The first edition of this paper was created in the late nineties and based on the behaviour of Delphi 2. While I had a need to use assembler in some of my projects, I quickly found out that there wasn't much documentation available on the topic of how to properly integrate assembler code into Delphi programmes. This paper was written with the intention of bridging that gap. It has since been through a number of revisions. Processors, and Delphi, have evolved significantly since I first wrote this paper, but it should retain a lot of its original value.
# Table of Contents

TABLE OF CONTENTS.......................................................................................................................... 3
INTRODUCTION........................................................................................................................................ 4
BASIC PRINCIPLES.................................................................................................................................. 6

CHAPTER 1: GENERAL CONTEXT OF ASSEMBLER CODE .................................................................... 7
  1.1. WHERE TO LOCATE THE ASSEMBLER CODE ........................................................................... 7
  1.2. LABELS........................................................................................................................................ 7
  1.3. LOOPS.......................................................................................................................................... 9
  1.4. ENTRY AND EXIT CODE ............................................................................................................ 10
  1.5. REGISTER PRESERVATION ......................................................................................................... 11

CHAPTER 2: PASSING PARAMETERS............................................................................................. 12
  2.1. CALLING CONVENTIONS ........................................................................................................... 12
  2.2. PASSING PARAMETERS IN REGISTERS .................................................................................... 12
  2.3. USING THE STACK FOR PARAMETER PASSING ....................................................................... 15
  2.4. PASSING BY VALUE VERSUS PASSING BY REFERENCE ............................................................ 17

CHAPTER 3: LOCAL VARIABLES .................................................................................................... 20
  3.1. LOCAL VARIABLES AND THE STACK FRAME ......................................................................... 20
  3.2. SIMPLE TYPES AS LOCAL VARIABLES .................................................................................... 22
  3.3. RECORDS AS LOCAL VARIABLES ............................................................................................. 23
  3.4. HEAP ALLOCATED TYPES AS LOCAL VARIABLES ..................................................................... 24

CHAPTER 4: RETURNING RESULTS ............................................................................................ 26
  4.1. RETURNING INTEGERS AS IMMEDIATE VALUES ..................................................................... 26
  4.2. RETURNING BOOLEANS AS IMMEDIATE VALUES ..................................................................... 27
  4.3. RETURNING REAL NUMBERS .................................................................................................... 28
  4.4. RETURNING CHARACTERS ......................................................................................................... 30
  4.5. RETURNING A LONG STRING ................................................................................................... 31

FURTHER READING.............................................................................................................................. 35

TABLE 1: USE OF CPU REGISTERS ............................................................................................... 36
TABLE 2: CALLING CONVENTIONS ............................................................................................... 38
TABLE 3: PARAMETER PASSING ..................................................................................................... 39
TABLE 4: RETURNING RESULTS .................................................................................................... 42
ABOUT THE AUTHOR.......................................................................................................................... 44
Introduction

Many programmers still associate assembler with a difficult, low-level kind of programming. Most of them also think it is incomprehensible and impossible to master. In reality, things are not that bad and these perceptions are mostly founded in unfamiliarity. It is quite possible to learn how to write good assembly code without being a genius. On the other hand, don't think a few lessons in assembly will leave you producing faster code than the average Delphi/Pascal equivalent. The reason is that when you write Delphi code, you are competing with an efficient and experienced assembler programmer: the compiler. Overall, the code produced by it is efficient and fast enough for most applications.

Most people grab for assembler in order to achieve better performance for their software. However, the key to performance is mostly design, not choice of language. There is little point in trying to fix a bad algorithm through assembler. So, the first thing to do when you are experiencing bottlenecks, is to review your code architecture and Pascal implementation, rather than immediately taking recourse to assembler. Writing it in assembler is not going to turn a bad algorithm or a bad approach into a good one.

Similarly, most programmers erroneously believe that writing code in assembler by definition means it's fast and assume that code written in assembler is by definition faster than compiled Delphi/Pascal code. That is certainly not the case. Badly written assembler routines will perform on an inferior level and can even cause strange bugs and problems in your application.

There is however a good case for hand crafting assembler code in certain cases. Delphi's (Pascal) language is a general programming language and certain specialised tasks could be done better in assembler. Also, taking advantage of processor specific features might require manual code design.

It is not within the scope of this article to teach you the principles of assembler programming. There are other information resources out there discussing assembler programming. See Further Reading to find some pointers to relevant material.

Before resorting to rewriting code in assembler, investigate first where your bottlenecks are and why. A profiler might be a useful tool during this analysis. Once you have identified the cause, take a step back and look at the structure and algorithm. Often, you can get much better performance by revising the algorithm or the application design, rather than just throwing in some assembler. On the other hand, in some particular cases, assembler might indeed be a better and even simpler choice.

Once you conclude that assembler is actually needed, you should take time proper to draw up a plan for your code and design the algorithm. Only when you
have a clear idea of what you want to do and how you will be implementing it, you can start coding. If you don't think about these issues, you'll end up with spaghetti code, convoluted statements and an unmaintainable program. Not to mention the possibility of introducing nasty bugs.
Basic Principles

During the implementation phase, there are several general principles you should follow. The first important rule is: keep your routines as short as possible. Assembler should be used for some real core activity in which performance and simplicity are essential. So, in most cases, you can manage with fairly short and specialised routines. If you see that you have plenty of long pieces of assembler in your project, you are probably over-enthusiastic about it.

Secondly, keep the routines readable by commenting them in a meaningful way. Because of the basic linear nature of assembly code, the individual statements are quite easy to read. Comments are needed to clarify how you want to implement the algorithm, so that a third party will immediately understand what is going on. Your comments need to add valuable information for the readers of your code. So, the following is clearly wrong:

```
inc edx {increase edx}
```

Such comments are totally pointless, since the instruction itself is making it blatantly clear that you are incrementing the edx register. Comments should indicate the inner workings of the algorithm and/or provide information not otherwise immediately clear from the code and its context, rather than just rephrase what the mnemonics already show. So, the above could, for instance, be replaced by something like this:

```
inc edx {go to the next element in our product table}
```

Thirdly, you should avoid the use of slow instructions wherever possible. In general, the simple instructions are preferable over the complex opcodes, since on modern cpus the latter are implemented in microcode. Google for Agner Fog's publications on the topic of code optimisation, which discuss this and many other very useful aspects of how processors behave.

The final general principle is predictable: test your assembler routines thoroughly and continuously. For routines written in assembler, the compiler will produce substantially less warnings and error messages, and will offer no feedback with regard to the basic logic of the assembler code. Unused variables and wrong use of pointers will not be detected as easily as in Pascal code. So, prepare for intensive testing and debugging of your code. It will take more time and effort when compared to debugging regular Pascal code. That is yet another reason to keep these routines as short and readable as possible.

If after all of this, you still want to go ahead with writing assembler code for your Delphi/Pascal programs, reading this article is probably a good first step. I have designed it to be as generic as possible.
Chapter 1: General Context of Assembler Code

To successfully use assembler code inside your Delphi projects, you need to understand how to call routines written in assembler, have access to the parameters passed and be able to return a result. This chapter discusses where assembler code should be located and what its basic structure should look like. It also explains the compiler's behaviour in generating entry and exit code.

Note: In this document, I will always specifically indicate the calling convention used in the examples. Although in the case of register this is superfluous (since it is the default convention that is used when no calling convention is specifically indicated), it contributes to the readability (look at it as an additional comment if you want) as it reminds the reader that parameters might be located in registers. Credit for this tip goes to Christen Fihl.

1.1. Where to locate the assembler code

Delphi uses the `asm` statement to indicate an assembler block. The end of the block is indicated by an `end` statement:

```pascal
procedure SomeProcedure; register;
asm
  {Code goes here}
end;
```

It is possible to nest `asm` blocks inside a Pascal function or procedure, but that approach is not recommended. You should isolate your assembler code inside separate function or procedure blocks. First of all, inserting assembler inside a regular Pascal function will interfere with the compiler's optimisation and variable management activities. As a result, the generated code will be far from optimal. Variables are likely to be pushed out of their register, requiring saving on the stack and reloading afterwards. Also, nesting inside a Pascal block forces the compiler to adapt its generated code to your assembler code. Again, this interferes with the optimisation logic and the result will be quite inefficient. So, the rule is to put assembler code in its own separate function/procedure block. There is also a design aspect: the readability and maintainability of your code will benefit greatly when all assembler is clearly isolated in dedicated, well-commented blocks.

1.2. Labels

Labels are tags that mark locations in your code. The most common reason for having labels is to have a point of reference for branching. There are two kinds
of labels you can use in your assembler code: Pascal-style labels and local assembly labels. The former type requires you to declare them in a label section first. Once declared, you can use the label in your code. The label must be followed by a colon:

```pascal
label
  MyLabel;
asm
  ...
  mov ecx, {Counter}
MyLabel:
  ... {Loop statements}
dec ecx
jnz MyLabel
...
end;
```

The above example illustrates how to declare a label `MyLabel`, marking a position in your program (`MyLabel:`) and moving to the label's location in a jump instruction (`jnz MyLabel`).

The same can be achieved in a slightly simpler way, by using local labels in your assembler code. Local labels do not require a declaration, rather you simply insert the label as a separate statement. Local labels must start with the `@` sign, and are again followed by a colon. Because `@` can't be part of an Pascal identifier, you can use local labels only within an `asm...end` block. Sometimes, you will see labels prefixed by a double `@` sign in code in this document. This is a convention I use a lot and it draws attention to the labels immediately, but it is not required (some assemblers use the `@@` to identify special purpose labels, like `@@:` for an anonymous label). Below is an example of the same logic as above, but using a local label:

```pascal
asm
  ...
  mov ecx, {Counter}
@MyLabel:
  ... {Loop statements}
dec ecx
jnz MyLabel
...
end;
```

Neither kind of label is intrinsically better than the other. There is no advantage in code size or speed of course, since labels are only reference points for the compiler to calculate offsets and jumps. The difference between Pascal-style and local labels in assembler blocks is a relic from the past and is fading away. As a consequence, even Pascal-style labels are "local" in the sense that it is not possible to jump to a label outside the current function or procedure block. That is just as well, since that would be a perfect scenario for disaster.
1.3. Loops

Often, the assembler code will be designed to achieve the highest speed possible. And quite often also, processing large amounts of data inside loops will be the task. When loops are involved, you should implement the loop itself in assembler too. That is not difficult and otherwise you will be wasting a lot of execution time because of calling overheads. So, instead of doing:

```pascal
function DoThisFast(...): ...; register;
asm
  ...{Here comes your assembler code}
  ...
end;

procedure SomeRoutine;
var
  I: Integer;
begin
  I:=0;
  ...
  while I<{NumberOfTimes} do begin
    DoThisFast(...);
    inc(I);
  end;
  ...
end;
```

You should implement the loop *inside* the assembler routine:

```pascal
function DoThisFast(...): ...; register;
asm
  ...{Here comes your main assembler code}
  ...
  dec ecx
  jnz @@loop
  ...
end;

procedure SomeRoutine;
begin
  ...
  DoThisFast(...);
  ...
end;
```

Note that in the example above, the loop counter counts downwards. That is because in this way, you can simply check the zero flag after decrementing to see if the end of the loop has been reached. By contrast, if you simply start off
with ecx=0 and then count upwards, you will need an additional compare instruction to check whether or not to continue the loop:

```
mov ecx, 0
@@loop:
...
ic ecx
cmp ecx, {NumberOfTimes}
jne @@loop
```

Alternatively, you can subtract the NumberOfTimes from 0 and then increase the loop index until zero is reached. This approach is especially useful if you use the loop index register simultaneously as an index to some table or array in memory, since cache performance is better when accessing data in forward direction:

```
xor ecx, ecx
sub ecx, {NumberOfTimes}
@@loop:
...
ic ecx
jnz @@loop
```

Remember however that in this case, your base register or address should point to the end of the array or table, rather than to the beginning, and you will be iterating through the elements in reverse order.

1.4. Entry and exit code

Another important aspect to be aware of, is that the compiler will automatically generate entry and exit code for your assembler block. This will only happen if there is a need for a stack frame, i.e. if parameters are passed to the routine via the stack, or if local data is stored on the stack. You'll find that very often this is indeed the case, and consequently entry and exit code will be generated. The compiler produces the following entry code:

```
push ebp
mov ebp, esp
sub esp, {Size of stack space for local variables}
```

This code preserves ebp, and then copies the stack pointer into the ebp register. Subsequently ebp can be used as the base register to access information on the stack frame. The sub esp line reserves space on the stack for local variables as required. The exit code pattern is as follows:

```
mov esp, ebp
pop ebp
ret {Size of stack space reserved for parameters}
```
This exit sequence will clean up the space allocated for local parameters by copying ebp (pointing at the beginning of the stack frame) back to the stack pointer. This deallocates the space used for local variables. Next, ebp is restored to the value it had upon entry of the routine. Finally, control is returned to the caller, adjusting the stack again for any space allocated for parameters passed to the routine. This parameter cleanup in the ret instruction is required for all calling conventions except cdecl. In all cases except cdecl, the called function is responsible for cleaning up the stack space allocated for parameters, and thus the ret instruction will include the necessary adjustment. In case of cdecl, however, the caller performs the cleanup.

If your function or procedure has neither local variables nor parameters passed to it via the stack, then no entry and exit code will be produced, except for the ret instruction that is always generated.

1.5. Register preservation

When using registers inside your function or procedure, please note that only the registers eax, ecx and edx can be freely modified. All other registers must be preserved. That means that if you use any of the other registers inside your routine, you must save them and restore them before returning from your routine.

You must not change the contents of any of the segment selectors: ds, es and ss all point to the same segment; cs has its own value; fs refers to the Thread Information Block (TIB) and gs is reserved. The esp register points to the top of the stack, of course, and ebp is made upon entry to point to the current stack frame as a result of the default entry code generated by the compiler. Since each pop and push operation will change the content of the esp register, it is usually not a good idea to access the stack frame directly through esp. Rather, you should reserve ebp for that purpose. Table 1 summarises register usage.

Apart from the register context, you can assume that the direction flag is cleared upon entry and if you change it (which I don't recommend), you should restore its cleared state prior to returning (by using the cld instruction). Finally, you should be careful about changing the FPU control word. Although this allows you to change the precision and rounding mode for floating point arithmetic, and permits you to mask certain exceptions, you will be drastically influencing the way calculations in your entire application are performed. Whenever you decide it is necessary to change the FPU control word, make sure you restore it as soon as possible. When you are using Comp or Currency types, make sure you don't reduce floating point precision.
Chapter 2: Passing Parameters

Most routines programmers write are passed one or more parameters as input. This chapter gives an overview of how such parameters can be passed to your assembler routine. Many routines will be functions that also have to return a result to the caller, but the process of returning a result is discussed in Chapter 4.

2.1. Calling Conventions

One of the most common operations in the processing of code is calling a subroutine that carries out a given task. In order to do that, the main code needs to hand over control to the subroutine, allow the subroutine to execute, and then return to where it left the main execution path. This process requires agreement between the caller and the callee about how to pass information to the subroutine, how to return results where applicable and who is responsible for the memory allocation and cleanup. Various conventions exist to deal with invoking subroutines, commonly known as calling conventions. Delphi supports a wide set of them: register, pascal, cdecl, stdcall, and safecall. Obviously, caller and callee need to use the same calling convention in order for this to work properly.

As already mentioned, calling conventions define a number of different aspects of subroutine invocation:

- Where parameters are located: in registers or on the stack
- In which order parameters are passed: right-to-left or left-to-right
- Who is responsible for cleaning up the parameters afterwards, the caller or the callee

Table 2 gives an overview of each of the Delphi supported calling conventions.

2.2. Passing parameters in registers

There is only one calling convention in Delphi, register, that uses the CPU registers to pass parameters to a subroutine. There are three registers available for this purpose: eax, edx and ecx and they are used up in this order (because ecx is used last, that register remains available the longest, which is quite convenient since you will often want to use it as a loop counter variable). If you are passing more parameters than there are registers available, the remaining ones will be passed on the stack as described later. When you use the register calling convention for methods, however, eax contains a pointer to Self and thus only two registers are available for parameter passing. Not all data types can be passed in a register. Table 3 gives a clear overview of which
types can be passed in a register. Please note that when passing parameters by reference, you are in fact passing a pointer to the variable in question. Since pointer types qualify for passing in a register, variables passed by reference always qualify for passing in a register, with the exception of method pointers.

If the number of parameters passed is lower than or equal to the number of registers available (three for standalone routines, two for methods), then there is no need to set up a stack frame for parameter passing. This can save overhead when calling the routine. Be careful however, because parameter passing is not the only reason for setting up stack frames: if you declare local variables, a stack frame is also required and thus extra overhead to manage the stack frame is still incurred.

In addition, for many structured types, the data itself actually resides on the stack or on the heap and the variable is a pointer to the actual data. Such a pointer occupies 32-bits and therefore will fit into a register. This means that most parameter types will qualify for passing through registers, although method pointers (consisting of two 32-bit pointers, one to the object instance and one to the method entry point) will always be passed on the stack.

This article is based on 32-bit modes, so registers are 32 bits wide. When passing information that doesn't occupy the whole register (byte- and word-sized values for example), the normal rules apply: bytes go in the lowest 8 bits (for example al) and words in the lower word of the register (for example ax). Pointers are always 32-bit values and thus occupy the whole register (for example eax). In case of byte- or word-sized variables, the content of the rest of the register is unknown and you should not make any assumptions about its state. For instance, when passing a byte to a function in al, the remaining 24 bits of eax are unknown, so you cannot assume them to be zeroed out. You can use an and operation to make sure the remaining bits of the register are reset:

```assembly
and eax,$FF {Unsigned byte value in AL, clears 24 highest bits}
```

or

```assembly
and eax,$FFFF {unsigned word value in AX, clears 16 highest bits}
```

When passing signed values (shortint and smallint), you might want to expand them to a 32-bit value for easier computation, but in doing so you need to retain the sign. To expand a signed byte to a signed double word, you need two instructions:

```assembly
cbw  {extends al to ax}
cwde {extends ax to eax}
```

The importance of not relying on the remainder bits having a specific value can be easily demonstrated. Write the following test routine:
function Test(Value: ShortInt): LongInt; register;
asm
end;

Next, drop a button and a label on a form and put the following code in the button’s OnClick event:

var
  I: ShortInt;
begin
  I:= -7;
  Label1.Caption := IntToStr(Test(I));
end;

Run the project and click the button. The Test routine receives a ShortInt through al. It returns an integer in the eax register (returning results is discussed in Chapter 4), which should be unchanged since the subroutine returns immediately. You can easily observe that eax has undefined content upon return. Now change the test function as follows and run the project again:

function Test(Value: ShortInt): LongInt; register;
asm
  cbw
  cwde
end;

The Test routine now returns the correct value.

In summary, when using the register calling convention, up to three parameters can be passed in the eax, edx and ecx registers. So, the following declaration:

procedure DoSomething(First: Integer; Second: ShortInt;
  Third: Pointer); register;
asm
  ...
end;

will put First in eax, Second in dl and Third in ecx. Next, here is an example of a method declaration:

procedure TSomeClass.DoSomething(First, Second: Integer);
register;
asm
  ...
end;

In this case, eax will contain Self, edx contains First, while Second is stored in ecx.
Please note that since `register` will result in parameters being passed in registers, you will lose that parameter information as soon as you override the register's contents. Take the following code:

```delphi
procedure DoSomething(AValue: Integer); register;
asm
    {eax will contain AValue}
    ...
    mov eax, [edx+ecx*4] {eax gets overwritten here}
    ...
end;
```

After `eax` gets overwritten, you no longer have access to the `AValue` parameter. If you need to preserve that parameter, make sure to save the contents of `eax` on the stack or in local storage for use afterwards. And don't fall into the common trap to do the following later on in your code:

```delphi
mov eax, AValue
```

because the compiler will, for the above line, simply generate the following code:

```delphi
mov eax, eax
```

as the compiler only knows, from the chosen calling convention, that `AValue` was passed in `eax` to the subroutine.

### 2.3. Using the stack for parameter passing

All calling conventions might use the stack for passing some parameters. While the `register` convention tries to use CPU registers first, not all variable types qualify for passing in a register and sometimes you will need to pass more parameters than there are registers available. All other calling conventions will pass all their parameters on the stack to the callee.

As explained in the previous chapter, the compiler will generate entry and exit code to manage the stack frame. As a result, `ebp` is initialised as base pointer to the stack frame, allowing easy access to parameters and other information on the stack (including local variables as explained in Chapter 3). When you refer to parameters that reside on the stack, the compiler will generate the appropriate offset from `ebp`. Have a look at the following declaration:

```delphi
function Test(First, Second, Third: Integer): Integer; pascal;
```

The calling convention is `pascal`, which means that prior to the call to the subroutine, the caller pushes three parameters on the stack in the order that they are declared (remember that the stack grows downwards, which means the first parameter is located at the highest address):
Next, the `call` instruction will push the return address onto the stack and then hands over execution to the subroutine, so immediately after entry, the stack looks as follows:

![Stack Diagram]

The compiler generated entry code (see Chapter 1) saves the current value of `ebp` and subsequently copies the value of `esp` to `ebp` so that the latter can from now on be used to access the parameter data on the stack frame:

![Stack Diagram]

From this point on, we can access the parameters on the stack frame as offsets from `ebp`. Because the return address sits on the stack between the current top-of-stack and the actual parameters, we access the parameters as follows:

\[
\begin{align*}
\text{First} & = \text{ebp} + \$10 \ (\text{ebp} + 16) \\
\text{Second} & = \text{ebp} + \$0C \ (\text{ebp} + 12) \\
\text{Third} & = \text{ebp} + \$08 \ (\text{ebp} + 8)
\end{align*}
\]

However, you can simply refer to these parameters by name. the compiler will replace each parameter with the correct offset from `ebp`. So, in the example above, writing the following:

```
mov eax, First
```

will be translated by the compiler into:

```
```
This will save you the headache from calculating the offsets yourself and it is also much more readable, so you should use the names of the parameters that are passed on the stack in your code wherever possible (practically always) instead of hard coding the offsets. Be careful however: if you use the register calling convention, the first set of parameters will be passed in registers. For those parameters that are passed in registers, you should use correct register to refer to the variable, to prevent ambiguities in your code. Take the following example:

```delphi
procedure DoSomething(AValue: Integer); register;
```

Since this declaration uses the register calling convention the AValue parameter will be passed into the eax register. It is probably wise to explicitly write eax in your code to refer to this parameter. It will help you to spot the following potential bug:

```delphi
mov eax, AValue
```

which on the basis of the declaration above would result in the following code to be generated:

```delphi
mov eax, eax
```

In summary: for parameters passed in a register, you should use the register to refer to it. For parameters passed via the stack, use the variable name to refer to it (and don't use the ebp register, so it remains available for access that information).

Stack space is always allocated in 32-bit chunks, and therefore the data passed on the stack will always occupy a dword multiple. Even if you pass a byte to the procedure, 4 bytes will be allocated on the stack with the three most significant bytes having undefined content. You should never assume that this undefined portion is zeroed out or has any other specific value.

### 2.4. Passing by value versus passing by reference

Earlier on, I mentioned the difference between passing by value and passing by reference. When passing by reference (using the var directive) or, in some instances, when using const, no data is copied and handed over to the routine in question, but rather a pointer to the original data is passed on. This difference is of course quite important, not in the least because of the non-localised effects changes to that data might cause. Take for instance the following function declaration:

```delphi
function MyFunction(I: Integer): Integer; register;
```
As the register calling convention is used, the value of the I parameter will be passed in the eax register (see table 3). So, given I=254, eax will upon entry contain the value $000000FE, passing I by value. However, if we change the declaration as follows:

```delphi
function MyFunction(var I: Integer): Integer; register;
```

the eax register no longer contains the value of I, but rather a pointer to the memory location where I is stored, for example $0066F8BC. Passing parameters by reference using var or const is done by means of a 32-bit pointer.

When you use const, indicating that you a variable is used for read-only access only, the compiler uses either method. The wording in Delphi’s online help can be misleading and some people assume that const always results in passing a 32-bit pointer to the actual value, but that is not correct. You can use table 3 for guidance.

By using const, the programmer informs the compiler that the data is only going to be read. Please note however that within an asm..end block, the compiler will not prevent you from writing code that violates this read-only characterisation of const parameters in cases where these are pointers to structured data like AnsiStrings or records. Be careful to honour the read-only character of the information passed using const in your assembler code, otherwise you could introduce nasty bugs. And of course, it would be extremely poor design to label information read-only, yet then proceed to change it. This is especially important when you are using reference counted types like AnsiString that use copy-on-write semantics.

All of the above means that you will have to carefully take into account the differences between passing by value and passing by reference. For example, imagine a function that calculates the sum of an integer with 12. In case of passing the integer parameter by value, the code should look as follows:

```delphi
function MyFunction(I: Integer): Integer; register;
asm
  add eax, 12
end;
```

I will discuss returning results in Chapter 4. For now it is sufficient to know that the result in this case will be returned to the caller via the eax register. As you can see, the value of I is taken directly from the eax register. But if we change the function to pass the information by reference, we would get something like this:

```delphi
function MyFunction(var I: Integer): Integer; register;
asm
```

Guido Gybels: Using Assembler in Delphi
mov eax, [eax] {Load the value of I through the pointer}
add eax, 12
end;

Because `eax` does not contain the value of `I`, but rather a pointer to the memory location where `I` is stored, we retrieve the value through the received pointer.
Chapter 3: Local Variables

Just as in any Pascal routine, you can use local variables inside your assembler code. Local variables are declared in the same way as for Pascal routines, by a `var` section. In this chapter, I will give a detailed overview of how local variables are implemented and used in `asm` blocks.

3.1. Local variables and the stack frame

Storage space for local variables is allocated on the stack, by means of the compiler generated entry code (and freed again in exit code). Please note that for some complex types like `AnsiStrings`, the space allocated is for a pointer to the actual data (strings reside on the heap and the string variable is a mere pointer to the data) and further action will be needed to allocate and assign the actual data. Continuing the narrative from the previous chapter, have a look at the following code:

```pascal
procedure DoSomething(First, Second, Third: Integer); pascal;
var
  SomeTemp: Integer
asm
  ...
end;
```

From that previous chapter, we already know that using the `pascal` calling convention will result in parameters being pushed onto the stack prior to invoking the procedure. The call instruction will push the return address onto the stack. Next, entry code will cause the value of `ebp` to also be pushed onto the stack. Then, `ebp` is set up as base pointer for accessing the data on the stack frame. At this point, the stack frame looks therefore as follows:

![Stack Frame Diagram](image)

Because we have also declared a local variable, `SomeTemp`, the compiler will add code (for instance `push ecx`) to reserve space on the stack for said variable:
As stated before, ebp contains a base pointer for accessing data on the stack frame. Since the stack grows downwards, higher addresses contain parameters, while lower addresses contain local variables. In our particular example, the stack frame has the following slots allocated:

**Parameters:**
- First = ebp + $10 (ebp + 16)
- Second = ebp + $0C (ebp + 12)
- Third = ebp + $08 (ebp + 8)

**Local Variables:**
- SomeTemp = ebp - $04 (ebp - 4)

A next local variable will be allocated at ebp -8 and so on. Just as with parameters on the stack, you can (and should) use the variable name to refer to the actual location on the stack:

```
mov eax, SomeTemp
```

which will be translated by the compiler into:

```
mov eax, [ebp-4]
```

Please note that the content of these variables is generally not initialised, and you should treat it as being undefined. It is your task to initialise them when and if required.

Because using local variables cause overhead for creating and managing the stack frame, it is worth analysing your algorithm carefully to determine whether or not you need local storage. Clever use of available registers and smart code design can often avoid the need for local variables altogether. Apart from avoiding overhead for allocating and managing local variables, moving data between registers is significantly faster than accessing data in main memory (but beware of stalls and other performance hits, for instance by reading a register immediately after writing it). When you are writing Object Pascal code, the Delphi compiler will perform optimisations by trying to use registers wherever feasible. Loop counter variables are a particular case in point and you, too, should favour registers for such usage. Of course, inside an
asm..end block, you are on your own and the compiler will not perform such optimisations for you. Well structured code will therefore aim to use registers as much as possible, especially for data that is used most often.

3.2. Simple types as local variables

Quite a number of data types will require simply allocation of space on the stack frame when you declare a local variable. ShortInt, SmallInt, LongInt, Byte, Word, DWord, Boolean, ByteBool, WordBool, LongBool, Char, AnsiChar and WideChar all belong to this category.

While not all of these types are 32-bits wide, reservation of stack space will always happen in chunks of 32-bits at a time. That means that if you use smaller types, like for instance byte or word, the unused part of the allocated space is undefined. For instance, if you declare a local variable as follows:

```pascal
var
 AValue: ShortInt;
```

While AValue only requires one byte, a full dword is allocated on the stack frame. This behaviour ensures that data on the stack is always aligned on a dword boundary, which improves performance and makes the logic to calculate variable locations easier (and allows for easy use of scaling in indirect addressing). You should however not use the remainder part of the allocated space (the padding) since this ultimately an implementation issue. Future compiler versions might behave differently. If you need additional storage space, simply use the appropriate, larger type.

Please note that this rule for dword allocation does not apply to local variables of type record, even though they are also stored on the stack frame. Their member fields' alignment depends on the state of the alignment switch ({$A} directive) and the use of the packed modifier. This is discussed in more detail in the next paragraph.

This alignment behaviour is another good reason to refer to local variables using their variable name, rather than manually calculating the offset yourself. The compiler will calculate the correct offset for you. In the example above, AValue occupies only one byte. Hence, only the lowest byte of the allocated dword is used. So, this instruction:

```pascal
mov al, AValue
```

will result in the compiler generating the following code:

```assembly
mov al, [epb-$01]
```
In Pascal routines, outside `asm...end` blocks, the compiler might optimise a local variable into a register, in which case no space for it will be allocated on the stack frame. While such optimisation does not happen inside your own `asm...end` blocks, you should be aware of this behaviour when observing compiler generated code through the CPU window. Similarly, sometimes the compiler will generate code that uses `esp` directly, rather than an offset from `ebp`, thus saving the need for initialisation of `ebp`. As argued before, it's not generally advisable to use `esp` directly in your own assembler code, as it makes it extremely hard to read and maintain the code and it is prone to introducing subtle coding errors that will be hard to find and debug. While study compiler generated code can be very instructive, remember that you are not a machine, but a human programmer. Machines are good in making sure they calculate the right offsets and the like - humans mostly are not. It's likely that in most cases stack frame overhead will not constitute a bottleneck, especially if you design your code carefully. If you identify that stack frame overheads cause performance issues in your application, you should reconsider your algorithm. It might sound very obvious, but too many programmers end up at some point optimising code that has no effect on the overall application's performance.

### 3.3. Records as local variables

Just as in the case of simple types, local record variables are stored on the stack frame. In that respect, they are not fundamentally different in their usage from simple types (see previous paragraph). However, the compiler's record alignment mechanism is more complex. This can seriously complicate things for the programmer if he/she is coding offsets directly.

There are two key factors that define the compiler's record alignment behaviour: the alignment directive (`{$A}` or `{ALIGN}`) and the `packed` modifier. Furthermore, the actual alignment of the record member fields is dependent on the field type. For example, let's consider the following record declaration:

```pascal
TMyRecord = record
  FirstValue: DWord;
  SecondValue: Byte;
  ThirdValue: DWord;
  FourthValue: Byte;
end;
```

The alignment boundary for each member field of the record depends on its type and its size. In the example above, `FirstValue` and `ThirdValue` are of type `DWord`, which is a 32-bit type. With alignment on, they will be aligned to `dword` boundaries. Since in between those two members, there is a byte-sized field, `SecondValue`, the compiler will add three padding bytes, thus ensuring
that ThirdValue is properly aligned. The following picture shows the memory allocation for this record in the aligned state:

By adding the packed modifier to the record declaration, the record's member fields are no longer aligned. You can see the result in the following illustration, as the padding bytes are no longer present:

Similarly, when alignment is turned off by using the {$A-} directive, even without the packed modifier there will be no padding between record member fields. Fortunately, just as for simple types, you can refer to record member fields by their names, and the compiler will calculate the correct offsets for you. However, always make sure you use operands of the proper size, i.e. specify the operand size explicitly. In that way, your code will continue to work correctly even when alignment is changed or the packed modifier is introduced at a later stage:

```assembly
mov eax, DWORD PTR [ARecord.FirstValue]
mov al, BYTE PTR [ARecord.Byte]
```

3.4. Heap allocated types as local variables

Dynamic variables, long strings, wide strings, dynamic arrays, variants, and interfaces in Delphi are all variable types that are stored in heap memory. In order to use them, you use a reference variable, i.e. a pointer to the actual variable data. In assembler, you will be responsible for the allocation and management of the memory and its contents.
In other words, if you use heap allocated types as local variables, memory will be allocated for the reference (the pointer) to that variable on the stack frame, but you are responsible for the actual allocation and deallocation of the memory and for initialising the contents. In Pascal, most of these types are largely automatically managed, so allocation and deallocation happens behind the scenes. In assembler blocks, that is obviously not the case.

You can call `GetMem` to allocate memory and return a pointer to the newly allocated memory. You need to pass the amount of memory needed in `eax` and upon return from `GetMem` the `eax` register will contain the pointer, which you can then store in the appropriate slot on the stack frame.
Chapter 4: Returning Results

In many cases, you will return a result from your assembler routines to the caller. In previous chapters, we have discussed how to pass parameters to a function and how to use local variables in your assembler code. This chapter will cover returning results to the caller.

4.1. Returning integers as immediate values

Delphi provides several different integer types. We use the term integer in this chapter mostly in its general meaning of "whole number", using a standard font in lowercase. Delphi also has a generic type called Integer. When I refer to the Delphi-specific generic data type, I will spell it Integer in a monospaced font and with an uppercase "I".

There are 8-bit, 16-bit, 32-bit and 64-bit integer types in Delphi. Some of these types are signed, whereas others are unsigned. Unsigned integers always represent positive whole numbers. Signed types have a sign bit, which if set indicates a negative value and when cleared indicates a positive value. Negative values are represented as the two's complement of the absolute value. The Delphi integer types are Shortint (8-bit, signed), Smallint (16-bit, signed), Longint (32-bit, signed), Int64 (64-bit, signed), Byte (8-bit, unsigned), Word (16-bit, unsigned) and Longword (32-bit, unsigned). In addition, Delphi also has the generic types Integer and Cardinal, which correspond on a 32-bit platform to respectively a signed 32-bit value and an unsigned 32-bit value. So, Integer is on a 32-bit platform the same as Longint, whereas Cardinal is the same on a 32-bit platform as Longword.

There are several other data types in Delphi that map to one of the above integer types. Many of these additional types are provided to offer a type of the same name as in C-declarations, often for conformity with the Windows API. For example, DWORD and UINT are the same as Longword, whereas SHORT is the same as Smallint, etc.

In general, returning integers is straightforward: you store the value in the eax register before returning to the caller. If the return type is smaller than eax, only the al (8 bits) or ax (16 bits) portion of the register is valid, the contents of the remainder of the register are ignored. See Table 4 for a detailed overview. The only exception to this rule are 64-bit integers. On a 32-bit platform, these are returned in edx:eax, with edx containing the most significant part.

Please note that the Comp type, which also represents a 64-bit integer, does not behave like other integers. It is a type that uses the floating point unit of the processor and as such follows the conventions for real types. You should have
little use for the Comp type, which is maintained for backward compatibility. It is recommended to use Int64 instead. On several processors, Int64 will also yield better performance for integer arithmetic, as it uses the CPU registers, rather than the FPU.

The following code demonstrates how to return an integer from assembly code. It returns the number of set bits in the AValue parameter as an unsigned 8-bit value (we don't need the larger range of 16 or 32 bit integers, since the returned value will fall in the range 0-32).

```pascal
function CountBits(const AValue: Longword): Byte;
asm
  mov ecx, eax
  xor al, al
  test ecx, ecx
  jz @@ending
@@counting:
  shr ecx, 1
  adc al, 0
  test ecx, ecx
  jnz @@counting
@@ending:
end;
```

4.2. Returning booleans as immediate values

Returning a boolean value is quite simple too. Again, the result goes in the eax register, or a subset of that register - similar to returning integers. Delphi has several boolean types. Boolean is the "proper" boolean type: it only has two possible values, true and false. Within Delphi, you should always use this type. The other boolean types, ByteBool, WordBool and LongBool are provided for compatibility with other languages and for calling the Windows API.

The Boolean type is really an enumerated type. It occupies a byte of memory. As this is not an ordinal type, you should only use the predefined constants True and False in assignments. Do not assign ordinal values to a Boolean as this relies on an implementation feature, namely the values that the compiler uses to represent the True and False values of a Boolean. While it is unlikely that this will change in future versions of Delphi, relying on it being future proof is still bad practice, architecturally ugly and also less clear than using the predefined boolean constants True and False. As a Boolean requires 8 bits, you return it in al:
function DoSomething(...): Boolean;
asm
  ...  
  mov al, True
  ...
end;

In contrast, ByteBool, WordBool and LongBool, provided for compatibility with other languages and calling the Windows API, are essentially ordinal types. They occupy respectively 8 bits, 16 bits and 32 bits and are thus returned correspondingly in al, ax or eax. They are considered true if their ordinality is non-zero and false otherwise. The compiler will perform any necessary conversions between these types and Boolean where required.

Table 4 provides the full overview.

4.3. Returning real numbers

Real numbers in Delphi are implemented as floating point types. Unfortunately, all too many programmers nowadays don't properly understand floating point representation. You can read my brief introduction on the topic.

The basic mechanism for returning real values from your assembler code is to put the result in the ST(0) register of the FPU, which corresponds to the top of the FPU stack.

Even though Delphi supports several floating point formats, such as single (7-8 significant digits, occupies 4 bytes) and double (15-16 significant digits, occupies 8 bytes of memory), internally the FPU always stores and handles floating points as 80-bit values. Delphi's Extended type (19-20 significant digits, uses 10 bytes of memory) maps onto this format. Note that all these real types are returned to the caller as a value in ST(0). It is only when the result is subsequently stored in memory or passed along to another part of the program that it effectively is transformed in its 4, 8 or 10 byte encoding. My article on floating point values on my website discusses these formats in some more detail.

There are however other considerations to take into account when working with real numbers. The Intel FPU has a control register that controls precision and rounding and also has exception masks to steer FP exception handling. The different precision and rounding methods allow a programmer fine control over the behaviour of the FPU and can be important for code compatibility with other systems or legal and other standards. It is therefore important to understand that it might not be sufficient to declare a result of a certain data type (say, single or double) only. Instead, you might need to explicitly configure the control register.
The precision control bits in the FPU control register are highly relevant in this respect. Under normal circumstances, this two bit binary value ought to be set to 11 at all times, indicating 64-bit mantissa precision. If the precision control bits are set to a lower precision, the FPU will reduce precision during computation and hence your result will be less precise than you would have expected. The theory of floating point arithmetic and the details of the Intel FPU are out of scope for this article, but you should familiarise yourself intimately with the topics before attempting to write elaborate floating point code or whenever you need to produce results in line with specific guidelines or standards. Delphi has several supporting functions and variables, such as Get8087CW, Set8087CW, SetPrecisionMode, etc. See online help for more information. Note that many libraries and even calls to OS functions can change the value of the FPU control word. If you change the control word inside your own code, it is good practice to make sure you restore it to its previous state when you are done. This ought to be done outside your time critical FP code, since setting the control word causes, on many processors, considerable stall if the control word is read immediately afterwards, which is the case for most FP instructions.

In addition to single, double and extended, the Real48, Comp and Currency types are also returned in ST(0). Even though Comp represents a 64-bit integer, it is a type that uses the FPU, rather than the CPU registers and as such is manipulated using FPU instructions. Currency is a fixed-point type mainly designed for monetary calculations, but as with Comp it is in fact a FPU based type. Note that Currency is scaled by $10^4$. Hence, a Currency value of 5.4321 is stored in ST(0) as 54321.

Anyone wishing to use FP math for monetary applications ought to make sure they fully understand the nature of floating point arithmetic. My article on floating point values provides a brief introduction to the topic and contains links to further reading material. Using scaled integers might be a better approach for such applications. Also, Intel CPUs support BCD encoding and arithmetic, which is useful for these purposes. Unfortunately, the Delphi language has no support for BCD types, so you will need to encode and decode BCD data yourself.

Unless needed for compatibility with other applications or environments, you should avoid the non-native Real48 type altogether. This type is not supported in hardware, so all manipulation has to be done in code, which makes it very slow. Convert a Real48 into a native float immediately after receiving it, using the System unit’s _Real2Ext function. When invoking _Real2Ext, eax contains a pointer to the Real48 value. Upon return, ST(0) is loaded with the value. You can then use the FPU to perform the required calculations. If you need to hand the result back as a Real48 type, call _Ext2Real, also in the
System unit, which will convert the value in ST(0) back into a Real48 value.
eax should contain a pointer to a 6-byte wide memory location where the
converted value will be stored. Note that in Delphi versions before Delphi 4, this
non-native 6-byte type was called Real instead of Real48. From Delphi 4
onwards, Real acts as a generic type for real numbers, at present implemented
as a Double.

Table 4 summarises the rules for returning results, including real types.

To conclude this section on returning real numbers, below is a full working
example. The function CalcRelativeMass below demonstrates floating point
arithmetic in assembler within a Delphi environment. The function takes two
parameters, mass and velocity of a body, and calculates the relative mass as
per the theory of relativity.

```pascal
function CalcRelativeMass(m, v: Double): Double; register;
const
  LightVelocity: Integer = 299792500;
asm
  {Calculate the relative mass according to the following
  formula: Result = m / Sqrt(1-v²/c²), where c = the
  velocity of Light, m the mass and v the velocity of
  an object}
  fild LightVelocity
  fild LightVelocity
  fmulp {Calculate c²}
  fld v
  fld v
  fmulp {Calculate v²}
  fxch
  fdivp {v²/c²}
  fld1
  fxch
  fsubp {ST(0)=1-(v²/c²)}
  fsqrt {Root of ST(0)}
  fld m
  fxch
  fdivp {divide mass by root result}
end;
```

### 4.4. Returning characters

Delphi currently offers two fundamental character types: **AnsiChar** is an 8-bit
caracter, whereas **WideChar** is a 16-bit Unicode character. There is also a
generic type, **Char**, which is mapped to **AnsiChar**. As with integer types, it is
good practice to use the fundamental types in your own assembly code.

As **AnsiChar** is an 8-bit type, you return it in al. **WideChar** is a 16-bit Unicode
character and is returned in ax.
The Delphi online help makes a bit of a mess of things by stating on the one hand that `WideChar` is a fundamental type, and on the other hand talking about its "current implementations [sic]", thereby alluding that future versions might change its 16-bit nature, as if it were some generic type. In practice, there is little choice but to consider `WideChar` as a fundamental, 16-bit Unicode type.

See also Table 4, which provides an overview of the rules for returning results.

### 4.5. Returning a long string

A very common type used in Delphi applications is the long string: `AnsiString`. This type is essentially an array of `AnsiChar` characters, but with some additional complexity. The `AnsiString` type has two important properties from the point of view of the assembler programmer. Firstly, it is a reference counted type. The reference counting mechanism has several advantages. If the same string is used in different places, the same instance can be shared, rather than having to allocate memory for each identical string. By using reference counting with a copy-on-write algorithm, a copy of a string in memory is only made when it is absolutely necessary, which reduces overhead and enhances performance. Reference counting also allows for automated memory management. When the reference count reaches zero, indicating no one is using the string anymore, it is automatically cleaned up. However, within our assembler code, we don't have the compiler's support for this, so we often must handle memory allocation and reference counting manually.

The second important feature of long strings is that they are stored in heap memory. The string variable is a pointer to the string on the heap. As with reference counting, and in contrast to Pascal code, we will need to explicitly deal with memory management issues for our long strings.

As we create and manipulate long strings in assembler code, we must ensure that those strings will continue to function properly for the rest of our Delphi code, outside the `asm` block. This requires us to allocate memory for the strings properly, through Delphi functions, and that we must consider the reference counting carefully when we create, process and return long strings.

In contrast to ordinal types, long strings are not returned in a register, rather the function behaves as if an additional `var` parameter was declared after all the other parameters. In other words, an `additional` parameter is passed to your function by reference. Chapter 2 describes parameter passing in detail, including the the issues related to passing variables by reference and the differences associated with the various calling conventions.

The approach of using an additional parameter might seem like a needless complication, but in fact it is quite clever: by passing this additional `var`
parameter, responsibility for decreasing the reference count is handed over to the caller. This makes it quite easy to return long strings from assembler code, while at the same time making sure that the reference count is suitably adjusted after the caller is done with it.

With regard to the process of allocating memory for long strings in assembler code, you should study the various routines provided in System.pas for this purpose. For example, you can call LStrSetLength to set the length of the Result string, before filling it with content. A major drawback of this approach is that it can easily be broken as Delphi itself evolves. System.pas gets special treatment at compile time and these internal routines might be changed at some point as Delphi evolves. One way around this is to create the string elsewhere in Pascal, then just hand an already allocated long string to your own assembler code. Writing code that ports well is an important consideration. Clearly the very choice for assembler means that code will be much more closely tied to a specific platform and compiler, but within those limitations, programmers should still endeavour to write code that is as readable and future proof as possible.

The following example illustrates how it can be done. The procedure FillWithPlusMinus fills the long string passed to it with a pattern. In this case, the string itself is already allocated before calling the procedure, thus avoiding the need to call System.pas routines.

```pascal
procedure FillWithPlusMinus(var AString: Ansistring); register;
asm
  push esi
  mov esi, [eax]  {esi now points to our string}
  test esi,esi {if nil, then exit}
  jz @@ending
  mov edx, [esi-4] {edx = length of the string}
  mov eax,'+-+' {pattern to use}
  mov ecx, edx {length in counter register}
  shr ecx,2 {divide, we process 4 bytes at once}
  test ecx,ecx
  jz ecx,ecx
  @@loop:
  mov [esi],eax
  add esi,4
  dec ecx
  jnz @@loop
  @@remain: {fill the remaining bytes}
  mov ecx, edx
  and ecx, 3
  jz @@ending
  @@loop2:
  mov BYTE PTR [esi],al
  shr eax,8
  inc esi
  dec ecx
```

Guido Gybels: Using Assembler in Delphi

32
The above example does not need to call any of the System.pas routines, but it still relies on some implementation specific behaviour, namely where it retrieves the long string’s length. Long strings are preceded by two extra dwords, a 32-bit length indicator at offset -4 and a 32-bit reference count at offset -8. There is no guarantee that this scheme will remain unchanged forever.

To use the above function, allocate a string and then call the routine:

```delphi
procedure DoSomething;
var
  ALine: AnsiString;
begin
  ...
  SetLength(ALine, {Required Length});
  FillWithPlusMinus(ALine);
  ...
end;
```

Alternatively, we could opt to call the appropriate System.pas routine for setting a long string’s length (LStrSetLength), as illustrated in the PlusMinusLine function for Delphi 7 below. From chapter 2 you will remember that with the register calling convention, LineLength, as the first parameter, will go in eax. The Result string is, as explained above, passed as an extra var parameter. In this case our Result parameter is the second parameter, so in case of the register calling convention it will go into edx. Note that we also call UniqueStringA to ensure our result has a reference count of 1. This is necessary, because our manipulation of the string’s content are unknown to the compiler.

```delphi
function PlusMinusLine(LineLength: Integer): AnsiString;
register;
asm
  push esi
  push ebx
  mov ebx, eax  {ebx=LineLength}
  mov esi, edx  {esi=pointer to Result}
  xchg edx, eax {eax=pointer to Result, edx=length}
  call System.@LStrSetLength
  mov ecx, [esi]
  jecxz @ending {if nil, then exit}
  mov eax, esi
  call System.@UniqueStringA
  mov esi, [esi] {esi = first character of string}
  mov eax, '+-+-' {pattern to use}
  mov ecx, ebx {length in counter register}
  shr ecx,2 {divide, we process 4 bytes at once}
  test ecx,ecx
```
jz @@remain
@@loop:
  mov [esi],eax
  add esi,4
  dec ecx
  jnz @@loop
@@remain: {fill the remaining bytes}
  mov ecx, ebx
  and ecx, 3
  jz @@ending
@@loop2:
  mov BYTE PTR [esi],al
  shr eax,8
  inc esi
  dec ecx
  jnz @@loop2
@@ending:
  pop ebx
  pop esi
end;

The example now uses the standard Result mechanism for returning data. It is called simply with the required length to obtain a string:

```pascal
procedure DoSomething;
var
  ALine: AnsiString;
begin
  ...
  ALine:=PlusMinusLine({Required Length});
  ...
end;
```

The above example above was written in Delphi 7. It should be very similar for most other versions of Delphi, although the names for the internal functions do differ somewhat between Delphi versions. Inspecting System.pas should help you in identifying the appropriate function. You can also use the ctrl-left button shortcut on the name of Pascal functions like UniqueString to jump to their System.pas implementations.
Further Reading

The Art of Assembly

Agner Fog: Optimizing subroutines in assembly language: An optimization guide for x86 platforms

Intel 64 and IA-32 Architectures Software Developer’s Manuals
http://www.intel.com/products/processor/manuals/
Table 1: Use of CPU registers

This table summarises register usage in Delphi 32-bit applications. The first column lists the different CPU registers. The second column shows what the register contains upon entry and the third indicates what the register holds upon exit. The fourth column tells you whether or not you are allowed to use the register inside your own code and the last column indicates if you need to preserve the content of the register (save it upon entry and restore it before leaving).

<table>
<thead>
<tr>
<th>Entry</th>
<th>Exit</th>
<th>Use allowed?</th>
<th>Preserve?</th>
</tr>
</thead>
<tbody>
<tr>
<td>eax</td>
<td>Self (1), First parameter (2) or undefined (3)</td>
<td>Function result (4)</td>
<td>✔</td>
</tr>
<tr>
<td>ebx</td>
<td>Unknown</td>
<td>n/a</td>
<td>✔</td>
</tr>
<tr>
<td>ecx</td>
<td>Second parameter (1), Third parameter (2) or undefined (3)</td>
<td>n/a</td>
<td>✔</td>
</tr>
<tr>
<td>edx</td>
<td>First parameter (1), Second parameter (2) or undefined (3)</td>
<td>For Int64 result type: highest dword, otherwise n/a</td>
<td>✔</td>
</tr>
<tr>
<td>esi</td>
<td>Undefined</td>
<td>n/a</td>
<td>✔</td>
</tr>
<tr>
<td>edi</td>
<td>Undefined</td>
<td>n/a</td>
<td>✔</td>
</tr>
<tr>
<td>ebp</td>
<td>Stack frame pointer</td>
<td>Stack frame pointer</td>
<td>✔ (5)</td>
</tr>
<tr>
<td>esp</td>
<td>Stack pointer</td>
<td>Stack pointer</td>
<td>✔ (5)</td>
</tr>
<tr>
<td>cs</td>
<td>Code selector</td>
<td>n/a</td>
<td>✔</td>
</tr>
<tr>
<td>ds</td>
<td>Data selector</td>
<td>n/a</td>
<td>✔</td>
</tr>
<tr>
<td>es</td>
<td>Data selector</td>
<td>n/a</td>
<td>✔</td>
</tr>
<tr>
<td>fs</td>
<td>Thread information block (TIB) selector</td>
<td>n/a</td>
<td>✔</td>
</tr>
<tr>
<td>gs</td>
<td>Reserved</td>
<td>n/a</td>
<td>✔</td>
</tr>
<tr>
<td>ss</td>
<td>Stack selector</td>
<td>Stack selector</td>
<td>✔</td>
</tr>
</tbody>
</table>

(1) Upon entry of methods when using the Register calling convention.
(2) Upon entry of stand-alone functions and procedures while using the Register calling convention.
(3) Upon entry for methods and stand-alone functions and procedures in all calling conventions.
except Register.

(4) Only for Result types that qualify to go into a register. See table 4 for a complete overview of how results are returned to the caller.

(5) While you are allowed to use these registers inside your code, such practise is strongly discouraged. The stack pointer itself changes with every stack operation and its content is therefore highly volatile, making it difficult to manage in code. The ebp register is used to access the stack frame and other uses are therefore to be avoided.
Table 2: Calling Conventions

This table gives an overview of the calling conventions supported by the Delphi compiler. For parameter order, "left-to-right" means that the parameters are pushed into registers or on the stack in the order that they are declared. Correspondingly, "right-to-left" means that parameters are pushed in reverse order.

<table>
<thead>
<tr>
<th>Parameter order</th>
<th>Clean-up by</th>
<th>Common Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>register</td>
<td>left-to-right</td>
<td>Callee</td>
</tr>
<tr>
<td>pascal</td>
<td>left-to-right</td>
<td>Callee</td>
</tr>
<tr>
<td>cdecl</td>
<td>right-to-left</td>
<td>Caller</td>
</tr>
<tr>
<td>stdcall</td>
<td>right-to-left</td>
<td>Callee</td>
</tr>
<tr>
<td>safecall</td>
<td>right-to-left</td>
<td>Callee</td>
</tr>
</tbody>
</table>
Table 3: Parameter Passing

This table gives an overview of how parameters are passed. A distinction is made between passing by value, declared as const or by reference. The “Value in register?” column indicates whether or not a specific type qualifies for passing in a register if it is passed as a value (applicable only to the “By Value” and “Const” columns!). Passing by reference in the form of a 32-bit pointer always qualifies for passing in a register. This is possible for all types, except method pointers, that are always passed via the stack. For example, an int64 type does not qualify according to the table below for passing in a register when it is passed as a value. However, since passing it by reference means passing a 32-bit pointer to the int64 value, passing an int64 by reference qualifies for use in a register.

<table>
<thead>
<tr>
<th>Type</th>
<th>By Value</th>
<th>Const</th>
<th>Value in Register?</th>
<th>By Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>ShortInt</td>
<td>8-bit value&lt;sup&gt;(1)&lt;/sup&gt;</td>
<td>8-bit value&lt;sup&gt;(1)&lt;/sup&gt;</td>
<td>✓</td>
<td>32-bit pointer to 8-bit Value&lt;sup&gt;(1)&lt;/sup&gt;</td>
</tr>
<tr>
<td>SmallInt</td>
<td>16-bit value&lt;sup&gt;(1)&lt;/sup&gt;</td>
<td>16-bit value&lt;sup&gt;(1)&lt;/sup&gt;</td>
<td>✓</td>
<td>32-bit pointer to 16-bit Value&lt;sup&gt;(1)&lt;/sup&gt;</td>
</tr>
<tr>
<td>LongInt</td>
<td>32-bit value</td>
<td>32-bit value</td>
<td>✓</td>
<td>32-bit pointer to 32-bit Value</td>
</tr>
<tr>
<td>Byte</td>
<td>8-bit value&lt;sup&gt;(1)&lt;/sup&gt;</td>
<td>8-bit value&lt;sup&gt;(1)&lt;/sup&gt;</td>
<td>✓</td>
<td>32-bit pointer to 8-bit Value&lt;sup&gt;(1)&lt;/sup&gt;</td>
</tr>
<tr>
<td>Word</td>
<td>16-bit value&lt;sup&gt;(1)&lt;/sup&gt;</td>
<td>16-bit value&lt;sup&gt;(1)&lt;/sup&gt;</td>
<td>✓</td>
<td>32-bit pointer to 16-bit Value&lt;sup&gt;(1)&lt;/sup&gt;</td>
</tr>
<tr>
<td>Dword</td>
<td>32-bit value</td>
<td>32-bit value</td>
<td>✓</td>
<td>32-bit pointer to 32-bit Value</td>
</tr>
<tr>
<td>Int64</td>
<td>32-bit pointer to 64-bit value&lt;sup&gt;2&lt;/sup&gt;</td>
<td>32-bit pointer to 64-bit value&lt;sup&gt;2&lt;/sup&gt;</td>
<td>✗</td>
<td>32-bit pointer to 64-bit Value&lt;sup&gt;2&lt;/sup&gt;</td>
</tr>
<tr>
<td>Boolean</td>
<td>8-bit value&lt;sup&gt;(1)&lt;/sup&gt;</td>
<td>8-bit value&lt;sup&gt;(1)&lt;/sup&gt;</td>
<td>✓</td>
<td>32-bit pointer to 8-bit Value&lt;sup&gt;(1)&lt;/sup&gt;</td>
</tr>
<tr>
<td>ByteBool</td>
<td>8-bit value&lt;sup&gt;(1)&lt;/sup&gt;</td>
<td>8-bit value&lt;sup&gt;(1)&lt;/sup&gt;</td>
<td>✓</td>
<td>32-bit pointer to 8-bit Value&lt;sup&gt;(1)&lt;/sup&gt;</td>
</tr>
<tr>
<td>WordBool</td>
<td>16-bit value&lt;sup&gt;(1)&lt;/sup&gt;</td>
<td>16-bit value&lt;sup&gt;(1)&lt;/sup&gt;</td>
<td>✓</td>
<td>32-bit pointer to 16-bit Value&lt;sup&gt;(1)&lt;/sup&gt;</td>
</tr>
<tr>
<td>LongBool</td>
<td>32-bit value</td>
<td>32-bit value</td>
<td>✓</td>
<td>32-bit pointer to 32-bit Value</td>
</tr>
<tr>
<td>Data Type</td>
<td>By Value</td>
<td>Const</td>
<td>Value in Register?</td>
<td>By Reference</td>
</tr>
<tr>
<td>-------------------</td>
<td>---------------------------</td>
<td>----------------</td>
<td>--------------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>AnsiChar</td>
<td>8-bit value(^{(1)})</td>
<td>8-bit value(^{(1)})</td>
<td>✓</td>
<td>32-bit pointer to 8-bit Value(^{(1)})</td>
</tr>
<tr>
<td>WideChar</td>
<td>16-bit value(^{(1)})</td>
<td>16-bit value(^{(1)})</td>
<td>✓</td>
<td>32-bit pointer to 16-bit Value(^{(1)})</td>
</tr>
<tr>
<td>ShortString</td>
<td>32-bit pointer to the string</td>
<td>32-bit pointer to the string</td>
<td>✓</td>
<td>32-bit pointer to the string</td>
</tr>
<tr>
<td>AnsiString</td>
<td>32-bit pointer to the string</td>
<td>32-bit pointer to the string</td>
<td>✓</td>
<td>32-bit pointer to a string</td>
</tr>
<tr>
<td>Variant</td>
<td>32-bit pointer to the variant</td>
<td>32-bit pointer to the variant</td>
<td>✓</td>
<td>32-bit pointer to the variant</td>
</tr>
<tr>
<td>Pointer</td>
<td>32-bit pointer</td>
<td>32-bit pointer</td>
<td>✓</td>
<td>32-bit pointer to a pointer</td>
</tr>
<tr>
<td>Object reference</td>
<td>32-bit pointer to the object instance</td>
<td>32-bit pointer to the object instance</td>
<td>✓</td>
<td>32-bit pointer to an object instance</td>
</tr>
<tr>
<td>Class reference</td>
<td>32-bit pointer to the class</td>
<td>32-bit pointer to the class</td>
<td>✓</td>
<td>32-bit pointer to a class</td>
</tr>
<tr>
<td>Procedure pointer</td>
<td>32-bit pointer to the procedure/function</td>
<td>32-bit pointer to the procedure/function</td>
<td>✓</td>
<td>32-bit pointer to a procedure/function</td>
</tr>
<tr>
<td>Method pointer</td>
<td>2x 32-bit pointer(^{(3)})</td>
<td>2x 32-bit pointer(^{(3)})</td>
<td>✗</td>
<td>2x 32-bit pointer(^{(3)})</td>
</tr>
<tr>
<td>Set</td>
<td>8/16/32-bit value or 32-bit pointer(^{(4)})</td>
<td>8/16/32-bit value or 32-bit pointer(^{(4)})</td>
<td>✓</td>
<td>32-bit pointer to the set</td>
</tr>
<tr>
<td>Record</td>
<td>8/16/32-bit value or 32-bit pointer(^{(5)})</td>
<td>8/16/32-bit value or 32-bit pointer(^{(5)})</td>
<td>✓</td>
<td>32-bit pointer to the record</td>
</tr>
<tr>
<td>Static Array</td>
<td>8/16/32-bit value or 32-bit pointer(^{(6)})</td>
<td>8/16/32-bit value or 32-bit pointer(^{(6)})</td>
<td>✓</td>
<td>32-bit pointer to the array</td>
</tr>
<tr>
<td>Dynamic Array</td>
<td>32-bit pointer to the array</td>
<td>32-bit pointer to the array</td>
<td>✓</td>
<td>32-bit pointer to a Dynamic Array</td>
</tr>
<tr>
<td>Open Array</td>
<td>32-bit pointer</td>
<td>32-bit pointer</td>
<td>✓</td>
<td>32-bit pointer to an Open Array</td>
</tr>
<tr>
<td>Data Type</td>
<td>By Value</td>
<td>Const</td>
<td>Value in Register?</td>
<td>By Reference</td>
</tr>
<tr>
<td>------------</td>
<td>----------</td>
<td>-------</td>
<td>--------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>Single</td>
<td>32-bit value</td>
<td>32-bit value</td>
<td>✗</td>
<td>32-bit pointer to 32-bit Value</td>
</tr>
<tr>
<td>Double (Real48)</td>
<td>64-bit value</td>
<td>64-bit value</td>
<td>✗</td>
<td>32-bit pointer to 64-bit Value</td>
</tr>
<tr>
<td>Extended</td>
<td>80-bit value $^8$</td>
<td>80-bit value $^8$</td>
<td>✗</td>
<td>32-bit pointer to 80-bit Value $^8$</td>
</tr>
<tr>
<td>Currency</td>
<td>64-bit value</td>
<td>64-bit value</td>
<td>✗</td>
<td>32-bit pointer to 64-bit Value</td>
</tr>
</tbody>
</table>

$^7$ Data types that occupy less than 32-bits will still take up 32-bits. The actual value is stored in the lowest parts of the stack location or register and the content of the remaining part is undefined and should be treated as such at all times.

$^8$ The pointer points to the lowest dword of the value. The highest dword is stored in the next location.

Method pointers are always passed on the stack. They consist of an instance pointer, which is pushed before the actual method pointer, which means the method pointer sits on the lowest address on the stack.

If the Set contents fit into a byte/word/dword, its value is passed immediately, respectively as a 8/16/32 bit value. Otherwise, a 32-bit pointer to the set is passed.

If the record contents fit into a byte/word/dword, the data is passed immediately, respectively as a 8/16/32 bit value. Otherwise, a 32-bit pointer to the record is passed.

If the array contents fit into a byte/word/dword, the data is passed immediately, respectively as a 8/16/32 bit value. Otherwise, a 32-bit pointer to the array is passed.

Open arrays are passed as 2 parameters: the first one is the pointer to the actual array, the second one is the number of elements in the array. As such, passing an open array parameter actually occupies 2 parameter slots. For instance: if you use the register calling convention and you pass one open array parameter eax will contain the pointer to the array and edx will contain the number of elements. See Chapter 2 for details about calling conventions. Also note that open array parameters reside on the stack, so refrain from using very large arrays.

While the value itself occupies only 10 bytes, 12 bytes are actually allocated (3 dwords). The content of the last 2 bytes should be considered undefined.
Table 4: Returning Results

This table gives an overview of how to return results from your assembler routines in Delphi 32-bit applications. The first column lists the different data types. The second column shows where such a type is returned. The third column indicates the nature of the returned result.

<table>
<thead>
<tr>
<th>Returned where?</th>
<th>What?</th>
</tr>
</thead>
<tbody>
<tr>
<td>shortInt</td>
<td>al</td>
</tr>
<tr>
<td>SmallInt(1)</td>
<td>ax</td>
</tr>
<tr>
<td>LongInt(2)</td>
<td>eax</td>
</tr>
<tr>
<td>Byte</td>
<td>al</td>
</tr>
<tr>
<td>Word</td>
<td>ax</td>
</tr>
<tr>
<td>LongWord(3)</td>
<td>eax</td>
</tr>
<tr>
<td>Int64</td>
<td>edx:eax</td>
</tr>
<tr>
<td>Boolean</td>
<td>al</td>
</tr>
<tr>
<td>ByteBool</td>
<td>al</td>
</tr>
<tr>
<td>WordBool</td>
<td>ax</td>
</tr>
<tr>
<td>LongBool</td>
<td>eax</td>
</tr>
<tr>
<td>Single</td>
<td>ST(0)(^{(4)})</td>
</tr>
<tr>
<td>Double</td>
<td>ST(0)(^{(4)})</td>
</tr>
<tr>
<td>Extended</td>
<td>ST(0)(^{(4)})</td>
</tr>
<tr>
<td>Real48</td>
<td>ST(0)(^{(4)})</td>
</tr>
<tr>
<td>Comp</td>
<td>ST(0)(^{(4)}) (^{(5)})</td>
</tr>
<tr>
<td>Currency</td>
<td>ST(0)(^{(4)})</td>
</tr>
<tr>
<td>AnsiChar</td>
<td>al</td>
</tr>
<tr>
<td>WideString</td>
<td>ax</td>
</tr>
<tr>
<td>Byte</td>
<td>al</td>
</tr>
<tr>
<td>AnsiString</td>
<td>extra var parameter(^{(7)})</td>
</tr>
</tbody>
</table>

\(^{(1)}\) The **SHORT** type is the same as SmallInt  
\(^{(2)}\) The **Integer** generic type is currently mapped to LongInt  
\(^{(3)}\) The **DWORD** and **UINT** types are the same as Longword. Also, the generic type **Cardinal** is...
on the 32-bit platform mapped to a Longword

(4) The FPU registers are 80-bits. All results are returned in ST(0). It's only the storage or manipulation that will define them as single, double, etc.

(5) Even though Comp represents a 64-bit integer, it is a type that is manipulated in the FPU, as opposed to the other integer types, which are manipulated using the CPU. As such, Comp follows the rules for real numbers in terms of manipulation and passing around.

(6) Whole number scaled by 10,000

(7) See Chapter 4 for a detailed explanation.
About the Author

Guido Gybels is a veteran Information and Communication Technology (ICT) expert and senior manager, with a proven track record of delivering award-winning innovation, research and development activities, software and hardware engineering projects, standardisation and policy and regulatory strategy in ICT. A former Director of New Technologies and Director of Technology, he is an accomplished senior manager with in-depth experience of planning, reporting, line management and controlling large budgets.

Over the last two decades of his professional career, Guido Gybels has been at the forefront of digital technologies for desktop and mobile alike, with a special interest in Internet applications, usability and user-focused design. His ongoing role as a board member of the Centre for Usable Home Technologies (CUHTec) at the University of York (UK) is testament to his strong commitment to harness modern technology and new media to deliver innovative new solutions in the real world.

Guido combines strong technical knowledge and experience with well-developed leadership skills, strategic thinking, an evidence-based approach to management and an uncompromising commitment to excellence.

In addition to being a very accomplished expert and leader in ICT, Guido Gybels has also demonstrated great political and public relations skills. He has extensive experience as a media spokesperson, successfully delivering high profile media coverage in the broadsheets, on radio and television. He acts as an expert advisor to both the UK government and the European Commission, in which role he interacts directly with senior civil servants, MPs, Ministers and European Commissioners.

While being atypical for the profession, his educational background indicates an agile and curious mind, having studied and published in such diverse fields as history, linguistics, geography, didactics, docimology and psychology.