInt` will refer to `ss:[bp][-2]`, and `callerLocalOptr` will refer to `ss:[bp][-6]`. Esp makes no guarantees that these locations hold meaningful values; it is your responsibility to make sure that you are describing the actual stack frame.

Note that if a routine declares a parameter or local variable, but never accesses it, Esp will generate a warning at assembly time. If you declare a variable that is used only by other routines which inherit the stack frame, you can disable the warning by using the **ForceRef** macro.

3.2 LMem Heaps and Chunks

It is worth paying special attention to Local Memory heaps as they are used in Esp. The heaps themselves are always the same, whether they are created and accessed from Esp code or from Goc code; however, they are manipulated in a somewhat different way.



First, a review of the basics of LMem is in order. A local memory heap is held within a single “global” block. The block is actually divided into two main portions, a handle table and the heap itself. Chunks are assigned in the “heap” portion of the block. A chunk’s location within a heap may change from time to time. For example, when *any* chunk is allocated or expanded in an LMem heap, *every* chunk in the heap may be moved to accommodate the change. Thus, the address of a chunk is very volatile. However, every chunk has a *handle* which is unchanging.

3.2

The location of the LMem heap itself is also volatile. Whenever a chunk is allocated or expanded, the LMem heap may have to be expanded to make room for it. This can cause the LMem heap to move on the global heap.

In Goc, chunk handles are treated as opaque tokens. To get the address of a chunk, you call **LMemDeref()**, passing the global handle of the LMem heap and the chunk handle of the chunk. **LMemDeref()** returns the address of the chunk, as a far-pointer (i.e. segment:offset). You end up having to do this over and over; whenever you allocate or resize a chunk, the pointers to every other chunk are invalidated.

In Esp, chunk handles are not treated as opaque. A chunk handle is a near-pointer; that is, it is an offset from the beginning of the LMem heap. The chunk handle is a pointer to an entry in the LMem heap’s handle table. The entry contains the offset from the beginning of the heap to the chunk itself (See Figure 2-2 on page * 72). Whenever you need to get the address of the chunk, you simply use the chunk handle to find the current offset of the chunk.

The Esp versions of most LMem routines require you to pass the segment address of the LMem heap in **ds**. Some routines may cause the block to be moved. These routines will generally fix **ds** so it still points to the heap after the call, even if the heap moved. Furthermore, if **es** was the same as **ds** at the time of the call, the routines will generally fix **es** as well. Some other routines (notably **ObjMessage**) fix these registers if you pass a special flag instructing them to do so. You should check the reference for any routine which can shuffle an LMem heap to see if it automatically fixes these registers. Remember, if you store the segment address of the heap, that stored address will not be fixed up; you will have to fix the stored address by hand.



3.2

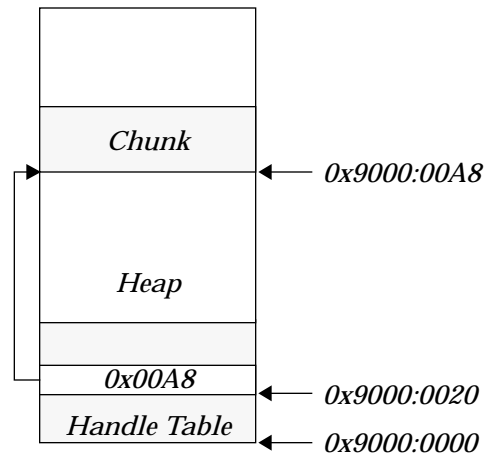


Figure 2-2 Structure of an LMem Heap

The LMem heap is divided into a handle table and the heap proper. The chunk handle is an offset to an entry in the handle table; that entry contains the offset to the chunk itself.

In this example, we assume that the heap is at location 0x9000; we wish to find the chunk with handle 0x0020.

Code Display 2-6 Using an LMem Heap

```
; This is an example of how to use an LMem heap in Esp. In this example,
; chunkHandleOne, chunkHandleTwo, and heapHandle are word-sized local variables.
; At the start of this code, chunkHandleOne and heapHandle are already set;
; chunkHandleTwo contains no meaningful value.

; First, we want to get access to the chunk specified by chunkHandleOne:

    mov     bx, heapHandle           ; bx = global handle of LMem heap
    call    MemDerefDS              ; ds:[0] = LMem heap
    mov     si, chunkHandleOne       ; ds:*si = chunk
    mov     si, ds:[si]              ; ds:[si] = chunk

; ds:[si] is now the address of the chunk. We can read from or write to the chunk
; at will. Now we want to allocate another chunk. ds still has the segment address
; of the LMem heap.
```



```

    clr     al                      ; Clear all object flags
    mov     cx, MY_CHUNK_SIZE      ; cx = size of chunk
    call    LMemAlloc              ; ax = chunk handle of new chunk
    mov     chunkHandleTwo, ax     ; store the new handle

; Note that the call to LMemAlloc may have moved the LMem heap. LMemAlloc
; automatically fixes ds (and, if appropriate, es); however, if I'd stored the
; address of the heap, the stored address would now be invalid.

; Now I want to look at the first chunk again. However, LMemAlloc can shuffle the
; heap, so I need to dereference the handle again:
    mov     si, chunkHandleOne
    mov     si, ds:[si]

```

3.3

These principles apply wherever LMem heaps are used. For example, DB items are stored in *item blocks*, which are a kind of LMem heap. When you lock a DB item, you are given the item's chunk handle, as well as the segment address of the item block. You must dereference the chunk handle the same way you would any other chunk handle.

Object Blocks are also LMem heaps. Whenever a message is sent to an object, that object may be resized; this can shuffle the block, or cause the object block to move on the global heap.

3.3 Objects and Classes

One big difference between Esp and other assemblers is Esp's support for object-oriented programming. OOP is at the heart of GEOS, and Esp is designed to make all the power of OOP available without sacrificing the efficiency of assembly language.

Using objects and classes is simple. The issues can be divided into three categories: Declaring objects; creating new classes; and handling messages. Each issue will be treated in its own section. This section also contains a review of the structure of GEOS objects, and how objects are handled in Esp.



3.3.1 Object Structure

This section is a review of how GEOS works with objects. Before you read this, you should be familiar with “GEOS Programming,” Chapter 5 of the Concepts Book.

3.3

An object’s instance data is stored as a chunk in an object block. An object block is a special kind of LMem heap. Whenever an instance chunk is created or expanded, all the other chunks in that object block may be moved, and the block itself may move on the global heap. When this happens, you can get the address of the instance chunk by dereferencing the chunk handle, as with any chunk.

The instance data for objects is divided into *master groups*. Master groups are discussed in section 5.3.2.2 of chapter 5 of the Concepts book.

In order to gain access to the instance data for an object, you have to find the offset to the master section containing that class’s instance data. For example, suppose you need to examine the *HT_myDatum* field of an instance of **HelloTriggerClass**. **HelloTriggerClass** is subclassed from **GenTriggerClass**, which is itself subclassed from **GenClass**. **GenClass** is the first master class above **HelloTriggerClass**; thus, all the instance data defined by **GenClass**, **GenTriggerClass**, and **HelloTriggerClass** is in the same master group.

Esp automatically defines *HT_myDatum* as the displacement from the start of the master group to the *HT_myDatum* field. In order to actually examine the field, you have to know where the master group begins. The displacement to the master section can change during the life of the object; however, you can find the displacement by looking at a specific place in the object. To allow you to find the displacement, Esp defines a constant instructing you where to look. In this case, the constant would be called `HelloTrigger_offset`. This constant says where in the object you should look to find the displacement to the master group containing **HelloTriggerClass**’s instance data. (In this case, `HelloTrigger_offset` is the same as `GenTrigger_offset` and `Gen_offset`.) To actually examine the field, you would take the steps shown in “Finding a Class’s Master Section”, Code Display 2-7.



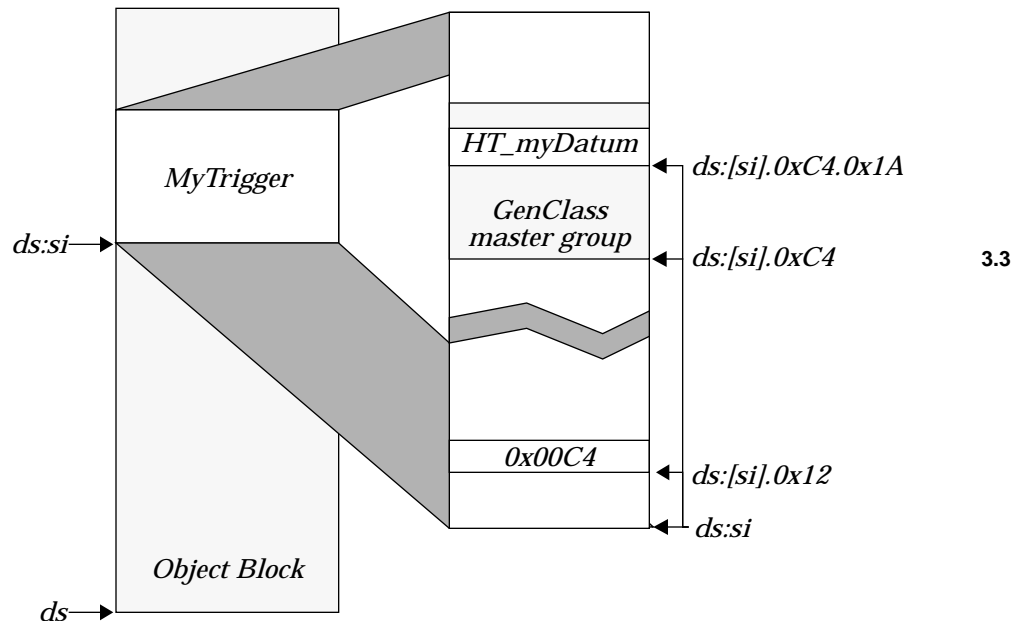


Figure 2-3 Finding an Instance Datum in a Master Section

We need to find *HT_myDatum*, a field in *HelloTriggerClass*. The first master class above *HelloTriggerClass* is *GenClass*, so the field will be in the *GenClass* master section.

In this example, we assume that *ds* points to the beginning of the object block, and *ds:[si]* is the beginning of the instance chunk. *HelloTrigger_offset* is assumed to be *0xA2*. *HT_myDatum* is assumed to be *0x1A*; that is, *HT_myDatum* starts *0x1A* bytes from the beginning of the master section.

Code Display 2-7 Finding a Class's Master Section

```
; We want to examine an instance datum of an object. In particular, we want to
; load the word-sized HT_myDatum field of a HelloTrigger object into ax. ds:[bx]
; points to the beginning of the object's instance chunk.

; First, we need to get the address of the master section. As noted, ds:[bx]
; points to the beginning of the instance chunk.
    mov     di, ds:[bx].HelloTrigger_offset
```

Routine Writing

* 76

```
; Now, ds:[bx][di] points to the beginning of the master section. To load the
; datum, we add a displacement to this:
    mov     ax, ds:[bx][di].HT_myDatum
```

3.3

When a message handler is called, it is given a pointer to the beginning of the instance chunk. However, whenever an object does anything which could shuffle the heap, it should redetermine this address, as described above.

If there is no master class in an object's class hierarchy, all instance data will be located at a constant offset from the start of the chunk. In this case, the name of an instance field will evaluate to the displacement from the beginning of the chunk to the field. Esp will not define constants of the form *MyClass_offset*; you can simply use the name of the instance data field as the displacement from the start of the instance chunk.

For example, if **GoodbyeClass** is a subclass of **MetaClass** (and is not itself a master class), and *G_aDatum* is an instance data field in that class, then *G_aDatum* will evaluate to the displacement from the beginning of the instance chunk to the data field.

If you're writing code for an "masterless" object, it's a good idea to use assertions to make the code more robust. For example, let's suppose we want to load *G_aDatum* into **ax**. The normal approach would be this:

```
; ds:[si] points to start of instance chunk
mov     ax, ds:[si].G_aDatum
```

There is one danger with this code. In a future rewrite, **GoodbyeClass** or one of its superclasses might be made into a master class. In this case, the code would be incorrect; *G_aDatum* would be the displacement from the start of the master group, not from the start of the instance chunk. For this reason, it's a good idea to use Esp directives to make sure that **GoodbyeClass** has no master class. One way to do this is to check that *Goodbye_offset* is undefined:

```
ifdef Goodbye_offset
    .err  <GoodbyeClass now has a master offset!>
endif
mov     ax, ds:[si].G_aDatum
```



If any class in **GoodbyeClass**'s class hierarchy becomes a master class, *Goodbye_offset* will become defined, and this code will generate an error.

3.3.2 Messages

Writing code for objects is much like any other coding. Indeed, almost all GEOS code is run by one object or another; either it is a message handler (or *method*), or it is a routine which is run by a method, directly or indirectly. There are a few conventions which apply specifically to methods; other than that, methods can be treated just like other routines.

3.3

3.3.2.1 Handling Messages

Writing methods (also known as “message handlers”) is little different from writing other Esp procedures. This section describes GEOS's conventions for handling messages, and how to write methods. “Sending Messages”, section 3.3.2.2 on page 81, describes how to send messages.

A method looks much like any other routine. However, its first line (the line containing the method's name) is slightly different:

```
<MethodName>      method <static|Dynamic> <Class> \
                   <MsgName> [ , <MsgName>]*
```

MethodName

This is the name of the method which will handle the message. By convention, the method's name is a combination of the class name and the message name; for example, the **HelloTriggerClass** handler for MSG_GEN_TRIGGER_SEND_ACTION would be named **HelloTriggerSendAction**. Nevertheless, you may give a method any name you could give to a routine.

static|dynamic

This is one of the reserved words *static* or *dynamic*. The difference between static and dynamic methods is discussed on page 80.

Class

This is the class for this method.

MsgName This is the message which activates this method. A single method may handle several different messages. Note that the message ID is passed to the method in **ax**, so a method can easily determine at run-time which message was sent.

Calling Conventions

3.3

Every message has its own calling conventions. These conventions specify such things as what values are to be passed in the “argument” registers (or on the stack), under what circumstances the message should be called, and what values will be returned (if the message is “called”, as opposed to “sent”). These conventions depend on how the message will be used.

If you are adding a new message for a class you have defined, you can set these conventions as you see fit. You should simply document what the method’s conventions are, so all code which sends the message will do it properly.

If you are *subclassing* a message (i.e. writing a new handler for a message which was defined by one of your object’s superclasses), you will have to adhere to existing conventions. You may also have to call the superclass if you want the existing behavior to be preserved. You can do this by calling **ObjCallSuperNoLock**. This routine is generally called either at the beginning or at the end of the method.

Register Usage

Methods are generally called indirectly; that is, a process sends a message by calling a kernel routine (as described in “Sending Messages” on page 81). and the kernel performs necessary bookkeeping, then calls the method. When a method is called, certain registers are set up automatically. If the recipient object is not a process object, the registers are set up like this:

ds	This register contains the segment address of the recipient’s object block.
es	This register contains the segment address of the block containing the recipient’s class definition.
si	This contains the chunk handle of the object. (Together, ds:si constitute the optr of the recipient object.)



bx	This is the offset to the beginning of the instance chunk; that is, ds:[bx] is a pointer to the instance chunk. (Thus, bx is the same as the value pointed to by ds:[si] , and *ds:[si] = ds[bx] .)	
di	If the class's instance data is in a master section, di will be the offset from the start of the block to the master section. If the class has no master, di will be the offset to the start of the instance chunk; that is, di will be the same as bx . In either case, you can find instance data fields by using a displacement from ds:[di] .	3.3
ax	This is the message number. Sometimes a single method will handle several different messages; such a method can examine ax to figure out precisely which method was sent.	
cx, dx, bp	These registers are passed intact from the message sender. See, however, "Passing Arguments On the Stack" on page 86.	

Note that some of the values passed in the registers are very volatile. For example, anything that can change the object block can potentially invalidate **ds** (since the block itself may be moved), as well as **bx** and **di** (since the chunks may be moved around within the block. Note in particular that if you send a message to any object in the same block, that can change the recipient's size, thus invalidating these registers (unless the message is passed on the queue). In these cases, you can recalculate the values for the registers with a few simple steps. Most routines which can move an object block (such as **ObjMessage**) can be instructed to fix **ds** when necessary. As to the other registers, you can fix them by dereferencing the chunk handles, like this:

```

; bx and di were invalidated
mov  bx, ds:[si]          ; ds:[bx] = instance chunk
mov  di, bx
add  bx, MyClass_offset; or whatever the class
                           ; offset is named

```

This assumes that method's class is in a master group. If this is not the case, remove the last line.

Any message handler is allowed to destroy **bx**, **si**, **di**, **ds**, and **es** (though **ds** and **es** must always contain valid segment addresses, as discussed in section 2.2 of chapter 2); if the message was "sent", the kernel will restore these



registers before returning execution to the calling routine. If the message was “called”, **ax**, **cx**, **dx**, and **bp** are returned to the sender of the message.

Passing Arguments on the Stack

3.3

Methods are ordinarily passed their arguments in registers. In some cases, you may want to pass more data on the stack. GEOS permits this. A method should specify whether it expects to be passed data on the stack; it is the responsibility of whoever sends a message to be sure that the data is passed properly.

Note that the message sender may use a different stack than the recipient does. In this case, the kernel will automatically copy the data to the recipient’s stack. The recipient may not make any assumptions about where in the stack the data is; it might be far from the top of the stack.

If data is passed on the stack, some of the registers have different meanings:

ss:bp	This will point to the lowest byte of the data passed on the stack. This may be anywhere on the stack; you may not make any assumptions about where on the stack the data is.
dx	This contains the number of bytes passed on the stack.
cx	This is passed through intact from the sender, just like always.

(Note that Goc does not support passing arguments both in **cx** and on the stack. Therefore, if the message may be sent from Goc code, you must assume that **cx** contains garbage.)

Esp does not define variable names for the passed parameters. You may declare these names by hand. Most commonly, a structure is passed on the stack; you can then access the fields of the structure by using the field names as displacements from **ss:[bp]**.

Static and Dynamic Methods

Most methods are called only by the kernel. Usually, you run a method by sending the appropriate message to an appropriate object. This allows GEOS to arrange the methods however it wishes within a resource. Such methods are said to be *dynamic* in nature.



If you wish, you may declare a method to be *static*. You may call a static method the same way you call any other routine. However, the kernel will not set up any registers before or after the method is called. If you make a call to a static method, the registers will be passed, as-is, to the method, and will be returned, as they are, from the method. You must therefore set up any registers that need to be passed to the method, and see to it that you preserve any important registers that the method destroys.

Process Class Messages

3.3

The application's process object, like any other object, may handle messages. Because the process class is a special kind of object, some of the conventions are slightly different. In particular, two of the registers passed have different values:

ds	This holds the segment address of the geode's dgroup resource. dgroup is notionally the "instance data" for the process object.
si	This register is passed intact from the message sender to the method. Thus, you may use this register for another passed parameter.

The other registers have the usual values.

3.3.2.2 Sending Messages

By now you should be very familiar with sending messages to objects. However, there are enough differences between Esp and Goc in how they handle messages that a review is appropriate.

Messages are sent to objects. Every object is run by a thread; that is, the object's methods are executed by that thread and use the thread's stack, as are the routines called by those messages. Every thread maintains a queue of messages which have been sent to that thread's objects from outside the thread. If you are running under thread A, and you send a message to an object which is run by thread B, the message will ordinarily be put at the end of thread B's queue. If you send the message to an object which is run by thread A, the calling routine will block while the message is handled, after which it will resume; again, this behavior can be overridden, forcing the message to be put on the queue.



Remember that when an object receives a message, it may be changed; for example, it might have to build out master sections of its instance data. This means that the instance chunk might grow, and therefore the object block might be shuffled. The object block might even be moved on the global heap. If the message is sent via the queue, you needn't worry about this. However, if the sender blocks until the message is handled—whether because the message is sent within the thread, or because it is sent as a “call-and-return”—there is a danger that the recipient object block might be shuffled or moved while the sender is blocked.

3.3.2.3 ObjMessage

ObjMessage, MessageFlags

The basic way of sending a message in Esp is with the routine **ObjMessage**. This routine is the counterpart to the Goc commands **@call**, **@send**, et al.

You specify the recipient of the message by passing the object's optr in **bx:si**. (If you are sending a message to a process object, you can pass the process ID in **bx**; in this case, **si** is passed intact to the method.) Pass the message ID in **ax**. **cx**, **dx**, and **bp** are passed intact to the method; you can use these to pass arguments to the message.

di contains a set of **MessageFlags**. This is a word-sized record that specifies how **ObjMessage** should behave. The record has the following fields:

- | | |
|----------------|--|
| MF_CALL | If this bit is set, the caller will block until the message is handled, even if the message is being sent to another thread. When ObjMessage returns, ax , cx , dx , and bp will contain whatever values they had when the method returned. |
| MF_FORCE_QUEUE | If this bit is set, the message will be sent via the thread's event queue, even if the message is being sent within a thread. This bit is incompatible with MF_CALL. |
| MF_STACK | If this bit is set, the caller is passing parameters to the method on the stack. ss:bp must point to the last argument pushed on the stack, and dx must contain the number of bytes passed on the stack. See “Passing Arguments On the Stack” on page 86 below. |



MF_RETURN_ERROR

If this bit is set, **ObjMessage** will indicate an error in sending the message by returning a **MessageError** value in **di**; if there is no error, **di** will be set to **MESSAGE_NO_ERROR**. If it is not set, **ObjMessage** will call **FatalError** if an error occurs.

MF_CHECK_DUPLICATE

This bit may be set only if **MF_FORCE_QUEUE** is also set. If this bit is set, **ObjMessage** will check if there is a matching message on the recipient's message queue (i.e. a message with the same message ID which is being sent to the same object, though the passed arguments and registers may be different). If it is, this message will be discarded (unless **MF_REPLACE** is also set). (See "Manipulating Messages on the Queue" on page 89.) This bit is incompatible with **MF_STACK**.

3.3

MF_CHECK_LAST_ONLY

This bit modifies **MF_CHECK_DUPLICATE** to check only the last message in the recipient's queue.

MF_REPLACE

This bit modifies **MF_CHECK_DUPLICATE** such that if a duplicate message is waiting on the event queue, the one on the queue will be discarded, and this message will be placed on the queue.

MF_CUSTOM This bit is reserved for use by the kernel.

MF_DISCARD_IF_NO_MATCH

This bit modifies **MF_CHECK_DUPLICATE** to discard the passed message if there *isn't* a matching message on the queue. It must be passed with **MF_CUSTOM**.

MF_MATCH_ALL

This bit is reserved for use by the kernel.

MF_INSERT_AT_FRONT

If this bit is set and the message is being sent via the queue, the message will be put at the head of the event queue (instead of the tail). This bit may only be set only if **MF_FORCE_QUEUE** is also set.

MF_FIXUP_DS

If this bit is set, **ObjMessage** will fix **ds** so it points to the same block as it did before the call, even if that block moved during

the call. **ds** must point to a memory block the first word of which is the block's global handle (e.g. an object block or other LMem heap).

MF_FIXUP_ES

If this bit is set, **ObjMessage** will fix **es** as per MF_FIXUP_DS. You must pass MF_FIXUP_DS along with this flag.

MF_CAN_DISCARD_IF_DESPERATE

If this bit is set, **ObjMessage** can discard the message if the system is running out of handles.

MF_RECORD If this bit is set, **ObjMessage** will behave like the **@record** Goc command; that is, it will pack the message into a recorded event and return its handle. The message can be dispatched with **MessageDispatch**. The only flag which can be combined with MF_RECORD is MF_STACK. (See "Recording and Dispatching Messages" on page 88.)

ObjMessage may be called with interrupts disabled; however, it will return with interrupts enabled. The status flags will be destroyed. The return values for **ObjMessage** depend on the **MessageFlags** passed.

If MF_RETURN_ERROR is passed, **di** will return a member of the **MessageError** enumerated type. The following errors may be returned:

MESSAGE_NO_ERROR

ObjMessage was able to deliver the message properly. (This does not guarantee that the handler was able to handle the message without error.) MESSAGE_NO_ERROR is guaranteed to be equal to zero.

MESSAGE_NO_HANDLES

The system was running low on handles, and MF_CAN_DISCARD_IF_DESPERATE was passed; accordingly, **ObjMessage** discarded this message.

If MF_CALL is passed, **ax**, **cx**, **dx**, **bp**, and **cf** will return whatever values were left in them by the message handler. The other status flags will be destroyed.

If MF_CUSTOM is passed, the address of the callback routine will be popped off the stack.



If MF_RECORD is passed, the handle of the encapsulated message will be returned in `di`; all other registers will be unchanged.

Calling and Sending

In Goc, there is a clear distinction between sending a message and using a message as a “call”. In Esp, that distinction remains, although the boundaries are a little more fluid.

When a message is sent with the flag MF_CALL, **ObjMessage** behaves like the Goc command **@call**. The sending routine halts until the method has returned. If the message is handled within the same thread, this is almost exactly like a routine call-and-return. If, on the other hand, the message is sent to another thread, the calling thread immediately blocks, and the message is put on the recipient’s queue. When the method returns, the caller’s execution resumes. In either case, as soon as **ObjMessage** returns, you may assume that the message has been handled (if there is a handler for that message).

3.3

If a message is sent with MF_CALL, the method may return values. When **ObjMessage** returns, `ax`, `cx`, `dx`, and `bp` will contain whatever values were in them when the method returned. `CF` will likewise be returned from the method. Note that when the method is called, it is passed `ax`, `cx`, `dx`, and `bp` exactly as they were passed to **ObjMessage**; therefore, if the method does not change these registers, they will be returned intact. (The case is slightly different for `bp` if arguments were passed on the stack; see “Passing Arguments On the Stack” on page 86).

If a message is sent with MF_FORCE_QUEUE, the sender does not block immediately. Instead, **ObjMessage** sets up an event structure for the message and puts it on the recipient thread’s event queue; **ObjMessage** then returns. At some unspecified time in the future, the recipient thread will handle the message. **ObjMessage** will not change any registers.

If neither MF_CALL nor MF_FORCE_QUEUE is passed, the message may or may not go by the queue. If the message is sent within the thread, the sender will block and the message will be handled immediately, bypassing the queue. In this respect, the message is handled much as if MF_CALL were passed; however, no values are returned, and all registers remain unchanged. If the message is sent to another thread, it is added to the recipient’s event queue, exactly as if MF_FORCE_QUEUE were passed.

3.3

As noted above, whenever an object handles a message, the object's block may be compacted or moved. This would invalidate any segment pointers to that block. This can be a problem if the message is sent within a thread, as the sender may have the same object block locked down. You can instruct **ObjMessage** to automatically fix **ds** so it points to the same block after the call as it did before. To do this, pass the flag **MF_FIXUP_DS** to **ObjMessage**. You may pass this *only* if **ds** points to a block whose first word is the block's global handle (i.e. **ds[0]** is the block's global handle). This is true of all LMem blocks (including object blocks). If you pass **MF_FIXUP_DS** and **ds** does not point to an appropriate block, the behavior is undefined.

If you pass **MF_FIXUP_DS**, you may also pass **MF_FIXUP_ES**. This instructs **ObjMessage** to fix **es** as well as **ds**. Both **ds** and **es** must point to blocks which start with the block's global handle (e.g. LMem heaps, including object blocks and DB item blocks); they need not point to the same block.

Note that you cannot fix **es** without also fixing **ds**. If you want to fix **es** but not **ds** (e.g. if **ds** does not point to an LMem heap), you should use the **segxchg** macro to swap **ds** and **es**, then call **ObjMessage** with **MF_FIXUP_DS**, then use **segxchg** again.

Passing Arguments On the Stack

Messages can pass up to three words of data in registers (in **cx**, **dx**, and **bp**). This is enough for most messages. However, some may need to pass more. In this case, the message can pass arguments on the stack.

ObjMessage requires that all arguments passed on the stack be contiguous, but they need not be at the top of the stack. If you pass arguments on the stack, you must set **ss:[bp]** to point to the last argument pushed on the stack, and **dx** must contain the total size of those arguments, in bytes. (**cx** may be used to pass an additional argument to the method.)

If the message is not sent via the queue (i.e. the message is sent within one thread, and is not sent with **MF_FORCE_QUEUE**), the arguments are left where they are on the stack, and **ss:[bp]** is passed through to the method unchanged (as are **cx** and **dx**). However, the arguments might not be anywhere near the top of the stack; for example, kernel routines may set up stack frames before the method is called. Thus, the method can access the arguments with effective addresses, but not by popping them off the stack.



If the message is sent via the queue, the kernel copies all the arguments from the stack into the messages event structure, then places that structure on the queue. When the message comes to the head of the queue, the kernel copies all the arguments onto the recipient's stack, setting `ss:[bp]` to point to the last argument pushed, as before. (Note that this means `bp` may be different when the method is called then it was when the message was sent.) `cx` and `dx` will be left unchanged.

When **ObjMessage** returns, it leaves the stack unchanged. This is useful if the sender will be passing the same arguments on the stack to several different methods.

3.3

Note that when a message is sent to another thread, `bp` might change during the send. When the sender calls **ObjMessage**, `bp` is an offset into the *sender's* stack; when the method is called, `bp` is an offset into the *recipient's* stack. This is different from usual messaging behavior; when `MF_STACK` is not passed, `bp` is always passed to the method unchanged. This presents a problem when a message is sent across threads with both `MF_CALL` and `MF_STACK`. Ordinarily, when `MF_CALL` is passed, `bp` is passed unchanged to the method, and when the method returns, the `bp` is likewise returned to the caller. If the method doesn't want to return a value in `bp`, it can simply leave `bp` unchanged, and the original value will be returned to the caller. If, on the other hand, values are passed on the stack, the method may be passed a different `bp` than the one passed by the sender. In this case, if the method changes `bp`, the kernel will assume that `bp` contains a return value, and will return that value to the sender. If, on the other hand, the method returns the same value of `bp` that it was passed, the kernel will assume that the method did not intend to return a value in `bp`. In this case, it will restore `bp` to the value the sender had in it, i.e. the offset to the passed arguments on the sender's stack. `ax`, `cx`, and `dx` are returned exactly as if `MF_STACK` had not been passed.

For example, suppose **HelloProc** sends a message to **GoodByeTrigger** (run by a different thread), passing arguments on the stack, and using the `MF_CALL` flag. When **HelloProc** calls **ObjMessage**, `bp` contains an offset in **HelloProc's** stack (perhaps `0x0100`). **ObjMessage** copies the arguments onto **GoodbyeTrigger's** stack and sets `bp` appropriately (perhaps to `0x005b`). It then calls the appropriate method. When the method returns, the kernel examines the contents of `bp`. If `bp` is still set to `0x005b`, the kernel will assume that no value is being returned there. It will therefore change `bp`



back to 0x0100 before restarting **HelloProc**. If, on the other hand, it finds that **bp** has been changed (perhaps to 0x0060), the kernel will assume that **bp** contains a return value, and will propagate that value to **HelloProc**; that is, when **ObjMessage** returns, **bp** will have its new value (in this case, 0x0060).

Recording and Dispatching Messages

3.3

`MessageDispatch`, `MessageSetDestination`, `ObjGetMessageInfo`,
`ObjFreeMessage`

You can use **ObjMessage** to encapsulate a message for later delivery (much like the **@record** command in Goc). To do this, call **ObjMessage** with the flag **MF_RECORD**. **ObjMessage** will set up an event structure for the message, recording the values of all the appropriate registers; if arguments are passed on the stack, those arguments will be recorded as well.

ObjMessage will return the handle of the recorded message. You can now change the registers or pop arguments off the stack; all the values will have been stored in the recorded message. If you record a message, the only flag which you may pass to **ObjMessage** (besides **MF_RECORD**) is **MF_STACK**.

To send an encapsulated message, call **MessageDispatch**. This message is passed two arguments. The handle of the encapsulated message is passed in **bx**. A word of **MessageFlags** is passed in **di**. The **MessageFlags** have the same meaning as they do for **ObjMessage**, with one exception: If **MF_RECORD** is passed to **MessageDispatch**, the encapsulated message will be preserved intact, so it can be dispatched again; if this flag is not set, the encapsulated message will be destroyed, and the handle will be freed.

If you pass the flag **MF_CALL** to **MessageDispatch**, the sender will block until the message is handled, and values will be returned in **ax**, **cx**, **dx**, and **bp** (just as with **ObjMessage**). Note that when the method is called, these registers will be set with the values stored in the encapsulated message; thus, even if the method doesn't change these registers, they will end up with different values than when **MessageDispatch** was called. If you do not pass **MF_CALL**, all registers will be preserved intact, as always.

When you encapsulate a message, the destination is stored in the recorded message. If you wish to change the destination, you can call **MessageSetDestination**. This routine is passed two arguments. You must pass the handle of the encapsulated message in **bx**, and the **optr** of the



argument in **cx:si**. The destination of the encapsulated message will be changed accordingly; the old destination will be lost.

To find out information about an encapsulated message, call **ObjGetMessageInfo**. This routine is passed the handle of the encapsulated message. It returns three values:

ax	The message ID.	
cx:si	The optr of the destination object.	3.3
CF	This flag is set if the message has arguments on the stack; otherwise it is cleared.	

When you are done with an encapsulated message, you should free it. You will ordinarily do this by dispatching it without the MF_RECORD flag; **MessageDispatch** will automatically free the event after sending it. If you wish to free an encapsulated message without sending it, call **ObjFreeMessage**. This routine is passed the handle of the message (in **bx**); it does not change any registers.

Manipulating Messages on the Queue

When you send a message via the queue, you can't be sure how long it will be before the message is handled. This can lead to unnecessary duplication. For example, suppose you change a Vis object twice in succession. After the first change, you send it a MSG_VIS_INVALIDATE, which goes on the object's queue. After you make the second change, you send another MSG_VIS_INVALIDATE, which also goes on the queue. If the first MSG_VIS_INVALIDATE has not yet been processed, then the object will be invalidated twice in a row, even though there was no change between the two invalidations; that is, the second invalidation is a waste of time.

To avoid situations like this, you can send a message with the flag MF_CHECK_DUPLICATE. (This flag must be accompanied by MF_FORCE_QUEUE.) If you send this, the messenger will check to see if there is another similar message waiting on that thread's queue. (A message is considered "similar" if it has the same message ID and the same destination, even if the passed arguments are different.) If such a message is waiting on the queue, then the new message will be discarded; it will not be added to the queue.



You can modify this behavior by passing `MF_REPLACE`. If you pass this flag, the latter message will supercede an existing message on the queue; that is, the pre-existing message will be discarded, and the new one will take its place in the queue. (The only difference between the two messages will be in the arguments passed.)

Another way to change the behavior of `MF_CHECK_DUPLICATE` is to pass `MF_DISCARD_IF_NO_MATCH`. If this flag is passed, the kernel will discard the message if there is no matching event on the queue. This flag is generally accompanied by `MF_REPLACE`; in this case, the message will replace an existing message on the queue, but will be discarded if no such message exists. (Note that if `MF_CHECK_DUPLICATE` and `MF_DISCARD_IF_NO_MATCH` is passed but `MF_REPLACE` is not, the message will always be discarded.)

3.3.2.4 Other Ways of Sending Messages

`ObjCallInstanceNoLock`, `ObjcallInstanceNoLockES`,
`ObjCallSuperNoLock`

ObjMessage is a general-purpose routine for sending messages. It makes very few assumptions about the recipient; in particular, it does not assume that the recipient's data is locked in memory. GEOS also provides some special-purpose routines for sending messages. These routines are faster, though they may only be used in certain circumstances. (If they are used improperly, behavior is undefined.)

Objects may often need to send messages to themselves, or to other objects in the same resource. In this case, they can use the routine

ObjCallInstanceNoLock. This routine is considerably faster than **ObjMessage**, because it makes two assumptions about the recipient:

- ◆ The recipient object must be run by the thread which calls **ObjCallInstanceNoLock**.
- ◆ The recipient's object block must be locked or fixed in memory.

Both of these conditions are met when the recipient object is in the same resource as the sender (or the special case of this in which an object sends a message to itself).

ObjCallInstanceNoLock is passed the following arguments:



- ax** This register contains message ID number.
- ds** This register must contain the *segment address* of the recipient's object block, which must be locked on the global heap.
- si** This register contains the recipient's chunk handle.
- cx, dx, bp** These registers are passed through to the recipient method.

ObjCallInstanceNoLock immediately blocks and calls the appropriate method (exactly as if **ObjMessage** had been called with MF_CALL and MF_FIXUP_DS). If you need to pass arguments on the stack, you will have to do this by hand, i.e. set **ss:[bp]** to point to the last argument on the stack, and set **dx** to contain the number of data bytes on the stack. Since the recipient is guaranteed to use the same stack as the sender, the arguments won't have to be copied from one stack to another.

3.3

ObjCallInstanceNoLock returns the following values:

- ax, cx, dx, bp, CF** Returned from the method. If no method is called, the registers are returned unchanged, and **CF** is cleared.
- ds** Fixed up to point to the same object block, even if the block moved during the call.
- es** If this register pointed to the object block, the value is invalidated (since the object block may have moved).

If you will need to have **es** fixed up, call **ObjCallInstanceNoLockES**. This routine is exactly the same as **ObjCallInstanceNoLock**, except that both **ds** and **es** are automatically fixed. If you call this routine, **es** *must* point to some kind of LMem heap (or another memory block in which the first word of the block is the block's global handle).

Esp provides a routine which corresponds to the Goc command **@callsuper**. This command is **ObjCallSuperNoLock**. It is just like **ObjCallInstanceNoLock**, with one exception: The message is handled by the object's superclass, not by its class. **ObjCallSuperNoLock** takes all the same arguments as **ObjCallInstanceNoLock**, and one more besides: **es:[di]** must be the address of the class structure for the object (or any other class in the object's class hierarchy). The message will be handled as if the object belonged to the *superclass* of the class specified by **es:[di]**.



Routine Writing

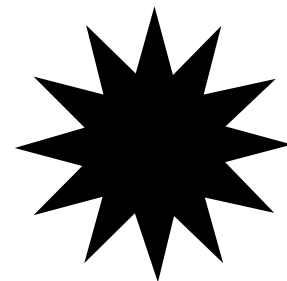
* 92

ObjCallSuperNoLock is most often used by objects sending messages to themselves. If you subclass a message, but want to maintain the default behavior for that message, you can just call **ObjCallSuperNoLock** to call your superclass's handler for the message. In this case, you can usually leave all your registers as they were when your message was called.

3.3



The UI Compiler



4

4.1	UIC Overview	95
4.2	Declaring Classes	97
4.2.1	Declaring Fields.....	98
4.2.2	Changing a Default Value.....	100
4.3	Creating Objects and Chunks	102
4.3.1	Setting Up a Resource	102
4.3.2	Creating Objects	103
4.3.2.1	Setting Up Parent-Child Links	104
4.3.2.2	Hints and Vardata	105
4.3.3	Creating Chunks	105
4.3.4	Creating VisMonikers.....	107

Assembly language is, by nature, very low-level. It does not ordinarily provide support for object-oriented programming. GEOS has had to add special features to its assembly language to fully support OOP.

You have seen some of this support already. The GEOS kernel manages objects, and provides routines for sending messages. Applications can call special routines to create, manipulate, or destroy objects. However, most applications will need to have many objects created before they start running. In particular, most applications will need to have many UI objects created before they start running. You can also declare new object classes in a UIC file, though you only have to do that if you will want to create objects from those classes at compile-time.

4.1

This is where the *User-Interface Compiler* (UIC) comes in. With UIC, you can specify in your source code what objects need to be created when the application is launched. All these objects will be created at compile time, and stored in the executable file. When the application is launched, the objects will be there, ready to be used.

The UIC is most commonly used to create user-interface objects. It can, however, create any kind of object, of any class (whether a GEOS system class, or a class you define yourself). It can also create other chunks that might be kept in an object-block (e.g. Vis monikers).

For assembly reference information about the various GEOS classes, see the PCGEOS\INCLUDE*.DEF and *.UIH files.

4.1 UIC Overview

Essentially, UIC reads files written in Espire, a special object-specification language, and writes special GEOS object-assembly files. These files (which have a **.rdf** suffixes) are automatically included when the executable is compiled.

When you write an application in assembly, you can specify objects by putting them in a UIC source file. This file's name should be **<geodename>.ui**. When



you run **mkmf**, it will automatically set up the makefile to include the proper calls to UIC. When you compile the application, the **.rdf** file will be generated and incorporated in the application.

UIC incorporates the C preprocessor. This means you can use the standard C preprocessing directives. In particular, you can write UIC header files, and include them in a **.uih** file with the **#include** directive. These header files customarily have the suffix **.uih**. Every **.ui** file must include the standard GEOS header file **generic.uih**; this file contains UI information about all the standard GEOS Gen and Vis classes.

Comments follow the C convention; i.e. they begin with **/*** and end with ***/**. As in C, newlines are treated as whitespace, not as statement terminators.

Some of the conventions of **.ui** (and **.uih**) files are different from elsewhere in GEOS. First of all, the names of all classes are shortened; they do not contain the word “class”. For example, in a **.uih** file, **GenTriggerClass** would be called just “GenTrigger”.

Also, in a **.ui** file, the names of instance data fields are truncated; the initial capital letters, followed by an underscore, are removed. For example, the class **GenTriggerClass** has a field named *GTI_destination*; in a **.ui** file, this field would be called just “destination”. This convention is followed with all GEOS classes; you should follow it with any classes you create. (If you’re ever unsure what the Espire field-name is for a class, you can look in the class’s **.uih** file in PCGEOS\INCLUDE\OBJECTS.

Finally, the names of flags in an instance data record have different conventions. In C and assembly files, the name of a flag would begin with the initials of the instance data field, followed by the name of the flag, in capital letters. For example, in assembly, the **GenClass** record *GI_attrs* has a field named *GA_READ_ONLY*. The corresponding field in a **.ui** file would be called “readOnly”. Members of enumerated types have similarly altered names. For example, in assembly, a field might be called *MET_AN_ENUM_VALUE*; if the type were declared in a **.uih** file, the member would be called “anEnumValue”.



4.2 Declaring Classes

You can create new classes by specifying them in your **.ui** file. You do this by writing special Espire directives; these tell Esp how to create objects of that class. These directives are often put in a **.uih** file which is included by the application's **.ui** file.

You don't always have to do this for every class. In particular, you don't have to declare your process class in the **.uih** file; and you don't have to declare a class if you will not need to create objects of that class at compile-time. However, if you will want to create objects from the class's subclass at compile time, you must declare both the class and the subclass in your **.uih** file.

4.2

If you specify a class in your **.uih** file, you must still declare it in your regular assembly code, as described in section 2.4 of chapter 2.

The class specification begins with a line like this:

```
class <classRoot> = <superClassRoot> [, master]
                                [,variant]{
```

classRoot This is the name of the class you are declaring, *without* the word "class". For example, if you are declaring **MyTriggerClass**, the **<classRoot>** would be **MyTrigger**.

superClassRoot

This is the name of the class's superclass, again without the word "class"; e.g. if the superclass is **GenTriggerClass**, this would be **GenTrigger**.

If the class is a master class or a variant class, you must specify that on this line, e.g.

```
class MyMasterVis = Vis, variant {
```

would be the first line in the specification for master class **MyMasterVisClass**, which is subclassed from **VisClass**.

After the top line, you specify all the instance data fields for the class. You may also change default values for fields inherited from the class's superclass.



4.2.1 Declaring Fields

You must list all the instance data fields that are added with that subclass. The basic format for specifying a field is:

```
<fieldRoot> = <fieldType> : <defaultValue>
```

4.2

fieldRoot This is the name of the instance data field, without the acronym of the class (as discussed in section 4.1 on page 95). For example, if the field is called *MCI_aField* in the MASM file, it would be called “aField” here.

fieldType This may be a simple type or a defined type. Simple types are the same as in MASM, except that they end with “Comp”; for example, the MASM type “byte” corresponds to the Espire type “byteComp”.

defaultValue This optional field specifies the default value of the instance data field. If you create an object of this class in a **.ui** file and do not specify a value for the field, the default value will be used.

For example, in MASM code, the class **GenViewControlClass** has a field with the following definition:

```
GVCi_scale          word    100
```

The Espire definition of the class, in a **.uih** file, has this corresponding line:

```
scale = wordComp : 100
```

The default value can also be an expression:

```
myField = byteComp : (3 * 20)
```

If the field contains an enumerated type, the format is this:

```
<fieldRoot> = enumComp <size> [( <first> [, <step> ] ) ]  
    { <member>, <member>... } : <default>;
```

fieldRoot This is the name of the instance data field, without the acronym of the class (as discussed in section 4.1 on page 95). For example, if the field is called *MCI_aField* in the MASM file, it would be called “aField” here.



size	This is the size of the enumerated type. It may be <code>byte</code> , <code>word</code> , or <code>dword</code> .
first	If this is present, it specifies the value of the first member of the enumerated type. The default first value is zero.
step	If this is present, it specifies the step between successive members of the enumerated type. The default step is one.
member	This is the name of a member of the enumerated type. The name is altered from its assembly form, as noted in section 4.1 on page 95; for example, if the member's name in assembly is its name in Espire will be "blueEnum". You must list all members of the enumerated type, in the same order in which they appear in the type's assembly specification.
default	This specifies the default value of the instance field.

4.2

If the field contains an record, the format is this:

```
<fieldRoot> = bitFieldComp <size>
               {<field>, <field>...}
               : <default>, <default>...;
```

fieldRoot	This is the name of the instance data field, without the acronym of the class (as discussed in section 4.1 on page 95). For example, if the field is called <i>MCI_aField</i> in the MASM file, it would be called "aField" here.
size	This is the size of the record. It may be <code>byte</code> , <code>word</code> , or <code>dword</code> .
field	This is the name of the flag. The name is changed from the MASM form, as noted above. For example, if the flag (in MASM) is called <code>MBF_A_BITFIELD_FLAG</code> , in Esp it would be called "aBitFieldsFlag".
default	This may be one or more flags in the record. By default, the flags listed here will set, and all the other flags will be cleared.

A field may be more than one bit wide. If a field in the record is specified like this:

```
<field>:<width>
```

then **width** will be the width of the field in bytes. Fields in a record may also contain a range of enumerated values. The field would be specified like this:

```
<field>:<width>={<value>, <value>...}
```



Each **value** would be a possible setting for that field.

If a field is more than one bit wide, you specify its default value with “<fieldname> <value>”; the value may be either an integer, or the enumerated type specified for that field. If you do not specify a default value, the field will default to zero.

For example, the object **GenDocumentControl** has a field with the following definition:

4.2

```
dcAttributes = bitFieldComp word {
    multipleOpenFiles,
    mode:2 = {viewer, sharedSingle,
              sharedMultiple},
    dosFileDenyWrite, vmFile, native,
    supportsSaveAsRevert, documentExists,
    currentTask:4 = {none, new, open,
                    useTemplate, saveAs, dialog},
    doNotSaveFiles }
: mode sharedSingle, vmFile, supportsSaveAsRevert,
currentTask new;
```

In this case, each field in the record is one bit wide, except for *mode*, which is two bits wide, and *currentTask*, which is four bits wide. By default, *mode* is set to *sharedSingle* (i.e. 1), and *currentTask* is set to *new* (i.e. 1); the flags *vmFile* and *supportsSaveAsRevert* are set; and all other flags are cleared.

4.2.2 Changing a Default Value

When you create a class, you may wish to change the default values of instance fields inherited from a superclass. The format for doing this is:

```
default <fieldRoot>      = <value>;
```

fieldRoot This is the name of the instance data field, as given in the superclass’s Espire declaration.

value This is the new default value for that field. As noted, if you want the default value to be interpreted by MASM, you should surround it in quotes, like so:

```
default      superField = "6 * SOME_MASM_CONSTANT";
```



If the field is a record, you may wish to turn on or off some of the flags, while leaving the rest unchanged. You can do that with a line like this:

```
default <recordRoot> = default + <flagName>,
                        - <flagname>... ;
```

recordRoot

This is the name of the instance data field, as given in the superclass's Espire declaration.

flagName This is the flag to turn on or off. If the flag is preceded by a "+", the flag's default value will be *set*; if it is preceded by a "-", its default value will be *clear*.

4.2

For example, the line

```
default superRecord = default +aFlag, -anotherFlag;
```

changes the default value of the superclass's field *superRecord*. In the subclass's *superRecord* field, *aFlag* is now on by default, and *anotherFlag* is off. All the other flags have the same default values as they have in the superclass.

Code Display 3-1 Modifying a Superclass

```
/* We are creating a subclass of GenTriggerClass. This class will have a new
 * field, and will change the default values of one of GenTriggerClass's fields.
 */

#include "generic.uih" /* This has the Espire definition of GenTriggerClass */

Class MyTrigger = GenTrigger {

/* Change the default values of a fields: */
    genStates = default +enabled;

/* And add a new field */
    myDatum      = wordComp : 0;

}

/* The .def file would have the corresponding Esp class definition; this would be
 * something like:

MyTriggerClass  class GenTriggerClass
    MTI_myDatum      word
```



```
MyTriggerClass  endc
```

4.3 Creating Objects and Chunks

4.3

The whole point of the UIC is that it lets you create objects in your geode's source code, instead of having to instantiate them at run-time. You can specify whole object-blocks, with a complete set of parent-child linkages, in your source file; the compiler will turn these into GEOS blocks.

Besides specifying objects, you can specify other chunks that should go in an object block. For example, you may want to put some text into a chunk in an object block; that way, a resource editor can modify the text (if e.g. you are translating the application for another country). You may also set up data resources, i.e. LMem heaps that contain chunks, but no objects.

4.3.1 Setting Up a Resource

```
start, end
```

Every object must be in an object block. Non-object chunks may be in object blocks, or they may be in non-object resources (i.e. LMem heaps). You can create these resources with the **start** and **end** directives. Every object in a **.ui** file must be between a **start** and the corresponding **end**.

The **start** and **end** directives have the following format:

```
start <resourceName> [, <resourceFlag>];  
/* object definitions... */  
end <resourceName>;
```

resourceName

This is the name of the resource. The first time you “start” that resource, UIC outputs control information for the LMem heap. You may start and stop a resource several times in a **.ui** file.

resourceFlag

This is one of the three words “data”, “ui”, or “app”. “data”



indicates that the block contains only non-object chunks. “ui” indicates that the resource is an object block which should be run by the user-interface thread. “app” indicates that the resource is an object block which should be run by the application thread.

A single resource may “start” and “end” many times in a **.ui** file. Thus, you can group your object declarations in whichever order is clear or convenient, instead of being forced to group them by resource.

4.3

4.3.2 Creating Objects

Creating objects in Espire is simple. You just specify the name of the object, and the initial settings for any fields which do not have the default settings. The UIC translates this into appropriate Esp directives.

The basic format of an object definition is:

```
<objName> = <className> [<ObjChunkFlag>]* {
    /* instance data...*/
}
```

objName This is the name of the object.

className This is the name of the object's class, as defined in the **.uih** file.

ObjChunkFlag

This may be one or more flags of the **ObjChunksFlags** bitfield, specified with Espire conventions. This is typically either **vardataReloc** or **ignoreDirty**, or both.

The object's class must have been defined in a **.uih** file, which must have been included. If it is a GEOS standard class, you can simply include the file **generic.uih**.

You need not specify all instance data fields for the object. If you do not specify a field, the field will have its default value.

To initialize a field, put in a line like

```
<fieldName> = <value>;
```

fieldName This is the name of the instance data field, as specified in the class's Espire specification.



value If this is a value which must be interpreted by Esp (not UIC), surround it with "double quotes". For example, suppose the field's value is a constant which is only known by the assembler (perhaps because it's defined in a **.def** file). You would then surround the constant with double quotes:

```
myField      = "MFC_CONSTANT_QUUX_FACTOR";
```

4.3

You can turn on or off certain bits in a record, while leaving the rest of the flags in their default settings. You do this in much the same way you do it when specifying classes, i.e.

```
<record> = default + <flagName>, - <flagname>... ;
```

record

This is the name of the instance data field, as given in the class's Espire declaration.

flagName This is the flag to turn on or off. If the flag is preceded by a "+", the flag's default value will be *set*; if it is preceded by a "-", its default value will be *clear*.

For example, the line

```
aRecord = default +aFlag, -anotherFlag;
```

specifies that the field *aRecord* should have its default settings, except that the field *aFlag* should be set, and *anotherFlag* should be cleared.

4.3.2.1 Setting Up Parent-Child Links

Gen and Vis objects are arranged in a hierarchy of children. GEOS implements this with special linkings to the first child and the next sibling. However, you need not be concerned with this. To set up an object's children, you need only use the Espire **children** directive.

To specify an object's children, put the following line in the object's data:

```
children = <childName> [, <childName>]*;
```

childName

These are the names of the children, in order, separated by commas.

UIC automatically sets up the parent's and children's links to each other in the proper way.



4.3.2.2 Hints and Vardata

You may specify an object's hints and other vardata in the .ui file. You can do this by putting the "hints" directive in the instance-data section. This directive has the following format:

```
hints = {
    <hintOrVardataName> [{ <value> }]
    /* repeat as necessary... */
}
```

4.3

hintOrVardataName

This is the name of the hint or vardata field, as specified in the class's *assembly* definition.

value

This field is optional. If the vardata field takes a value, you may specify it here. Everything between the curly braces is written to the .rdf file, i.e. it is not interpreted by the UIC.

4.3.3 Creating Chunks

Chunks are very much like objects. They may be placed in an object block, and referenced by name. They may also be placed in LMem data blocks.

To create a chunk which contains a string, use this format:

```
chunk <chunkName> = "Text...";
```

chunkName

This is the name of the chunk. The name can be used as an optr to the chunk.

UIC will create a chunk containing the text as a null-terminated string.

If a chunk contains some other kind of data, the format is this:

```
chunk <chunkName> = {
    <dataType> <value>
    /* repeat as necessary */
}
```



chunkName

This is the name of the chunk. The name can be used as an optr to the chunk.

dataType

This is the type of data. This is a standard Esp data type, not an Espire type. It may also be an application-defined structure, record, etc.

value

This is the value of the data. It is specified as in Esp.

4.3

Note that the initializers are evaluated in Esp, not in UIC. They should be specified as if they were Esp global variables, as described in section 2.3.1 of chapter 2.

Sometimes you will want an object's instance data field to contain an optr to a chunk created just for that field. If the chunk doesn't need a name, and is used only by that object, you can define the chunk in that instance field's initializer, like so:

```
<field> = chunk {  
    /* chunk data */  
}
```

or

```
<field> = chunk "String..."
```

field

This is the name of the instance data field. This field must be able to contain an optr.

The chunk is created in the same resource as the object. The chunk is unnamed, and the field will contain an optr to the chunk. That is, the Espire code

```
AnObject = MyVis {  
    chunkPtr = chunk {  
        dw    1, 2, 3, 4  
    }  
}
```

is almost functionally identical to



```
AnObject = MyVis {
  chunkPtr = MyChunk;
}

chunk MyChunk = {
  dw      1, 2, 3, 4
}
```

The only difference is that in the second example, the name **MyChunk** evaluates to an optr to the chunk. (This allows you to examine the chunk by name in Swat.)

4.3

4.3.4 Creating VisMonikers

VisMonikers are created much the way they are in Goc. As in Goc, a **VisMoniker** may be a single moniker, or a list of monikers; if it is, the system will choose whichever moniker is most appropriate for the specific UI and monitor.

If the moniker is a simple text moniker, the format is

```
visMoniker <monikerName> = "Text moniker";
```

monikerName

This is the name of the moniker. The name can be used as an optr to the moniker.

This creates a simple text moniker. To create a more elaborate moniker, with special attributes, use this format:

```
visMoniker <monikerName> = {
  [<attr>      = <initializer>;]*
  "Text moniker"; /* This is optional */
}
```

monikerName

This is the name of the moniker. The name can be used as an optr to the moniker.

attr

This is the name of a VisMoniker attribute. These are described in “Visual Monikers” in “GenClass,” Chapter 2 of the Object Reference Book.



initializer This is the value to which the field should be set.

If a field in an object is of type **VisMonikerComp**, it may be initialized with the name of a **VisMoniker**, or you may create a **VisMoniker** on the fly, like this:

```
<fieldName> = "Text moniker";
```

or

```
<fieldName> = {  
    /* Attributes & initializers */  
}
```

4.3

fieldName This is the name of the instance data field. The field must be of type **VisMonikerComp**.

UIC will automatically create a chunk with the specified **VisMoniker**, and set the instance data field to point to that chunk.

You may wish to have an instance data field contain a list of **VisMonikers**. GEOS will then automatically use whichever moniker is most appropriate. To do so, initialize the instance data field like this:

```
<fieldName> = list {  
    <monikerName> [, <monikerName>]*  
}
```

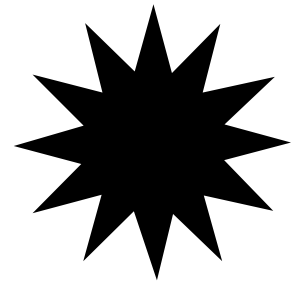
fieldName This is the name of the instance data field. The field must be of type **VisMonikerComp**.

monikerName

This is the name of a **VisMoniker**. The moniker may be in the same or a different resource.

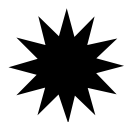


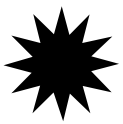
Mixing C and Assembly



E.1	Adding Esp Code to a Goc Geode.....	111
E.2	Writing an Esp Library.....	113

E





You may sometimes wish to combine Goc and Esp code in a single application. There are two main times when you may want to do this. You may be writing an application in Goc, but find that the application is spending a lot of time in a few critical routines; in this case, you may be able to improve efficiency by rewriting those few routines in assembly. If you don't want to rewrite the rest of the application, you will have to write those routines so they can be called from C.

On the other hand, you may be writing a library that has a lot of time-consuming routines. In this case, you may find it worthwhile to rewrite the entire library in assembly, while preserving a Goc interface. That way, all the applications that use the library will be able to take advantage of the efficiency of assembly code. (For example, most GEOS system libraries are written in assembly.)

In particular, if you design a new object class that is being used by many different applications, it may be worthwhile to write a library which defines that class of object. You can write all the method code in assembly, while providing a Goc interface; this lets every application that uses the class take advantage of assembly's efficiency.

E.1 Adding Esp Code to a Goc Geode

Most people will find it easiest to write applications in Goc. For most purposes, Goc is efficient enough; after all, whenever an application is running a system routine, or sending a message to a system-defined object, it is most likely executing assembly code.

However, some applications may have very computation-heavy, time-consuming routines. This can be exacerbated if the application is intended to run on a slower platform, or if the time-consuming routines cannot (for some reason) be compiled efficiently. In this case, you can sometimes improve efficiency significantly by rewriting just those routines in assembly language.



All the routines in any single code resource must be written in the same language (Gcc or Esp). You will therefore have to segregate your Esp routines into one or more resources. Since you lose efficiency if resources are too small, it may be best to simply put all the Esp routines into a single resource. Simply write an assembly file with the routines; then declare all the routines in a C header file as “extern”. **mkmf** will automatically generate appropriate instructions to compile and link the two resources.

You should write the Esp routines so they conform to C pass-and-return conventions. Ordinarily, we recommend that you write routines using the “pascal” convention. Under this convention, arguments are pushed on the stack in the same order that they are passed. For example, if the routine **MyFunc()** has the following declaration:

```
extern word
    _pascal MyFunc(word    firstVar,
                    byte    secondVar,
                    dword   thirdVar);
```

then *firstVar* would be pushed on the stack first; next *secondVar* would be pushed (taking up a full word, even though it's only one byte long); then *thirdVar* would be pushed (first the high word, then the low word). When the assembly routine returned, it would simply load the return value into **ax**, and this value would be returned to the caller. (Byte-sized values are returned in **al**; dword-sized values are returned in **dx:ax**.)

If you use the pascal calling conventions, you can simply use Esp's techniques for declaring passed arguments; declare them in the same order as in the C declaration. For example, suppose the C declaration of **MyFunc()** is as shown above. In the assembly resource, you could write **MyFunc()** like this:

```
MyFunc proc far        firstVar:word, \
                        secondVar:byte, thirdVar:dword

    .enter

; routine code... return value ends up in ax

    .leave

    ret

MyFunc endp
```



If you must have the routine use C calling conventions, remember that the arguments are passed in the opposite order they are declared. This is useful if the routine has a variable number of arguments, but in other situations, it's just a nuisance.

E.2 Writing an Esp Library

You may wish to write a library in Esp whose routines can be called by either Goc or Esp code. This is very much like writing a library in Goc. Simply write all the exported routines to use pascal pass-and-return conventions.

Remember to write both a Goc header file and an Esp header file; that way, an application can include whichever of these is appropriate. Make sure that both of these header files are maintained in tandem, and accurately reflect the state of the library.

If you are writing an object library, you will need to write an Espire header file (**.uih**), as well as the Goc and Esp header files. All three header files must be maintained in tandem. Espire is discussed in "The UI Compiler," Chapter 4.



Mixing C and Assembly

* 114

 Esp