

Assembly in Qbasic

Petter Holmberg

PDF Conversion 2003 by Thomas Antoni - thomas@antonis.de --- www.antonis.de

What is this?

This document is a collection of the 6 tutorials written by Petter Holmberg, and edited by myself (zkman) for publication from November 1998 to April 1999 in Qbasic: the Magazine, which is located at qbtm.quickbasic.com. This thoroughly covers the use of low-level assembly in the Qbasic programming language. It is intended for users of the Microsoft QuickBasic programming language that are experienced in it's use.

Can I post this on my site?

The tutorials (chapters) in this document are (c) 1998, 1999 Petter Holmberg, and may only be distributed on the following conditions:

This file remains in it's .zip which also contains `absasm21.bas`, retains this document in its entirety, and has the copyright definitions in place.

Who is Petter Holmberg?

Petter Holmberg is a student in Northern Sweden, and a member of the Enhanced Creations programming group (they're located at ec.quickbasic.com). His credentials include the Absolute Assembly program for incorporating assembly into a Qb program, a jpeg loader in Qbasic, work on some of the subs included in DirectQB (the Allegro of the Qbasic online community) and much of the blazing fast engine for Enhanced's new game currently referred to as "Project rt". We are proud to have him as a writer for qbasic: the magazine.

What is Qbasic: the Magazine?

Qbasic: the Magazine is an online e-zine devoted to game programming in Qbasic. We cover everything from assembly such as this to MIDI creation, sprite art, game design, and technical issues, as well as providing news and reviews on various qbasic programs. It is free, and can be viewed at qbtm.quickbasic.com.

Getting Started

Chapter 1

Hello everyone!

This article is written for all of you who wants to learn how to program in assembler in order to enhance your QuickBASIC programs. I know this is a dream for many QB programmers, but they feel it's too complicated to learn, and they haven't found any good sources of information to get started with. If you are one of these programmers, this article is written for you. You will find that it's not easy to learn assembly language programming, but you will probably also find that it's much easier than you first thought. This article will not delve too deeply into assembly language programming, but it will give you a solid start to work on.

So what is assembler then?

The early ancestors of today's computers, developed in the period of about 1940 to 1960, was a real pain to program. The circuits in these computers could perform simple arithmetic operations, they could take data as input, write data as output, and do other operations needed to solve problems for the people that had built them. In order to make the computers understand what they should do, they needed to be fed with instructions. These instructions was given to the computers as series of codes. Let's say the number 1 was the code for adding, the number 2 was the code for subtracting, and the number 3 was the code for outputting the result. The programmers would figure out a program, input it into the computer by turning switches or making holes in paper cards and feed them to the computer. If the program didn't work, the programmers had to go through each instruction again and see where the error was, and then reprogram the computer again. Not very convenient, especially as the programs were all written as a series of ones and zeroes. In order to make programming easier, they started writing the programs in hexadecimal numbers instead of binary numbers. That changed 4 binary digits into one hexadecimal, making the programs shorter and easier to read. But the programs was still just a sequence of numbers, hard to remember and understand for any programmer. So someone had the great idea that they would instead write the instructions as short words, that could be translated directly into numbers and fed to the computer. So instead of saying 1 for an addition, the programmers could say "add", and instead of 2 for subtraction, they could say "sub". Now you could see more clearly what the program did, and finding errors was not as hard anymore. The assembly language was invented.

Later on, computer engineers found out that you could actually make programming a lot easier if you rewrote long sequences of assembly instructions into codes much more like human language. They were called high-level programming languages, and BASIC was one of the first ones. Today's microprocessors still perform their duties as a series of simple instructions, such as "add" and "sub", but programming languages like BASIC makes sure that we usually shouldn't have to worry about it.

Why do I need to learn assembler?

There are many reasons to use a high-level language like BASIC instead of assembler: A simple instruction such as PRINT could in assembler be more than 100 lines of code. It is therefore pretty obvious that BASIC programs are easier to write and debug, and you don't have to worry about what the processor actually does when it writes a letter on the screen. It just works. Another reason to use high-level languages is that you could easily convert your BASIC program on your PC to work on an Amiga computer, using an Amiga BASIC compiler. If you had wrote your program in assembler you would find that the Amiga wouldn't understand it, because it's CPU doesn't work like a PC processor. There are still reasons to use assembler instead of a high-level language: QuickBASIC cannot do everything. There are sometimes things you want to do with the computer that no BASIC instruction can do, and you often find that your BASIC program needs to do so many calculations that the program gets slow. The problem is that such an instruction as PRINT takes many possibilities into account. It makes sure you have a valid string to print, it checks what screen mode you use and what color you want to print the text in and so on. Usually you know all these details when you want to print the text, and you don't need the processor to perform all these checks. The only way to remove them is to use assembler code instead of

PRINT. There's no point in writing a full program in assembler. Only use it when you need to do something really fast or something really low-level.

What do I need to know?

When you write a BASIC program, you don't really need to know much about how the computer works. In assembler you work with the computer on its own level, and therefore you need to know what you're actually doing. You don't need to know very much to get started though, and you will learn the rest as you're learning assembler. The first thing that you will find useful to know is how to count in the binary and hexadecimal system instead of the decimal. This is pretty easy to learn. Usually we count in the decimal system. We then have 10 numbers, ranging from 0 to 9. The lowest number we could use is 0, and as we count upwards we use the numbers 1, 2, 3, 4, 5, 6, 7, 8 and 9. That's all the numbers we have, so in order to continue we need to use two numbers. We reset the 9 to 0, and add a 1 to the right of it. The first number is now worth 10 times the second one. We can now use all combinations of numbers up to 99, and then we need to reset them and add a third number. This suggests that the number 1234 can be expressed as $1 \cdot 10^3 + 2 \cdot 10^2 + 3 \cdot 10^1 + 4 \cdot 10^0$. See the pattern? What if you didn't have 10 numbers to play with? Well, it works just as fine anyway. The binary system, on which computer technology is based, has only 2 possible numbers, 0 and 1.

You start counting from 0, and when you reach 1 you have used all of your numbers and need to add a second one, and you get the number 10. Each new number is worth 2 times the number to the right. The binary number 10110 can thus be expressed as $1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$, or 22. The hexadecimal system works with 16 different numbers. Since we only have invented 10 symbols for numbers, we use letters to represent the higher numbers. The hexadecimal system therefore uses the numbers 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F. The hexadecimal number F3 can therefore be expressed as $15 \cdot 16^1 + 3 \cdot 16^0$, or 243. It's easier to understand if we put the three systems in a table for comparison:

Decimal	Hexadecimal	Binary
0	0	0
1	1	1
2	2	10
3	3	11
4	4	100
5	5	101
6	6	110
7	7	111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111
16	10	10000

As you can see, the number F in hexadecimal is the same as the number 1111 in binary, and this shows why the hexadecimal system is often used in assembly language programming instead of the decimal. If you want the binary number 1111000011110000, you can write it in hexadecimal as F0F0. As you can see, it's easy to convert binary numbers to hexadecimal and hexadecimal numbers to binary.

The number of different digits you can use is called the base of the counting system. You can use any number as a base. If your number in any counting system is, say, 3 digits long, it can be expressed as: $a \cdot \text{base}^2 + b \cdot \text{base}^1 + c \cdot \text{base}^0$, where a, b, and c are your three digits. The most important thing when using different systems simultaneously is to keep track of what system you use for a certain number. For example, is the number 10 the usual decimal for 10, or the binary version of the decimal number 2? If you still haven't understood this, read it again until you do or ask someone who understands it to explain it to you. It's very useful to know about this when you program in assembler.

The second thing that is necessary to know when programming in assembler is the PC memory architecture. I'm not going to explain this in detail, because it's a complicated issue.

A PC has 640 kilobytes of basic memory, and additional megabytes in special memory circuits that you can insert into the computer yourself. The terms EMS and XMS refers to this extra memory. That is not the memory I'm going to talk about here. The interesting thing is the basic 640 kilobytes that every PC has. You need to know how to find a certain position in the memory if you want to use it, and you need to know how to do this if you are going to be an assembly programmer.

Each position in the memory have an address, a number telling the computer where to read or write data. It would have been easy if this address would just have been a number from 0 to 640k, but that's not the system used. A memory position is described by two numbers, called the segment address and the offset address. The actual memory position is a combination of the segment and the offset address.

The segment address describes the memory as groups of 16 bytes. The first byte in the memory, byte 0 if you like, has the segment address 0. The segment address 1 is the 16th byte in memory, and the segment address 2 is the 32nd byte in memory. The offset address is a number telling you how far from the segment position in memory the byte you want is. So if you want to access byte 3 in memory, you use the segment address 0, and the offset address 3. Together they form a number pointing at an exact memory position. Written as a formula this can be expressed as: $\text{actual memory address} = \text{segment} \cdot 16 + \text{offset}$. if you want to access byte 20 in the memory, you use the segment 1, giving you the position 16, and the offset 4, adding 4 bytes to the position, for the final number 20. But you can also use a segment address of 0, and the offset 20, giving you the same memory position! The segment and the offset address numbers can both range from 0 to 65535, giving you several possible combinations when you want to use a certain memory position. This system makes it a little complicated to understand memory addressing to beginners. You can see what segment and offset a certain BASIC variable is located at by using the functions VARSEG and VARPTR. Try it!

Now you might be wondering how it is possible for both the segment and offset variables to be 65535. That gives you the biggest possible memory position of: $65535 \cdot 16 + 65535 = 1114095$, which is bigger than 640k. Well, this memory certainly exists, but it is not

accessible as the first 640k of memory, and I'm not going to delve deeper into this here and now. Later on, I will discuss memory access in more detail.

Again, if you didn't understand this, read it again, and if that didn't help, ask someone to explain it to you. This was all for the first part of this article: A very brief introduction to what's about to come. The next time I will start describing the basics of assembler and how you use it in QuickBASIC. Make sure you understand the different numbering systems and the memory addressing scheme until then.

The Basics

Chapter 2

Last time I discussed the history of assembler and told you where to use it and not. I also gave you some information about the binary and hexadecimal system, and briefly explained how the base memory is addressed. This background information was needed to give you a good start in the learning process. This time I will teach you the basics of the assembly language and show you how to use it in QuickBASIC. Let's get to it!

How the heck can I execute assembly code in QuickBASIC? This might be the first question you're asking yourselves. How can you make QuickBASIC understand assembly code? Well, you can't. QuickBASIC can only understand regular BASIC expressions. However, it is possible to make QuickBASIC execute snippets of machine language code in a program. Machine language is the only language the processor really understands, and the BASIC code you usually see is translated into machine language instructions when you run the program. But as assembly code basically is machine language represented in a more humane way, all you need is a program that translates your assembly code into machine code, and the knowledge on how to get QuickBASIC to run that machine code.

When converting a program written in a high-level language such as QuickBASIC to machine code, you say that you compile a program. When you do the same with an assembly program, you say that you assemble the program. A program that does this is called an assembler. Do not mix up the expressions here! Now you may be thinking that you don't have an assembler on your hard drive, but that's where you're wrong. All Microsoft operating systems, from MS-DOS to Win98 have a program called DEBUG somewhere. This program was included in Microsoft OS's as a tool for advanced users, and it has the possibility to convert raw assembly code to machine code and the reverse. This program can be very useful, but it's also very hard to use. Luckily, you won't need to worry about that. I will explain this later.

There are two ways to run machine code in QuickBASIC. I will start by only explaining the first one. QBASIC and QuickBASIC both have a built-in function called CALL ABSOLUTE. With CALL ABSOLUTE you can execute a machine language routine and then return to QuickBASIC. If you use the standard QBASIC, you can use CALL

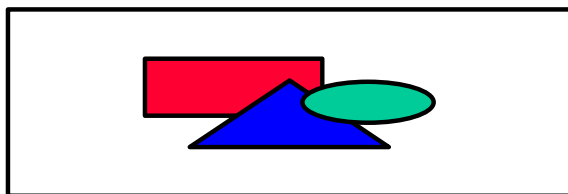
ABSOLUTE directly, but in QuickBASIC it's included in the external library QB.QLB/QB.LIB. So if you use QuickBASIC, you must start it with the syntax QB /L, which includes this library.

To begin with, we will work with DEBUG and CALL ABSOLUTE as our tools to learn assembler. But as I told you, DEBUG is hard to use, so I have created a program that makes everything a whole lot easier. It is called Absolute Assembly and is included in the zip file this document is contained in. The last version was written many months ago, but it works fine. This program relieves you from the pain of using DEBUG manually to create a program. It takes a raw text file with assembly code as input, and through DEBUG generates a snippet of QuickBASIC code in a file of your choice. I'll give you the details as we continue.

The basics of assembler

Now we are ready to begin discussing some serious stuff: The first steps into the asm world! First of all: When working with assembler, you mainly process a lot of numbers. These numbers need to be stored somewhere. You can of course use the memory to store your data, but there's another way to do it: Through registers. Registers are a bit like variables, but they're not stored in the memory as variables are. They are stored in the microprocessor, where they can be accessed instantly and effectively. However, there are only a few of them, so you'll have to use them carefully and keep track of what you're doing with them. Many registers also have special uses, so you can, and must, use them only in certain places. This may seem a little confusing to you right now, but you will soon understand how it works.

There are four basic registers that can be used for almost anything. They are called AX, BX, CX and DX. You can think of these registers as INTEGER variables in QuickBASIC. They are small memory cells that can store a 16-bit large number. If you want to use only one of the two bytes in these registers, you can do so by calling them AH/AL, BH/BL, CH/CL, and DH/DL. The H and L stands for "high" and "low". So you can use only the upper 8 bits of the AX register by calling it AH, and the lower 8 bits by calling it AL. On 386 computers and later, you can also call these registers EAX, EBX, ECX and EDX, and use them to store 32-bit large numbers. I'll better draw this to make you understand it:



Writing data into AL doesn't affect AH, but it affects AX and EAX. The same rule goes for BX, CX and DX. Remember that this is all just different names to access different parts in the same register. As DEBUG cannot handle any 386 or above processor instructions, we won't be using 32 bits registers as long as we're working with it.

Although these four general purpose registers can be used for almost everything, they also have special purposes. The A, B, C and D in the registers actually stand for Accumulator, Base, Counter and Data. I will tell you when they should be used when we come to such situations. There are many other registers with more special uses that you will need to learn,

but I will return to them later when we need them. Now we're going to learn our first assembly instruction!

MOV, your key to data transfer

The most common and important assembly instruction is called MOV. Assemblers in most platforms have this instruction. Its purpose is to move or copy values between memory and registers. As you probably guessed, MOV is short for move. Most assembly instructions are three letters long. The name is a little misleading, because moving a value would mean taking it away from the source, but MOV actually copies the value. The general syntax for MOV is:

```
MOV destination, source
```

Beginners always tend to mix up the positions of the source and destination with MOV. It may seem more natural to put the source first, but if you think about it you'll see that you do the same in BASIC. (destination = source) The source can be a direct number, a register or a value in the memory. The destination can be a register or a memory position. Here are some examples: If you want to put the value 8 in the AX register, you type:

```
MOV AX, 8
```

If you would like to copy the contents of the CH register into BL, you type:

```
MOV CH, BL
```

A thing that you cannot do is to move a 16-bit value into an 8-bit register. An instruction like MOV AL, BX is therefore not possible. But what about values in the memory? Well, then you must learn to use three new registers: DS, SI and DI.

Accessing the memory:

In order to be able to read and write in the memory, you need the special memory addressing registers. If you want to read data from the memory, you must put the memory address into the two registers DS and SI. Their full names are the Data Segment register and the Source Index register. In DS, you should put the segment address of the memory position you want to read from, and in SI you should put the offset address. How this works was discussed in part 1 of this tutorial.

Suppose you want to copy byte number 18 in the memory into AL. Then you need the following assembly code:

```
MOV BX, 1
MOV DS, BX
MOV SI, 2
MOV AL, [SI]
```


This requires some explaining. First of all: It's impossible to move direct values into the DS register. Don't ask me why, but it has to do with the Intel PC processor architecture. So what you need to do is to put a value in one of the general purpose registers, here I used BX, often used in this situation, and then copy the contents of that register into DS. That explains the first two lines. Next we put a value into the SI registers. Luckily this can be done directly. We wanted memory position 18, and now the DS register is 1 and the SI register is 2. Remember that the actual memory position is the segment times 16 plus the offset. In our case this gives us $1 \times 16 + 2 = 18$. The final line copies the byte located at this memory position into AL. The brackets [] around SI means that the computer should fetch the byte at the memory position pointed out by SI, (DS is automatically assumed to hold the correct segment address) instead of the value in SI itself. Without the brackets, the value of SI, 1, would get into AL. Well, actually that wouldn't work because AL is only 8 bits and SI is 16 bits. If it had been the whole AX register it would work though. But with the brackets, the value from memory position $DS \times 16 + SI$ is read.

If you want to do the opposite, write to memory, you use DI instead of SI. DI is short for Destination Index. Writing data is done in almost the same way as reading data. If you would like to write the value 5 into the same memory position as we were reading from in the previous example, you type this:

```
MOV BX, 1
MOV DS, BX
MOV DI, 2
MOV AL, 5
MOV [DI], AL
```

Easy, huh? Here the DS register is also used for the segment address. In this example, AL, an 8 bit register part, is used, and so 8 bits will be written to memory. If you had changed AL to AX, 16 bits would have been written.

Now you should know how to read and write values from/to registers, and how to access specific memory positions. But you can't do much with it yet. Now it's time to learn how to call asm routines from QB!

The interface

Now you need to have both DEBUG and Absolute Assembly, the two programs I talked about earlier. You won't have to know how to use DEBUG, because Absolute Assembly will do all the dirty work for you. You can use your favorite text editor when creating your assembly routines. Just make sure the code is saved in a file of the standard ASCII .TXT format. When you want the code to be fed into QB, you just start Absolute Assembly. You will be asked to type in the name of the text file containing the asm source, the QB program file to put the code in, and the name of a code string. Make sure you have saved the BAS file in standard ASCII format (this only applies to QuickBASIC users). The code string is a string variable name that is going to be used in the QB program to access the code. If you're making an asm routine to draw pixels on the screen, you should call it drawpixel\$ or something like that. You don't have to type in the \$ sign in Absolute Assembly. Next, you will be asked to answer yes or no to two questions. The first asks you if you want to append the code to the basic source file. If you answer no, your BAS file will be cleared before the code is written to it. If you answer yes, the ASM code will end up in the bottom of the file, without erasing its

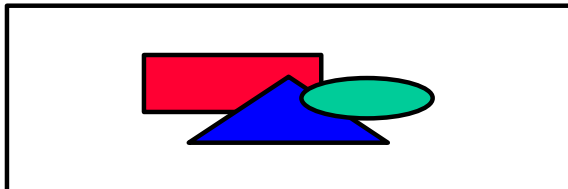
old contents. The other question is if you want to add CALL ABSOLUTE lines to the program. This is a little hard to explain right now. I'll get back to it soon.

Our first assembly routine

It's time to try writing an asm routine for QB. The first test routine won't do anything, it's just a test. First, start your favorite text editor and type in the following lines:

```
PUSH BP
MOV BP, SP
POP BP
RETF
```

Now you wonder what this means, but don't worry, I will explain it to you. The first line is a new assembly instruction for you: PUSH. This introduces a new part of assembly programming. PUSH is an instruction used to put values on the stack. What is the stack then? Well, it's a part of the memory that can be used to store temporary values. You will often have the need to keep track of more numbers than there are registers, and the most convenient way to go then is to use the stack. The stack is a place where you can shuffle away values until you want to use them. PUSH is the instruction you use to copy a value to the stack. When you want to get the value back, you use the opposite of PUSH, an instruction called POP. You can see that POP is used on the third line of the program. There's a special register that keeps track of where in the memory the stack is located. It is called the Stack Pointer, or SP. When you use PUSH, the value goes to the memory address in SP. Then SP is changed, so that it points at a new memory position in the stack. This works a little strange. You can think of the stack as a stack of plates. When you use PUSH, it's like you were putting a new plate on the stack. When you use POP, you remove it, revealing the plate underneath. So you must keep order of the values you put on the stack.



Consider this example:

```
PUSH AX
PUSH BX
POP AX
POP BX
```

First, AX is copied to the stack, and then BX. When the first POP instruction is called, the value that was last put on the stack, the BX value, is returned to AX. When the second POP is called, the first value of AX gets into BX. So this example will actually swap the values in

AX and BX, using the stack. If you want the values to get back in the right order, you must POP them in the opposite order:

```
PUSH AX
PUSH BX
POP BX
POP AX
```

This would correctly return the values to their original registers. This is a technique called LIFO, and it stands for Last In, First Out. Get it? Of course, there's no reason to PUSH and POP values like in the example above, but if you needed AX and BX for other things between the PUSH and POP calls, you would find it very useful.

There's a strange thing about the stack that you need to know also. The image of a stack of plates is not entirely correct. The value in the SP register is not increased after each PUSH, it's actually decreased! So it would be more like a stack of plates turned upside down, even though earthly physics obviously wouldn't accept stacks of plates hanging upside down on the roof. :-) This is not something you need to care about now though.

Now, let's return to the test program. As you can see, the PUSH instruction uses a register called BP. This is another new thing for you. BP is short for Base Pointer, and it is a value that points to the base of the stack. So in the plate example, it would point at the plate in the bottom of the stack, (if you ignore the upside down-thing for a while). It's essential that this value stays the same before and after the call to the asm routine, because QB uses it too. So therefore we always PUSH it in the beginning of the routine, and then POP:s it back in the end.

Now we have come to the second line. By now you should understand what it does: It puts the value of the SP register in BP. Now the computer will think that the bottom of the stack is at what's actually the top of the stack. In the next part of this tutorial series I will explain why we do this. After the first two lines, we would be free to write anything, but as we don't want the routine to do anything yet, we'll just POP the old value of BP back and return to QB. The last line with an instruction called RETF, which stands for Return Far, will make sure we get back to the BASIC program.

Let's try this out:

- First, type the four lines into a text file and save it. Let's call it ASMTEST.TXT!
- Then: Run Absolute Assembly.
- First you will be asked to type in the name of the asm source file. Type ASMTEST.TXT. Usually, assembly source files have the extension .ASM, but it doesn't matter what name you use.
- Then you type in the name of the BASIC source file. Since we don't have one, just type ASMTEST.BAS, and this file will be created.
- Then you must type in the name of the code string. call it test.
- The program now asks if you want to append the asm code to the BASIC file. Since the BAS file is empty, press the N key for no.
- Finally the program asks if you want to add CALL ABSOLUTE lines. Press Y.

- Now DEBUG should be executed by the program, and it will prompt you what is happening. If everything was done correctly, the program will ask you if you want to convert another file. Press N and exit the program.
- Now open the file ASMTEST.BAS in QB. You will see this:

```
' ----- Created with Absolute Assembly 2.1 by Petter Holmberg, -97. ----- '
test$ = " "
test$ = test$ + CHR$( &H55 )           ' PUSH BP
test$ = test$ + CHR$( &H89 ) + CHR$( &HE5 ) ' MOV BP,SP
test$ = test$ + CHR$( &H5D )           ' POP BP
test$ = test$ + CHR$( &HCB )           ' RETF

offset% = SADD(test$)
DEF SEG = VARSEG(test$)
CALL ABSOLUTE(offset%)
DEF SEG
' ----- Created with Absolute Assembly 2.1 by Petter Holmberg, -97. ----- '
```

As you see, there's now a string variable called test\$. For each line, numbers (converted to ASCII codes) are added to the string. On the right, you can see the assembly instructions you typed in earlier as comments. For each line, one assembly instruction, or more correctly, one machine language equivalent to an assembly instruction, is added to the string. Because we answered yes to the question if we wanted to add CALL ABSOLUTE lines to the BAS file, there are also four other lines under the test\$ declaration. the offset% variable gets the offset address of the string test\$, and the DEF SEG instruction makes sure the default segment is the segment of the test\$ string. (DEF SEG in QB is almost the same as typing MOV DS, BX in assembler)

And now comes the CALL ABSOLUTE call. This line will execute the code located at the start of the test\$ string. As this test program doesn't really do anything, you won't see anything happening. Finally, the second DEF SEG resets the default QB segment. Run the program and make sure it actually works! It won't do anything, but just the fact that it didn't crash is enough to make an assembly programmer happy!

This is the end of the second part of my assembly tutorial. I've tried to go slowly in the beginning so that you would understand everything, but now the vital basics of assembler should be crystal clear for you. The next time we can start the fun! I will teach you more assembler instructions, and we will start writing programs that can do something useful, such as manipulating BASIC variables and returning the answer.

Mathematics

Chapter 3

It's time for the third part of my assembly tutorial. I've had some very positive response to the two earlier parts, mainly from people saying it's the first assembly tutorial they've read where they understand everything. I'm glad to hear this, because that was my intention with this tutorial series. There are too many texts that explain things in such a hurry that you have a problem understanding it.

In the previous part of this tutorial, I showed you how to use my program Absolute Assembly - with a little help from Microsoft's DEBUG - to make an assembly routine run in QBASIC. Using that knowledge, we can now start writing assembly and see it run. In order to do that we need to know more assembly instructions, so that's what this part mainly is about. When you've read through it you will have enough knowledge to start making your own experiments. But first, let's repeat the important parts from the last time!

The registers

As discussed in the last part of this tutorial, registers are an important part of assembly programming. Registers are memory cells in the CPU that keep track of numbers important to the computer. Some of them have specific tasks to perform, while others can be used freely by the assembly programmer. There are four basic registers that you will use often: AX, BX, CX and DX. They work like integer variables in QBASIC. There's also a register called DS, used to store a memory segment address when reading/writing from/to the memory. The two registers SI and DI are used to store the offset address. There are other important registers that still have to be discussed, but we'll get to that later on.

The stack

The stack is an important part of assembly programming. It's an area in memory allocated by the program as a place to store temporary values that can't fit into the registers. A register can be put on the stack with the assembly instruction PUSH. It is returned with an instruction called POP. The stack has to be properly maintained, or else the program will most likely crash. When writing assembly routines in QBASIC, the stack must be in the same state before and after the routine was called. Two specific stack registers were discussed: SP and BP. More about the stack will be discussed in this part.

MOV

The most universal of assembly instructions, MOV, was also explained. This is the assembly instruction used to exchange values between the registers and the memory. MOV is used very frequently in assembly programs as you soon will see.

I also introduced a sequence of four assembly instructions that you will use a lot in the future:

```
PUSH BP
MOV BP, SP
POP BP
RETF
```

They will be the base for many of the assembly routines that you will write, and you will soon understand why.

Now it's time for more assembly instructions.

Arithmetic operations

One of the most primary requirements a programming language must fulfill is the ability to perform calculations on numbers. Since assembler is a direct translation from machine language, the language of the CPU, it naturally has assembly instructions for the most primary arithmetic operations. First of all, you must have the ability to perform additions and subtractions between numbers. This is pretty easy to do in assembler. The instructions needed are ADD and SUB. I believe it's pretty clear what they stand for. The general syntax for these instructions are:

ADD Destination, source

And:

SUB destination, source

Just like with MOV, the destination is where the result of the operation is placed. It can be a register or a pointer to a memory address. The source can be a register, a memory pointer or a direct value. Let's consider an example: If you would like to see the result of a subtraction of 4 from the value 5 in the AX register, you could test it like this:

```
MOV AX, 5
SUB AX, 4
```

The number 5 gets into the AX register with MOV, and then it's subtracted by 4 using SUB. The result, 1, will be in the AX register. Another example: Let's imagine you want to add 2 to the byte value at memory address 3:3, and put the answer in the CH register. Then you could write:

```
MOV BX, 3
MOV DS, BX
MOV SI, 3
MOV CH, (SI)
ADD CH, 2
```

As explained in the last part of this tutorial, the first four lines are an example of reading a byte from the memory and putting it into a register. The ADD instruction then adds 2 to the value located in CH.

There are also two special case instructions for additions and subtractions. If you only want to add or subtract the number 1 from a value, you can use the instructions INC or DEC (short for Increase and Decrease). These instructions are performed faster by the processor than ADD and SUB, at least by older processors. They are also very easy to use. They only take one argument: The destination. It's recommended that you use INC X/DEC X instead of ADD X, 1/DEC X, 1. An example: You want to increase the value in the AL register with 1. The you just type:

```
INC AX
```

Could it be any easier?

There are more things you probably want to do. How about multiplication and divisions? Well, you can certainly do that too, using the instructions MUL and DIV. They are a little harder to use though. The MUL instruction only takes one operand, like this:

MUL source

The source cannot be a direct number, it has to be either a register or a memory pointer. If the source value is an 8 bit number, it will be multiplied with the value in the AL register, and the answer gets into the AX register. This is because the product of two 8 bit numbers can be 16 bits long. For example, suppose we want to multiply 100 with 200. We can do that like this:

```
MOV AL, 100
MOV DL, 200
MUL DL
```

The result, 20000, will fill up the whole AX register since it wouldn't fit into one byte. What about 16 bit numbers then? Well, since DEBUG cannot handle 32 bit registers, the solution looks somewhat different. If we use a 16 bit number as source, it is multiplied with the whole AX register and the result goes into DX:AX. That means that the high 16 bits goes into DX and the low 16 bits goes into AX. In many cases you, the programmer, knows that a multiplication won't have a product that exceeds 16 bits, so you can ignore the DX register. But if you have an important number in the DX register before the multiplication it will be lost anyway. Division is similar to multiplication. The syntax for DIV is:

DIV source

Like with MUL, the source must be a register or a memory pointer, not a direct value. If the value is 8 bits, the whole 16 bit number in AX is divided by the source. The quotient of the division goes into AL and the remainder into AH. If the value is 16 bits, the number in the DX:AX register pair is divided by the source and the quotient goes into AX and the remainder into DX. This tells us that DIV can only perform integer division, like the \ sign in QBASIC, which is unfortunately true. For the sake of clarity, let's make an example: We want to divide 5 by 2, which would result in a quotient of 2 and a remainder of 1. Let's try it:

```
MOV AL, 5
MOV DL, 2
DIV DL
```

Now the number 2 would be in AL and the number 1 in AH. As you can see, multiplications and divisions are trickier than additions and subtractions. Multiplications and divisions are also very slow compared to additions and subtractions. If you want to make a really fast assembly routine, try to avoid MUL and DIV as much as possible. One of the tricks to do that will be showed later.

There's also another thing about multiplications and divisions I have to tell you about. The MUL and DIV instruction only works with unsigned values, which means values that are positive. If you want to do multiplications or divisions with signed values, values that are negative, you must instead use the instructions IMUL and IDIV. They work with both positive and negative numbers, and works in the same way as MUL and DIV. But if you know that you only have positive values, use MUL and DIV for clarity purposes. That was the important stuff about arithmetic operations in assembler.

Logical instructions

There's another set of instructions that are very important in assembler: The logical instructions. They are not as easy to understand as the arithmetic ones, but they are extremely useful. These instructions also exists in QBASIC, so you can easily play around with them in order to understand how they work.

There are four arithmetic instructions in assembler: AND, OR, XOR and NOT. They have the same names in QBASIC. The processor has no problems executing these instructions, because it works with logical instructions for almost everything it does. Basic operations like additions and subtractions are performed as a series of logical instructions within the small semiconductive transistors in the CPU. Executing logical assembly instructions are therefore also one of the fastest operations the CPU can perform. Let's begin exploring these instructions!

Logical instructions doesn't treat the numbers in the computer as numbers, they operate on the individual bits in the number. So you won't understand the logical instructions if you look at what happens to the numbers you pass to them. You must study the individual bits.

The first logical instruction we will look at is NOT. It's a little different from the other three, and it's also the easiest to understand. NOT inverts all the bits in a number. So if you perform a NOT on the binary 8 bit number 00001111, you will get the result 11110000. If you NOT the number 10101100, you get 01010011 and so forth. The NOT instruction only takes one argument:

NOT destination

The destination can be a register or a memory pointer. The other three instructions, AND, OR and XOR, takes two arguments:

AND destination, source

OR destination, source
XOR destination, source

AND works like this: The destination value is compared bitwise against the source value, and only in the case where both corresponding bits are 1, the result will be 1. Otherwise it will be 0. So if you AND the value 11110000 with 10101010, you will get the result 10100000. 11001100 AND 10101010 will give the result 10001000 and so forth. The name of the instruction can be used as a remainder of its use: If bit A AND bit B is a 1, the result will be 1. Otherwise it will be 0.

OR is similar to AND, but it works in the opposite way. If both bits are 0, the result will be 0. Otherwise it will be 1. Or you can also say that if at least one of the bits is 1, the result will be 1. 11110000 OR 10101010 is therefore 11111010, and 11001100 OR 10101010 is 11101110. If you want to use the name as a remainder, you can say that if bit a OR bit b is a 1, the result will be 1. Otherwise it will be 0.

XOR is perhaps the most interesting logical instruction. The name stands for eXclusive OR. It works like OR except for one detail: If both bits are 1, the result is 0. You can also say that the two compared bits must be different if the result should be a 1. 11110000 XOR 10101010 is therefore 01011010, and 11001100 XOR 10101010 is 01100110. If this has made you dizzy, we better summarize the four logical instructions in a table:

AND:

0 AND 0 = 0
0 AND 1 = 0
1 AND 0 = 0
1 AND 1 = 1

OR:

0 OR 0 = 0
0 OR 1 = 1
1 OR 0 = 1
1 OR 1 = 1

XOR:

0 XOR 0 = 0
0 XOR 1 = 1
1 XOR 0 = 1
1 XOR 1 = 0

NOT:

NOT 0 = 1
NOT 1 = 0

Now, what use can we have for logical instructions then? Well, there are several neat things you can do with them. AND can be used as a filter if you only want to read special bits in a number. For example, if you have a number in the AL register and you want to take away the four highest bits in the number. Then you only AND it with the binary number 00001111. Let's suppose the number in AL was 10010011. The result will then be 00000011. Notice how the four high bits have been filtered away. If you check the table above you can probably figure out why this works.

We covered the binary numbering system in part 1 of this tutorial, and if you remember it, you know that the bits in a binary number are "worth" 1, 2, 4, 8, 16, 32, 64, 128 and so forth if you count from right to left. Notice how only one of these numbers are odd? That's right: the first one. This implies that if a number is odd, it MUST have the binary digit worth 1 set. With a little help from AND, you can then test if any number is odd. Let's test the number 5. 5 in binary is 00000101. If we AND this with 00001111, we get the result 00000001. The result is 1, so 5 must be odd. If we test 10 instead, we get: 00001010 AND 00000001 = 00000000. The result is 0, so 10 must be an even number. Test this in QBASIC!

XOR also have some nice uses. One of the most common ones uses the fact that if both bits compared with XOR are the same, the resulting bit is 0. This means that if you compare a number against itself, the result must always be 0 no matter what the number is, because all bits always are the same. For example, 11001100 XOR 11000011 = 00000000. This works no matter what number you use. Remember what I said about the speed of logical instructions? These two facts together suggests that:

```
XOR AL, AL
```

is faster than:

```
MOV AL, 0
```

And that is certainly true! At least with older processors. This trick is often used by assembly programmers. Rather than just setting a register to 0, you XOR the register with itself, thus getting the same result faster. XOR can also be used for very simple data encryption. Take any number and XOR it with a number X. You will probably get a result that makes no sense. If you then XOR it again with the same number X, you get the original number back! Cool, huh? Let's test it: We have the top secret number 10011011. Now we use the secret pre key 10110110 to encrypt it, using XOR. The result is 00101101. you can't see any connection to the original number, can you? Then we "unlock" the number with our pre key and XOR again. The result is now 10011011. Wow! Our secret number is back! Of course you won't see Pentagon using this not too sophisticated encryption scheme for their secret documents, but maybe you'll have some private use for it? Test this in QBASIC too and make sure I'm not lying to you!

Shift and rotation operations

Finally I thought I'd go through another part of assembly programming: Shifts and rotations. These operations also work with the individual bits, and they also have the nice properties of being really fast, just like the logical instructions.

I'll begin by explaining shifts. Shift instructions are not too hard to understand. Their purpose is to move all the bits in a number a certain amount of bit positions in a certain direction. There are two basic shift instructions SHL and SHR, short for Shift Left and Shift Right. If you have the binary number 00111001 and shift it left one position, you get the number 01110010. If you shift it right one position, you get the number 00011100. Get it? It's like taking away one bit at one end of the number, move all the others one step to fill up the hole, and put a 0 in the empty place at the other end.

The shift instructions are not too hard to use. The basic syntax for the two shift instructions are:

```
SHL source, count  
SHR source, count
```

The source can be a register or a memory pointer. The count value tells the CPU the amount of bits to shift. Here comes a little quirk though: The 8086 processor only allowed the count to be the number 1 or the contents of the CL register. The instruction SHL AL, 2 was therefore not valid. Later Intel processors can use any direct number. BEBUG however, only supports the basic 8086/8088 assembly instructions, so we must use the CL register if we want to shift a number more than one bit position. If we want to shift the contents of the BH register four steps to the right, we must then type:

```
MOV CL, 4  
SHR BH, CL
```

What can we use shift instructions to then? Well, there's a very neat use for it that often comes in handy. Remember what I said about the slowness of multiplications and divisions? Well, certain multiplications and divisions can be done with shift instructions, making them even faster than additions and subtractions! In the binary world, you double the value of a bit if you move it one step to the left. The binary number 100 is two times as big as the number 10 and so forth. Therefore, shifting a number one step to the left is the same as multiplying it by two. If you shift it two steps you multiply it by four and so on. So if you want to multiply a number with 2^x , for example 128, you can use a SHL instruction instead. Let's try it! Suppose we want to multiply the number 10 with 8, you can either type the slow:

```
MOV AL, 10  
MOV DL, 8  
MUL DL
```

Or, you could type the much faster:

```
MOV AL, 10  
MOV CL, 3  
SHL AL, CL
```

Get it? The same goes for division, but then you use SHR instead. The only thing you have to watch out for is that the shift instructions may push some ones over "the edge" of the register, and then you will get an incorrect answer. So make sure the numbers you want to multiply doesn't get bigger than 8 bits. Or 16 bits if you're dealing with bigger numbers.

Rotations works like shifts, but they don't throw away any bits. The bits that disappears on one side of the number, are put on the other side. So if you rotate the number 10100110 two steps to the left, the result will be 10011010. The 10 that was pushed away at the left edge, are moved to the new, empty spaces at the right edge. A rotation of a byte eight steps in any direction would not modify the byte at all because all the bits would be rotated to the same positions that they were at from the beginning. The two instructions needed are ROL and ROR, short for Rotate Left and Rotate Right. The basic syntax is exactly the same as for SHL and SHR, and the count value can be either 1 or the contents of the CL register. If you want to rotate the byte in AL four steps to the right, you simply type:

```
MOV CL, 4  
ROR AL
```

Passing values between QBASIC and the assembly program

Okay: Now We've been going through 16 basic assembly instructions: ADD, SUB, INC, DEC, MUL, IMUL, DIV, IDIV, AND, OR, XOR, NOT, SHL, SHR, ROL and ROR. With these at your hand you can do many things. Now you probably want to test this in reality, using QBASIC. But there's no way you can watch the results of these calculations in QBASIC yet. Therefore, we must learn how to pass variables between QBASIC and assembly routines.

Last time, you saw how an assembly routine could be called with the CALL ABSOLUTE keyword. The syntax for CALL ABSOLUTE is:

```
CALL ABSOLUTE(offset)
```

Where offset is the offset address of the string/array that contains the machine pre you want to execute. The segment address must be set with a DEF SEG before the call. If you want to pass variables to the routine, you do so by putting them before the offset specification. This is what I mean: Suppose you want to pass the integer variables a% and b% to the assembly routine. You then type:

```
CALL ABSOLUTE(a%, b%, offset%)
```

This will ensure that the a% and b% variables are passed to the assembly routine. Exactly how this works will be explained in just a minute. First I must point out that CALL ABSOLUTE can only pass variables of the type INTEGER. LONG variables, SINGLE and DOUBLE variables, strings, user data type variables and arrays can NOT be passed to your assembly routines. If you have the need to do that you must instead pass two integer variables, describing the segment and offset address of the variable you really want to send.

How can you read the variables in your assembly pre then? Now it's time to look back on the four golden lines that were presented in the last part:

```
PUSH BP  
MOV BP, SP  
POP BP
```

RETF

When QBASIC executes a CALL ABSOLUTE instruction, the segment and offset address of the next QBASIC instruction is pushed onto the stack. That is 4 bytes. If you add variables to the CALL ABSOLUTE line, these are also pushed onto the stack, BEFORE the BASIC segment/offset pair. They are pushed in the order they appear inside the parentheses after the CALL ABSOLUTE statement. Now, the first assembly instruction above pushes the BP register onto the stack. BP tells the program on what offset the bottom of the stack is located. Then, the contents of SP is copied into BP. SP tells the program where the top of the stack is. Now the computer thinks that the stack starts in the end. This comes in handy, because it is possible to fetch a byte from the memory like this:

MOV BX, (BP)

If we use the BX register as destination, we can get the two bytes located at the memory position SS:BP. The SS register is a new register to you. It contains the segment address of the stack. It's rarely used by assembly programmers though. Anyway, it is also possible to get the two bytes at a position RELATIVE to SS:BP. Let's say we want the word (a two byte number is called a word in assembler) 5 bytes above SS:BP. We then type:

MOV BX, (BP + 5)

If we want the word 8 bytes below SS:BP, we type:

MOV BX, (BP - 8)

Now, remember that I said last time that the stack is like a stack of plates turned upside-down, i.e. the stack grows DOWNWARDS as you push values onto it. The values already pushed on the stack are thus at higher memory addresses than the current SP value. Since we put the contents of the SP register in BP, we can now find our variables by searching at memory addresses above SP:BP. The current value of BP points at the value that was last pushed onto the stack. We just pushed the original BP value, so that's what we'll find that. At BP+2, we'll find the next value. Before QBASIC gave our assembly routine the control of the program flow, CALL ABSOLUTE pushed the segment and offset of the next QBASIC instruction onto the stack, so on BP+2 and BP+4 you'll find these values, numbers that are of no interest to us. It is after that, at BP+6 and above, that we'll find our variables. If you wrote:

CALL ABSOLUTE(myvariable%, offset%)

myvariable% would be located at BP+6. If you wrote:

CALL ABSOLUTE(x%, Y%, z%, offset%)

x% would be at BP+10, y% at BP+8 and z% at BP+6. This may sound confusing, but QBASIC pushes the variables in the order they appear inside the parentheses. x% will therefore have the highest memory address in the stack. The last variable will always be the closest one to BP. If you use PUSH in your own assembly pre, you can read them without using POP in the same way you read the QBASIC variables. The first value you push will be at BP-2, the second one at BP-4 and so on.

Let's suppose you want to get the value of the variable x% into the AX register. Then you only type:

```
PUSH BP
MOV BP, SP
MOV BX, (BP + 6)
MOV AX, BX
```

Right?

Wrong! You will discover that the value in AX is not the value you had in the x% variable in QBASIC. Why?

As if our problems weren't enough, QBASIC hasn't passed the value of x% to the stack. The number at BP+6 is the OFFSET address of the x% variable in the memory. This makes things a little harder to grasp, but as you'll see, this can be very useful.

But first, let's solve this new problem in QBASIC. If you use the QBASIC BYVAL clause before the variable, your problems will be solved. This is what I mean:

```
CALL ABSOLUTE(BYVAL x%, offset%)
```

Now, QBASIC won't push the offset of the variable x% on the stack. It now pushes the actual value of x%. Now you can use these lines to get the value of x% in the AX register:

```
PUSH BP
MOV BP, SP
MOV BX, (BP + 6)
MOV AX, BX
```

Wow! Now we can pass variables from QBASIC to an assembly routine. Now we can use ADD, OR, SHL or any other instruction to modify the values in cool ways. Well, even if that's certainly true, we won't have much use for it if we couldn't pass the modified values back to QBASIC again. How can we do that then? Well, we need to know the memory addresses of the QBASIC variables from within our assembly routine in order to change them. If we just let QBASIC push the values of some variables onto the stack, we can read them, but we cannot return any values, because we don't know where the variables reside. This is the reason QBASIC as default pushes the offset of the variables instead of their values. Let's try to solve the problem without using

BYVAL

it's actually not that hard to get the value of variable x% into AX without using BYVAL. We only need some extra brackets on the final row:

```
PUSH BP
MOV BP, SP
MOV BX, (BP + 6)
MOV AX, (BX)
```

Now, the third line won't get the actual value of x%, but the offset address where x% can be found. BX now knows where the variable is. The last line doesn't just copy the value of BX like before. Now it gets the word located at the MEMORY POSITION pointed out by BX. Wow! That's the value that was in x% from the beginning! Are you getting dizzy yet? ;-) If you're confused, just read the last lines again and again until you get it.

Now you know how to get values of QBASIC variables into a registers both with and without BYVAL. When your assembly routine only needs to read values from QBASIC, you can use any of these two methods. I usually use BYVAL, Because it's easier to read the values from the assembly pre then. When you want to pass values back to QBASIC, you don't have any option: You cannot use BYVAL. Passing values to QBASIC is not any harder than to read them though. Let's imagine we want to do the opposite of the previous example of getting the value of the x% variable into AX: Getting the value of AX into x%, readable by QBASIC: You call the routine like before, but without BYVAL:

```
CALL ABSOLUTE(x%, offset%)
```

Then, you only need to type this in assembler:

```
PUSH BP
MOV BP, SP
MOV BX, (BP + 6)
MOV (BX), AX
```

The only line that has been changed from the last example is the last one. Instead of getting the value at the offset of BX into AX, we now put the value of AX at the offset of BX. When the control has been returned to QBASIC again, you'll find that the x% variable has changed! Now the topic of variable passing is almost completed. Just a few more things:

First, let's return to the last two lines in the set of four lines that I showed you already in the last part of this tutorial:

```
POP BP
RETF
```

When you've done what you wanted in your assembly routine, you must return to QBASIC properly. As we pushed the original value of BP before and changed it, we must change it back before we get back to QBASIC. The POP instruction resurrects the old value. Then comes the RETF instruction. RETF, short for Return Far, will POP back the segment and offset that CALL ABSOLUTE pushed onto the stack, and jump to that memory position. The control has returned to QBASIC! But when you passed variables to your assembly routine, CALL ABSOLUTE pushed them on the stack also. RETF alone won't clean up the mess you left in the stack and QBASIC will lock up your computer when it gets the wrong values from the stack. In order to fix things up, you must tell RETF to POP away the extra words that you put there by passing variables to your assembly pre. This is easy to do. Let's say you only passed one variable. One integer variable takes two bytes, so you change the RETF to RETF 2, and two extra bytes will be popped away into cyberspace! If you passed four variables, you must take away $2 * 4 = 8$ bytes. RETF 8 fixes it! And finally: Remember that you can ONLY use BX when reading values relative to the offset of BP!

An example program

We've been going through a lot in this issue, so a short example program to demonstrate this would be a good idea. I thought we could write a simple assembly program that could add two QBASIC variables together and return the answer in a third variable. Not too exciting, but it's a good exercise. Let's begin in QBASIC: First we type in the core program:

```
CLS
PRINT "This program adds two numbers together through an assembly routine."
PRINT
INPUT "Type the value of number 1: "; a%
INPUT "Type the value of number 2: "; b%

' We'll put some assembly pre here later!
CALL ABSOLUTE(a%, b%, c%, offset%)

PRINT
PRINT "The result of the addition is"; c%
Now we have the skeleton pre for our program. Let's start a text editor and write the assembly pre:
PUSH BP
MOV BP, SP
MOV BX, [BP+0A]
MOV AX, (BX)
MOV BX, (BP + 8)
MOV CX, (BX)
ADD AX, CX
MOV BX, (BP + 6)
MOV (BX), AX
POP BP
RETF 6
```

If you feel confused, here's some explanations:

The first two lines should be familiar to you by now. The two following lines puts the value of the variable a% in the AX register. This variable is located at BP+10. DEBUG handles all numbers as hexadecimal, so we cannot write 10, as it would interpret that into the decimal number 16. So we write 0A instead. Just A would have been enough, but I usually put a 0 in the beginning, just because it looks nice :-)

The a% variable now is in AX. The following two lines puts the contents of the b% variable in CX in the same way. They are then added together with ADD. Now we have the result in AX. BX then gets the offset address of the variable c%, and the contents of AX is copied to that memory position with the last MOV instruction. BP is then popped back to its old value and RETF returns the control to QBASIC, popping away 6 extra bytes to get rid of the three variables that CALL ABSOLUTE put there in the beginning. 11 lines just to add two numbers together! Well, that's as easy as it gets in assembler.

In order to make this program run, you must now use Absolute Assembly like I showed you in the last part of this tutorial. Call the pre string add\$, answer yes to the question about

appending the destination file and answer no to the question about adding call absolute pre to the program, and then you'll get the following QBASIC program:

```
CLS
PRINT "This program adds two numbers together through an assembly routine."
PRINT
INPUT "Type the value of number 1: "; a%
INPUT "Type the value of number 2: "; b%

' We'll put some assembly pre here later!
CALL ABSOLUTE(a%, b%, c%, offset%)

PRINT
PRINT "The result of the addition is"; c%

' ----- Created with Absolute Assembly 2.1 by Petter Holmberg, -97. ----- '

add$ = ""
add$ = add$ + CHR$(&H55)
' PUSH BP
add$ = add$ + CHR$(&H89) + CHR$(&HE5)
' MOV BP,SP
add$ = add$ + CHR$(&H8B) + CHR$(&H5E) + CHR$(&HA)
' MOV BX,[BP+0A]
add$ = add$ + CHR$(&H8B) + CHR$(&H7)
' MOV AX,[BX]
add$ = add$ + CHR$(&H8B) + CHR$(&H5E) + CHR$(&H8)
' MOV BX,[BP+08]
add$ = add$ + CHR$(&H8B) + CHR$(&HF)
' MOV CX,[BX]
add$ = add$ + CHR$(&H1) + CHR$(&HC8)
' ADD AX,CX
add$ = add$ + CHR$(&H8B) + CHR$(&H5E) + CHR$(&H6)
' MOV BX,[BP+06]
add$ = add$ + CHR$(&H89) + CHR$(&H7)
' MOV [BX],AX
add$ = add$ + CHR$(&H5D)
' POP BP
add$ = add$ + CHR$(&HCA) + CHR$(&H6) + CHR$(&H0)
' RETF 0006
' ----- Created with Absolute Assembly 2.1 by Petter Holmberg, -97. ----- '
```

Now, move up the pre declaration above the CALL ABSOLUTE line and add a DEF SEG to set the segment address of the string before the call, and you'll end up with this program:

```
CLS
PRINT "This program adds two numbers together through an assembly routine."
PRINT
```

```

INPUT "Type the value of number 1: "; a%
INPUT "Type the value of number 2: "; b%

add$ = ""
add$ = add$ + CHR$(&H55)
' PUSH BP
add$ = add$ + CHR$(&H89) + CHR$(&HE5)
' MOV BP,SP
add$ = add$ + CHR$(&H8B) + CHR$(&H5E) + CHR$(&HA)
' MOV BX,[BP+0A]
add$ = add$ + CHR$(&H8B) + CHR$(&H7)
' MOV AX,[BX]
add$ = add$ + CHR$(&H8B) + CHR$(&H5E) + CHR$(&H8)
' MOV BX,[BP+08]
add$ = add$ + CHR$(&H8B) + CHR$(&HF)
' MOV CX,[BX]
add$ = add$ + CHR$(&H1) + CHR$(&HC8)
' ADD AX,CX
add$ = add$ + CHR$(&H8B) + CHR$(&H5E) + CHR$(&H6)
' MOV BX,[BP+06]
add$ = add$ + CHR$(&H89) + CHR$(&H7)
' MOV [BX],AX
add$ = add$ + CHR$(&H5D)
' POP BP
add$ = add$ + CHR$(&HCA) + CHR$(&H6) + CHR$(&H0)
' RETF 0006
DEF SEG = VARSEG(add$)
CALL ABSOLUTE(a%, b%, c%, SADD(add$))
DEF SEG
PRINT
PRINT "The result of the addition is"; c%

```

Notice how I exchanged the offset% variable in the CALL ABSOLUTE line with SADD directly. It's not necessary to put the offset in a variable before using it with CALL ABSOLUTE.

Test this program and you'll see that it works, at least when the numbers you add won't give a result that is above the 16 bit limit. This program can now serve as a base for more experiments. You can test all the instructions I've been presenting in this part of the tutorial with only small modifications of the assembly pre in this example. I encourage you to do so. It's the best way to learn how to use them.

Phew! That was all for this part of my assembly tutorial, the longest one so far. Now you know 20 assembly instructions: MOV, PUSH, POP, RETF, ADD, SUB, INC, DEC, MUL, IMUL, DIV, IDIV, AND, OR, XOR, NOT, SHL, SHR, ROL and ROR. You also know of 10 registers: AX, BX, CX, DX, DS, SI, DI, SS, BP and SP. There are more assembly instructions and registers to cover, but with the ones you master now, together with the knowledge on how to use CALL ABSOLUTE, you know enough to start writing some basic assembly routines yourself. There are many more things to know about assembly programming though. In the next part of this tutorial, we'll look at the possibilities to control

the program flow, learn more about memory access, discover the extra features of Absolute Assembly, and finally, open the door into one of the most interesting parts of assembly programming in QBASIC, using an assembly instruction that will make you sit up for whole nights programming!

Until the next time, experiment with the new things I've presented in this part until you feel familiar with them. Now we're really getting somewhere

Program Flow

Chapter 4

The fourth part of my assembly tutorial is here! The last part was really huge and covered many ways to manipulate registers and QB variables. I hope you've done some test programs during the last month to test what you've learnt. Now it's time for something new!

Until now, we've only seen simple assembly programs that basically only could manipulate numbers in different ways. As interesting this may be, it would be great to know a little more, wouldn't it?

Controlling the program flow

In QBASIC, we're used to instructions that can control the way that snippets of code are run. I'm talking about instructions like GOTO, IF, FOR/NEXT and DO/LOOP. These kinds of instructions are essential in all program languages. Naturally, they also exist in assembler.

Many professional programmers don't like BASIC because of one particular instruction. I guess most of you knows what I'm talking about... GOTO! If you don't know the sad story of GOTO, here it is:

Before QuickBASIC was created, no BASIC compiler was procedural, i.e. they didn't support the creations of SUBs and FUNCTIONs. You had to build your program around a messy structure of jumps back and forth through the code. Some common routines could be called with GOSUB/RETURN which made it a little easier to keep up a good program structure, but sooner or later you still ended up with a really messy program that was hard to debug and update unless you planned the program very well. The term "spaghetti code" is often used to describe program code that is really messy, with jumps between lines all over the place. Most old BASIC programs looked like that, so naturally the BASIC language wasn't the choice of professional programmers. When you're coding in QBASIC/QuickBASIC you should never use GOTO and, if you can avoid it, not GOSUB/RETURN either. With a good procedural structure you will never need them, and the program won't turn into spaghetti. In assembler, you don't have that luxury. You **MUST** use instructions similar to GOTO and GOSUB if you want to get anything done!

Let's begin by explaining the equivalent to GOTO:

GOTO in asm

Whenever you need to make an unconditional jump in an assembly program, you use the instruction JMP. (JMP stands for JuMP, as you probably guessed). The general syntax is:

JMP offset

Where offset is a number that describes the offset in bytes to the byte where the machine language equivalent of the JMP instruction is located. The number can also be in a register or at a specific position in the memory. Does it sound complicated to you? Yes, I thought so! Actually, it's a pain to use JMP like this. If you want to make a correct jump, you must go through all of the asm code between the JMP instructions and the destination instruction and count the number of bytes they take up. And if you need to insert new assembly instructions in the middle of a program using JMP, you'll mess up everything and you have to recalculate all the offsets.

In DEBUG, you can always see the memory position of every assembly instruction, so in order to make it easier to use jumps, you just type the memory position of the instruction you want to jump to, and DEBUG translates this into an offset for you. This makes it easier to use JMP, but you cannot say it has become very much easier.

One of my primary concerns when writing Absolute Assembly 2.0, the first really useful version, was to make it much easier to use JMP. So I included the support for line labels, just like the ones you use in QBASIC. Absolute Assembly and DEBUG together takes care of the translation to offset numbers for you. With Absolute Assembly, JMP is as easy to use as GOTO is in BASIC. Consider this very short program:

```
LineLabel: MOV AX, 1  
JMP LineLabel
```

This program moves a 1 into AX, at a line labeled LineLabel, and then the JMP instruction makes the program go back to that line again. 1 is moved to AX again, and the same jump is performed again. As you easily can understand, this program would result in an infinite loop if executed. Since I didn't spend too much time perfecting the label feature of Absolute Assembly, there are some limits to the ways that you can use them. Read the notes in the beginning of the program source for more information. You should use JMP carefully in order to avoid spaghetti code. Now when we're at it, let's look at another feature of Absolute Assembly:

Comments

It's hard to understand source that other people have written. Most of the time it's also hard to understand the code that you've written yourself after a couple of weeks. Understanding assembly code, even if you wrote it yourself only an hour ago can be a nightmare! Just look at the routine that we did in the end of the last part of this tutorial series:

```

PUSH BP
MOV BP, SP
MOV BX, [BP+A]
MOV AX, [BX]
MOV BX, [BP+8]
MOV CX, [BX]
ADD AX, CX
MOV BX, [BP+6]
MOV [BX], AX
POP BP
RETF 6

```

Do you remember what it did? Can you instantly explain how it works? I can't. Due to this problem, I knew that it was necessary to allow comments in Absolute Assembly. In BASIC, the "" sign, or the older REM instruction can be used for commenting. In the most popular assembler's, like Microsoft's MASM and Borland's TASM, (I'll get back to them in another part of this tutorial series) the semicolon, ";", is used for comments, so I made this an Absolute Assembly standard too. Commenting assembly code is as simple as commenting BASIC code.

Let's try:

```

; Assembly routine that adds two integer variables together:
PUSH BP ; Allow the reading of variables from QBASIC:
MOV BP, SP
MOV BX, [BP+A] ; Move variable 1 into AX:
MOV AX, [BX]
MOV BX, [BP+8] ; Move variable 2 into CX:
MOV CX, [BX]
ADD AX, CX ; Add AX and CX together and put the result in variable 3:
MOV BX, [BP+6]
MOV [BX], AX
POP BP ; Get back to QB:
RETF 6

```

Wow! What a difference a few comments can make, right? Now the purpose and the basic functions of the routine are clearly explained. The details of each operation can now be understood by examining very few lines of code. This commented version of the addition routine would be correctly handled by Absolute Assembly. It just ignores everything written on a line after a semicolon has been encountered. Now, let's get back to those jumps!

CALL and RET

Instead of using only JMP to jump around in your asm code, you can use the assembly instructions CALL and RET. They are the asm equivalents to GOSUB and RETURN in BASIC. So if you have a routine that needs to be used several times in your code, you can put it in a subroutine in the end of your asm code and use CALL to get there. The syntax for CALL is:

CALL address

Just like with MOV, you would have to calculate the memory offset to the asm instruction that you want to jump to, but with Absolute Assembly all you have to do is to specify a line label.

When a CALL is executed, some things happen that you maybe will find interesting: First of all, the CPU pushes the offset address of the asm instruction after CALL on the stack, thus reducing the SP register by two. This is important to know if you're using the stack in the program. Then, the IP register, which always contains the offset address of the current machine language instruction being executed, is changed to the offset address of the assembly instruction you want to jump to. The IP register cannot be changed manually. The only way to modify it is to use JMP, CALL or similar instructions.

When you've jumped to a subroutine using CALL and want to get back again, you must use the instruction RET, short for RETurn. RET will pop the address of the instruction after the CALL instruction back from the stack and change IP. The next assembly instruction to be executed is the one after CALL. The syntax for RET is simply:

RET number

The number can usually be left out, but if you have pushed additional numbers on the stack inside the subroutine, you can let the RET instruction pop them away for you.

If you look at the example program used in the description of JMP above, you can see the instruction RETF 6. RETF works just like RET, but with the difference that it returns from a FAR call, i.e. a call that has been made from another segment address in the memory. It's possible to specify a full memory address, containing both segment and offset after CALL, but we won't need to do that in Absolute Assembly programs. However, I strongly suspect that QBASIC executes such a CALL instruction when you use CALL ABSOLUTE. Far calls require that both the segment and offset of the next asm instruction are pushed on the stack, so also the CS register, containing the segment address of the instruction currently being executed, is pushed. That's four bytes instead of two, and that's why you have to use RETF instead of RET. The number 6 after RETF pops away 6 extra bytes from the stack. That's the three integer variables that was passed from the BASIC program.

This program is a simple, useless example of using CALL and RET:
(The numbers to the right specify in which order the instructions are executed)

```
PUSH BP ; 1
MOV BP, SP ; 2
MOV AX, 1 ; 3
CALL Subroutine ; 4
MOV CX, 3 ; 7
POP BP ; 8
RETF 2 ; 9
```

```
Subroutine:
MOV BX, [BP+6] ; 5
RET ; 6
```

Just as a reminder: The four lines that was presented to you earlier in this tutorial series as a base for all your assembly routines are not necessary if you don't pass any BASIC variables to the routine. Thus, three of the four lines below won't be necessary:

```
PUSH BP
MOV BP, SP
POP BP
RETF x
```

The only important instruction is RETF. You don't need the others. All right, now on to something else.

Conditional jumps

It's very likely that your assembly routines sometimes has to do different things depending on, say, the numbers you've put in the variables you pass to them. In QBASIC, you frequently have to use IF/ELSE or SELECT CASE to control such things. How can we do this in assembler? The answer is: Through conditional jumps!

If you want to control the program flow depending on input data, you can use the assembly instruction CMP, which is short for CoMPare. The syntax is:

CMP destination, source

The destination and source can be registers, immediate numbers or memory pointers.

CMP actually works a little like SUB. It subtracts the source from the destination. The difference is that it doesn't store the result in the destination like the SUB instruction would do. What's the use of it then? Well, something very important actually happens, but you can't see it. Now I have no choice but to present another new feature of assembler to you: The flags.

The flags are a very important part of assembly programming, even though it's something you rarely have to worry about. The flags are all located in a register, and that register is simply called the FLAGS register. This register is different from all of the others, because its individual bits all have separate tasks, and they are very important for the execution of a program. Each bit in the FLAGS register is called a flag, and they all have names. I'm not going to present them to you here because you'll never need to use most of them, but the important thing to know is that, almost every assembly instruction modifies some of the flags in different ways. One example is SUB. There's one flag called the Sign Flag, SF, and it will be set to 1 if the result of the subtraction gets negative or 0 if it gets positive. One of the few times you really use of the FLAGS register is when you push or pop it. PUSH FLAGS and POP FLAGS are valid instructions, and they're used when you need to preserve the state of the flags to a later time. So, even though CMP won't store the result of a subtraction, it will

modify the flags in the same way that SUB would do. What use can we have of this then?
Well, here comes the answer:

There are a number of assembly instructions that can be used to perform conditional jumps in the code. Their names all begin with a J, for Jump.

Here are the most common ones:

Name: Description:

JB Jump if Below
JBE Jump if Below or Equal
JE Jump if Equal
JAE Jump if Above or Equal
JA Jump if Above
JL Jump if Less (signed)
JLE Jump if Less or Equal (signed)
JGE Jump if Greater or Equal (signed)
JG Jump if Greater (signed)
JNB Jump if Not Below
JNBE Jump if Not Below or Equal
JNE Jump if Not Equal
JNAE Jump if Not Above or Equal
JNA Jump if Not Above
JNL Jump if Not Less (signed)
JNLE Jump if Not Less or Equal (signed)
JNGE Jump if Not Greater or Equal (signed)
JNG Jump if Not Greater (signed)

How do these instructions work then? Well, the general syntax for all of them is:
Jxxx linelabel

The linelabel thing works just like it does with JMP.

The idea is that you should use CMP to compare two operators, and then use one of the conditional jump instructions to go where you want to go depending on the state of the flags that a CMP between the two operators changed. Let's try an example!

Suppose you have two numbers in AX and BX. If the number in AX is greater than the one in BX, CX should be set to 1. If not, CX should be left unchanged. Then you could use this code snippet to test it:

```
.  
.   
.   
CMP AX, BX    ; Compare AX against BX.  
JBE NotGreater ; IF AX is below or equal to BX; skip the next line.  
MOV CX, 1     ; Set CX to 1. (This line is only executed if AX > BX)
```


NotGreater: ; The label used for the skipping of the previous line.

.
. .
.

Get it? Now CX will only be changed if AX is greater than BX, because if it's below or equal to BX, one line will be skipped.

As you can see, I've put a space before the MOV CX, 1 instruction. I usually do this when writing conditional jump code in asm, just to make it look more like in QBASIC where you often do this between IF and END IF. This is just one of my tricks to make asm code more readable so you don't have to care about it.

There's another thing you may be wondering about. In the list of jump instructions above, some of the descriptions have the comment "(signed)" in them. As I mentioned briefly in the previous tutorial, a signed number is the same as a negative number. I'll wait with the explanation of how negative numbers are stored, but it's important that you know when to use what jump instruction. If you want to compare two numbers where one or both of them are negative, you must use a conditional jump instructions that can handle signed numbers. So instead of using JA (Jump if Above), you use JG (Jump if Greater) and so forth. The JE and JNE instructions works with all types of numbers. I promise to explain the nature of signed and unsigned numbers later, and then you'll understand why you need so many different jump instructions.

Let's try another example just for the sake of clarity: Consider the following code snippet:

.
. .
.
CMP AX, 0
JL Negative
MOV BX, CX
JMP EndOfTest
Negative:
MOV BX, DX
EndOfTest:
. .
.
.

What this code snippet does is the following: It first tests if AX is 0. If it is less than zero, i.e. if it's negative, BX will be set to the value in DX. If it wasn't negative, no jump will occur and BX will get the value in CX instead. But in order to avoid setting BX to DX right after that, which would destroy everything, an unconditional jump to the code after that line must be made. It's a bit ugly, but that's the only way to do it. If it makes you feel any better you can think of the "JL Negative" instruction as IF, the "Negative:" label as ELSE, and the "EndOfTest:" label as END IF.

There's another assembly instruction that can be used for conditional jumps: TEST. The syntax for TEST is:

TEST destination, source

TEST is used exactly like CMP and for the same purpose. The difference is that CMP performs a subtraction between the two operands, but TEST performs an AND between them. This can be useful if your conditional jumps depends on the bit settings of the operands instead of the value of the whole operand.

Loops

Another important feature of QBASIC and other high-level languages is the ability to execute a code snippet repeatedly, a feature known as looping. In QBASIC, you have the FOR and NEXT instructions for loops that you want to perform a certain number of times, and DO/LOOP and WHILE/WEND for loops that should run until something special occurs. Slow loops are one of the major reasons of slow program execution, and one of the major reasons to use assembler in QBASIC is to speed up things. Therefore, you'll soon discover that the most important parts of your program to rewrite in assembler often are the loops.

You already know how to perform DO/LOOP type of loops in assembly. You can use JMP together with conditional jumps, like this:

```
.  
.   
.   
XOR AX, AX ; Set AX to 0.  
StartOfLoop:  
INC AX      ; Increase AX.  
CMP AX, 10  ; If AX is 10: Get out of the loop.  
JE EndOfLoop  
JMP StartOfLoop ; Jump back to the start of the loop.  
EndOfLoop:  
.   
.   
. 
```

This example would increase AX ten times and then continue the program. Note how I use three spaces before the instructions inside the loop here, just like loops are usually written in QBASIC. This is just another way of making the code more readable. Feel free to invent your own tricks if you don't like mine. (This article is formatted to html, which unfortunately makes 3 blank spaces a wicked pain to code, which is why you'll notice only one space-editor)

Although this is an acceptable way of performing loops in asm, there is a special loop instruction that does the job even better. Let's take a look at it!

The special loop instructions I'm talking about are made for the FOR/NEXT type of loops, i.e. loops that you want to run a certain number of times. The CX register plays an important role here. It's the register used to store the number of times a loop should be executed.

The instruction used to perform loops is... LOOP! The general syntax is:

LOOP Label

The Label operator works the same as with other jump instructions. Here's an example of using LOOP:

```
.  
.   
.   
MOV CX, 10 ; Loop ten times.  
StartOfLoop: ; This label specifies the start of the loop.  
ADD AX, 4 ; A useless asm instruction.  
SUB BX, DX ; Another useless asm instruction.  
LOOP StartOfLoop ; Jump back to the StartOfLoop label.  
.   
.   
.
```

This example demonstrates how you use the CX register and LOOP to make a set of asm instructions execute a certain number of times. You just put the number specifying how many times you want to execute the loop in the CX register. When the LOOP instruction is executed, the number in CX is decremented by one (without touching the stack), and if the result is above zero, a jump back to the specified label is performed. This means that the lines between the line label you jump to and the JUMP instruction will be executed the same number of times as the number you set CX to before the loop. The only bad thing with this loop technique is that the CX register is occupied as long as you're inside the loop. You can use CX for other stuff, but then you would have to save its value somewhere else (the stack for example) and restore the value before the LOOP instruction.

Limits of jumps

All of the asm instructions performing jumps (JMP, CALL, LOOP and so on) have a limit: The jumps cannot be too long. What I mean is that you can't jump across too many assembly instructions. Or actually, the limit depends on the number of bytes the machine language equivalents to the assembly instructions takes up. The limit is between 128 bytes upwards to 127 bytes downwards. It's not easy to know how many bytes your assembly instructions occupy, but it generally varies between one and five. An instruction such as PUSH AX only takes up one byte, but an instruction like MOV [BP+10], BX needs three bytes. Usually this limit is nothing to worry about. 100 bytes are a lot of assembly code. I've never had a problem with this limit.

Memory transfers fast and easy

I thought we should make a really cool example of assembler in QBASIC this time. But for that example, we need to know three new assembly instructions. There are lots of asm instructions for handling strings. I'm not going to discuss all of them here, but there are three of them that are really useful: LODS, STOS and MOVS. Their respective syntaxes are:

```
LODSx  
STOSx
```

MOVSx

As you can see, there are no operands or anything. The "x" should be replaced with either a B for Byte or a W for Word. The first instruction, LODS, works like this: If you use LODSB, a byte from the address pointed out by DS:SI will be loaded into AL, and SI will be increased by one. (Actually it may be decreased instead if the Direction Flag, DF, is set.) LODSW works in the same way except that a whole word (two bytes) will be loaded into the entire AX register and SI will be increased (or decreased) by two. STOS is the opposite of LODS: It copies the value of AL/AX to the address pointed out by ES:DI, and increases/decreases DI by one or two depending on if you use STOSB or STOSW.

MOVS is a combination of LODS and STOS. A byte or a word is loaded from DS:SI, but it doesn't go to a register. Instead it is copied to ES:DI and both SI and DI are incremented. MOVSB/MOVSX can therefore be used to transfer bytes from one position in the memory to another without touching the registers, something you cannot do with the standard MOV. We're going to use MOVS in the example routine I'm now going to present to you.

An example program

To summarize this part of my assembly tutorial, we're going to make a really cool example program. You may not have realized it, but you already have a great knowledge of assembly programming. If you don't believe me, you'll only have to look at this example:

What we're going to do is an assembly version of the popular QBASIC instruction PUT. I'm not talking about the file handling version, but the graphics instruction used to put a sprite on the screen.

We're only going to use SCREEN 13 for this example, since this screen mode is really easy to use. The whole screen is built up by a 320 * 200 pixel bitmap with 256 possible colors. Each pixel occupies one byte in the VGA memory, and that memory starts at the address A000:0000h. The first byte contains the index of the pixel at coordinates 0,0, the second one of the pixel at 0,1, the 320th one of the pixel at 1,0 and so forth.

The sprite is stored in a QBASIC array, where the first word specifies the width of the sprite times 8. The second word specifies the height of the sprite, and then the pixel indexes comes as bytes stored in the same way as the screen 13 bitmap.

It would be nice with a routine that was a little better than PUT. Of course it will be faster than PUT, but let's add another feature. Many times you wish that you could draw sprites with an "invisible" color. This means that one of the colors in the sprite won't be drawn on the screen. With this feature, your sprites can have irregular edges and "holes" in them because a certain color will be skipped when the sprite is drawn on the screen. We'll use color 0 as the "invisible" color.

Before we start writing the asm code, we'll make a demo program in QBASIC to test it with:

```
' Demonstration of using assembler in QBASIC to put a sprite in SCREEN 13
' much faster than with PUT and with an invisible color.
SCREEN 13
' Initialization of the assembly routine:
```

```

' (Here we're goint to put stuff later on.)
DIM testsprite%(513)
DIM background%(513)
LINE (0, 0)-(31, 31), 32, BF
LINE (4, 4)-(27, 27), 0, BF
LINE (8, 8)-(23, 23), 40, BF
LINE (12, 12)-(19, 19), 0, BF
GET (0, 0)-(31, 31), testsprite%
' Demonstrate the routine:
CLS
LINE (0, 24)-(319, 199), 32
LINE (0, 199)-(319, 24), 40
spritex% = 144
spritey% = 96
COLOR 40
PRINT " Demo of PUT routine in assembler:"
COLOR 32
PRINT "Use the cursor keys to move the sprite."
PRINT "Pressing Escape exits the demonstration."
GET (spritex%, spritey%)-(spritex% + 31, spritey% + 31), background%
' (Here we're going to make a call to the asm PUT routine later.)

DO
key$ = INKEY$

SELECT CASE key$
' Up:
CASE CHR$(0) + "H":
PUT (spritex%, spritey%), background%, PSET
IF spritey% > 30 THEN spritey% = spritey% - 10
GET (spritex%, spritey%)-(spritex% + 31, spritey% + 31), background%
' (Here we're going to make a call to the asm PUT routine later.)
' Down:
CASE CHR$(0) + "P":
PUT (spritex%, spritey%), background%, PSET
IF spritey% < 158 THEN spritey% = spritey% + 10
GET (spritex%, spritey%)-(spritex% + 31, spritey% + 31), background%
' (Here we're going to make a call to the asm PUT routine later.)
' Left:
CASE CHR$(0) + "K":
PUT (spritex%, spritey%), background%, PSET
IF spritex% > 10 THEN spritex% = spritex% - 10
GET (spritex%, spritey%)-(spritex% + 31, spritey% + 31), background%
' (Here we're going to make a call to the asm PUT routine later.)
' Right:
CASE CHR$(0) + "M":
PUT (spritex%, spritey%), background%, PSET
IF spritex% < 278 THEN spritex% = spritex% + 10
GET (spritex%, spritey%)-(spritex% + 31, spritey% + 31), background%
' (Here we're going to make a call to the asm PUT routine later.)

```

```
END SELECT
LOOP UNTIL key$ = CHR$(27)
```

That's it! Let's save this code in the file PUTDEMO.BAS.

Now we need to figure out what input values the asm routine needs:

First of all we need to pass to the assembly routine the x and y coordinates of the screen where we want the sprite to be drawn. The asm routine also need to know where in the memory to find the sprite. The sprite should be stored in an array, and all arrays start with the offset address 0 in the memory, so we just need to pass the offset address. The asm routine also need to know the dimensions of the sprite, but that's stored in the sprite data so we can obtain these values inside the assembly routine itself. So, let's make the call to it look something like this:

```
CALL ABSOLUTE(BYVAL x%, BYVAL y%, BYVAL VARSEG(testsprite%(0)),
SADD(asmput$))
```

Now we can start typing in the asm code in our favourite text editor. Let's take it step by step! First we want to allow the reading of QBASIC variables:

; A PUT routine with clipping:

```
PUSH DS ; Push DS and BP and move SP into BP.
PUSH BP
MOV BP, SP
```

As you can see, we do not only push BP, but also DS. It's necessary to preserve the value of DS in all assembly routines called by QBASIC if you're going to use it, or else your computer will crash.

Now we need to point DS:SI to the start of the sprite in the memory: This requires that we get the segment address of it from QBASIC. Where in the stack can we find it? Well, first we remember from the last part of this tutorial series that QBASIC first pushes the variables we passed to the routine in left to right order and then pushes four extra bytes on the stack, so the computer knows how to get back to the BASIC program. Then we push DS and BP on the stack ourselves. Since the stack grows downwards, this means that the last variable would be found at $BP+2+2+4=BP+8$, the middle one at $BP+2+2+4+2=BP+10$ and the first one at $BP+2+2+4+2+2=BP+12$. Since DEBUG treats all numbers as hexadecimal, we will have to define the stack offsets of the variables in the following ways:

```
x% = BP+0C
y% = BP+0A
VARSEG(testsprite%(0) = BP+08
```

Calculating stack offsets like this may seem a little tricky, but you'll get the hang of it after a while. Now, let's load the address of the sprite into DS:SI!

```
MOV BX, [BP+08] ; Get the segment address of the sprite.
```

```
MOV DS, BX
XOR SI, SI ; Set SI to 0.
```

Now we know where to begin fetching the data. But where should we put it? In order to determine the correct position on the screen, we need to know the pixel co-ordinates of the upper left corner of the sprite. These numbers should've been passed to the routine and pushed on the stack. It's just to go and get them:

```
MOV DX, [BP+0C] ; DX = X coordinate.
MOV AX, [BP+0A] ; AX = Y coordinate.
MOV BX, AX ; BX = AX = Y coordinate.
```

And why do I load the y co-ordinate into two registers, you ask? Since the screen pixels are stored in the memory from left to right, row by row from the top to the bottom, the correct memory position to begin moving data to depending on the coordinates x and y is: $A000h:y * 320 + x$. As you can see, the offset calculation includes a multiplication. Even though this only needs to be calculated once per call to the routine, making the use of MUL despite its slowness acceptable, we're going to use another method which is much more interesting. Remember from the last part of this tutorial that you could use shift instructions for multiplications and divisions if the number to multiply or divide by was in the series of numbers defined as 2^x , like 2, 4, 8, 16, 32, 64, 128, 256 and so on. 320 isn't one of these numbers, but it can be expressed as the sum of two of them: $320 = 256 + 64$. This suggests that if we multiply the y coordinate first with 256 and then with 64 and add the results together, it will be the same as if we multiplied it with 320 in the first place. And that is certainly true! So with two shifts and one addition, we can perform a multiplication with 320 really fast:

```
MOV CL, 8 ; Multiply y with 256.
SHL AX, CL
MOV CL, 6 ; Multiply y with 64.
SHL BX, CL
ADD AX, BX ; Add the results together.
```

Now you can see why the y coordinate needed to be copied to more than one register. We need the value twice.

Now we have $y * 320$ stored in AX and x stored in DX. Now we only need to add them together to get the correct memory offset for the sprite. The segment address should be set to A000h, and then we have the correct destination memory address in ES:DI:

```
MOV BX, A000 ; ES = A000h.
MOV ES, BX
ADD AX, DX ; DI = y * 320 + x.
MOV DI, AX
```

All right. Now we have DS:SI pointing at the start of the sprite data and ES:DI pointing at the screen memory position where the data should be written.

Now it would be nice to know the width and height of the sprite. This is stored in the beginning of the sprite data. First comes the width of the sprite. We load the first two bytes of

the sprite data into AX, using LODSW. The number we get is eight times bigger than the actual width. Therefore we need to do a division by 8 to get it right. Luckily, this division can be handled with a shift instruction. Then we move the value to BX, where it will be stored:

```
LODSW ; Get width info from sprite data.  
MOV CL, 3 ; Divide the number by 8 to get correct width.  
SHR AX, CL  
MOV BX, AX ; Store the width in BX.
```

Next comes the height. It's simple to retrieve it from the sprite data, since it comes after the width and is stored in the correct form. We won't need any shifts to correct it. We won't support sprites higher than 200 pixels, so only one byte needs to be stored. Let's keep the height in AH. AL must be left free for later:

```
LODSW ; Store the height in AH  
MOV AH, AL
```

Finally, we need to know what 320 - the sprite width is, because this number will be used in the drawing loop. This is simple to do. We can use DX to store this number, and the sprite width is already in BX. So all we need to do is this:

```
MOV DX, 140 ; DX = 320 - Sprite width  
SUB DX, BX
```

Now we have loaded all the data we need. We still haven't put anything in CX, and that's good because we're going to need it in the drawing loop. AL will also be used, since we use LODSB in the code.

Let's take a look at the registers and see how many we have left:

```
DS = Source segment  
SI = Source offset  
ES = Destination segment  
DI = Destination offset  
AH = Sprite height  
AL = Reserved for drawing loop  
BX = Sprite width  
CX = Reserved for drawing loop  
DX = 320 - Sprite width
```

Phew! We made it without having to use the stack to stuff away data. All of the basic registers are used. If we would have had to load more information, using the stack would have been inevitable.

OK! Now it's time for the main loop. Actually it has to be two loops inside each other. I'll show it first and explain it later:

```
Yloop: CMP AH, 0 ; Stop drawing if y is zero.  
JE EndOfDrawing  
MOV CX, BX ; CX = Sprite width.  
XLoop:
```



```

LODSB ; Load a pixel from DS:SI into AL.
CMP AL, 0 ; Is the pixel color 0?
JE SkipPixel
STOSB ; No: Copy the pixel in AL to ES:DI.
DEC DI
SkipPixel:
INC DI ; Yes: Increase DI by one.
LOOP XLoop
ADD DI, DX ; Move screen memory pointer to next line.
DEC AH ; Decrease height.
JMP YLoop
EndOfDrawing:

POP BP ; Return to QBASIC.
POP DS
RETF 6

```

All right! What this loop does is the actual drawing process. First we test if the height of the sprite (a number located in AH) is 0. If it is, we will instantly jump out of the loop and return to QBASIC since there's nothing to draw. If it's over 0, CX will be loaded with the width of the sprite. Now we can load the first pixel from the sprite into AL, using LODSB. Remember that this also increases SI by 1. We test to see if this pixel is 0, the invisible color. If it is, we increase DI by 1, thus skipping a pixel on the screen. If it isn't 0, we use STOSB instead to copy the pixel to the screen and decrease DI by 1 to compensate for the increase that comes below it. Then we use LOOP to repeat this process. Each time CX will be decreased until it reaches 0. When this first happens, the first column of the sprite has been drawn. Now we continue below the LOOP instruction, where 320 - the sprite width is added to DI. This ensures that the next column will be drawn in the correct position on the screen. Finally we decrease AH containing the sprite height by 1 and return to the start of the outer loop. This process will of course continue until AH is 0, and the sprite drawing will be complete! All that's left to do is to exit from the routine, returning us to the demo program.

Now paste together the pieces of code we've collected and save it in a file called ASMPUT.ASM. Then run Absolute Assembly. Make ASMPUT.ASM your assembly source file, PUTDEMO.BAS your QBASIC destination file, make sure it appends the code to PUTDEMO.BAS instead of erasing its original contents and skip the adding of call absolute code. We can do that ourselves. Now start QBASIC and make sure the asm code was added to the program. Take the added code and move it up to the beginning of the demo program, replacing the line saying: ' (Here we're going to put stuff later on.) Then, you look up the five places saying: ' (Here we're going to make a call to the asm PUT routine later.) and replace them with the following lines:

```

DEF SEG = VARSEG(asmput$)
CALL ABSOLUTE(BYVAL spritex%, BYVAL spritey%, BYVAL
VARSEG(testsprite%(0)), SADD(asmput$))
DEF SEG

```

And that's it! Test the program and see the asm code we've written in action!

There are some limits to this sprite drawing routine though: It won't work in any other screen mode than SCREEN 13, because the screen memory works differently for the other modes, and the routine lacks border checking, i.e. If you put the sprite on a position where parts of it is "outside" the screen borders, it will not be drawn correctly. If you feel like it, you can try writing a sprite routine that "clips" the sprite correctly at the screen edges. But anyway, it draws sprites in a way that the standard PUT cannot do, and it's much faster too! Neat, huh? :-)

Oh no! I've done it again! These tutorial parts are growing for each new month :-) Last time I promised to cover a bit more than I actually did in this part, but the things that you've just read are enough for now. Now go and experiment with assembly flow control on your own and see if you can come up with something cool! We've come a long way now, but basically we still only know how to do clever calculations and memory transfers in asm. This can be used for much, but there's still more to learn. The next time I will introduce some ways to communicate with the different parts of your computer in assembler. This is the part of programming that is usually referred to as... I/O!

Controlling I/O

Chapter 5

Welcome to the fifth part of my assembly tutorial series! During the last months, we've gone through some rough material, but now you should know almost all the basics of assembly programming. We've looked at calculations, memory transfers and flow control. This is enough to make some really cool assembly routines. But there are other aspects of programming in assembler. In this part we will look at the possibilities to do a little more with the assembly language. You can certainly do more than manipulate numbers and the memory in asm, so that's what we're going to look at this time. And maybe we'll also pick up some extra info that we lost on the way here...

Communicating with the outside world

The demo program in the last part gave a taste of what you can do with nothing more than some simple assembly code. It only moved data between two memory positions, but the destination happened to be the video memory in the VGA mode 13h- or SCREEN 13 as QB programmers knows it. Thus, we could detect the result on the output device known as the monitor. But there are other devices that it would be cool to be able to access in asm, such as the keyboard, mouse and the sound card. Can we do this? Certainly!

The ports

The I/O devices on the PC can be accessed through something called the I/O ports. Through these, the CPU communicates with the different parts of your computer. By reading data

from, or writing data to these ports, we can access the hardware we want to control. Doing this in assembly is very easy. There are two asm instructions that you use for this task, and their names are simply IN and OUT. The syntaxes for these instructions are:

IN source, destination
OUT destination, source

These instructions have equivalents in QBASIC. There you use them in a very similar way. In QBASIC they're called INP and OUT, and the syntax is the same too!

The source for IN and destination for OUT are values specifying the port number. There are many I/O ports, so the number is 16 bits long. You can use a direct number as destination, but it's also common to MOV the number to DX first and use it as source/destination. The values used as destination for IN and source for OUT can be either bytes or words. You should use AX or AL for these tasks. As an example: Suppose you want to write a 1 to port 0487h and then read a value from the same port. Then you could use this program:

```
MOV DX, 0487 ; Port number.  
MOV AL, 1 ; Number to write to the port.  
OUT DX, AL ; Write to the port.  
IN DX, AL ; Read from the port.
```

Don't test this program!!! It was only an example and it won't work. It's a bad idea to experiment with the I/O ports at all if you don't know what you're doing. If you're unlucky you may damage your hard drive or some other sensitive piece of hardware in your computer.

Another important thing to know is that you can't read and write to all ports. Some only works one way, so you may be able to read values from them but not write, or write to them but not read. This depends on the purpose of the port. Some devices, such as the VGA card and many soundboards uses only a few I/O ports for dozens of system functions. This is made possible through a clever technique called indexing: One port is used as an index port and another as a data port. First you write a value to the index port telling the hardware what system function you want to use, and then the data port can work in different ways depending on the value sent to the index port. Exactly how this works varies with the device you're accessing.

Ok, now I've rambled on enough. What can we do with IN and OUT then? Well, you can do really much, but the problem is that it's REALLY hard to control most I/O devices in this way. But let's look at an example anyway:

In VGA mode 13h (SCREEN 13), color 0 is usually black. But it's possible to set color 0 to white, red, yellow or any other color by changing the palette registers. The palette tells the VGA card how much red, green and blue to put on the screen for a certain color index. As default, the intensities are set to 0 for red, green and blue. Using two I/O ports assigned to the VGA card we can change the palette setting of color 0. It's not difficult at all. First, we write a 0 to port 03C8h, telling the VGA card both that we want to change the palette and that we want to change color 0. Then we write three successive values to port 03C9h, telling the VGA card the new intensities for the red, green and blue components of the color. The maximum value possible is 63, so if we set all three components to 63 we'll make color 0 look white:

```
MOV DX, 03C8 ; Set the port number to 03C8h
OUT DX, 0 ; Tell the VGA card that we want to change color 0
INC DX ; Set the port number to 03C9h
OUT DX, 3F ; Set red component to 63
OUT DX, 3F ; Set green component to 63
OUT DX, 3F ; Set blue component to 63
```

If you happened to be in screen mode 13h looking at a blank, black screen, it would become bright white after these six lines of asm instructions.

All right, now you know that you can program I/O devices through the I/O ports. Some I/O devices are fairly simple to program this way, but most I/O devices are a nightmare to access this way. The mouse is a great example of such a device. You must access it in different ways depending on what type of mouse you have, what port it's connected to your computer through and so on. Is there a solution to this problem? Luckily, the answer is yes!

Interrupts

The solution to the problems you may have accessing different I/O devices is something called interrupt calls. But interrupt calls are so much more than that, as you soon will see.

Basically, interrupts are small machine language routines stored in the conventional memory. Most interrupts controls different I/O devices in your computer, but there are interrupts with other functions too. Some interrupts are initialized by the BIOS of your computer when you start it up, some are initialized by the operating system and others are initialized by different device drivers.

Interrupts can be called by a program like we will do, but they may also be called automatically by the computer when something special happens, such as the pressing of a key on the keyboard. Interrupts called by programs are called software interrupts and interrupts called by the hardware are called hardware interrupts. You can have use of both types.

When an interrupt is called, the computer immediately interrupts what it's doing and runs the interrupt code. When the interrupt code has been executed, the control is returned to the code that was interrupted. Guess why they call it interrupts ;-)

Calling interrupts in assembler is easy. You can do it through an instruction called INT. The syntax is:

INT number

The number you pass with the INT instruction is the number of the interrupt you want to call. The number should be a one byte immediate number. Many times you can access many different subfunctions through one interrupt. By setting some of the registers before the INT call, you can tell the interrupt code what subfunction you want to access and what parameters you want to pass to that subfunctions. Like with the I/O ports, some interrupt routines returns data to you, some wants you to send data to them and some works both ways. You can do many different things with interrupts, but let's begin by using them to access an I/O device that you usually cannot access in QB: The mouse!

The mouse can be controlled through interrupt 33h. There are lots of subfunctions assigned to that interrupt. You tell the interrupt what subfunction you want to use by putting a number in the AX register. Additional registers can be used to pass parameters to the subfunction. The subfunction may also return data to you in the registers.

First of all, it would be nice to know if we have a mouse installed and a working mouse driver. The very first subfunction of int 33h can do this test for you. This subfunction has the number 0, and you don't have to send any parameters with it. This asm code would make the call:

```
MOV AX, 0 ; Access subfunction 0 of int 33h (detect mouse)
INT 33 ; Make the interrupt call
```

When the interrupt code has been executed, the line after the INT call would be called. The interrupt has now returned the current status of the mouse in the AX register, and additional information in the BX register. The AX registers tells you if there's a mouse installed. If it's set to -1, a working mouse was detected. If it's 0, you're apparently without a working mouse. If the subfunction found a mouse, you can also see how many buttons the mouse has by looking at the number in the BX register.

When you have detected a working mouse, you might want to display the mouse cursor on the screen. This can be done with subfunction 1. All you have to do is to set AX to 1 and call the interrupt:

```
MOV AX, 1 ; Access subfunction 1 of int 33h (display mouse cursor)
INT 33 ; Make the interrupt call
```

If you want to hide the mouse cursor, you can call subfunction 2 in the same way.

Now you may want to get some information about the current position of the mouse cursor and the state of the mouse buttons. Subfunction 3 takes care of this for you:

```
MOV AX, 3 ; Access subfunction 3 of int 33h (get mouse status)
INT 33 ; Make the interrupt call
```

After this call, BX should tell you the current state of the mouse buttons. If bit 0 of BX is set, the left button is currently being pressed, bit 1 returns the state of the right button and bit 2 returns the state of the middle button if you have a three-button mouse.

The current coordinates of the mouse cursor can be found in CX and DX. CX contains the horizontal coordinate and DX contains the vertical coordinate.

Another interesting interrupt is interrupt 21h. This interrupt has tons of subfunctions installed by DOS. And don't worry Win95/98/NT users, you have them too!

Int 21h has many DOS-assigned subfunctions. For example, you can change the default directory, open and read/write data in files and so on. But there are also a couple of functions that does some other neat things for you, such as printing a string of text on the screen. Let's try it out!

Int 21h, subfunction 09h can be used to print a string on the screen. Here, you'll need to put the number 09h in the AH register, the segment address of the string in DS and the offset address of the string in DX. The string must be terminated with a \$ sign, so if you want to print "Hello World!" on the screen, the string should be: Hello World!\$. This example routine could print such a string on the screen for you.

```
MOV DX, [BP+08] ; Set DX to offset address of string
MOV BX, [BP+0A] ; Set DS to segment address of string
MOV DS, BX
MOV AH, 9 ; Call the string printing function
INT 21
```

Another interesting interrupt is interrupt 10h. It contains subfunctions initialized by your BIOS. This is a great way of creating compatibility between different PC's. Even if two PC users have different BIOS manufacturers with hardware working in different ways, they can both access the same functions in the same ways, using the same interrupts. The difference lies in the actual machine language routines assigned to those interrupts, but the user doesn't need to worry about them.

Int 10h contains many interesting subfunctions for accessing video services, such as the ones you use to change screen modes. Through int 10h you can access many more screen modes than you can do using the SCREEN keyword in QB, such as hi-res VESA modes!

The ultimate resource when you need to look up information about interrupts is Ralph Brown's interrupt list. No assembly programmer should be without it! It tells you what the different interrupts and subfunctions do, what you should set the registers to when calling them and what data they return. You can download it from the resources section at the Enhanced Creations website, or from many other sites. Just search for it and you'll get thousands of hits! It's a big download, but it's worth it!

I remember when I first discovered interrupts. It was so much fun, because I instantly got access to functions and I/O devices that aren't possible to access in QB. Interrupts give you a lot of power as a programmer, and they make life a whole lot easier for you if you program in assembler. I'll leave you to explore the world of interrupt functions yourselves now. It can be really fun! Int 10h, 21h and 33h are good starting points.

I said before that it was possible to do much more with interrupts than this. Actually, you can do a lot more. It's possible to tamper with hardware interrupts as well, and if you want to, you can make your own interrupt functions and override existing interrupts with your own routines! The example program this month is a good example of advanced interrupt handling. I thought we should write our own keyboard handler. As you may have noticed when working in QBASIC, the keyboard handling is great, unless you want to make an action game. It's impossible to detect multiple keypresses with INKEY\$, something you often need in games.

We can solve this problem with a little assembly code. The keyboard is controlled through int 09h, and it's a hardware interrupt that is called whenever a key is pressed or released. If we can override this interrupt routine with a keyboard routine that we write ourselves, we should be able to handle the keyboard input in a more game-oriented way.

The first thing we need to know is how we can make int 09h run our assembly code instead of the one it's running right now. Well, believe it or not, there is an interrupt subfunction assigned to make just this change for you! This subfunction lies under interrupt 21h. It can change the memory address of the code that should be executed when you make a certain interrupt call. However, it would be nice to know the address of the old keyboard routine so we can give it the control back when we exit our program. There's a subfunction under int 21h for that too. But let's begin by writing the actual keyboard handler:

Keyboard code

The first thing we need to keep in mind is that the keyboard handler can be called at any time during the execution of our program. It's you who decides by pressing or releasing a key on the keyboard. This means that whatever our keyboard handler does, the registers **MUST** be returned in the same state that they were before the interrupt call. Suppose the program you are running at a certain moment needs AL to be 3. But then the keyboard handler is called and changes AL to 2. When the interrupt routine ends, the program that was currently running will crash, since AL suddenly got changed. This problem is easy to fix though. All we need to do is to push all the registers we want to use in the beginning of the interrupt routine and pop them at the end. Even the flags must be left untouched, but the interrupt call itself pushes them for us so we don't have to worry about it. I don't think I've mentioned this before, but if you actually need to push and pop the flags sometime, you can do this with the instructions PUSHF and POPF.

```
PUSH DS ; Push everything we're going to modify on the stack
PUSH AX
PUSH BX
```

This is how I've thought the keyboard handler should work: We'll create a QB integer array with 128 elements. Each one will work as a keyboard "flag". So if a certain element is 0, the key with that scancode is not being pressed at the moment, but if it turns to 1, that key is being pressed. All QB arrays has a default offset address of 0, so all the asm routine needs to know is the segment address of the array. But how can we pass this value to the keyboard handler from BASIC? This is not a routine we call with CALL ABSOLUTE. Well, there's a really smart solution to this, but we'll wait with this problem. All we do is to set BX to an immediate number for the moment:

```
MOV BX, 1234 ; DS = SEG keyboard flag array (currently not implemented)
MOV DS, BX
```

Now comes the reading of the keyboard status. How do we do this? Through I/O ports of course! Exactly how this works is beyond my knowledge, I've only looked at other keyboard handlers to see how they do it. First we should read a byte from port 60h. We will also set AH to 0 for later use. If the value read from the port is bigger than 127, a key has been released. But first, we'll take care of keypresses:

```
IN AL, 60 ; Read a value from port 60h
XOR AH, AH ; Set AH to zero
CMP AL, 7F ; If a key has been released, jump to some later piece of code
JA Release
```

Now it's time to take care of the keyboard array. If the jump to the Release label wasn't done, it means a key has been pressed and we should set an element of the keyboard array to 1. The array consists of two-byte integers, and the number currently in AL is between 0 and 127. So if we multiply AL by two, we should get the correct offset address in the array. We want to write a 1 to this offset address since a key was pressed, so that's what we'll do:

```
SHL AL, 1 ; Multiply AL with 2 to get the correct offset address
MOV BX, AX ; BX = OFS keyboard flag array
MOV AL, 1
MOV [BX], AL ; Write a 1 to the keyboard flag array
JMP Endkey ; Skip the key release code
```

All right, that should take care of keypresses. Now it's time for releases. If a key is released, AL will contain a number between 128 and 255. If we subtract 128 from this, we get a number between 0 and 127 which is what we want. But we don't need to use a SUB here. All we need to do is to mask out the highest bit of AL, and we'll get the same result. We can do this by ANDing the number with 127. Then we'll do the same as we did with keypresses, except that we write a 0 to the array this time:

```
Release:
AND AL, 7F ; Get rid of highest bit in AL
SHL AL, 1 ; Multiply AL with 2 to get the correct offset address
MOV BX, AX ; BX = OFS keyboard flag array
MOV AL, 0
MOV [BX], AL ; Write a 0 to the keyboard flag array
```

Now we're done with the array handling. All we need to do is to exit our interrupt handler and return to the code that was interrupted by it. Well, not really. The keyboard needs some more attention before we can leave it. We need to tell the keyboard that we received the information it sent us and that we want it to tell us of later keystrokes too. This is also a matter of I/O port programming that I don't really understand. So let's just do it without asking any questions :-)

```
Endkey:
IN AL, 61 ; Reset the keyboard
OR AL, 80
OUT 61, AL
MOV AL, 20
OUT 20, AL
```

All right, NOW we can exit the keyboard handler. All we need to do is to POP back the registers we used and then we use the Interrupt Return instruction, IRET, to end the routine:

```
POP BX ; Pop back all the registers we used
POP AX
POP DS
IRET ; Exit the interrupt routine
```


All right! Now we have a complete keyboard handler except for that problem with the array segment. For some strange reason, this keyboard handler won't run through Absolute Assembly, even though I've made it work before. Since I didn't have the time to fix this problem, I'm going to give you the BASIC code as it would have looked if it had been working. We'll also add the keyflag array to the program. Save this in a file called KEYDEMO.BAS:

' A demonstration of advanced interrupt handling in QB:

DIM keydata%(127) ' Keyboard flag array

' ----- Created with Absolute Assembly 2.1 by Petter Holmberg, -97. ----- '

```
newkey$ = ""
newkey$ = newkey$ + CHR$(&H1E) ' PUSH DS
newkey$ = newkey$ + CHR$(&H50) ' PUSH AX
newkey$ = newkey$ + CHR$(&H53) ' PUSH BX
newkey$ = newkey$ + CHR$(&HBB) + CHR$(&H34) + CHR$(&H12) ' MOV BX, SEG
keydata%(0)
newkey$ = newkey$ + CHR$(&H8E) + CHR$(&HDB) ' MOV DS,BX
newkey$ = newkey$ + CHR$(&HE4) + CHR$(&H60) ' IN AL,60
newkey$ = newkey$ + CHR$(&H30) + CHR$(&HE4) ' XOR AH,AH
newkey$ = newkey$ + CHR$(&H3C) + CHR$(&H7F) ' CMP AL,7F
newkey$ = newkey$ + CHR$(&H77) + CHR$(&HA) ' JA Release
newkey$ = newkey$ + CHR$(&HD0) + CHR$(&HE0) ' SHL AL,1
newkey$ = newkey$ + CHR$(&H89) + CHR$(&HC3) ' MOV BX,AX
newkey$ = newkey$ + CHR$(&HB0) + CHR$(&H1) ' MOV AL,01
newkey$ = newkey$ + CHR$(&H88) + CHR$(&H7) ' MOV [BX],AL
newkey$ = newkey$ + CHR$(&HEB) + CHR$(&HA) ' JMP Endkey
newkey$ = newkey$ + CHR$(&H24) + CHR$(&H7F) ' Release: AND AL,7F
newkey$ = newkey$ + CHR$(&HD0) + CHR$(&HE0) ' SHL AL,1
newkey$ = newkey$ + CHR$(&H89) + CHR$(&HC3) ' MOV BX,AX
newkey$ = newkey$ + CHR$(&HB0) + CHR$(&H0) ' MOV AL,00
newkey$ = newkey$ + CHR$(&H88) + CHR$(&H7) ' MOV [BX],AL
newkey$ = newkey$ + CHR$(&HE4) + CHR$(&H61) ' Endkey: IN AL,61
newkey$ = newkey$ + CHR$(&HC) + CHR$(&H80) ' OR AL,80
newkey$ = newkey$ + CHR$(&HE6) + CHR$(&H61) ' OUT 61,AL
newkey$ = newkey$ + CHR$(&HB0) + CHR$(&H20) ' MOV AL,20
newkey$ = newkey$ + CHR$(&HE6) + CHR$(&H20) ' OUT 20,AL
newkey$ = newkey$ + CHR$(&H5B) ' POP BX
newkey$ = newkey$ + CHR$(&H58) ' POP AX
newkey$ = newkey$ + CHR$(&H1F) ' POP DS
newkey$ = newkey$ + CHR$(&HCF) ' IRET
```

' ----- Created with Absolute Assembly 2.1 by Petter Holmberg, -97. ----- '

Now, take a look at the line of asm code that was supposed to set BX to the segment address of keydata%() but only set it to 1234h:

```
newkey$ = newkey$ + CHR$(&HBB) + + CHR$(&H34) + CHR$(&H12) ' MOV BX, SEG
keydata%(0)
```

Look at the hexadecimal numbers: First comes a BBh, which is the machine language equivalent to MOV BX, immediate number. Then comes the numbers 34h and 12h. Didn't we just set BX to 1234h? There's an obvious pattern here! Here comes the trick: What if we changed the numbers a little here. If we put other numbers there, it wouldn't make any difference to the routine. What if we put the actual segment of the keyflag array there. It would work as good as any other number, but we would also get a working routine then. Of course we need to convert that number into a string, but that's really easy here. Just change the line to this:

```
newkey$ = newkey$ + CHR$(&HBB) + MKI$(VARSEG(keydata%(0))) ' MOV BX, SEG
keydata%(0)
```

And that's it! Everything will work fine now.

Ok, that was the actual keyboard handler. But we haven't installed it yet. We need one routine that can install it, and another one that can remove it. Let's begin with the first one:

The routine that initializes the keyboard handler will use two subfunctions of int 21h. The first one, subfunction 35h tells you the address of the current interrupt handler. If we set AH to 35h and AL to 9h (telling the subfunction that we want to get the address of int 9h, the keyboard handler) and call int 21h, it will return the current address of int 9h in ES:BX. All we need to do is to fetch these values and return them to two QB variables for storing. The other subfunction is subfunction 25h, which changes the address of the interrupt. (Actually, it's usually called the interrupt vector.) We only need to set AH to 25h, AL to 9h, DS:DX to the address of the newkey\$ string and call int 21h. When this is done, our keyboard handler has taken over the control. I'm not going to describe every step in this routine like I've done before. There's nothing new in this routine, so I'll leave you to figure out how it works. It can be a good exercise for you to figure out exactly how this routine works. Regardless if you care or not, just add this code to the BAS file:

```
newseg% = VARSEG(newkey$) ' Segment address of our keyboard handler
newofs% = SADD(newkey$) ' Offset address of our keyboard handler
```

```
' ----- Created with Absolute Assembly 2.1 by Petter Holmberg, -97. ----- '
```

```
initkey$ = ""
initkey$ = initkey$ + CHR$(&H1E) ' PUSH DS
initkey$ = initkey$ + CHR$(&H55) ' PUSH BP
initkey$ = initkey$ + CHR$(&H89) + CHR$(&HE5) ' MOV BP,SP
initkey$ = initkey$ + CHR$(&HB4) + CHR$(&H35) ' MOV AH,35
initkey$ = initkey$ + CHR$(&HB0) + CHR$(&H9) ' MOV AL,09
initkey$ = initkey$ + CHR$(&HCD) + CHR$(&H21) ' INT 21
initkey$ = initkey$ + CHR$(&H89) + CHR$(&HDF) ' MOV DI,BX
initkey$ = initkey$ + CHR$(&H8B) + CHR$(&H5E) + CHR$(&HE) ' MOV BX,[BP+0E]
initkey$ = initkey$ + CHR$(&H8C) + CHR$(&H7) ' MOV [BX],ES
initkey$ = initkey$ + CHR$(&H8B) + CHR$(&H5E) + CHR$(&HC) ' MOV BX,[BP+0C]
```

```

initkey$ = initkey$ + CHR$(&H89) + CHR$(&H3F) ' MOV [BX],DI
initkey$ = initkey$ + CHR$(&H8B) + CHR$(&H56) + CHR$(&HA) ' MOV DX,[BP+0A]
initkey$ = initkey$ + CHR$(&H8E) + CHR$(&HDA) ' MOV DS,DX
initkey$ = initkey$ + CHR$(&H8B) + CHR$(&H56) + CHR$(&H8) ' MOV DX,[BP+08]
initkey$ = initkey$ + CHR$(&HB4) + CHR$(&H25) ' MOV AH,25
initkey$ = initkey$ + CHR$(&HB0) + CHR$(&H9) ' MOV AL,09
initkey$ = initkey$ + CHR$(&HCD) + CHR$(&H21) ' INT 21
initkey$ = initkey$ + CHR$(&H5D) ' POP BP
initkey$ = initkey$ + CHR$(&H1F) ' POP DS
initkey$ = initkey$ + CHR$(&HCA) + CHR$(&H8) + CHR$(&H0) ' RETF 0008

```

' ----- Created with Absolute Assembly 2.1 by Petter Holmberg, -97. ----- '

' Save the state of the keyboard flags:

```
DEF SEG = 0
```

```
keyflags% = PEEK(&H417) AND &H70
```

```
DEF SEG
```

' Install our new keyboard handler:

```
offset% = SADD(initkey$)
```

```
DEF SEG = VARSEG(initkey$)
```

```
CALL ABSOLUTE(oldseg%, oldofs%, BYVAL newseg%, BYVAL newofs%, offset%)
```

```
DEF SEG
```

The three lines above the CALL absolute code may seem confusing, but here's the explanation: Before installing the new keyboard handler it's a good idea to save the state of the keyboard flags. Don't mix this up with the flag array that we created for our own keyboard handler. The keyboard flags I'm talking about right now are a couple of bits stored at a fixed position in the memory by the default keyboard handler. They tell you the state of some of the keys on your computer, such as the Num Lock, Caps Lock, Scroll Lock, Alt, Ctrl and Shift keys. If we don't save the state of these keys before we install our keyboard handler and we hold down, say, a shift key before the new keyboard handler is installed and release it while this keyboard handler is in control, the old keyboard handler will think it's still being held down when we remove the new keyboard handler. Wow, that DID sound confusing, didn't it? Well, don't mind about it. We just save the state of these flags before installing our keyboard handler so we can set it back to that state afterwards.

Now all we have left is the routine that removes our keyboard handler and returns the control to the old one, and some additional BASIC code to test our keyboard handler with.

We'll just make it a simple demo that constantly prints the state of every element in the array on the screen so we can see if our keyboard handler reacts on the pressing and releasing of different keys. It's also important that we end the loop when we see that the ESC key is being held down, or otherwise we'll get stuck in the demo program. With our new keyboard handler it's not possible to use Ctrl-Break to abort the program. Our keyboard handler has the full control over the keyboard, remember? The state of the ESC key is stored in keydata%(1), so all we need to do is to wait for it to become 1. The routine that removes the handler only needs to use in 21h, subfunction 25h to put the old interrupt address back. And finally, we must set those keyboard flags to the state that they were in before we installed our keyboard handler:

```
CLS
PRINT "Multiple keypress demo. Press the ESC key to exit..."

' Display all of the keyboard flags on the screen:
DO
LOCATE 3, 1
FOR i = 0 TO 127: PRINT keydata%(i); : NEXT i
LOOP UNTIL keydata%(1) ' Exit from the loop if the user presses the ESC key
```

' ----- Created with Absolute Assembly 2.1 by Petter Holmberg, -97. ----- '

```
remkey$ = ""
remkey$ = remkey$ + CHR$(&H1E) ' PUSH DS
remkey$ = remkey$ + CHR$(&H55) ' PUSH BP
remkey$ = remkey$ + CHR$(&H89) + CHR$(&HE5) ' MOV BP,SP
remkey$ = remkey$ + CHR$(&H8B) + CHR$(&H56) + CHR$(&HA) ' MOV DX,[BP+0A]
remkey$ = remkey$ + CHR$(&H8E) + CHR$(&HDA) ' MOV DS,DX
remkey$ = remkey$ + CHR$(&H8B) + CHR$(&H56) + CHR$(&H8) ' MOV DX,[BP+08]
remkey$ = remkey$ + CHR$(&HB4) + CHR$(&H25) ' MOV AH,25
remkey$ = remkey$ + CHR$(&HB0) + CHR$(&H9) ' MOV AL,09
remkey$ = remkey$ + CHR$(&HCD) + CHR$(&H21) ' INT 21
remkey$ = remkey$ + CHR$(&H5D) ' POP BP
remkey$ = remkey$ + CHR$(&H1F) ' POP DS
remkey$ = remkey$ + CHR$(&HCA) + CHR$(&H4) + CHR$(&H0) ' RETF 0004
```

' ----- Created with Absolute Assembly 2.1 by Petter Holmberg, -97. ----- '

```
' Remove our keyboard handler:
offset% = SADD(remkey$)
DEF SEG = VARSEG(remkey$)
CALL ABSOLUTE(BYVAL oldseg%, BYVAL oldofs%, offset%)
DEF SEG
```

```
' Restore the keyboard flags to their old state:
DEF SEG = 0
POKE (&H417), keyflags%
DEF SEG
```

And that's it! This was an example of advanced assembly handling, but it wasn't too hard to understand, was it? There's much you can do with custom interrupt handlers. You can also make interrupt handlers that doesn't take over the control of the old one, but merely runs first and then passes over the control to the old interrupt handler. Some old computer viruses worked like this, but you should use them for nicer things :-)

I think this is enough for now. I'm sure you already have ideas for what you want to use this newly acquired knowledge for. If you want to look more at I/O ports I suggest that you go and find some document about advanced soundcard- or video programming. Just don't blame me if you fry your monitor ;-)

If you want to play with interrupts more, download Ralph Brown's interrupt list and start with something easy, like making mouse routines or using

basic DOS interrupts. If you want to do something more advanced, try writing interrupt handlers for other things than the keyboard. One thing you can do is to make interrupt routines that are called automatically 18.2 times per second. This can be useful for many things.

If you run into problems, (it's easy to do when programming these things) try hard to find them on your own. I've discovered that it's easy to start making the same, small mistake over and over again when you're writing asm code. If you find these annoying habits of yours by your own, you will usually never make them again. It's easy to forget simple things, such as that a certain register won't work in certain conditions, how the stack looks at the moment and that you accidentally deletes some important number by overwriting it with another. The only thing you can do is to practise and practise until you get so used to assembly programming that you know where the potential traps are.

See ya next chapter, when we close out this series!

Closing Out

Chapter 6

Hello and welcome to the sixth part of my assembly tutorial series. This part is also the very last one. No!!! I hear you screaming ;-) But the reason for this is basically that I have covered almost all of the important aspects of assembly programming. There's more to learn- there's always more to learn, but I think you'll have more use of other documents from now on. This series has been concentrating on the general aspects of asm programming. Now you are ready to start exploring stuff that interests you in particular. Maybe you want to learn more about interrupts, or you may be interested only in I/O port programming. There are lots of good documents to download about every aspect of assembly programming. It's not necessary for me to explain everything.

So, what should this last part be about? For the first time in this series, it hasn't been easy for me to decide. Therefore, this part will cover some miscellaneous stuff, mostly about some of the general aspects of assembly programming. I'll try to share my experiences of asm programming with you so you don't have to all the mistakes I've done.

Numbers in assembler

When programming in assembler, it's very important to know how numbers are stored in the registers, the stack and the memory. It's especially important to know the difference between positive and negative numbers and how they are stored.

We begin by looking at positive integer values, i.e. the numbers 0,1,2,...,n: Positive integer values are the easiest ones to store. When you need to use a positive integer value in your asm code, the first thing you want to ask is: How high numbers do I need to store? It's easy to select the number of bits that needs to be used for storing a certain value. If you use n bits, the biggest number that can be stored is $2^n - 1$, so if you use 8 bits, you can store any number from 0 up to $2^8 - 1 = 255$. With 16 bits you can store any number up to 65535 and so forth. If you only need to use a number smaller than 100, you shouldn't use 16 bits to store it as 8 bits are enough.

The different bits in a binary number are numbered from right to left. The rightmost bit is called the least significant bit and it has the bit number 0. The leftmost bit is called the most significant bit. If it's an 8-bit number this bit is bit no. 7.

If you want to calculate what number you get if you set a certain bit to 1, you can calculate this with the following formula:

$$n = 2^b$$

where n is the number you want to know and b is the number of the bit. So if you set bit 5 to one, the number you get is $2^5 = 32$.

Numbers that can only be positive are called unsigned values. A signed value is a value that can be both positive and negative. It's called signed numbers because the most significant bit of such numbers tell the sign of the number. If we have a positive value, the most significant bit is 0, but if we have a negative value, the most significant bit is 1. So 0 means + and 1 means -. This means that a signed number needs one extra bit just to store the sign of the number, so we get a lower maximum value with a certain amount of bits if we use signed numbers than an unsigned number. The biggest signed number that can be stored with n bits is $2^{(n-1)} - 1$. So if we have an 8-bit signed number, it can have the maximum value of $2^{(8-1)} - 1 = 127$.

The smallest signed number that can be stored with n bits is $-(2^{(n-1)})$, so with 8 bits, you can store any negative integer down to $-(2^{(8-1)}) = -128$. Thus, an 8-bit signed number can store all integer values from -128 to 127. Positive values are stored in the same way no matter if the number is signed or unsigned, but negative values are stored in an interesting form: The most significant bit is set to 1 to indicate it's a negative number, and the rest of the bits represents the maximum possible value with those bits plus one minus the absolute value of the negative number plus one. So if you have the 8-bit binary number -10, the most significant bit will be a 1 to indicate it's a negative number, and the rest of the bits will represent the number $127 + 1 - 10 = 118$. This form of storing positive and negative numbers is called the two's complement.

If you want to convert a number from positive to negative or negative to positive, you could use the SUB instruction to subtract a value from 0, like this:

```
SUB AX, 0
```

This would work fine for all numbers. But you could also use the special assembly instruction NEG. NEG works just in the same way, but it's faster, takes less place in the memory and it's

easier to understand what happens when you see a NEG instruction than if you see a SUB instruction. The syntax for NEG is simply:

NEG destination

The destination operand can be either a register or a memory pointer. So if you have the number -10 in AL and execute the instruction NEG AL, AL will become 10.

Sometimes it's necessary to convert an 8-bit number to a 16-bit number or a 16-bit number to a 32-bit number. This is easy if you have a positive number. For example: If you have a positive integer in AL and you want that number to be treated as a 16-bit value in AX, you only need to make sure AH is 0. But what about negative numbers then? How do you convert a number in the two's complement form upwards? Luckily, there are two instructions that does this for you. They're called CBW and CWD, short for Convert Byte to Word and Convert Word to Double word. Their syntaxes are:

CBW

CWD

Note that they don't have any input/output operands. That's because these instructions only works with the AX register. If you have an 8-bit number that you want to convert to a 16-bit number, you should put it in AL, execute a CBW instruction, and you'll get the correct 16-bit value in AX, no matter if it's a positive or a negative integer you've converted.

If you want to convert a 16-bit value to 32 bits, you should put it in AX and execute a CWD instruction. The result will be stored in DX:AX.

When to create an assembly routine and how to do it

A little assembly code can often mean a huge improvement for a BASIC program. However, it's important to know when to use it and when to avoid it. There's no use to rewrite everything in assembler. There are two reasons you may have to write an assembly routine: It will speed up your program, or it will enable you to do something that you cannot do in QuickBASIC.

The speed issue is the most common reason for using assembly code. It's a well known fact that in an average program, 90% of the execution time is used to execute 10% of the code. By rewriting those 10% of the code into assembler, 90% of the time, your program will run faster. It's important to learn how to find those 10% of source code. Usually, it's something that is repeatedly called by the main program loop and involves a lot of calculations. If you manage to pinpoint the part of your program that slows it down the most, you should consider rewriting it in assembler.

It's also a good idea to use assembler when you need to do something that is hard to handle in QuickBASIC. One example of this was the keyboard handling example in the previous part of this series. The keyboard functions in QuickBASIC doesn't let you control the keyboard at the level we often need for computer games, but with a little assembly, we get all the control we want.

When you know what you want to rewrite in assembler, you may think it's going to be easy to implement your idea into a working routine, but you may be surprised at how hard it is to write the routine. Transforming your ideas directly to assembler can be very hard because you have to take so many technical issues into account. You may find that you run out of registers, that you start writing really unstructured code or that you simply don't know how to do some things in assembly.

My experience in assembly programming has told me that the best way to go is this: First, start writing your routine in BASIC. This way, you'll be sure that you know every step in the process of writing the routine, and you don't have to mind about registers and other low-level stuff.

Then, start optimizing your code. Try to push your BASIC code to its limits. Even if the code still is terribly slow, you will discover how you can make it faster and better. If you wrote everything in assembler from the beginning, you probably would have missed these enhancements. Put comments everywhere in your source code so that you know what each line does.

When you have an optimized BASIC version, start rewriting it in assembler. You shouldn't try to make the final version right from the beginning though. Start by writing a pseudo-version, where you use variable names instead of registers and skip some low-level stuff like setting the memory registers correctly when you want to get a number from the memory. Make your assembly code similar to the BASIC version, so that you know exactly what you're doing. Put LOTS of comments into the code so that you know what every line does.

When you have your pseudo-version finished, start transforming it into a working version. Change variable names into registers, use the correct code to get a number from the memory and so forth. Put in even more comments, as the code will get harder to understand. The commenting should be so extensive that you can tell what each line of assembly code does just from looking at the comments. Divide the source into separate parts by using spacing and commenting, so that you can isolate the different parts of the routine just by looking at the layout of the source code. Even if you know what you're doing right now, you may have forgotten it after a few days. Looking at your own asm source without understanding it is really frustrating. DO NOT try to optimize your assembly code yet, try to make it work first. You will probably have a lot of bugs when you first run your routine, and you must fix them all before continuing.

Now when you have a working assembly routine, you may want to try optimizing it even further by using smart assembly code. This should be the final step in the creation of your routine. If you start optimizing right away, you're risking to lose control over your coding, but if you have a working routine without optimizations to look back on, you will still know what you're doing and you know that your routine works even if you don't manage to optimize it so much.

Make sure your comments still explains what the routine does. Your optimized code will be harder to understand than the original version, so you need to watch out.

If you follow these steps, you stand a good chance of getting a really well- written and terribly fast assembly routine!

Optimizing your assembly code

It's one thing to optimize a BASIC program by using assembly code, but you can also optimize your assembly code to gain even more speed.

Optimizing assembly code is a whole science itself, and I could probably write another tutorial series as long as this one just about asm optimization if I knew enough about it. There's so much to say about assembly optimization that I can only cover some of the basics here.

When optimizing an assembly routine, it's not enough to just make the separate instructions run faster by using smarter code. Many times you also need to look at the code as a whole and ask yourself what the code's actually doing. Your code may run at a near-optimal speed based on what it does, but maybe it's doing more than it has to. For example, you may have a routine that has a very time-consuming MUL instruction inside of a loop. You manage to make it run a lot faster by using a SHL instruction instead, and you think you've been very clever. But if you had taken another look, you might have discovered that you're actually doing the same multiplication over and over, and it would have been enough to do it once before the loop and save the result in a register for later use. By moving the MUL instruction above the loop you would probably have gained much more than by changing it to a SHL instruction.

Keep in mind though, that because your assembly code is so fast from the beginning compared to BASIC code, it's often unnecessary to mind about optimizing it. If your routine runs adequately fast for your program, it's better to leave it unoptimized since it's more readable that way. But it can also be really important to optimize your assembly code because it runs so often. Concentrate on optimizing loops and other assembly code that runs frequently. If you don't have room for all your variables within the registers, use the stack or the memory for the values you use the least, and save the free registers for the code inside loops.

Switching Registers

Sometimes you need to exchange the values of two registers. If you only use MOV to exchange them, you will need a free register for temporary storage of one of the values. For example, if we have one value in AX and one value in BX and we want them to switch places, we could do this:

```
MOV CX, AX ; Store value of AX temporarily in CX
MOV AX, BX ; AX = BX
MOV BX, CX ; BX = CX (which is the old value of AX)
```

If you need to do this and you have no free register, the only solution seems to be to push a value to free a register, but there's a better way. You can use the instruction XCHG, short for eXCHAnGe, to do it. The syntax for XCHG is:

XCHG destination, source

Where the source and destination operands can be registers or memory pointers. However, the source and the destination cannot both be memory pointers at the same time. With XCHG you won't need a temporary register, and thus the stack doesn't need to be used either.

Remove your jumps!

Another way to make your code faster is to avoid jumps in the code. This includes loops. Let's suppose you want to increase the AL register four times. The following solution seems obvious:

```
MOV CL, 4
IncLoop:
INC AL
LOOP IncLoop
```

But what's actually happening when the computer executes this code? Well, this is how the computer sees it:

```
Set CL to 4
Increase AL
Decrease CL
Is CL 0? No: Jump back one line
Increase AL
Decrease CL
Is CL 0? No: Jump back one line
Increase AL
Decrease CL
Is CL 0? No: Jump back one line
Increase AL
Decrease CL
Is CL 0? Yes: Continue execution
```

Now suppose that we rewrote the code into this:

```
INC AL
INC AL
INC AL
INC AL
```

When this code is executed, this is what the computer does:

```
Increase AL
Increase AL
Increase AL
Increase AL
```

As you can see, a loop isn't always the best choice. Replacing loops with repeated code is called unrolling. If the loop is executed so many times that it's not practical to unroll it completely, you can do a partial unrolling. For example, if you have a loop that executes 80

times, you can change it to a loop that executes 10 times with 8 copies of the loop code inside it.

It's not only certain instructions that makes an assembly routine slow: Wait states, bus transfers, the prefetch queue and dynamic RAM refreshes can also steal time, and avoiding these bad guys can be really hard.

For every new CPU that's released, assembly optimization becomes less important, not only because the processors get faster and faster, but also because they execute time consuming assembly instructions more efficiently. A MUL instruction for example, isn't as slow compared to an ADD instruction in a Pentium as it was on the 80286. On the other hand, the Pentium can execute several assembly instructions simultaneously if they fulfill certain requirements, so it's possible to optimize asm code for a Pentium by using these new features.

Finding errors

One of the biggest pains of writing assembly code is to find and eliminate the bugs. Once you're done writing an assembly routine and tests it for the first time, it almost never works properly. Since the error often hangs your computer so that you have to restart it, you'll probably get no hint at where the error is. All there is to do is to start going through your code and search for bugs. This can be very hard and time-consuming. Here are a few ways to make the debugging easier:

First of all, "execute" the code in your head and try to see what the routine is actually doing. Write it down on paper and you may discover that you're doing something wrong.

If you still have problems finding the errors, try to execute only a part of your program. Take the first snippet of code, insert a JMP to the end of the routine and return the values that you're working with to the BASIC program so you can look at them and see if they're correct so far. Then, move down the JMP instruction a couple of lines and see if everything still is correct. If you continue doing this, you will eventually find out where the computer hangs and where the error is.

Keep in mind that some assembly instructions doesn't work with certain combinations of registers, memory pointers and direct values. if DEBUG won't accept an instruction that may seem correct, it may be because you're using a combination of input/output values that cannot be handled by that instruction.

Also, make sure the call to and return from the assembly routine is done correctly. Many times, I've struggled to find the errors in completely correct assembly code that won't run, just to find out that the error was in the CALL ABSOLUTE line in QuickBASIC. Check this before you start searching for errors in the assembly code.

There's an error in CALL ABSOLUTE assembly routines that's so common it's important that you know about it. It's very nasty because you don't have to write a single line incorrectly to get this error. It has to do with the DS register. Be VERY careful when you set the DS register. You already know that it's important to push the value of DS before you change it and POP it back at the end of your program, but there's another thing you have to know about DS. When you set it to another value than the original you won't be able to fetch values passed from QuickBASIC. Consider the following code snippet:

```
PUSH DS ; Preserve DS
MOV BX, A000 ; Set DS to A000h
MOV DS, BX
MOV AX, [BP+08] ; Get a variable from QB
POP DS ; Restore DS
```

Even if this seems correct, AX won't get the value of a QB variable. The value it will receive is the value at DS:BP+8, which isn't the correct address. You must make sure DS points to the segment address it points to from the beginning if you want to read from or write to QB variables.

Doing nothing

If you feel an urge to do nothing for a while, the assembly language has an instruction just for that: NOP. The NOP instruction takes up one byte in the memory, and when it's executed... Well, nothing happens at all! The syntax for NOP is:

```
NOP
```

Even though it doesn't do anything, it takes up a little amount of time. Actually, NOP is equivalent to XCHG AL, AL, which, of course, doesn't change anything.

It may seem silly to have such an instruction in the assembly language, but there's actually some use for it sometimes, for example if you want to leave some empty space in a program for data to be initialized later on. The NOP instruction occupies one byte in the memory.

And that was the final thing I had to say. I hope you've enjoyed this tutorial series and that you've managed to understand everything. Thanks to everyone who's sent me emails with comments on and suggestions for this series. If you want to learn more about assembly programming, there's LOTS of good info to find on the Internet. For example, Check out:

<http://cs.smith.edu/~thiebaut/ArtOfAssembly/ArtofAsm.html>

I thought I could finish with a list of all the assembly instructions we've covered in this series. This list divides them into groups based on what they do and explains their different functions. Try to tell what they mean before looking at the explanations and see if you know them all!

Good luck with your assembly programming!

List of assembly instructions:

<i>Name</i>	<i>Meaning</i>
-------------	----------------

Data transfer:

MOV	Copy values between registers and the memory
PUSH	Put values on the stack
POP	Get values from the stack
LODS	Load string
STOS	Store string

MOVS	Move string
IN	Read values from I/O ports
OUT	Write values to I/O ports
NOP	Do nothing

Jumps:

CALL	Call subroutine
RET	Return from subroutine
RETF	Return from subroutine in another segment
JMP	Jump to another offset address
LOOP	Perform a loop
INT	Call interrupt routine
IRET	Return from interrupt routine

Conditional:

CMP	Compare two values
TEST	Compare two values
JB	Jump if Below
JBE	Jump if Below or Equal
JE	Jump if Equal
JAE	Jump if Above or Equal
JA	Jump if Above
JL	Jump if Less (signed)
JLE	Jump if Less or Equal (signed)
JGE	Jump if Greater or Equal (signed)
JG	Jump if Greater (signed)
JNB	Jump if Not Below
JNBE	Jump if Not Below or Equal
JNE	Jump if Not Equal
JNAE	Jump if Not Above or Equal
JNA	Jump if Not Above
JNL	Jump if Not Less (signed)
JNLE	Jump if Not Less or Equal (signed)
JNGE	Jump if Not Greater or Equal (signed)
JNG	Jump if Not Greater (signed)

Data manipulation:

ADD	Add two values together
SUB	Subtract value from another
INC	Increase value by one
DEC	Decrease value by one
MUL	Multiply to values together (unsigned)
IMUL	Multiply to values together (signed)
DIV	Divide value by another (unsigned)
IDIV	Divide value by another (signed)
NEG	Negate value (invert its sign)
SHL	Shift bits to the left (unsigned)
SHR	Shift bits to the right (unsigned)
SAL	Shift bits to the left (signed)
SAR	Shift bits to the right (signed)

ROL	Rotate bits to the left
ROR	Rotate bits to the right
AND	Logical AND (1 if both bits are 1)
OR	Logical OR (1 if one or both bits are 1)
XOR	Logical XOR (1 if one bit is 1 and one bit is 0)
NOT	Logical NOT (invert bit)
CBW	Convert byte value to word value (signed)
CWD	Convert word value to doubleword value (signed)

--- The End ---