Adam's Assembler Tutorial 1.0

PART I

Revision : 1.4
Date : 16-02-1996
Contact : blackcat@faroc.com.au
http://www.faroc.com.au/~blackcat

What is Assembler?

Assembler has got to be one of my favourite languages to work with. Not that it's an easy language at first, but when you become familiar with it, you'll realise just how logical it is.

Assembler is a low-level language which you can use to give you programs added speed on slow tasks. Basically it consists of statements which represent machine language instructions, and as it's nearly machine code, it's fast.

In the early days before the 8086 came about, yes, there were humans on the Earth back then, :), programming was not an easy task. When the first computers were developed, programming had to be done in machine code which was _not_ an easy task, and so Assembler was born.

Why use it?

As I said before, Assembler is fast. It also allows you to speak to the machine at hardware level, and gives you much greater control and flexibility over the PC. One of the other advantages of Assembler is that it allows you to impress your friends by entering pages of seemingly incomprehensible code. Watch them gather round you and be impressed/laugh at your nerdiness? :)

How did this tutorial come about?

Well, I had a couple of friends who wanted to learn Assembler to speed up their Pascal programs, so I gave them some Assembler Tutorials I had. While these tutorials had all the information you'd ever need, they were not written for the novice to easily understand, so I decided to write my own.

If you're using this tutorial and find it useful and informative, then please mail me. I appreciate feedback.

LESSON 1 - Registers

When you're working with Assembler, you'll have to use registers. You can think of these as variables already defined for you. The most common are listed below:

- AX - the accumulator. Comprises AH and AL, the high and low bytes of AX. Commonly used in mathematical and I/O operations.

- BX - the base. Comprises BH and BL. Commonly used as a base or pointer register.

- CX - the counter. Comprises CH and CL. Used often in loops.

- DX - the displacement, similar to the base register. Comprises DH and DL. I think you're getting the pattern now.

These registers are defined as general purpose registers as we can really store anything we want in them. They are also 16-bit registers, meaning that we can store a positive integer from 0 to 65535, or a negative integer from -32768 to 32768.

Incidently, the matter of the high and low byte of these resgisters has caused quite a bit of confusion in the past, so I'll try to give it some explaination here. AX has a range of 0 to FFFFh. This means that you have a range of 0 to FFh for AH and AL. (If you're a little concerned with the hex, don't worry. Next tutorial will cover it.)

Now if we were to store 0A4Ch in AX, AH will contain 0Ah, and AL will contain 4Ch. Get the idea? This is a pretty important concept, and I'll cover it in more depth next tute.

The segment registers: - ta da!

These are some other registers which we will not cover for the first few tutorials, but will look at in greater depth later. They are immensely handy, but can also be dangerous.

- CS - the code segment. The block of memory where the code is stored. DON'T fool around with this one unless you know what you are doing. I'm not all that sure that you can actually change it - I've never tried.

- DS - the data segment. The area in memory where the data is stored. During block operations when vast blocks of data are moved, this is the segment which the CPU commonly refers to.

- ES - the extra segment. Just another data segment, but this one is commonly used when accessing the video.

- SS - no, not the German army. This is the stack segment, in which the CPU stores return addresses from subroutines. Take care with this one. :)

Some others you will commonly use are:

- SI - the source index. Often used in conjuction with block move instructions. This is a pointer within a segment, usually DS, that is read from by the CPU.

- DI - the destination index. Again, you'll use it a lot. Another pointer within a segment, often ES, that is written to by the CPU.

- BP - the base pointer, used in conjunction with the stack segment. We won't be using it a lot.

- SP - the stack pointer, commonly used with the stack segment. DON'T fool around with this one until you are sure you know what you are doing.

By now you should understand what registers are. There are other registers too, and things known as flags, but we will not go into these as yet.

THINGS TO DO:

1) Learn the various registers off by heart.
2) Get a calculator that supports hexadecimal - damn handy, or a least an ASCII chart. That covers 0 - 255, or 0h to FFh.

LESSON 2 - The 8086 instruction set:

Okay, so you've learnt about registers, but how do you use them, and how do you code in Assembler? Well, first you'll need some instructions. The following instructions can be used on all CPU's from the 8086 up.

- MOV <dest>, <value> - MOVE. This instruction allows you to MOVE a value into a location in memory.

  EG: MOV AX, 13h

  This would move 13h (19 decimal) into the AX regsister. So if AX had previously held 0, it would now hold 13h.

  THIS ONLY MOVES THE VALUE INTO THE REGISTER, IT DOES NOT DO ANYTHING.

  EG: (In Pascal) AX := $13;

- INT <number> - INTERRUPT. This instruction generates an interupt. You can think of this as a bit like a procedure.

  EG: INT 10h

  Would generate interrupt 10h (16 decimal). Now what this would do depends on the contents of the AH register, among other things. For instance, if AX = 13h and interrupt 10h was generated, the video would be placed into 320x200x256 mode.

More accurately:

AH would equal 00 - set mode subfunction, and
AL would equal 13h - 320x200x256 graphics mode.

However, if AH = 2h, and interrupt 16h was generated, this would instruct the CPU to check if a keypress was waiting in the keyboard buffer.

If AH = 2h, and BH = 0h and interrupt 10h was generated, then the CPU would move the cursor to the X location in DL and the Y location in DH.

You can bear in mind, that AH contains the function to execute, and the other registers may contain any other data necessary.

DO NOT WORRY ABOUT THIS FOR NOW, WE WILL COVER IT IN GREATER DETAIL LATER.

- ADD <dest> <value> - ADD. This instruction adds a number to the value stored in dest.

```
EG: MOV AX, 0h  ; AX now equals 0h
    ADD AX, 5h  ; AX now equals 5h
    ADD AX, 10h ; AX now equals 15h
```

Pretty simple, huh?

- SUB <dest> <value> - SUBTRACT. I think you can guess what this does.

```
EG: MOV AX, 13h  ; AX now equals 13h  (19 dec)
    SUB AX, 5h   ; AX now equals 0Eh  (14 dec)
```

- DEC <register> - DECREMENTS something.

```
EG: MOV AX, 13h  ; AX now equals 13h
    DEC AX       ; AX now equals 12h
```

- INC <register> - INCREMENTS something.

```
EG: MOV AX, 13h  ; Take a guess
    INC AX       ; AX = AX + 1
```

- JMP <location> - JUMPS to a location.

```
EG: JMP 020Ah    ; Jump to the instruction at 020Ah
    JMP @MyLabel ; Jump to @MyLabel.
```

DON'T WORRY IF THIS IS A LITTLE CONFUSING - IT GETS WORSE! THERE ARE 28 DIFFERENT JUMP INSTRUCTIONS TO LEARN, MAYBE MORE. WE'LL COVER THEM LATER.

- CALL <procedure> - CALLS a subfunction.

```
EG: Procedure MyProc;

    Begin   { MyProc }
       { ... }
    End;    { MyProc }
```

```
Begin   { Main }
   Asm
      CALL MyProc   ; Guess what this does!
   End;
End.
```

```
OR: CALL F6E0h  ; Call subfunction at F6E0h
```

- LOOP <label> - LOOPS for a period of time.

```
EG: MOV CX, 10h  ; This is why CX is called the
                 ; COUNT register.  10h = 16

    @MyLabel:

    ; some stuff
    ; more stuff

    LOOP @MyLabel   ; Until CX = 0
                    ; Note: CX gets decremented
                    ; each time.  Don't DEC it
                    ; yourself.

; THIS WOULD LOOP 16 times - thats 10 in hex.
```

- LODSB - Load a byte
  LODSW - Load a word
  STOSB - Store a byte
  STOSW - Store a word

These instructions are used to put or get something in a location in memory. The ES:SI register, (remember we talked about this earlier as SI being the source index?), points to the location we want to get data from, and ES:DI points to where we will be putting information.

Anyway, imagine that we have the following set-up in memory:

Memory Location	06	07	08	09	10	11	12
Value	50	32	38	03	23	01	12

When we use LODSB or STOSB, it returns or gets a number in AL. So if ES:SI pointed to 07 and we executed a LODSB instruction, AL would now equal 32.

Now, if we pointed ES:DI to 11, put say, 50 in the AL register and executed STOSB, then the following would result:

Memory Location	06	07	08	09	10	11	12
Value	50	32	38	03	23	50	12

NOTE: When we use LODSB/STOSB we use AL. This is because we will be dealing with an 8-bit number, (a byte) only. We can store an 8-bit number in AL, AH, or AX, but we cannot store a 16-bit number in AH or AL because these are 8-BIT REGISTERS.

As a result, when we uses LODSW or STOSW, we must use AX and not

AL, as we will be getting/putting a 16-bit number.

- MOVSB - Move a byte
  MOVSW - Move a word

  As an example we'll get a byte from DS:SI and send it to ES:DI.

  At DS:SI:

Memory Location	06	07	08	09	10	11	12
Value	50	32	38	03	23	50	12

  At ES:DI:

Memory Location	06	07	08	09	10	11	12
Value	10	11	20	02	67	00	12

  If point DS:SI to location 07, point ES:SI to location 11 and execute MOVSB, the stuff at ES:DI will look like:

  At ES:DI:

Memory Location	06	07	08	09	10	11	12
Value	10	11	20	02	67	32	12

  I HOPE YOU GET THE GENERAL IDEA. HOWEVER, OF COURSE IT ISN'T THAT SIMPLE. MEMORY LOCATIONS AREN'T ARRANGED IN ARRAY FORM, ALTHOUGH I WISH THEY WERE. WHEN MOVING/GETTING/PUTTING YOU BE DEALING WITH A SEGMENT/ OFFSET LOCATION.

- REP - REPEAT for the number of times specified in the CX register.
  A REP in front of a MOVSB/LODSB/STOSB instruction would cause that instruction to be repeated. So:

  If CX = 5, and
  if ES:DI pointed to 1000:1000h,

  then REP STOSB would store what was in the AL register in the location 1000:1000h 5 times.

THINGS TO DO:

1) Memorise all the instructions above - it's not hard and there's not many there.

2) Make sure you understand the theory behind it.

COMING UP NEXT WEEK:

- Hexadecimal and what it is.

- Segments and offsets - we touched on them in this tute.
- Some more instructions.
- Some sample programs, and code you can use in your programs.

  Maybe a PutPixel, ClrScr, anything I think is useful.

If you wish to see a topic discussed in a future tutorial, then mail me, and I'll see what I can do.

Don't miss out!!! Download next week's tutorial from my homepage at:

- http://www.faroc.com.au/~blackcat

- Adam.

Adam's Assembler Tutorial 1.0

PART II

Revision : 1.4
Date : 17-02-1996
Contact : blackcat@faroc.com.au
http://www.faroc.com.au/~blackcat

Hello again, budding Assembler programmers. For those who missed the first issue, get it now at my homepage.

Anyway, last issue I said we'd be discussing hexadecimal, segments + offsets, some more instructions and some procedures containing assembler that you could actually use.

So, here we go with segments and offsets!

LESSON 3 - Segments and Offsets

Before we delve into the big, bad world of segments and offsets, there is some terminology you'll need to know.

- The BIT - the smallest piece of data we can use. A bit - one eigth of a byte can be either a 1 or a 0. Using these two digits we can make up numbers in BINARY or BASE 2 format.

  ```
  EG:     0000 = 0   0100 = 4   1000 = 8    1100 = 12   10000 = 16
          0001 = 1   0101 = 5   1001 = 9    1101 = 13   ...I think you
          0010 = 2   0110 = 6   1010 = 10   1110 = 14   get the idea...
          0011 = 3   0111 = 7   1011 = 11   1111 = 15
  ```

- The NIBBLE, or four bits. A nible can have a maximum value of 1111 which is 15 in decimal. This is where hexadecimal comes in. hex is based on those 16 numbers, (0-15), and when writing hex, we use the 'digits' below:

  0 1 2 3 4 5 6 7 8 9 A B C D E F

  Hexadecimal is actually quite easy to use, and just as a 'fun fact', I think the Babylonians - some ancient civilisation anyway - used a BASE-16 number system. Any historians out there who want to confirm this?

  IMPORTANT >>> A nibble can hold a value up to Fh <<< IMPORTANT

- The BYTE - what we'll be using most. The byte is 8 bits long - that's 2 nibbles, and is the only value you'll be able to put in one of the 8-bit registers, EG: AH, AL, BH, BL, ...

  A byte has a maximum value of 255 in decimal, 11111111 in binary, or FFh in hexadecimal.

- The WORD - another commonly used unit. A word is a 16-bit number, and is capable of holding a number up to 65535. That's 1111111111111111 in binary, and FFFFh in hex.

  Note: Because a word is four nibbles, it is also represented by four hexadecimal figures.

  Note: This is a 16-bit number, and this corresponds to the 16-bit registers. That's AX, BX, CX, DX, DI, SI, BP, SP, DS, ES, SS and IP.

- The DWORD, or double word consists of 2 words or 4 bytes or 8 nibbles or 32-bits. You will not use double words much in these tutorials, but we'll mention them later when we cover 32-BIT PROGRAMMING.

  A DWORD can hold from 0 to 4,294,967,295, that's FFFFFFFFh, or 11111111111111111111111111111111. I hope there's 32 one's back there.

  The DWORD is also the size of the 32-BIT extended registers, or EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP and EIP.

- The KILOBYTE, is 1024 bytes, _NOT_ 1000 bytes. The kilobyte is equal to 256 double-words, 512 words, 1024 bytes, 2048 nibbles or 8192 BITS. I'm not going to write out all the one's.

- The MEGABYTE, or 1024 kilobytes. That's 1,048,576 bytes or 8,388,608 bits.

Now we've covered the terminology, let's have a closer look at just how those registers are structured. We said that AL and AH were 8-bit registers, so shouldn't they look something like this?

```
              AH                                    AL
     ┌────┬──┬──┬──┬──┬──┬──┬────┐      ┌────┬──┬──┬──┬──┬──┬──┬────┐
     │   0│0 │0 │0 │0 │0 │0 │0   │      │   0│0 │0 │0 │0 │0 │0 │0   │
```

In this case, both AH and AL = 0, OR 00h and 00h. As a result, to work out AX we use: AX = 00h + 00h. When I say + I mean, 'just put together' not AX = AH PLUS AL.

So, if AH were to equal 00000011 and AL were to equal 0000100, to work out AX we must do the following.

1) Get the hex values for AH and AL.

```
   00000011 = 03h   00010000 = 10h
```

2) Combine them.

```
   AX = AH  + AL
   AX = 03h + 10h
   AX = 0310h
```

And there you have it. Not too tricky.

Okay, now lets look at a 16-bit register:

```
                                 AX
                    ┌────────────────────────┐
                    │                        │

              AH                         AL
     ┌────┬──┬──┬──┬──┬──┬──┬────┐  ┌────┬──┬──┬──┬──┬──┬──┬────┐
     │   0│0 │0 │0 │0 │0 │0 │0   │  │   0│0 │0 │0 │0 │0 │0 │0   │
```

So from that, we can see that AX = 00000000 and 00000000, or 0000000000000000.

Now lastly, lets see what a 32-bit register looks like:

```
        ┌─────────────────────────────────────────────────┐
        │                       EAX                       │
        ├────────────────────────┐                        │
        │           AX           │                        │
        ├────────────┬───────────┤                        │
        │ 00000000   │ 00000000  │ 00000000     00000000  │
        │    AH      │    AL     │                        │
        └────────────┴───────────┴────────────────────────┘
```

Not too difficult either, I hope. And if you got that, you're ready for SEGMENTS and OFFSETS.

A Segmented Architechture

Long, long ago, when IBM built the first PC, it wasn't feasible for programs to be above 1 megabyte - heck, the first XT's had only 64K of RAM! Anyway, seeing as the designers of the XT didn't envisage huge applications, they decided split memory up into SEGMENTS, measily small areas of RAM which you can JUST fit a virtual screen for 320x200x256 graphics mode in.

Of course, you can access more than a megabyte of RAM, but you have to split it up into segments to use it, and this is the problem. Of course, with 32-bit programming you can access up to 4GB of RAM without using segments, but that's another story.

Segments and offsets are just a method of specifying a location in memory.

```
EG:   3CE5:502A

      ^^^^ ^^^^
      SEG  OFS
```

Okay, here's the specs:

```
An OFFSET  = SEGMENT X 16
A  SEGMENT = OFFSET  / 16
```

Some segment registers are:

```
CS, DS, ES, SS and FS, GF  - Note: The last 2 are 386+ registers.
```

Some offset registers are:

```
BX, DI, SI, BP, SP, IP     - Note: When in protected mode, you can use any
                                   general purpose register as an offset
                                   register - EXCEPT IP.
```

Some common segments and offsets are:

    CS:IP - Addres of the currently executing code.
    SS:SP - Address of the current stack position.

    NOTE: DO NOT TAMPER WITH THESE!

So when we refer to segments and offsets, we do so in the form:

SEGMENT:OFFSET

A good example would be:

```
A000:0000 - which actually corresponds to the top left of the VGA screen in
            320x200x256 color mode.

            ** FUN FACT ** VGA RAM starts a A000h  :)
```

Phew! That was a lot for the second tute. However, we're not done yet. The AX, AH, AL thing is a concept you may not have grasped yet, so here we go:

```
    MOV   AX, 0     ; AX = 0
    MOV   AL, 0     ; AL = 0
    MOV   AH, 0     ; AH = 0
```

```
    MOV   AL, FFh   ; AL = FFh
                    ; AX = 00FFh
                    ; AH = 00h

    INC   AX        ; AX = AX + 1

                    ; AX = 0100h
                    ; AH = 01h
                    ; AL = 00h

    MOV   AH, ABh   ; AX = AB00h
                    ; AH = ABh
                    ; AL = 00h
```

Got it yet?

THINGS TO DO:

1) Learn the BIT/NIBBLE/BYTE... stuff off by heart.
2) Go back over the segment and offset examples.
3) Make sure you understand the relationship between AX, AH and AL.
4) How about some hex addition problems?

The Stack

The stack is a very useful feature which we can take advantage of. Think of it as stack of papers in an IN tray. If you put something on the top, it'll be the first one taken off.

As you add something to the stack, the stack pointer is DECREASED, and when you take it off, it is INCREASED. To explain this better, look at the diagram below:

```
┌───────────────────┐
│     The STACK     │
├───────────────────┤
│                   │      <<< When PUSHing a byte onto the stack, it goes
│ ■                 │          here - last on, first off.
│ ■                 │
│ ■                 │
│ ■                 │
│ ■                 │


│  SP               │      <<< The stack pointer moves downward.
└───────────────────┘
```

And in practice:

```
   MOV   AX, 03h   ; AX = 03h
   PUSH  AX        ; PUSH AX onto the stack
```

```
   MOV   AX, 04Eh  ; AX = 04Eh

                   ; Do anything...perform a sum?

   POP   AX        ; AX = 03h
```

Or:

```
   MOV   AX, 03h   ; AX = 03h
   PUSH  AX        ; Add AX to the stack

   MOV   AX, 04Eh  ; AX = 04Eh

                   ; Do anything...perform a sum?

   POP   BX        ; BX = 03h
```

You've just learnt two new instructions:

- PUSH <register>   - PUSHes something onto the stack, and
- POP <register>   - POPs it back off.

That's all you'll need to know about the stack - for now.

And lastly, some procedures which demonstrate some of this stuff. Note that the comments have been DELIBERATELY REMOVED. It is your task to try and comment them, and by comment I just mean write down what each instruction is doing. Note also, that some new instructions are introduced.

```
Procedure ClearScreen(A : Byte; Ch : Char);   Assembler;

Asm     { ClearScreen }
  mov   ax, 0B800h
  mov   es, ax
  xor   di, di
  mov   cx, 2000
  mov   ah, A
  mov   al, &Ch
  rep   stosw
End;    { ClearScreen }


Procedure CursorXY(X, Y : Word);   Assembler;

Asm    { CursorXY }
   mov   ax, Y
   mov   dh, al
   dec   dh
   mov   ax, X
   mov   dl, al
   dec   dl
   mov   ah, 2
```

```
   xor   bh, bh
   int   10h
End;    { CursorXY }


Procedure PutPixel(X, Y : Integer; C : Byte; Adr : Word);   Assembler;

Asm     { PutPixel }
   mov   ax, [Adr]
   mov   es, ax
   mov   bx, [X]
   mov   dx, [Y]
   xchg  dh, dl
   mov   al, [C]
   mov   di, dx
   shr   di, 2
   add   di, dx
   add   di, bx
   stosb
End;    { PutPixel }


Procedure Delay(ms : Word);   Assembler;

Asm     { Delay }
   mov   ax, 1000
   mul   ms
   mov   cx, dx
   mov   dx, ax
   mov   ah, 86h
   int   15h
End;    { Delay }
```

THINGS TO DO:

1) Go over the stack example. Make your own code example.
2) Comment the procedures above as best as you can. Try and guess what the new instructions do. It's not that hard.

COMING UP NEXT WEEK:

- Many more instructions, all the JUMPS.
- What are flags?
- The above procedures with comments.
- An assembler-only program. You'll need DEBUG at least, though TASM and TLINK would be a good idea.

If you wish to see a topic discussed in a future tutorial, then mail me, and I'll see what I can do.

Don't miss out!!! Download next week's tutorial from my homepage at:

- http://www.faroc.com.au/~blackcat

- Adam Hyde.

```
╔═════════════════════════════════════════════════════════╗
║              Adam's Assembler Tutorial 1.0              ╟──┐
║                                                         ║  │
║                        PART III                         ║  │
╚═════════════════════════════════════════════════════════╝  │
  └──────────────────────────────────────────────────────────┘
```

Revision : 1.3
Date   : 27-02-1996
Contact : blackcat@faroc.com.au
            http://www.faroc.com.au/~blackcat

Welcome to the third tutorial in the series. Last tutorial I said we'd be discussing some more instructions, flags and an actual assembler program.

During this tutorial, you will find "Peter Norton's Guide to Assembler", "Peter Norton's Guide to the VGA Card", or any of the "Peter Norton's Guide to..." books damn handy. You cannot program in Assembler without knowing what all the interrupts are for and what all the subfunctions are.

I recommend you obtain a copy as soon as possible.

An Assembler Program

I don't generally write code in 100% Assembler. It is much more convenient to use a high level language such as C or Pascal, and use Assembler to speed up the slow bits. However, you may wish to torture yourself and write an application completely in Assembler, so here is the basic template:

```
┌───────────────
│ DOSSEG     - tells the CPU how to sort the segment.  CODE, DATA + STACK
├──
│ MODEL      - declare the model we will use
├──
│ STACK      - how much stack will we allocate?
├──
│ DATA       - what's going into the data segment
```

```
├──
│ CODE       - what's going into the code segment
├──
│ START      - the start of your code
├──
│ END START  - the end of your code
└───────────────
```

FUN FACT: I know of someone who wrote a Space Invaders clone, (9K), all in Assembler. I have the source if anyone is interested...

Okay, now let's look at a sample program that'll do absolutely nothing!

```
    DOSSEG            ; Not really necessary
    .MODEL SMALL
    .STACK 200h
    .DATA
    .CODE

START:
    MOV   AX, 4C00h   ; AH = 4Ch, AL = 00h
    INT   21h
END START
```

Let's go over this in more detail. Below, each of the above statements are explained.

- DOSSEG - this sorts the segments in the order:

  Code segments;
  Data segments;
  Stack segments.

  Not really necessary, but leave it in while you are learning.

- MODEL - this allows the CPU to determine how your program is structured. You may have the following MODELs:

  1) TINY - both code and data fit in the same 64K segment.

  2) SMALL - code and data are in different segments, though each are less than 64K.

  3) MEDIUM - code can be larger than 64K, but data has to be less than 64K.

  4) COMPACT - code is less than 64K, but data can be greater than 64K.

  5) LARGE - code and data may be larger than 64K, though arrays cannot be greater than 64K.

  6) HUGE - code, data and arrays may be larger than 64K.

- STACK - this instructs the PC to set up a stack as large as the amount specified.

- DATA - allows you to create a data segment.

Basically, where all your data will go.

- CODE - allows you to create a code segment.

  Basically, where all your code will go.

- START - Just a label to tell the compiler where the main body of your program begins.

  - MOV AX, 4C00h ; AH = 4Ch, AL = 00h

  This moves 4Ch into ah, which coincidently returns us to DOS. When interrupt 21h is called and AH = 4Ch, back to DOS we go.

  - INT 21h

- END START - Do you have no imagination?

Okay, I hope you got all that, because now we're actually going to do something. Excited yet? :)

In this example we'll be using interrupt 21h, (the DOS interrupt), to print a string. To be precise, we'll be using subfunction 9h, and it looks like this:

- INTERRUPT 21h
- SUBFUNCTION 9h

Requires:

- AH = 9h
- DS:DX = FAR pointer to the string to be printed. The string must be terminated with a $ sign.

So here's the example:

```
    DOSSEG
    .MODEL SMALL
    .STACK 200h
    .DATA

OurString   DB  "This is a string of characters.  "
            DB  "Do you lack imagination?  Put something interesting here!$"

    .CODE

START:
    MOV   AX, SEG OurString     ; Move the segment where OurString is located
    MOV   DS, AX                ; into AX, and now into DS

    MOV   DX, OFFSET OurString  ; Offset of OurString -> DX
    MOV   AH, 9h                ; Print string subfunction
    INT   21h                   ; Generate interrupt 21h

    MOV   AX, 4C00h             ; Exit to DOS sufunction
    INT   21h                   ; Generate interrupt 21h
END START
```

If you assemble this with TASM - TASM WHATEVERYOUWANTTOCALLIT.ASM then link with TLINK - TLINK WHATEVERYOUCALLEDIT.OBJ you'll get an EXE file of about 652 bytes. You can use these programs in DEBUG with some modifications, but I'll leave that up to you. To work with standalone Assembler you _need_ TASM and TLINK, though I guess MASM <shudder> would do the same job OK.

Now lets go over the code in a bit more detail:

```
    MOV   AX, SEG OurString     ; Move the segment where OurString is located
    MOV   DS, AX                ; into AX, and now into DS

    MOV   DX, OFFSET OurString  ; Move the offset where OurString is located
    MOV   AH, 9h                ; Print string subfunction
    INT   21h                   ; Generate interrupt 21h
```

You'll notice we had to use AX to put the segment address of OurString in DS. You will discover that you cannot refer to a segment register directly in Assembler. In last tute's PutPixel procedure, I moved the address of the VGA into AX, and then into ES.

The SEG instruction is also introduced. SEG returns the segment of where the string OurString is located, and OFFSET returns, guess what?, the offset from the beginning of the segment to where the string ends.

Notice also that we used DB. DB is nothing special, and stands for Declare Byte, which is all it does. DW, Declare Word and DD, Declare Double word also exist.

You could have also put OurString in the code segment, the advantage being CS will be pointing to the same segment as OurSting, so you wont have to worry about finding the segment which OurString lies in.

The above program in the code segment would look like:

```
    DOSSEG
    .MODEL SMALL
    .STACK 200h
    .CODE

OurString     DB  "Down with the data segment!$"

START:
    MOV   AX, CS
    MOV   DS, AX

    MOV   DX, OFFSET Message
    MOV   AH, 9
    INT   21h

    MOV   AX, 4C00h
    INT   21h
END START
```

Simple, no?

We won't look at standalone Assembler programs again all that much, but most of the techniques we'll be using can be implemented in the basic Assembler standalone template.

So, what are flags?

This part's for my mate Clive who's been hassling me about flags for a while, so here we go Clive, with FLAGS.

I can't remember if we introduced the CMP instruction or not, CMP - (COMPARE), but CMP compares two numbers and reflects the comparison in the FLAGS. To use it you'd do something like this:

- CMP AX, BX

then follow with a statement like those below:

UNSIGNED COMPARISONS:

- JA - jump if AX was ABOVE BX;
- JAE - jump if AX was ABOVE or EQUAL to BX;
- JB - jump if AX was BELOW BX;
- JBE - jump if AX was BELOW or EQUAL to BX;
- JNA - jump if AX was NOT ABOVE BX;
- JNAE - jump if AX was NOT ABOVE or EQUAL to BX;
- JNB - jump if AX was NOT BELOW BX;
- JNBE - jump if AX was NOT BELOW or EQUAL to BX;
- JZ - jump if ZERO flag set - same as JE;
- JE - jump if AX is EQUAL to BX;
- JNZ - jump if ZERO flag NOT set - same as JNE;
- JNE - jump if AX is NOT EQUAL to BX;

SIGNED COMPARISONS:

- JG - jump if AX was GREATER than BX;
- JGE - jump if AX was GREATER or EQUAL to BX;
- JL - jump if AX was LOWER than BX;
- JLE - jump if AX was LOWER or EQUAL to BX;
- JNG - jump if AX was NOT GREATER than BX;
- JNGE - jump if AX was NOT GREATER or EQUAL to BX;
- JNL - jump if AX was NOT LOWER than BX;
- JNLE - jump if AX was NOT LOWER or EQUAL to BX;
- JZ - jump if ZERO flag set - same as JE;
- JE - jump if AX EQUALS BX;
- JNZ - jump if ZERO flag NOT set - same as JNE;
- JNE - jump if AX is NOT EQUAL to BX;

NOT SO COMMON ONES:

- JC - jump if CARRY flag set;
- JNC - jump if CARRY flag NOT set;
- JO - jump if OVERFLOW flag is set;
- JNO - jump if OVERFLOW flag NOT set;
- JP - jump if PARITY flag is set;

- JNP   - jump if PARITY flag is NOT set;
- JPE   - jump if PARITY is EVEN - same as JP;
- JPO   - jump if PARITY is ODD - same as JNP;
- JS   - jump if SIGNAL flag is NOT set;
- JNS   - jump if SIGNAL flag NOT SET.

Phew! My eyes have almost dried out after staring at this screen for so long!

Anyway, here's what they look like:

Flag	SF	ZF	--	AF	--	PF	--	CF
Bit	07	06	05	04	03	02	01	00

```
 Key:
------

SF - Sign flag;
ZF - Zero flag;
AF - Auxillary flag;
PF - Parity flag.
CF - Carry flag.
```

Note: THERE ARE MANY MORE FLAGS TO LEARN. They'll be covered in a later Tutorial.

THINGS TO DO:

1) Go over the basic Assembler frame and memorise it all.
2) Try writing a simple program that displays some _imaginative_ comments.
3) Learn the least cryptic JUMP statements off by heart.

Okay, last tute I gave you some pretty nifty procedures, and asked you to comment them. I didn't wnat a detailed explanation of what they did - you're not expected to know that yet - just a summary of what each instruction does.

EG:

```
   MOV   AX, 0003h   ; AX now equals 03h;
   ADD   AX, 0004h   ; AX now equals 07h;
```

So, here's the full set of procedures with comments:

```
{ This procedure clears the screen in text mode }

Procedure ClearScreen(A : Byte; Ch : Char);   Assembler;

Asm     { ClearScreen }
  mov   ax, 0B800h    { Move the video address into AX      }
  mov   es, ax        { Point ES to the video segment       }
```

```
  xor   di, di        { Zero out DI                           }
  mov   cx, 2000      { Move 2000 (80x25) into CX             }
  mov   ah, A         { Move the attribute into AH            }
  mov   al, &Ch       { Move the character to use into AL     }
  rep   stosw         { Do it                                 }
End;    { ClearScreen }
```

Explanation:

We zero out DI so it equals 0 - the left hand corner of the screen. This is where we will start filling the screen from.

We move 2000 into CX because we will be putting 2000 characters onto the screen.

```
{ This procedure moves the cursor to location X, Y }

Procedure CursorXY(X, Y : Word);   Assembler;

Asm    { CursorXY }
   mov   ax, Y              { Move Y value into AX           }
   mov   dh, al             { Y goes into DH                 }
   dec   dh                 { Adjust for zero based routine  }
   mov   ax, X              { Move X value into AX           }
   mov   dl, al             { X goes into DL                 }
   dec   dl                 { Adjust for zero based routine  }
   mov   ah, 2              { Call the relevant function     }
   xor   bh, bh             { Zero out BH - page 0           }
   int   10h                { Do it                          }
End;    { CursorXY }
```

Explanation:

The 'adjusting for the zero-based BIOS' is done because the BIOS refers to position (1, 1) as (0, 0), and likewise (80, 25) as (79, 24).

```
Procedure PutPixel(X, Y : Integer; C : Byte; Adr : Word);   Assembler;

Asm     { PutPixel }
   mov   ax, [Adr]          { Move the address of the VGA into AX }
   mov   es, ax             { Dump AX in ES                       }
   mov   bx, [X]            { Move X value into BX                }
   mov   dx, [Y]            { Move Y value into DX                }
   xchg  dh, dl             { From here onwards calculates the    }
   mov   al, [C]            { offset of the pixel to be plotted   }
   mov   di, dx             { and puts this value in DI.  We will }
   shr   di, 2              { cover this later - next tute - when }
   add   di, dx             { we cover shifts vs muls.            }
   add   di, bx
   stosb                    { Store the byte at ES:DI             }
End;    { PutPixel }
```

NOTE: I would be greatly interested in finding a PutPixel procedure faster than this one. I have seen an inline one which does this in about half the time, but even so, this one is pretty hot.

```
{ This procedure is a CPU independant delay function }

Procedure Delay(ms : Word);   Assembler;

Asm     { Delay }
   mov   ax, 1000           { Move the # of ms in a sec into AX   }
   mul   ms                 { Make AX = # of ms to wait           }
   mov   cx, dx             { Get ready for delay - put # of ms   }
   mov   dx, ax             { where necessary                     }
   mov   ah, 86h            { Create the delay                    }
   int   15h
End;    { Delay }
```

Just about all the fluid has left my eyes now - it's nearly midnight - so I'd better stop. Sorry that the comments are a bit short, but I need my sleep!

Next tutorial will cover:

- Shifts - what are they?
- Some CMP/JMP examples.
- How VGA memory is arranged, and how to access it.
- um, some other great topic.

Next week I'll make an effort to show you how to access memory quickly, ie the VGA, and give you some examples.

If you wish to see a topic discussed in a future tutorial, then mail me, and I'll see what I can do.

Don't miss out!!! Download next week's tutorial from my homepage at:

- http://www.faroc.com.au/~blackcat

See you next week!

- Adam.

Adam's Assembler Tutorial 1.0

PART IV

Revision : 1.5
Date   : 01-03-1996
Contact : blackcat@faroc.com.au
            http://www.faroc.com.au/~blackcat

Welcome back, budding Assembler coders. The tutorials seem to be getting popular now, and I've had some mail requesting me to cover the VGA so I'll give it a go. This is basically what I've been leading up to in my own disjointed way anyhow, as graphics programming is not only rewarding, it's also fun too! Well, I think it's fun. :)

Firstly though, we must finish off the CMP/JMP stuff, and cover shifts. When you're coding in Assembler, you'll find comparisons, shifts and testing bits are very common operations.

A Comparison Example

I won't bother going over the following example - it's fairly easy to understand and you should get the basic idea anyway.

```
   DOSSEG
   .MODEL SMALL
   .STACK 200h
   .DATA

FirstString    DB  13, 10, "Is this a great tutorial or what? :) - $"
SecondString   DB  13, 10, "NO? NO? What do you mean, NO?$"
ThirdString    DB  13, 10, "Excellent, let's hear you say that again.$"
FourthString   DB  13, 10, "Just a Y or N will do.$"
ExitString     DB  13, 10, "Fine, be like that!$"

   .CODE

START:
   MOV   AX, @DATA                     ; New way of saying:
   MOV   DS, AX                        ; DS -> SEG data segment

KeepOnGoing:
   MOV   AH, 9
   MOV   DX, OFFSET FirstString        ; DX -> OFFSET FirstString
   INT   21h                           ; Output the first message

   MOV   AH, 0                         ; Get a key - store it in AX
   INT   16h                           ; AL - ASCII code, AH - scan code
                                       ; It doesn't echo onto the screen
                                       ; though, we have to do that ourselves

   PUSH  AX                            ; Here we display the char - note that
   MOV   DL, AL                        ; we save AX.  Obviously, using AH to
   MOV   AH, 2                         ; signal to print a string destroys AX
   INT   21h
   POP   AX
```

```
   CMP   AL, "Y"                      ; Check to see if 'Y' was pressed
   JNE   HatesTute                    ; If it was, keep going

   MOV   AH, 9                        ; Display the "Excellent..." message
   MOV   DX, OFFSET ThirdString
   INT   21h
   JMP   KeepOnGoing                  ; Go back to the start and begin again

HatesTute:
   CMP   AL, "N"                      ; Make sure it was 'N' they pressed
   JE    DontLikeYou                  ; Sadly, it was equal

   MOV   DX, OFFSET FourthString      ; Ask the user to try again
   MOV   AH, 9
   INT   21h
   JMP   KeepOnGoing                  ; Let 'em try

DontLikeYou:
   MOV   DX, OFFSET SecondString      ; Show the "NO? NO? What..." string
   MOV   AH, 9
   INT   21h

   MOV   DX, OFFSET ExitString        ; Show the "Fine, be like that!" string
   MOV   AH, 9
   INT   21h

   MOV   AX, 4C00h                    ; Return to DOS
   INT   21h
END START
```

You should understand this example, play around with it and write something better. Those with a "Peter Norton's Guide to..." book or similar, experiment with the keyboard subfunctions, and see what other similar GetKey combinations exist, or better still, play around with interrupt 10h and go into some weird video mode - one which your PC supports! - and use some color.

Shifts

A simple concept, and one which I should have discussed before, but like I said - I have my own disjointed way of going about things.

First you'll need to understand some hexadecimal and binary arithmetic - a subject I _should_ have covered before. I usually use a scientific calculator - hey, I always use a calculator, I'm not stupid! - but it is good to be able to know how to multiply, add and convert between the various bases.

You also cannot use a calculator in Computing exams, not in Australia anyway.

CONVERTING BINARY TO DECIMAL:

Way back in Tutorial One we looked at what binary numbers look like, so imagine I have an eight-bit binary number such as:

```
 11001101
```

What is this in decimal??? There are a number of ways to convert such a number, and I use the following, which I believe is probably the easiest:

Binary Number	1	1	0	0	1	1	0	1
Decimal equivalent	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Decimal equivalent	128	64	32	16	8	4	2	1
Decimal value	128 +	64 +	0 +	0 +	8 +	4 +	0 +	1 = 205

Get the idea? Note for the last line, it would be more accurate to write:

$$1 \times 128 + 1 \times 64 + 0 \times 32 + 0 \times 16 + 1 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1$$
$$= 128 + 64 + 0 + 0 + 8 + 4 + 0 + 1$$
$$= 205$$

Sorry if this is a little confusing, but it is difficult to explain without demonstrating. Here's another example:

Binary Number	0	1	1	1	1	1	0	0
Decimal equivalent	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Decimal equivalent	128	64	32	16	8	4	2	1
Decimal value	0	+ 64 +	32 +	16 +	8 +	4 +	0 +	0 = 124

Note:

- You can use this technique on 16 or 32-bit words too, just work your way up. Eg: After 128, you'd write 256, then 512, 1024 and so on.

- You can tell if the decimal equivalent will be odd or even by the first bit. Eg: In the above example, the first bit = 0, so the number is EVEN. In the first example, the first bit is 1, so the number is ODD.

FUN FACT: In case you didn't already know, bit stands for Binary digIT. :)

CONVERTING DECIMAL TO BINARY:

This is probably easier than base-2 to base-10. To find out what 321 would be in binary, you'd do the following:

321	=	256 X 1
321 - 256 = 65	=	128 X 0
65	=	64 X 1
65 - 64 = 1	=	32 X 0
1	=	16 X 0
1	=	8 X 0
1	=	4 X 0
1	=	2 X 0

```
    1                      =      1  X  1
```

And you get the binary number - 101000001. Easy huh? Let's just try another one to make sure we know how:

```
    198                    =    128  X 1
    198 - 128 = 70         =     64  X 1
    70  -  64 =  6         =     32  X 0
    6                      =     16  X 0
    6                      =      8  X 0
    6                      =      4  X 1
    6   -   4 =  2         =      2  X 1
    2   -   2 =  0         =      1  X 0
```

And this gives us - 11000110. Note how you can check the first digit to see if you got your conversion right. When I wrote the first example, I noticed I had made a mistake when I checked the first bit. On the first example, I got 0 - not good for an odd number. I realised my mistake and corrected the example.

CONVERTING HEXADECIMAL TO DECIMAL:

Before we begin, you should know that the hexadecimal number system uses the 'digits':

```
   0         =  0 (decimal)  =     0 (binary)
   1         =  1 (decimal)  =     1 (binary)
   2         =  2 (decimal)  =    10 (binary)
   3         =  3 (decimal)  =    11 (binary)
   4         =  4 (decimal)  =   100 (binary)
   5         =  5 (decimal)  =   101 (binary)
   6         =  6 (decimal)  =   110 (binary)
   7         =  7 (decimal)  =   111 (binary)
   8         =  8 (decimal)  =  1000 (binary)
   9         =  9 (decimal)  =  1001 (binary)
   A         = 10 (decimal)  =  1010 (binary)
   B         = 11 (decimal)  =  1011 (binary)
   C         = 12 (decimal)  =  1100 (binary)
   D         = 13 (decimal)  =  1101 (binary)
   E         = 14 (decimal)  =  1110 (binary)
   F         = 15 (decimal)  =  1111 (binary)
```

You'll commonly hear hexadecimal referred to as hex, or base-16 and it is commonly denoted by an 'h' - eg 4C00h, or a '$', eg - $B800.

Working with hexadecimal is not as hard as it may look, and converting back and forth is pretty easy. As an example, we'll convert B800h to decimal:

FUN FACT: B800h is the starting address of the video in text mode for CGA and above display adaptors. :)

```
          B    = 4096 x B = 4096 x 11 = 45056
          8    =  256 x 8 =  256 x  8 =  2048
          0    =   16 x 0 =   16 x  0 =     0
          0    =    1 x 0 =    1 x  0 =     0

          So B800h = 45056 + 2048 + 0 + 0
                   = 47104
```

Note: For hexadecimal numbers greater than FFFFh (65535 decimal), you merely follow the same procedure as for binary, so for the fifth hexadecimal digit, you'd multiply it by 65535.

Hit 16 X X on your calculator, and keep hitting =. You'll see the numbers you'd need to use. The same applies for binary. Eg: 2 X X and = would give you 1, 2, 4, 8, 16... etc.

Okay, that seemed pretty easy. I don't even think we need a second example. Let's have a crack at:

CONVERTING DECIMAL TO HEXADECIMAL:

Again, the same sort of procedure as the one we followed for binary. So convert 32753 to hexadecimal, you'd do:

```
32753 / 4096                  =  7 (decimal) = 7h

32753 - (4096 x 7) = 4081

4081 /  256                   = 15 (decimal) = Fh

4081 - (256 x 15)  =  241

241 / 16                      = 15 (decimal) = Fh

241 - (16 x 15)    = 1

1 / 1                         =  1 (decimal) = 1h
```

So eventually we get 7FF1h as our answer. This is not a particularly nice process and requires some explanation.

1) When you divide 32753 by 4096 you get 7.9963379... We are not interested in the .9963379 rubbish, we just take the 7, as 7 is the highest whole number that we can use.

2) The remainder left over from the above operation is 4081. We must now perform the same operation on this, except with 256. Dividing 4081 by 256 gives us 15.941406... Again, we just take the 15.

3) Now we have a remainder of 241. Dividing this by 16 gives us 15.0625. We take the 15, and calculate the remainder.

4) Our last remainder just happens to be one. Dividing this by one gives us, you guessed it - one. YOU SHOULD NOT GET AN ANSWER TO SEVERAL DECIMAL PLACES HERE. IF YOU HAVE - YOU HAVE DONE THE CALCULATION WRONG.

It's a particularly nasty process, but it works. I do not use this except when I have to - I'm not crazy - I use a scientific calculator, or Windows Calculator <shudder> if I must.

Okay, now we've dealt with the gruesome calculations, you're ready for shifts. There are generally two forms of the shift instruction - SHL (shift left) and SHR (shift right). Basically, all these instructions do is shift and expression to the left or right by a number of bits. Their main advantage is their ability to let you replace slow multiplications with much faster shifts. You will find this will speed up pixel/line/circle algorithms by an amazing amount.

PC's are becoming faster and faster by the day - a little too fast for my liking. Back in the days of the XT - multiplication was _really_ slow - perhaps taking up to four seconds for certain operations. Not so much of this applies today, but it is still a good idea to optimize your code.

When we plot a pixel onto the screen, we have to find the offset for the pixel to plot. Basically, what we do is to multiply the Y location by 320, add the X location onto it, and add this to address A000h.

So basically, we get: A000:Yx320+X

Now, as fast as your wonderful 486 or Pentium machine is, this could be made a lot faster. Lets rewrite that equation above so we use some different numbers:

$$\text{Offset} = Y \times 2^{8} + Y \times 2^{6} + X$$

Or:

$$\text{Offset} = Y \times 256 + y \times 64 + X$$

Recognise those numbers? They look an awful lot like the ones we saw in that binary-to-decimal conversion table. However, we are still using multiplication. How can we incorporate shifts into the picture?

What about:

```
Offset = Y SHL 8 + Y SHL 6 + X
```

Now this is a _lot_ faster, as all the computer has to do is shift the number left - much better. Note that shifting to the left INCREASES the number, and shifting to the right will DECREASE the number.

Here's an example that may help you if you are still unsure as to what is going on. Let's say that we're working in base-10 - decimal. Now let's take the number 36 as an example. Shifting this number LEFT by 1, gives us:

```
  36  +  36                                                  = 72
```

Now SHL 2:

```
  36  +  36  +  36  +  36                                    = 144
```

And SHL 3:

```
  36 +  36   +  36  +  36  +  36  +  36  +  36  +  36   = 288
```

Notice the numbers forming? There were 2 36's with SHL 1, 4 36's with SHL 2 and 8 36's with SHL 3. Following this pattern, it would be fair to assume that 36 SHL 4 will equal 36 x 16.

Note however, what is really happening. If you were to work out the binary value of 36, which looks like this: 100100, and then shifted 36 LEFT by two, you'd get 144, or 10010000. All the CPU actually does it stick a few extra 1's and 0's in a location in memory.

As another example, take the binary number 1000101. If we were to shift it LEFT 3, we'd end up with:

```
        1 0 0 0 1 0 1
          <---------- SHL 3
  1 0 0 0 1 0 1 0 0 0
```

Now lets shift the number 45 RIGHT 2. In binary this is 101101. Hence:

```
        1 0 1 1 0 1
        SHR 2 ---->
            1 0 1 1
```

Notice what has occurred? It is much easier for the CPU to just move some bits around, (approximately 2 clock ticks), rather than to multiply a number out. (Can get to around 133 clock ticks).

We will be using shifts a lot when programming the VGA, so make sure you understand the concepts behind them.

PROGRAMMING THE VGA IN ASSEMBLER

I have received quite a bit of mail asking me to cover the VGA. So for all those who asked, we'll be spending most of our time, but not all, on programming the VGA. After all, doesn't everyone want to code graphics?

When we talk about programming the VGA, we are generally talking about mode 13h, or one of its tweaked relatives. In standard VGA this is the _only_ way to use 256 colors, and it's probably one of the easiest modes to use too. If you've ever tried experimenting with SVGA, you'll understand the nightmare it is for the programmer in supporting all the different SVGA cards that exist - except if you use VESA that is, which we'll discuss another time. The great thing about standard mode 13h is you know that just about every VGA card in existence will support it. People today often ignore mode 13h, thinking the resolution to be too grainy by today's standards, but don't forget that Duke Nukem, DOOM, DOOM II, Halloween Harry and most of the Apogee games use this mode to achieve some great effects.

The great thing about mode 13h - that's 320x200x256 in case you were unaware, is that accessing VGA RAM is incredibly easy. As 320 x 200 only equals 64,000, it is quite possible to fit the entire screen into one 64K segment - leaving out the hell of planes, (or should that be plains of Hell?), and masking registers.

The bad news is that standard mode 13h really only gives you one page to use, seriously hampering scrolling and page-flipping. We'll later cover how to get into your own modes - and mode X which will avoid these problems.

So, how do you get into the standard mode 13h?

The answer is simple. We use interrupt 10h - video interrupt, and call subfunction 00h - set mode. In Pascal, you could declare a procedure like this:

```
Procedure Init300x200;   Assembler;

Asm     { Init300x200 }
   mov   ah, 00h         { Set video mode }
   mov   al, 13h         { Use mode 13h   }
   int   10h             { Do it          }
End;    { Init300x200 }
```

You may also see:

```
   mov   ax, 13h
   int   10h
```

This is perfectly correct, and probably saves one clock tick by not putting 00h in AH and then 13h in AL, but it is more correct to use the first example.

Okay, so we're in mode 13h, but what can we actually do in it, other than look at a blank screen? We could go back to text mode by using:

```
   mov   ah, 00h
   mov   al, 03h
   int   10h
```

but that's a little dull. Why not plot a pixel?

There are a number of ways you could get a pixel on the screen. The easiest way in Assembler is to use interrupts. You do it like this in Pascal:

```
Procedure PutPixel(X, Y : Integer; Color : Byte);   Assembler;

Asm     { PutPixel }
   mov   ah, 0Ch        { Draw pixel subfunction        }
   mov   al, [Color]    { Move the color to plot in AL  }
   mov   cx, [X]        { Move the X value into CX      }
   mov   dx, [Y]        { Move the Y value into DX      }
   mov   bx, 1h         { BX = 1, page 1                }
   int   10h            { Plot it                       }
End;    { PutPixel }
```

However, even though this is in Assembler, it isn't particularly speedy. Why you ask? Because it uses interrupts. Interrupts are fine for getting in and out of video modes, turning the cursor on and off, etc... but not for graphics.

You can think of interrupts like an answering machine. "The CPU is busy right now, but if you leave your subfunction after the tone - we'll get back to you."

Not good. Let's use that technique we discussed earlier during shifts. What we want to do is put the value of the color we want to plot into the VGA directly. To do this, we'll need to move the address of the VGA into ES, and calculate the offset of the pixel we want to plot. An example of this is shown below:

```
Procedure PutPixel(X, Y : Integer; Color : Byte);   Assembler;

Asm     { PutPixel }
   mov   ax, 0A000h     { Move the segment of the VGA into AX,   }
   mov   es, ax         { and now into ES                        }
   mov   bx, [X]        { Move the X value into BX               }
   mov   dx, [Y]        { Move the Y value into DX               }
   mov   di, bx         { Move X into DI                         }
   mov   bx, dx         { Move Y into BX                         }
   shl   dx, 8          { In this part we use shifts to multiply }
   shl   bx, 6          { Y by 320                               }
   add   dx, bx         { Now here we add X onto the above,      }
   add   di, dx         { giving us DI = Y x 320 + X             }
   mov   al, [Color]    { Put the color to plot into AL          }
   stosb                { Put the byte, AL, at ES:DI             }
End;    { PutPixel }
```

This procedure is fast enough to begin with, though I gave out a much faster one a few tutorials ago which uses a pretty ingenious technique to get DI.

Okay, I think that's enough for this week. Have a play with the PutPixel routines and see what you can do with them. For those with a "Peter Norton's Guide to..." book, see what other procedures you can make using interrupts.

THINGS TO DO:

1) We covered a lot in this tutorial, and some important concepts were in it. Make sure you are comfortable with the comparisons, because we'll get into testing bits soon.

2) Make sure you understand the binary -> decimal, decimal -> binary, decimal -> hex and hex -> decimal stuff. Make yourself some example sums and test your answers with Windows Calculator.

3) You _must_ understand shifts. If you are still having problems, make some expressions up on paper, and test your answers with a program such as:

```
Begin   { Main }
   WriteLn(45 SHL 6);
   ReadLn;
End.    { Main }
```

   and/or Windows Calculator.

4) Have a look at the VGA stuff, and make sure you have grasped the theory behind it, because next week we're really going to go into it in depth.

Next week I'll also try to give some C/C++ examples as well as the Pascal ones for all you C programmers out there.

Next tutorial will cover:

- How the VGA is arranged
- How we can draw lines and circles
- Getting and setting the palette in Assembler
- Fades
- Some C/C++ examples

If you wish to see a topic discussed in a future tutorial, then mail me, and I'll see what I can do.

Don't miss out!!! Download next week's tutorial from my homepage at:

- http://www.faroc.com.au/~blackcat

See you next week!

- Adam.

" I _never_ write code with bugs, I just add some unintentional features! "

Adam's Assembler Tutorial 1.0

PART IV

Revision : 1.5
Date : 01-03-1996
Contact : blackcat@faroc.com.au
http://www.faroc.com.au/~blackcat

Welcome back, budding Assembler coders. The tutorials seem to be getting popular now, and I've had some mail requesting me to cover the VGA so I'll give it a go. This is basically what I've been leading up to in my own disjointed way anyhow, as graphics programming is not only rewarding, it's also fun too! Well, I think it's fun. :)

Firstly though, we must finish off the CMP/JMP stuff, and cover shifts. When you're coding in Assembler, you'll find comparisons, shifts and testing bits are very common operations.

A Comparison Example

I won't bother going over the following example - it's fairly easy to understand and you should get the basic idea anyway.

```
   DOSSEG
   .MODEL SMALL
   .STACK 200h
   .DATA

FirstString    DB  13, 10, "Is this a great tutorial or what? :) - $"
SecondString   DB  13, 10, "NO? NO? What do you mean, NO?$"
ThirdString    DB  13, 10, "Excellent, let's hear you say that again.$"
FourthString   DB  13, 10, "Just a Y or N will do.$"
ExitString     DB  13, 10, "Fine, be like that!$"

   .CODE

START:
   MOV   AX, @DATA                  ; New way of saying:
   MOV   DS, AX                     ; DS -> SEG data segment

KeepOnGoing:
   MOV   AH, 9
   MOV   DX, OFFSET FirstString     ; DX -> OFFSET FirstString
   INT   21h                        ; Output the first message

   MOV   AH, 0                      ; Get a key - store it in AX
   INT   16h                        ; AL - ASCII code, AH - scan code
                                    ; It doesn't echo onto the screen
                                    ; though, we have to do that ourselves

   PUSH  AX                         ; Here we display the char - note that
   MOV   DL, AL                     ; we save AX.  Obviously, using AH to
   MOV   AH, 2                      ; signal to print a string destroys AX
   INT   21h
   POP   AX

   CMP   AL, "Y"                    ; Check to see if 'Y' was pressed
   JNE   HatesTute                  ; If it was, keep going

   MOV   AH, 9                      ; Display the "Excellent..." message
   MOV   DX, OFFSET ThirdString
   INT   21h
   JMP   KeepOnGoing                ; Go back to the start and begin again

HatesTute:
   CMP   AL, "N"                    ; Make sure it was 'N' they pressed
   JE    DontLikeYou                ; Sadly, it was equal

   MOV   DX, OFFSET FourthString    ; Ask the user to try again
   MOV   AH, 9
   INT   21h
   JMP   KeepOnGoing                ; Let 'em try
```

```
DontLikeYou:
   MOV   DX, OFFSET SecondString     ; Show the "NO? NO? What..." string
   MOV   AH, 9
   INT   21h

   MOV   DX, OFFSET ExitString       ; Show the "Fine, be like that!" string
   MOV   AH, 9
   INT   21h

   MOV   AX, 4C00h                   ; Return to DOS
   INT   21h
END START
```

You should understand this example, play around with it and write something better. Those with a "Peter Norton's Guide to..." book or similar, experiment with the keyboard subfunctions, and see what other similar GetKey combinations exist, or better still, play around with interrupt 10h and go into some weird video mode - one which your PC supports! - and use some color.

Shifts

A simple concept, and one which I should have discussed before, but like I said - I have my own disjointed way of going about things.

First you'll need to understand some hexadecimal and binary arithmetic - a subject I _should_ have covered before. I usually use a scientific calculator - hey, I always use a calculator, I'm not stupid! - but it is good to be able to know how to multiply, add and convert between the various bases.

You also cannot use a calculator in Computing exams, not in Australia anyway.

CONVERTING BINARY TO DECIMAL:

Way back in Tutorial One we looked at what binary numbers look like, so imagine I have an eight-bit binary number such as:

11001101

What is this in decimal??? There are a number of ways to convert such a number, and I use the following, which I believe is probably the easiest:

Binary Number	1	1	0	0	1	1	0	1	
Decimal equivalent	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
Decimal equivalent	128	64	32	16	8	4	2	1	
Decimal value	128 +	64 +	0 +	0 +	8 +	4 +	0 +	1	= 205

Get the idea? Note for the last line, it would be more accurate to write:

```
   1 x 128 + 1 x 64 + 0 x 32 + 0 x 16 + 1 x 8 + 1 x 4 + 0 x 2 + 1 x 1
 =     128 +     64 +      0 +      0 +     8 +     4 +     0 +     1
 = 205
```

Sorry if this is a little confusing, but it is difficult to explain without demonstrating. Here's another example:

Binary Number	0	1	1	1	1	1	0	0
Decimal equivalent	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Decimal equivalent	128	64	32	16	8	4	2	1
Decimal value	0 +	64 +	32 +	16 +	8 +	4 +	0 +	0 = 124

Note:

- You can use this technique on 16 or 32-bit words too, just work your way up. Eg: After 128, you'd write 256, then 512, 1024 and so on.

- You can tell if the decimal equivalent will be odd or even by the first bit. Eg: In the above example, the first bit = 0, so the number is EVEN. In the first example, the first bit is 1, so the number is ODD.

FUN FACT: In case you didn't already know, bit stands for Binary digIT. :)

CONVERTING DECIMAL TO BINARY:

This is probably easier than base-2 to base-10. To find out what 321 would be in binary, you'd do the following:

```
    321                   =    256  X  1
    321 - 256 = 65        =    128  X  0
    65                    =     64  X  1
    65  -  64 = 1         =     32  X  0
    1                     =     16  X  0
    1                     =      8  X  0
    1                     =      4  X  0
    1                     =      2  X  0
    1                     =      1  X  1
```

And you get the binary number - 101000001. Easy huh? Let's just try another one to make sure we know how:

```
    198                   =    128  X 1
    198 - 128 = 70        =     64  X 1
    70  -  64 =  6        =     32  X 0
    6                     =     16  X 0
    6                     =      8  X 0
    6                     =      4  X 1
    6   -   4 =  2        =      2  X 1
    2   -   2 =  0        =      1  X 0
```

And this gives us - 11000110. Note how you can check the first digit to see if you got your conversion right. When I wrote the first example, I noticed I had made a mistake when I checked the first bit. On the first example, I got 0 - not good for an odd number. I realised my mistake and corrected the example.

CONVERTING HEXADECIMAL TO DECIMAL:

Before we begin, you should know that the hexadecimal number system uses the 'digits':

```
   0         =  0 (decimal)  =     0 (binary)
   1         =  1 (decimal)  =     1 (binary)
   2         =  2 (decimal)  =    10 (binary)
   3         =  3 (decimal)  =    11 (binary)
   4         =  4 (decimal)  =   100 (binary)
   5         =  5 (decimal)  =   101 (binary)
   6         =  6 (decimal)  =   110 (binary)
   7         =  7 (decimal)  =   111 (binary)
   8         =  8 (decimal)  =  1000 (binary)
   9         =  9 (decimal)  =  1001 (binary)
   A         = 10 (decimal)  =  1010 (binary)
   B         = 11 (decimal)  =  1011 (binary)
   C         = 12 (decimal)  =  1100 (binary)
   D         = 13 (decimal)  =  1101 (binary)
   E         = 14 (decimal)  =  1110 (binary)
   F         = 15 (decimal)  =  1111 (binary)
```

You'll commonly hear hexadecimal referred to as hex, or base-16 and it is commonly denoted by an 'h' - eg 4C00h, or a '$', eg - $B800.

Working with hexadecimal is not as hard as it may look, and converting back and forth is pretty easy. As an example, we'll convert B800h to decimal:

FUN FACT: B800h is the starting address of the video in text mode for CGA and above display adaptors. :)

```
          B    = 4096 x B = 4096 x 11 = 45056
          8    =  256 x 8 =  256 x  8 =  2048
          0    =   16 x 0 =   16 x  0 =     0
          0    =    1 x 0 =    1 x  0 =     0

          So B800h = 45056 + 2048 + 0 + 0
                   = 47104
```

Note: For hexadecimal numbers greater than FFFFh (65535 decimal), you merely follow the same procedure as for binary, so for the fifth hexadecimal digit, you'd multiply it by 65535.

Hit 16 X X on your calculator, and keep hitting =. You'll see the numbers you'd need to use. The same applies for binary. Eg: 2 X X and = would give you 1, 2, 4, 8, 16... etc.

Okay, that seemed pretty easy. I don't even think we need a second example. Let's have a crack at:

CONVERTING DECIMAL TO HEXADECIMAL:

Again, the same sort of procedure as the one we followed for binary. So convert 32753 to hexadecimal, you'd do:

```
32753 / 4096                  =  7 (decimal) = 7h

32753 - (4096 x 7) = 4081

4081 /  256                   = 15 (decimal) = Fh

4081 - (256 x 15)  =  241

241 / 16                      = 15 (decimal) = Fh

241 - (16 x 15)    = 1

1 / 1                         =  1 (decimal) = 1h
```

So eventually we get 7FF1h as our answer. This is not a particularly nice process and requires some explanation.

1) When you divide 32753 by 4096 you get 7.9963379... We are not interested in the .9963379 rubbish, we just take the 7, as 7 is the highest whole number that we can use.

2) The remainder left over from the above operation is 4081. We must now perform the same operation on this, except with 256. Dividing 4081 by 256 gives us 15.941406... Again, we just take the 15.

3) Now we have a remainder of 241. Dividing this by 16 gives us 15.0625. We take the 15, and calculate the remainder.

4) Our last remainder just happens to be one. Dividing this by one gives us, you guessed it - one. YOU SHOULD NOT GET AN ANSWER TO SEVERAL DECIMAL PLACES HERE. IF YOU HAVE - YOU HAVE DONE THE CALCULATION WRONG.

It's a particularly nasty process, but it works. I do not use this except when I have to - I'm not crazy - I use a scientific calculator, or Windows Calculator <shudder> if I must.

Okay, now we've dealt with the gruesome calculations, you're ready for shifts. There are generally two forms of the shift instruction - SHL (shift left) and SHR (shift right). Basically, all these instructions do is shift and expression to the left or right by a number of bits. Their main advantage is their ability to let you replace slow multiplications with much faster shifts. You will find this will speed up pixel/line/circle algorithms by an amazing amount.

PC's are becoming faster and faster by the day - a little too fast for my liking. Back in the days of the XT - multiplication was _really_ slow - perhaps taking up to four seconds for certain operations. Not so much of this applies today, but it is still a good idea to optimize your code.

When we plot a pixel onto the screen, we have to find the offset for the pixel to plot. Basically, what we do is to multiply the Y location by 320, add the X location onto it, and add this to address A000h.

So basically, we get: A000:Yx320+X

Now, as fast as your wonderful 486 or Pentium machine is, this could be made a lot faster. Lets rewrite that equation above so we use some different numbers:

$$\text{Offset} = Y \times 2^8 + Y \times 2^6 + X$$

Or:

$$\text{Offset} = Y \times 256 + y \times 64 + X$$

Recognise those numbers? They look an awful lot like the ones we saw in that binary-to-decimal conversion table. However, we are still using multiplication. How can we incorporate shifts into the picture?

What about:

```
Offset = Y SHL 8 + Y SHL 6 + X
```

Now this is a _lot_ faster, as all the computer has to do is shift the number left - much better. Note that shifting to the left INCREASES the number, and shifting to the right will DECREASE the number.

Here's an example that may help you if you are still unsure as to what is going on. Let's say that we're working in base-10 - decimal. Now let's take the number 36 as an example. Shifting this number LEFT by 1, gives us:

```
  36  +  36                                                 = 72
```

Now SHL 2:

```
  36  +  36  +  36  +  36                                   = 144
```

And SHL 3:

```
  36 +  36   +  36  +  36  +  36  +  36  +  36  +  36   = 288
```

Notice the numbers forming? There were 2 36's with SHL 1, 4 36's with SHL 2 and 8 36's with SHL 3. Following this pattern, it would be fair to assume that 36 SHL 4 will equal 36 x 16.

Note however, what is really happening. If you were to work out the binary value of 36, which looks like this: 100100, and then shifted 36 LEFT by two, you'd get 144, or 10010000. All the CPU actually does it stick a few extra 1's and 0's in a location in memory.

As another example, take the binary number 1000101. If we were to shift it LEFT 3, we'd end up with:

```
        1 0 0 0 1 0 1
          <---------- SHL 3
  1 0 0 0 1 0 1 0 0 0
```

Now lets shift the number 45 RIGHT 2. In binary this is 101101. Hence:

```
        1 0 1 1 0 1
        SHR 2 ---->
            1 0 1 1
```

Notice what has occurred? It is much easier for the CPU to just move some bits around, (approximately 2 clock ticks), rather than to multiply a number out. (Can get to around 133 clock ticks).

We will be using shifts a lot when programming the VGA, so make sure you understand the concepts behind them.

PROGRAMMING THE VGA IN ASSEMBLER

I have received quite a bit of mail asking me to cover the VGA. So for all those who asked, we'll be spending most of our time, but not all, on programming the VGA. After all, doesn't everyone want to code graphics?

When we talk about programming the VGA, we are generally talking about mode 13h, or one of its tweaked relatives. In standard VGA this is the _only_ way to use 256 colors, and it's probably one of the easiest modes to use too. If you've ever tried experimenting with SVGA, you'll understand the nightmare it is for the programmer in supporting all the different SVGA cards that exist - except if you use VESA that is, which we'll discuss another time. The great thing about standard mode 13h is you know that just about every VGA card in existence will support it. People today often ignore mode 13h, thinking the resolution to be too grainy by today's standards, but don't forget that Duke Nukem, DOOM, DOOM II, Halloween Harry and most of the Apogee games use this mode to achieve some great effects.

The great thing about mode 13h - that's 320x200x256 in case you were unaware, is that accessing VGA RAM is incredibly easy. As 320 x 200 only equals 64,000, it is quite possible to fit the entire screen into one 64K segment - leaving out the hell of planes, (or should that be plains of Hell?), and masking registers.

The bad news is that standard mode 13h really only gives you one page to use, seriously hampering scrolling and page-flipping. We'll later cover how to get into your own modes - and mode X which will avoid these problems.

So, how do you get into the standard mode 13h?

The answer is simple. We use interrupt 10h - video interrupt, and call subfunction 00h - set mode. In Pascal, you could declare a procedure like this:

```
Procedure Init300x200;   Assembler;

Asm     { Init300x200 }
   mov   ah, 00h         { Set video mode }
   mov   al, 13h         { Use mode 13h   }
   int   10h             { Do it          }
End;    { Init300x200 }
```

You may also see:

```
   mov   ax, 13h
```

```
   int   10h
```

This is perfectly correct, and probably saves one clock tick by not putting 00h in AH and then 13h in AL, but it is more correct to use the first example.

Okay, so we're in mode 13h, but what can we actually do in it, other than look at a blank screen? We could go back to text mode by using:

```
   mov   ah, 00h
   mov   al, 03h
   int   10h
```

but that's a little dull. Why not plot a pixel?

There are a number of ways you could get a pixel on the screen. The easiest way in Assembler is to use interrupts. You do it like this in Pascal:

```
Procedure PutPixel(X, Y : Integer; Color : Byte);   Assembler;

Asm     { PutPixel }
   mov   ah, 0Ch        { Draw pixel subfunction        }
   mov   al, [Color]    { Move the color to plot in AL  }
   mov   cx, [X]        { Move the X value into CX      }
   mov   dx, [Y]        { Move the Y value into DX      }
   mov   bx, 1h         { BX = 1, page 1                }
   int   10h            { Plot it                       }
End;    { PutPixel }
```

However, even though this is in Assembler, it isn't particularly speedy. Why you ask? Because it uses interrupts. Interrupts are fine for getting in and out of video modes, turning the cursor on and off, etc... but not for graphics.

You can think of interrupts like an answering machine. "The CPU is busy right now, but if you leave your subfunction after the tone - we'll get back to you."

Not good. Let's use that technique we discussed earlier during shifts. What we want to do is put the value of the color we want to plot into the VGA directly. To do this, we'll need to move the address of the VGA into ES, and calculate the offset of the pixel we want to plot. An example of this is shown below:

```
Procedure PutPixel(X, Y : Integer; Color : Byte);   Assembler;

Asm     { PutPixel }
   mov   ax, 0A000h     { Move the segment of the VGA into AX,   }
   mov   es, ax         { and now into ES                        }
   mov   bx, [X]        { Move the X value into BX               }
   mov   dx, [Y]        { Move the Y value into DX               }
   mov   di, bx         { Move X into DI                         }
   mov   bx, dx         { Move Y into BX                         }
   shl   dx, 8          { In this part we use shifts to multiply }
   shl   bx, 6          { Y by 320                               }
```

```
   add   dx, bx         { Now here we add X onto the above,      }
   add   di, dx         { giving us DI = Y x 320 + X             }
   mov   al, [Color]    { Put the color to plot into AL          }
   stosb                { Put the byte, AL, at ES:DI             }
End;    { PutPixel }
```

This procedure is fast enough to begin with, though I gave out a much faster one a few tutorials ago which uses a pretty ingenious technique to get DI.

Okay, I think that's enough for this week. Have a play with the PutPixel routines and see what you can do with them. For those with a "Peter Norton's Guide to..." book, see what other procedures you can make using interrupts.

THINGS TO DO:

1) We covered a lot in this tutorial, and some important concepts were in it. Make sure you are comfortable with the comparisons, because we'll get into testing bits soon.

2) Make sure you understand the binary -> decimal, decimal -> binary, decimal -> hex and hex -> decimal stuff. Make yourself some example sums and test your answers with Windows Calculator.

3) You _must_ understand shifts. If you are still having problems, make some expressions up on paper, and test your answers with a program such as:

   ```
   Begin   { Main }
      WriteLn(45 SHL 6);
      ReadLn;
   End.    { Main }
   ```

   and/or Windows Calculator.

4) Have a look at the VGA stuff, and make sure you have grasped the theory behind it, because next week we're really going to go into it in depth.

Next week I'll also try to give some C/C++ examples as well as the Pascal ones for all you C programmers out there.

Next tutorial will cover:

- How the VGA is arranged
- How we can draw lines and circles
- Getting and setting the palette in Assembler
- Fades
- Some C/C++ examples

If you wish to see a topic discussed in a future tutorial, then mail me, and I'll see what I can do.

Don't miss out!!! Download next week's tutorial from my homepage at:

- http://www.faroc.com.au/~blackcat

See you next week!

- Adam.

" I _never_ write code with bugs, I just add some unintentional features! "

Adam's Assembler Tutorial 1.0

PART V

Revision : 1.5
Date : 15-03-1996
Contact : blackcat@faroc.com.au
http://www.faroc.com.au/~blackcat

Well, another week or so seems to have gone by... Another week I should have been using to accomplish something useful. Anyway, it seems that the tutorials have gained a bit more popularity, which is good.

I've also received some demo code from someone who seems to have found the tutorials of some use. Please, if you attempt something either with the help of the tutorials or on your own, please send it to me. I like to see what people have made of my work, or just how creative you all are. If you write something that I think could be useful for others to learn from, or is just pretty cool, I'll stick it up on my web site.

Note that I included a starfield demonstration in this week's tutorial just for the hell of it. You can run STARS.EXE, or look at STARS.PAS for the full source. It's only a simple demo, but it can be used to achieve some very nice effects.

Now, this week we're firstly going to list a summary of all the instructions that you should have learnt by now, and a few new ones as well. Then we'll take a look at how the VGA is arranged, and cover a simple line routine.

THE INSTRUCTION SET SUMMARY

■ ADC <DEST>, <SOURCE> - Name: Add with Carry

Type: 8086+

Description: This instruction adds <SOURCE> to <DEST> and adds the value stored in the carry flag, which will be a one or a zero to <DEST> also.

Basically, DEST = DEST + SOURCE + CF

EG: ADC AX, BX

■ ADD <DEST>, <SOURCE> - Name: Add

Type: 8086+

Description: This instruction adds <SOURCE> and <DEST>, storing the result in <DEST>.

EG: ADD AX, BX

■ AND <DEST>, <SOURCE> - Name: Boolean AND

Type: 8086+

Description: This instruction performs a bit by bit comparison of <DEST> and <SOURCE>, storing the result in <DEST>.

```
EG: AND 0, 0     = 0
    AND 0, 1     = 0
    AND 1, 0     = 0
    AND 1, 1     = 1
```

■ BT <DEST>, <BIT NUMBER> - Name: Bit Test

Type: 80386+

Description: This instruction tests <BIT NUMBER> of <DEST> which can either be a 16 or 32-bit register or memory location. If <DEST> is a 16-bit number then <BIT NUMBER> can range from 0 - 15, else if <DEST> is a 32-bit number, then <BIT NUMBER> may have a value from 0 to 31.

The value held in <BIT NUMBER> of <DEST> is then copied into the carry flag.

```
EG: BT   AX, 3
    JC   WasEqualToOne
```

■ CALL <DEST> - Name: Procedure Call

Type: 8086+

Description: This instruction simply calls a subroutine. In more technical terms, it pushes the address of the next instruction, IP, onto the stack, and then sets the instruction pointer, IP, to the value specified by <DEST>.

EG: CALL MyProc

- CBW

- Name: Convert Byte to Word
  Type: 8086+

  Description: This instruction extends the byte in AL to AX.

```
EG: MOV   AL, 01h
    CBW
    ADD   BX, AX   ; Do something with AX
```

- CLC

- Name: Clear Carry Flag
  Type: 8086+

  Description: This instruction clears the carry flag in the flags register to 0.

  EG: CLC

- CLD

- Name: Clear Direction Flag
  Type: 8086+

  Description: This instruction clears the direction flag in the flags register to 0. When the direction flag is 0, any string instructions increment the index registers SI and DI.

  EG: CLD

- CLI

- Name: Clear Interrupt Flag
  Type: 8086+

  Description: This instruction clears the interrupt flag in the flags register to 0, thus disabling hardware interrupts.

  EG: CLI

- CMC

- Name: Complement the Carry Flag
  Type: 8086+

  Description: This instruction checks the value currently held in the carry flag. If it is 0 - it becomes a 1 and if it is 1 - it becomes a 0.

```
EG: BT   AX, 1    ; Test bit 1 of AX
    JC   WasOne
    JMP  Done

    WasOne:
    CMC           ; Return CF to 0

    Done:
```

■ CMP <VALUE1>, <VALUE2> - Name: Compare Integer
Type: 8086+

Description: This instruction compares <VALUE1> and <VALUE2> and reflects the comparison in the flags.

EG: CMP AX, BX

See also the Jcc instructions.

■ CWD - Name: Convert Word to Doubleword
Type: 8086+

Description: This instruction extends the word in AX to the DX:AX pair.

EG: CWD

■ DEC <VALUE> - Name: Decrement
Type: 8086+

Description: This instruction subtracts one from the value held in <VALUE> and stores the result in <VALUE>.

EG: DEC AX

■ DIV <VALUE> - Name: Unsigned Division
Type: 8086+

Description: This instruction divides <VALUE> by either AX for a byte, DX:AX for a word or EDX:EAX for a doubleword.

For a byte, the quotient is returned in AL and the remainder in AH, for a word the quotient is returned in AX and the remainder in DX and for a DWORD, the quotient is returned in EAX and the remainder in EDX.

```
EG: MOV   AX, 12
    MOV   BH, 5
    DIV   BH
    MOV   Quotient, AL
```

```
MOV   Remainder, AH
```

■ IN <ACCUMULATOR>, <PORT> - Name: Input from I/O port
Type: 8086+

Description: This instruction reads a value from one of the 65536 hardware ports into the specified accumulator.

AX and AL are commonly used for input ports, and DX is commonly used to identify the port.

```
EG: IN    AX, 72h

    MOV   DX, 3C7h
    IN    AL, DX
```

■ INC <VALUE> - Name: Increment
Type: 8086+

Description: This instruction adds one to the number held in <VALUE>, and stores the result in <VALUE>.

```
EG: MOV   AX, 13h   ; AX = 13h
    INC   AX        ; AX = 14h
```

■ INT <INTERRUPT> - Name: Generate an Interrupt
Type: 8086+

Description: This instruction saves the current flags and instruction pointer on the stack, and then calls <INTERRUPT> based on the value in AH.

```
EG:   MOV   AH, 00h   ; Set video mode
      MOV   AL, 13h   ; Video mode 13h
      INT   10h       ; Generate interrupt
```

■ Jcc - Name: Jump if Condition
Type: 8086+

I'm not going to repeat myself for all 32 of them, just look in Tutorial Three for the entire list of them. Bear in mind that it would be a good idea to call CMP, OR, DEC or something similar before you use one of these instructions. :)

```
EG: DEC   AX
    JZ    AX_Has_Reached_Zero
```

■ JMP <DEST> - Name: Jump
Type: 8086+

Description: This instruction simply

loads a new value, <DEST>, into the instruction pointer, thus transferring control to another part of the code.

EG: JMP   MyLabel

■ LAHF - Name: Load AH with Flags
Type: 8086+

Description: This instruction copies the low bytes of the flags register into AH. The contents of AH will look something like the following after the instruction has been executed:

Flag	SF	ZF	--	AF	--	PF	--	CF
Bit	07	06	05	04	03	02	01	00

You may now test the bits individually, or perform an instruction similar to the follow to get an individual flag:

```
EG: LAHF
    SHR   AH, 6
    AND   AH, 1   ; AH now contains the ZF flag.
```

■ LEA <DEST>, <SOURCE> - Name: Load Effective Address
Type: 8086+

Description: This instruction loads the memory address that <SOURCE> resides in, into <DEST>.

EG: I use   LEA   SI, Str in a procedure of mine which puts a string on the screen very fast.

■ LOOP <LABEL> - Name: Decrement CX and Branch
Type: 8086+

Description: This instruction is a form of the For...Do loop that exists in most high-level languages. Basically it loops back to a label, or memory offset, until CX = 0.

```
EG: MOV   CX, 12

    DoSomeStuff:
       ;...
       ;...
       ;... This will be repeated 12 times

    LOOP DoSomeStuff
```

■ Lseg <DEST>, <SOURCE>

- Name: Load Segment Register
  Type: 8086+

  Description: This instruction exists in several forms. All accept the same syntax, in which <SOURCE> specifies a 48-bit pointer, consisting of a 32-bit offset and a 16-bit selector. The 32-bit offset is loaded into <DEST>, and the selector is loaded into the segment register specified by seg.

  The following forms exist:

  ```
  LDS
  LES
  LFS     * 32-bit
  LGS     * 32-bit
  LSS
  ```

  ```
  EG: LES   SI, A_Pointer
  ```

■ MOV <DEST>, <SOURCE>

- Name: Move Data
  Type: 8086+

  Description: This instruction copies <SOURCE> into <DEST>.

  ```
  EG: MOV   AX, 3Eh
      MOV   SI, 12h
  ```

■ MUL <SOURCE>

- Name: Unsigned Multiplication
  Type: 8086+

  Description: This instruction multiplies <SOURCE> by the accumulator, which depends on the size of <SOURCE>.

  If <SOURCE> is a byte then:

  * AL is the multiplicand;
  * AX is the product.

  If <SOURCE> is a word then:

  * AX is the multiplicand;
  * DX:AX is the product.

  If <SOURCE> is a doubleword then:

  * EAX is the multiplicand;
  * EDX:EAX is the product.

  Note: The flags are left in an un-touched state except for OF and CF, which are cleared to 0 if the high byte, word or dword of the product is 0.

```
EG: MOV   AL, 3
    MUL   10
    MOV   Result, AX
```

■ NEG <VALUE> - Name: Negate
Type: 8086+

Description: This instruction subtracts <VALUE> from 0, resulting in a two's complement negation of <VALUE>.

```
EG: MOV   AX, 03h
    NEG   AX        ; AX = -3
```

■ NOT <VALUE> - Name: Boolean Complement
Type: 8086+

Description: This instruction inverts the state of each bit in the operand.

```
EG: NOT   CX
```

■ OR <DEST>, <SOURCE> - Name: Boolean OR
Type: 8086+

Description: This instruction performs a boolean OR operation between each bit of <DEST> and <SOURCE>, storing the result in <DEST>.

```
EG: OR 0, 0     = 0
    OR 0, 1     = 1
    OR 1, 0     = 1
    OR 1, 1     = 1
```

■ OUT <PORT>, <ACCUMULATOR> - Name: Output to Port
Type: 8086+

Description: This instruction outputs the value in the accumulator to <PORT>. Using the DX register to pass the port to OUT, you may access up to 65,536 ports.

```
EG: MOV   DX, 378h
    OUT   DX, AX
```

■ POP <REGISTER> - Name: Pop Register
Type: 8086+

Description: This instruction pops the current value off the stack, and places it into <REGISTER>.

EG: POP   AX

■ POPA — Name: Pop All General Registers
Type: 80186+

Description: This instruction pops all the 16-bit general purpose registers off the stack, except for SP.

It is the same as:

```
POP   AX
POP   BX
POP   CX
...
```

EG: POPA

■ POPF — Name: Pop Stack into Flags
Type: 8086+

Description: This instruction pops the low byte of the flags off the stack.

EG: POPF

■ PUSH <REGISTER> — Name: Push Register
Type: 8086+

Description: This instruction pushes <REGISTER> onto the stack.

EG: PUSH AX

■ PUSHA — Name: Push All General Registers
Type: 80186+

Description: This instruction pushes all 16-bit general purpose registers onto the stack.

It is the same as:

```
PUSH  AX
PUSH  BX
PUSH  CX
...
```

EG: PUSHA

■ PUSHF — Name: Push Flags
Type: 8086+

Description: This instruction pushes the low byte of the flags of the stack.

EG: PUSHF

- REP — Name: Repeat String Prefix
  Type: 8086+

  Description: This instruction will repeat the following instructing for the number of times specified in the CX register.

  EG:
```
MOV   CX, 6
REP   STOSB    ; Store 6 bytes
```

- RET — Name: Near Return from Subroutine
  Type: 8086+

  Description: This instruction returns IP to the value it had held before the last CALL instruction. RET, or RETF for a far jump, must be called when using stand alone assembler.

  EG: RET

- ROL <DEST>, <VALUE> — Name: Rotate Left
  Type: 8086+

  Description: This instruction rotates <DEST> <VALUE> times. A rotation is achieved by shifting <DEST> once, then transferring the bit shifted off the high end to the low-order position of <DEST>.

  EG: ROL   AX, 3

- ROR <DEST>, <VALUE> — Name: Rotate Right
  Type: 8086+

  Description: This instruction rotates <DEST> <VALUE> times. A rotation is achieved by shifting <DEST> once, and transferring the bit shifted off the low end to the high-order position of <DEST>.

  EG: ROR   BX, 5

- SAHF — Name: Store AH in Flags
  Type: 8086+

  Description: This instruction loads the contents of the AH register into bits 7, 6, 4, 2 and 0 of the flags register.

  EG: SAHF

■ SBB <DEST>, <SOURCE> - Name: Subtract with Borrow
Type: 8086+

Description: This instruction subtracts <SOURCE> from <DEST>, and decrements <DEST> by one if the carry flag is set, storing the result in <DEST>.

Basically, <DEST> = <DEST> - <SOURCE> - CF

EG: SBB   AX, BX

■ SHL <DEST>, <VALUE> - Name: Shift Left
Type: 8086+

Description: This instruction shifts <DEST> left by <VALUE>. I'm not going to go into the theory behind shifts again. If you are unsure as to what this instruction does, please refer to Tutorial Four.

EG: SHL   AX, 5

■ SHR <DEST>, <VALUE> - Name: Shift Right
Type: 8086+

Description: This instruction shifts <DEST> right by <VALUE>. Please refer to Tutorial Four for the theory behind shifts.

EG: SHR   DX, 1

■ STC - Name: Set Carry Flag
Type: 8086+

Description: This instruction assigns the value of the carry flag to one.

EG: STC

■ STD - Name: Set Direction Flag
Type: 8086+

Description: This instruction sets the value of the carry flag to one. This instructs all string operations to decrement the index registers.

```
EG: STD
    REP STOSB   ; DI is being decremented
```

■ STI - Name: Set Interrupt Flag
Type: 8086+

Description: This instruction sets the value of the interrupt flag to one, thus allowing hardware interrupts to occur.

```
EG: CLI      ; Stop interrupts
    ...      ; Perform crucial function
    STI      ; Enable interrupts
```

■ STOS - Name: Store String

Type: 8086+

Description: This instruction exists in the following forms:

```
STOSB      - Store a byte       - AL
STOSW      - Store a word       - AX
STOSD      - Store a doubleword - EAX
```

The instructions write the current contents of the accumulator to the memory location pointed to by ES:DI. It then increments or decrements DI according to the operand used, and the value in the direction flag.

```
EG: MOV   AX, 0A000h
    MOV   ES, AX
    MOV   AL, 03h
    MOV   DI, 0
    STOSB           ; Store 03 at ES:DI,
                    ; which just happens
    ; to be at the top of the screen in
    ; mode 13h
```

■ SUB <DEST>, <SOURCE> - Name: Subtract

Type: 8086+

Description: This instruction subtracts <SOURCE> from <DEST>, storing the result in <DEST>.

```
EG: SUB   ECX, 12
```

■ TEST <DEST>, <SOURCE> - Name: Test Bits

Type: 8086+

Description: This instruction performs a bit-by-bit AND operation on <SOURCE> and <DEST>. The result is reflected in the flags, and they are set as the would be after an AND operation.

```
EG: TEST   AL, 0Fh   ; Check to see if any
                     ; bits set in the low
                     ; nibble of AL
```

■ XCHG <VALUE1>, <VALUE2> - Name: Exchange

Type: 8086+

Description: This instruction exchanges the values in <VLAUE1> and <VALUE2>.

EG: XCHG AX, BX

- XOR <DEST>, <SOURCE> - Name: Exclusive Boolean OR
  Type: 8086+

  Description: This instruction performs a bit-by-bit exclusive OR operation on <SOURCE> and <DEST>. The operation is defined as follows:

  ```
  XOR   0, 0    = 0
  XOR   0, 1    = 1
  XOR   1, 0    = 1
  XOR   1, 1    = 0
  ```

  EG: XOR AX, BX

Phew! What a lot there are, and we only covered the basic ones! You are not expected to understand each and every one of them though. You probably saw words like 'Two's Complement', and thought - "What the hell does that mean?".

Do not worry about them for now. We'll continue at our usual pace, and introduce the new instructions above one by one, explaining them as we go. If you already understand them now, then this is an added bonus. You will also notice that there were a lot of 8086 instructions above. There are actually very few instances where it is necessary to use a 386 or 486 instruction, let alone Pentium instructions.

Anyway, before we press on with the VGA, I'll just list the speed at which each of the above instructions execute at, so you can use this to gauge how fast your Assembler routines are.

Instruction	386 Clock Ticks	486 Clock Ticks
ADC	2	1
ADD	2	1
AND	2	1
BT	3	3
CALL	7+m	3
CBW	3	3
CLC	2	2
CLD	2	2
CLI	5	3
CMC	2	2
CMP	2	1

CWD	2	3
DEC	2	1
DIV	-	-
- Byte	9-14	13-18
- Word	9-22	13-26
- DWord	9-38	13-42
IN	12/13	14
INC	2	1
INT	depends	depends
Jcc	-	-
- Branch	7+m	3
- No Branch	3	1
JMP	7+m	3
LAHF	2	3
LEA	2	1
LOOP	11	6
Lseg	7	6
MOV	2	1
MUL	-	-
- Byte	9-14	13-18
- Word	9-22	13-26
- DWord	9-38	13-42
NEG	2	1
NOT	2	1
OR	2	1
OUT	10/11	16
POP	4	1
POPA	24	9
POPF	5	9
PUSH	2	1
PUSHA	18	11
PUSHF	4	4
REP	depends	depends
RET	10+m	5
ROL	3	3
ROR	3	3
SAHF	3	2
SBB	2	1
SHL	3	3
SHR	3	3
STC	2	2
STD	2	2
STI	3	5
STOS	4	5
SUB	2	1
TEST	2	1
XCHG	3	3
XOR	2	1

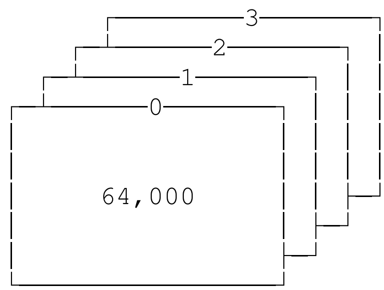
Note: m = Number of components in next instruction executed.

Ugh, I never want to see another clock-tick again! Now, on with the fun stuff - the VGA!

You've probably noticed by now that your video card has more than 256K of RAM. (If you haven't, then these tutorials are probably not for you.) Even if you have only 256K of RAM, like my old 386, you'll still be able to get into mode 13h - 320x200x256. However, this raises some questions.

Multiply 320 by 200 and you'll notice that you only need 64,000 bytes of memory to store a single screen. (The VGA actually gives us 64K, which is 65,536 bytes for the unaware.) What happened to the remaining 192K or so?

Well, the VGA is actually arranged in bitplanes, like this:



Each plane being 64,000 bytes long. Here's how it works:

```
 A pixel at 0, 0 is mapped in plane 0 at offset 0;
 A pixel at 1, 0 is mapped in plane 1 at offset 0;
 A pixel at 2, 0 is mapped in plane 2 at offset 0;
 A pixel at 3, 0 is mapped in plane 3 at offset 0;
 A pixel at 4, 0 is mapped in plane 0 at offset 1   ... and so on ...
```

Because of the pixels being chained across all four planes, it is impossible to use multiple pages in mode 13h without having to resort to using a virtual screen, or something similar.

The automatic mapping of the pixels is handled completely by the video card, so you can blindly work away without even knowing about the four bitplanes if you wish.

We'll go onto how we can get around this, by entering a special display mode, known as Mode X, later, but for now, let's just see what we can do in plain old mode 13h.

DRAWING LINES

We've gone a little over the size that I'd planned to go to for this tutorial, and I had intended to cover Bresenham's Line Algorithm, but that'll have to wait till next week. However, I will cover how to draw a simple horizontal line in Assembler.

An Assembler Horizontal Line Routine:

First we'll need to point ES to the VGA.

This should do the trick:

```
MOV   AX, 0A000h
MOV   ES, AX
```

Now, we'll need to read the X1, X2 and Y values into registers, so something like this should work:

```
MOV   AX, X1    ; AX now equals the X1 value
MOV   BX, Y     ; BX now equals the Y  value
MOV   CX, X2    ; CX now equals the X2 value
```

It will be necessary to work out how long the line is, so we'll use CX to store this, seeing as: i) CX already holds the X2 value, and ii) we'll be using a REP instruction, which will use CX as a counter.

```
SUB   CX, AX    ; CX = X2 - X1
```

Now we'll need to work out what DI will be for the very first pixel we'll be plotting, so we'll use what we did in the PutPixel routine:

```
MOV   DI, AX    ; DI = X1
MOV   DX, BX    ; DX = Y
SHL   BX, 8     ; Shift Y left 8
SHL   DX, 6     ; Shift Y left 6
ADD   DX, BX    ; DX = Y SHL 8 + Y SHL 6
ADD   DI, DX    ; DI = Y x 320 + X
```

We have the offset of the first pixel now, so all we have to do is put the color we want to draw in, in AL, and use STOSB to plot the rest of the line.

```
MOV   AL, Color ; Move the color to plot with into AL
REP   STOSB     ; Plot CX pixels
```

Note that we used STOSB because it will increment DI for us, thus saving a lot of MOV's and INC's. Now, depending on what language you'll use to implement this in, you'll get something like:

```
void Draw_Horizontal_Line(int x1, int x2, int y, char color);
{
asm {
   mov   ax, 0xa000
   mov   es, ax        ; Point ES to the VGA

   mov   ax, x1        ; AX = X1
   mov   bx, y         ; BX = Y
   mov   cx, x2        ; CX = X2

   sub   cx, ax        ; CX = Difference of X2 and X1

   mov   di, ax        ; DI = X1
   mov   dx, bx        ; DX = Y
   shl   bx, 8         ; Y SHL 8
   shl   dx, 6         ; Y SHL 6
   add   dx, bx        ; DX = Y SHL 8 + Y SHL 6
   add   di, dx        ; DI = Offset of first pixel

   mov   al, color     ; Put the color to plot in AL
```

```
    rep   stosb         ; Draw the line
    }
 }
```

We'll now we've covered how to draw a simple horizontal line. The above routine isn't blindingly fast, but it isn't all that bad either. Just changing the calculation of DI part to look like the fast PutPixel I gave out in Tutorial Two would probably double the speed of this routine.

My own horizontal line routine is probably about 4 to 5 times as fast as this one, so in the future, I'll show you how to optimize this one fully. Next week we'll also cover how to get and set the palette, and how we can draw circles. I'm sorry it didn't make it into this tutorial, but this one sort of grew a bit...

THINGS TO DO:

1) Write a vertical line routine based on the one above. Clue: You'll need to increment DI by 320 at some stage.

2) Go over the list of Assembler instructions, and learn as many as you can.

3) Have a look at the Starfield I wrote, and try to fix the bugs in it. See what you can do with it.

Sorry again that I didn't include the things I said I would last week, but as I said, the tutorial just grew, and I'm a bit behind with some other projects I'm supposed to be working on.

Next week's tutorial _will_ include:

- Line algorithms and examples;
- A circle algorithm;
- The palette;
- Something else that you ought to know...

If you wish to see a topic discussed in a future tutorial, then mail me, and I'll see what I can do.

Don't miss out!!! Download next week's tutorial from my homepage at:

- http://www.faroc.com.au/~blackcat

See you next week!

- Adam.

```
╔═══════════════════════════════════════════════════════════╗
║                Adam's Assembler Tutorial 1.0              ╟──┐
║                                                           ║  │
║                          PART VI                          ║  │
╚═══════════════════════════════════════════════════════════╝  │
  └────────────────────────────────────────────────────────────┘
```

```
Revision :  1.3
Date     :  13-04-1996
Contact  :  blackcat@faroc.com.au
            http://www.faroc.com.au/~blackcat
```

Hello again, Assembler coders. This edition is a little late, but I had a lot of other things to finish, and I'm working on a game of my own now. It's a strategy game, like Warlords II, and I think I'm going to have to write most of the code for it in 640x480, not my beloved 320x200 - but I may change my mind. Heck, the amount of games I started writing but never got around to finishing is pretty large, so this one may not get all that far.

Anyway, I said we'd be having a look at some line/circle routines this week, so here we go...

Last week we came up with the following horizontal line routine -

```
   mov   ax, 0A000h
   mov   es, ax          ; Point ES to the VGA

   mov   ax, X1
   mov   bx, Y           ; BX = Y
   mov   cx, X2          ; CX = X2

   sub   cx, ax          ; CX = Difference of X2 and X1

   mov   di, ax          ; DI = X1
   mov   dx, bx          ; DX = Y
   shl   bx, 8           ; Y SHL 8
   shl   dx, 6           ; Y SHL 6
   add   dx, bx          ; DX = Y SHL 8 + Y SHL 6
   add   di, dx          ; DI = Offset of first pixel

   mov   al, color       ; Put the color to plot in AL
   rep   stosb           ; Draw the line
```

Now although this routine was much faster than the BGI routines, (or whatever your compiler provides), it could be improved upon greatly. If we go through the routine with the list of clock ticks provided in the last tutorial, you'll see that it chews up quite a few.

I'll leave optimization up to you for now, (we'll get onto that in a later tutorial), but either replacing the STOSB with MOV ES:[DI], AL or STOSW will speed things up quite a bit. Don't forget that if you decide to use a loop, to whack words onto the VGA, you will have to decrement CX by one.

Now, lets get on to a vertical line. We'll have to calculate the offset of the first pixel as we did with the horizontal line routine, so something like this should do:

```
   mov   ax, 0A000h      ; Put the VGA segment into AX
   mov   es, ax          ; Point ES to the VGA

   mov   ax, Y1          ; Move the first Y value into AX
   shl   ax, 6           ; Y x 2 to the power 6
   mov   di, ax          ; Move the new Y value into DI
   shl   ax, 2           ; Now we have Y = Y x 320
   add   di, ax          ; Add that value onto DI
   add   di, X           ; Add the X value onto DI
```

Now a bit of basic housekeeping...

```
   mov   cx, Y2          ; Store Y2 in CX
   mov   al, Color       ; Store the color to plot with in AL
   sub   cx, Y1          ; CX = vertical length of line
```

And now the final loop...

```
Plot:
   mov   es:[di], al     ; Put the a pixel at the current offset
   add   di, 320         ; Move down to the next row
   dec   cx              ; Decrement CX by one
   jnz   Plot            ; If CX <> 0, then keep on plotting
```

Not a fantastic routine, but it's pretty good all the same. Note how it was possible to perform a comparison after DEC CX. This is an extremely useful concept, so don't forget that it is possible.

Have a play around with the code, and try and speed it up. Try other methods of finding the offset, or different methods of flow control.

Now, that was the easy stuff. We are now going to have a look at a line routine capable of drawing diagonal lines.

The following routine was picked up from SWAG, author unknown, and is an ideal routine to demonstrate a line algorithm. It is in great need of optimization, so this can be a task for you - if you wish. Some of the points needing addressing are:

1) Whoever wrote it had never heard of XCHG - this would save quite a few clock ticks;

2) It makes one of the great sins of unoptimized code - it will say, move a value to AX, and then perform an operation involving AX in the next instruction, thus incurring a penalty cycle. (We'll talk about this next week).

3) It works with BYTES not WORDS, so the speed of writing to the VGA could be doubled if words were used.

4) And the biggest sin of all, it uses a MUL to find the offset. Try using shifts or an exchange to speed things up.

Anyway, I put the comments in, and I feel that it is fairly self explanatory as it is, so I won't go over how it works. You should be able to pick that up for yourself. Work through the routine, and see how the gradient for the line is worked out.

```
Procedure Line(X1, Y1, X2, Y2 : Word; Color : Byte);   Assembler;

Var
   DeX          : Integer;
   DeY          : Integer;
   IncF         : Integer;

Asm     { Line }
   mov   ax, [X2]       { Move X2 into AX                                    }
   sub   ax, [X1]       { Get the horiz length of the line -  X2 - X1        }
   jnc   @Dont1         { Did X2 - X1 yield a negative result?               }
   neg   ax             { Yes, so make the horiz length positive             }

@Dont1:
   mov   [DeX], ax      { Now, move the horiz length of line into DeX        }
   mov   ax, [Y2]       { Move Y2 into AX                                    }
   sub   ax, [Y1]       { Subtract Y1 from Y2, giving the vert length        }
   jnc   @Dont2         { Was it negative?                                   }
   neg   ax             { Yes, so make it positive                           }

@Dont2:
   mov   [DeY], ax      { Move the vert length into DeY                      }
   cmp   ax, [DeX]      { Compare the vert length to horiz length            }
   jbe   @OtherLine     { If vert was <= horiz length then jump              }

   mov   ax, [Y1]       { Move Y1 into AX                                    }
   cmp   ax, [Y2]       { Compare Y1 to Y2                                   }
   jbe   @DontSwap1     { If Y1 <= Y2 then jump, else...                     }
   mov   bx, [Y2]       { Put Y2 in BX                                       }
   mov   [Y1], bx       { Put Y2 in Y1                                       }
   mov   [Y2], ax       { Move Y1 into Y2                                    }
                        { So after all that.....                             }
                        { Y1 = Y2 and Y2 = Y1                                }

   mov   ax, [X1]       { Put X1 into AX                                     }
   mov   bx, [X2]       { Put X2 into BX                                     }
   mov   [X1], bx       { Put X2 into X1                                     }
   mov   [X2], ax       { Put X1 into X2                                     }

@DontSwap1:
```

```
   mov   [IncF], 1      { Put 1 in IncF, ie, plot another pixel             }
   mov   ax, [X1]       { Put X1 into AX                                    }
   cmp   ax, [X2]       { Compare X1 with X2                                }
   jbe   @SkipNegate1   { If X1 <= X2 then jump, else...                    }
   neg   [IncF]         { Negate IncF                                       }

@SkipNegate1:
   mov   ax, [Y1]       { Move Y1 into AX                                   }
   mov   bx, 320        { Move 320 into BX                                  }
   mul   bx             { Multiply 320 by Y1                                }
   mov   di, ax         { Put the result into DI                            }
   add   di, [X1]       { Add X1 to DI, and tada - offset in DI             }
   mov   bx, [DeY]      { Put DeY in BX                                     }
   mov   cx, bx         { Put DeY in CX                                     }
   mov   ax, 0A000h     { Put the segment to plot in, in AX                 }
   mov   es, ax         { ES points to the VGA                              }
   mov   dl, [Color]    { Put the color to use in DL                        }
   mov   si, [DeX]      { Point SI to DeX                                   }

@DrawLoop1:
   mov   es:[di], dl    { Put the color to plot with, DL, at ES:DI          }
   add   di, 320        { Add 320 to DI, ie, next line down                 }
   sub   bx, si         { Subtract DeX from BX, DeY                         }
   jnc   @GoOn1         { Did it yield a negative result?                   }
   add   bx, [DeY]      { Yes, so add DeY to BX                             }
   add   di, [IncF]     { Add the amount to increment by to DI              }

@GoOn1:
   loop  @DrawLoop1     { No negative result, so plot another pixel         }
   jmp   @ExitLine      { We're all done, so outta here!                    }

@OtherLine:
   mov   ax, [X1]       { Move X1 into AX                                   }
   cmp   ax, [X2]       { Compare X1 to X2                                  }
   jbe   @DontSwap2     { Was X1 <= X2 ?                                    }
   mov   bx, [X2]       { No, so move X2 into BX                            }
   mov   [X1], bx       { Move X2 into X1                                   }
   mov   [X2], ax       { Move X1 into X2                                   }
   mov   ax, [Y1]       { Move Y1 into AX                                   }
   mov   bx, [Y2]       { Move Y2 into BX                                   }
   mov   [Y1], bx       { Move Y2 into Y1                                   }
   mov   [Y2], ax       { Move Y1 into Y2                                   }

@DontSwap2:
   mov   [IncF], 320    { Move 320 into IncF, ie, next pixel is on next row  }
   mov   ax, [Y1]       { Move Y1 into AX                                   }
   cmp   ax, [Y2]       { Compare Y1 to Y2                                  }
   jbe   @SkipNegate2   { Was Y1 <= Y2 ?                                    }
   neg   [IncF]         { No, so negate IncF                                }

@SkipNegate2:
   mov   ax, [Y1]       { Move Y1 into AX                                   }
   mov   bx, 320        { Move 320 into BX                                  }
   mul   bx             { Multiply AX by 320                                }
   mov   di, ax         { Move the result into DI                           }
   add   di, [X1]       { Add X1 to DI, giving us the offset                }
   mov   bx, [DeX]      { Move DeX into BX                                  }
   mov   cx, bx         { Move BX into CX                                   }
   mov   ax, 0A000h     { Move the address of the VGA into AX               }
   mov   es, ax         { Point ES to the VGA                               }
```

```
   mov   dl, [Color]   { Move the color to plot with in DL                  }
   mov   si, [DeY]     { Move DeY into SI                                   }

@DrawLoop2:
   mov   es:[di], dl   { Put the byte in DL at ES:DI                        }
   inc   di            { Increment DI by one, the next pixel                }
   sub   bx, si        { Subtract SI from BX                                }
   jnc   @GoOn2        { Did it yield a negative result?                    }
   add   bx, [DeX]     { Yes, so add DeX to BX                              }
   add   di, [IncF]    { Add IncF to DI                                     }

@GoOn2:
   loop  @DrawLoop2    { Keep on plottin'                                   }

@ExitLine:
                       { All done!                                          }
End;
```

I don't think I made any mistakes with the commenting, but I am pretty tired, and I haven't handy any caffeine for days - let alone hours, so if you spot a mistake - please let me know.

I was going to include a Circle algorithm, but I couldn't get mine to work in Assembler - all the floating point math might have something to do with it. I could include one written in a high level language, but this is meant to be an Assembler tutorial, not a graphics one. However, if enough people shout for one...

THE INS AND OUTS OF IN AND OUT

IN and OUT are a very important part of Assembler coding. They allow you to directly send/receive data from any of the PC's 65,536 hardware ports, or registers. The basic syntax is as follows:

- IN <ACCUMULATOR>, <PORT> - Name: Input from I/O port
  Type: 8086+

  Description: This instruction reads a value from one of the 65536 hardware ports into the specified accumulator.

  AX and AL are commonly used for input ports, and DX is commonly used to identify the port.

  ```
  EG: IN    AX, 72h

      MOV   DX, 3C7h
      IN    AL, DX
  ```

■ OUT <PORT>, <ACCUMULATOR> - Name: Output to Port
Type: 8086+

Description: This instruction outputs the value in the accumulator to <PORT>. Using the DX register to pass the port to OUT, you may access up to 65,536 ports.

```
EG: MOV   DX, 378h
    OUT   DX, AX
```

Okay, that wasn't very helpful, as it didn't tell you much about how to use it - let alone what to use it for. Well, if you intend to work with the VGA much, you'll have to be able to program its internal registers. Similar to the registers that you've been working with up until now, you can think of changing them just like interrupts, except: 1) You pass the value to the port, and that's it; and 2) It is pretty near instantaneous.

As an example, we'll cover how to set and get the palette by directly controlling the VGA's hardware.

Now, the VGA has a lot of registers, but the next three you'd better get to know pretty well:

- ■ 03C7h - PEL Address Register (Read)
  Sets the palette in read mode
- ■ 03C8h - PEL Address Register (Write)
  Sets the palette in write mode
- ■ 03C9h - PEL Data Register (Read/Write)
  Read in, or write 3 RGB values, every 3rd write, the index, or color you are setting, is incremented by one.

What all this means is -

If we were to set a color's RGB value, we'd send the value of the color we wanted to change to 03C8h, then read in 3 values from 03C9h. In Assembler, we'd do this:

```
mov   dx, 03C8h        ; Put the DAC read register in DX
mov   al, [Color]      ; Put the color's value in AL
out   dx, al           ; Send AL to port DX
inc   dx               ; Now use port 03C9h
mov   al, [R]          ; Put the new RED value in AL
out   dx, al           ; Send AL to port DX
mov   al, [G]          ; Put the new GREEN value in AL
out   dx, al           ; Send AL to port DX
mov   al, [B]          ; Put the new BLUE value in AL
out   dx, al           ; Send AL to port DX
```

And that would do things nicely. To read the palette, we'd do this:

```
mov   dx, 03C7h        ; Put the DAC write register in DX
mov   al, [Color]      ; Put the color's value in AL
out   dx, al           ; Send AL to port DX
add   dx, 2            ; Now use port 03C9h
```

```
in    al, dx           ; Put the value got from port DX in AL
les   di, [R]          ; Point DI to the R variable - this came from Pascal
stosb                  ; Store AL in R

in    al, dx           ; Put the value got from port DX in AL
les   di, [G]          ; Point DI to the G variable
stosb                  ; Store AL in G

in    al, dx           ; Put the value got from port DX in AL
les   di, [B]          ; Point DI to the B variable
stosb                  ; Store AL in B
```

Note how that routine was coded differently. This was originally a Pascal routine, and as Pascal doesn't like you messing with Pascal variables in Assembler, you have to improvise.

If you are working in stand alone Assembler, then you can code this much more efficiently, like the first example. I left the code as it was so those who are working with a high-level language can get around a particularly annoying problem.

Now you have seen how useful IN and OUT can be. Directly controlling hardware is both fast and efficient. In the next few weeks, I may include a list of some of the most common ports, but if you have a copy of Ralf Brown's Interrupt List, (available at X2FTP), you will already have a copy.

Note: You can find a link to Ralf's Interrupt List on my homepage.

A bit more on the FLAGS register:

Now, although we have been using the flags register in almost all our code up until this point, I haven't really gone into depth about it. You can work blissfully unaware of the flags, and compare things without knowing what's really happening, but if you want to get further into Assembler, you'll need to know some more.

Back in Tutorial Three, I gave an extremely simplistic view of the FLAGS register. In reality, the FLAGS, or EFLAGS register is actually a 32-bit register, although only bits 0-18 are used. We really don't need to know any of the flags above bit 11 for now, but it's good to know they are there.

The EFLAGS register actually looks like this:

```
18  17  16  15  14  13  12   11  10  09  08  07  06  05  04  03  02  01  00
AC  VM  RF  --  NT  IO/PL    OF  DF  IF  TF  SF  ZF  --  AF  --  PF  --  CF
```

Now, the flags are as follows:

- AC - Alignment Check (80486)
- VM - Virtual 8086 Mode
- RF - Resume Flag
- NT - Nested Task Flag

- IOPL - I/O Privilege Level - has a value of 0,1,2 or 3 thus 2 bits big

- OF - Overflow Flag
  This bit is set to ONE if an arithmetic instruction generated a result that was too large or too small to fit in the destination register.

- DF - Direction Flag
  When set to ZERO, string instructions, such as MOVS, LODS and STOS will increment the memory address they are working on by one. This means that say, DI, will be incremented when you use STOSB to put a pixel at ES:DI. Setting the bit to ZERO will decrement the memory address after each call.

- IF - Interrupt Enable Flag
  When this bit is set, the processor will respond to external hardware interrupts. When the bit is reset, hardware interrupts are ignored.

- TF - Trap Flag
  When this bit is set, an interrupt will occur immediately after the next instruction executes. This is generally used in debugging.

- SF - Sign Flag
  This bit is changed after arithmetic instructions. The bit receives the high-order bit of the result, and if set to ONE, it indicates that the result of the operation was negative.

- ZF - Zero Flag
  This bit is set when arithmetic instructions generate a result of zero.

- AF - Auxiliary Carry Flag
  This bit indicates that a carry out of the low-order nibble of AL occurred in an arithmetic instruction.

- PF - Parity Flag
  This bit is set to one when an arithmetic instruction results in an even number of 1 bits.

- CF - Carry Flag
  This bit is set when the result of an arithmetic operation is too large or too small for the destination register or memory address.

Now, of all those above, you really won't have to worry too much about most of them. For now, just knowing CF, PF, ZF, SF, IF, DF and OF will be sufficient. I didn't give the first few comments as they are fairly technical, and are used mostly in protected mode and complex situations. You shouldn't have to know them.

You can, if you wish, move a copy of the flags into AH with LAHF - (Load AH with Flags) - and modify or read individual bits, or change the status of bits more easily with CLx and STx. However you plan to change the flags, remember that they can be extremely useful in many situations.

(They can also be very annoying when late at night, lines start drawing backwards, and you spend an hour wondering why - then remember that you forgot to clear the direction flag!)

I think we've covered quite a few important topics in this tutorial. Brush up on the flags, and go over the largish line routine, as it is an excellent example of flow control. Make sure your skills at controlling instruction flow are perfected.

Next week I'll try to tie all the topics we've covered over the last few weeks together, and present some form of review of all that you've learnt. Next week I'll also go into optimization, and how you can speed up all the code we've worked with so far.

Next week's tutorial will include:

- A review of all you've learnt
- Optimization
- Declaring procedures in Assembler
- Linking your code to C/C++ or Pascal

If you wish to see a topic discussed in a future tutorial, then mail me, and I'll see what I can do.

Don't miss out!!! Download next week's tutorial from my homepage at:

- http://www.faroc.com.au/~blackcat

See you next week!

- Adam.

Adam's Assembler Tutorial 1.0

PART VII

Revision : 1.3
Date : 01-05-1996
Contact : blackcat@faroc.com.au
http://www.faroc.com.au/~blackcat

Hi again, and welcome to the seventh instalment of the Assembler Tutorials.

These tutorials seem to be coming out at an irregular rate, but people are screaming at me for things I haven't done, and I'm still working on projects of my own. I hope to spit these tutes out fortnightly.

Now this week we'll be covering two pretty important topics. When I first began playing around with Assembler I soon realised that Turbo Pascal, (the language I was working with then), had a few limitations - one of them being that it was, and still is, a 16-bit language. This meant that if I wanted to play around with super-fast 32-bit screen writes, I couldn't. Not even with the built in Assembler, (well, not easily anyway).

What I needed to do was to write code separately in 100% Assembler, then link it to Turbo. This isn't a particularly hard task, and is one of the things I'm going to try and teach you today.

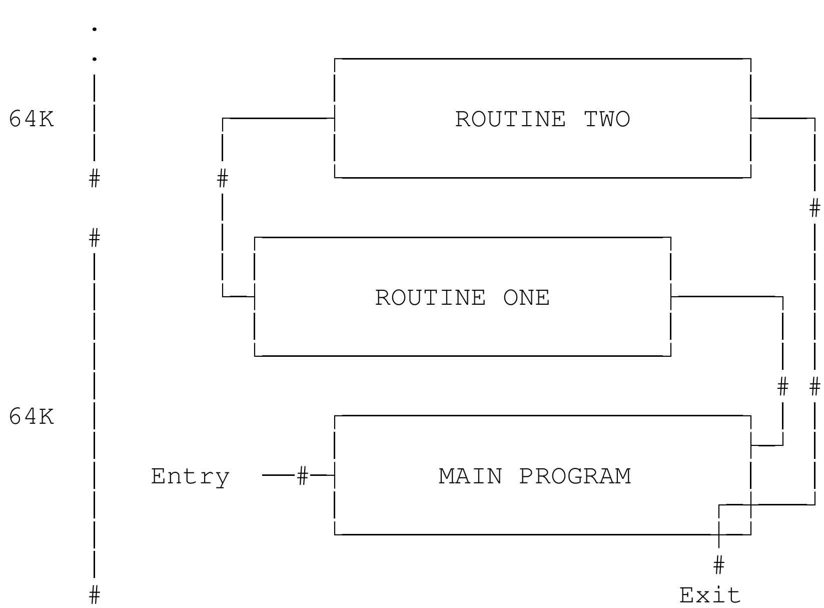
The other advantage of writing routines in stand alone Assembler is that you can also link the resulting object code to another high-level language, like C.

WRITING EXTERNAL CODE FOR YOUR HIGH LEVEL LANGUAGE

Before we begin, you'll need an idea of what far and near calls are. If you already know, then skip ahead past this little section.

As we discussed before, the PC has a segmented architecture. As you know, you can only access one 64K segment at a time. Now if you are working on code less than 64K in size, or in a language that takes care of all your worries for you, you don't need to worry so much. However, when working in Assembler, we do.

Imagine we have the following program loaded into memory:



When a JMP is executed to transfer control to Routine One, this will be a near call. We do not leave the segment that the main body of the program is located in, and so when the JMP or CALL is executed, and CS:IP is changed by JMP, only IP need be changed, not CS.

The offset changes, but not the segment.

Now jumping to Routine Two would be different. This leaves the current segment, and so both parts of the CS:IP pair will need to be altered. This is a far call.

The problem occurs when the CPU encounters a RET or RETF at the end of the call. Let's say by accident you put RET at the end of Routine Two instead of RETF. When the CPU saw RET it would only POP IP off the stack, and so your machine would probably crash, as CS:IP would now be pointing to garbage.

This point is especially important when linking to a high-level language. Whenever you write code in Assembler and link it to say, Pascal, remember to use the {$F+} compiler directive, even if it wasn't a far call. This way, after Turbo has called the routine, it'll pop both CS and IP, and everything will be fine.

Failure to do so is at your own risk!

Okay, let's return to the stand alone model we saw in Tutorial Three. I don't remember rightly, but I think it went something like this:

```
    DOSSEG
    .MODEL SMALL
    .STACK 200h
    .DATA
    .CODE

START:


END START
```

Now, I think it's time you graduated from using that skeleton. Let's look at other ways we can set up a skeleton routine:

```
    DATA     SEGMENT WORD PUBLIC

    DATA     ENDS


    CODE     SEGMENT WORD PUBLIC
     ASSUME  CS:CODE, DS:DATA

    CODE     ENDS

    END
```

This is an obviously different skeleton. Note how I omitted the period in front of DATA and CODE. Depending on which assembler/linker you use, you may need to use a period or you may not. TASM, the assembler I use, supports both of these formats, so pick one that both you and your assembler are happy with.

Note also the use of DATA SEGMENT WORD PUBLIC. Firstly, WORD tells the assembler to align the segment on word boundaries.

FUN FACT: You needn't worry about this for now, but Turbo Pascal does this anyway, so putting BYTE instead of word would make no difference. :)

PUBLIC allows the compiler you use, to access any variables you may wish to place in the data segment. If you do not want your high-level language to have access to any variables you may declare, then omit this. If you will not be needing access to the data segment anyway, then don't bother with the whole DATA SEGMENT thing.

Now, onto the code segment. Generally, you will need to include this in all the code you write. :) The assume statement will also be pretty standard in all you'll work with. You can also expect to see CSEG and DSEG instead of CODE and DATA. Note that again this is declared public. This is where all our routines will go.

So, how do I declare external procedures?

Okay, for this example, we're going to use a few simple routines similar to those in the MODE13H Pascal library. (Available from my homepage).

If you remember, the procedures looked a bit like this:

- Procedure PutPixel(X, Y : Integer; Color : Byte);
- Procedure InitMCGA;
- Procedure Init80x25;

Fitting these in our skeleton gives us this:

```
CODE     SEGMENT WORD PUBLIC
 ASSUME  CS:CODE, DS:DATA

 PUBLIC  PutPixel
 PUBLIC  InitMCGA
 PUBLIC  Init80x25

CODE     ENDS


END
```

Now, all we have to do is to code 'em. But hang on a minute - the PutPixel routine had PARAMETERS. How do we use these in external code???

This is the tricky bit. What we do is push these values onto the stack, simply saying -- PutPixel(10, 20, 15); -- will do this for us. It's getting them off that's harder. What I generally do, and I suggest you do, is make sure that you DECLARE ALL EXTERNAL PROCEDURES FAR. This makes working with the stack so much easier.

FUN FACT: Remember that what's first on the stack is LAST OFF. :)

When you call PutPixel, the stack will be changed. As this is a far call, the first four bytes will hold CS:IP. The bytes from then on will hold your parameters.

To cut a long story short, let's say the stack used to look like this:

```
   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ...
```

After calling -- PutPixel(10, 20, 15); -- some time later, it may look like this:

```
   4C EF 43 12 0F 00 14 00 0A 00   9E F4 3A 23 1E 21 ...

   ^^^^^^^^^^^ ^^^^^ ^^^^^ ^^^^^   ^^^^^^^^^^^^^^^^^
      CS:IP    Color   Y     X      Some other crap
```

Now, to complicate things, the CPU stores words on the stack with THE LEAST SIGNIFICANT PART FIRST. This doesn't bother us too much, but if you muck around with a debugger without realising this, you're gonna get really confused.

Note also, that when Turbo Pascal puts one byte data types on the stack, they will chew up TWO BYTES, NOT ONE. Don't you just _love_ the way the PC is organised? ;)

Now, all that I've said up until this point only applies to value parameters - PARAMETERS YOU CANNOT CHANGE. When you muck around with REFERENCE PARAMETERS, like -- MyProc(Var A, B, C : Word); -- each parameter now uses FOUR BYTES of stack space, two for the segment and two for the offset of where the variable is located in memory.

So if you pushed a variable that was held in, say, memory address 4AD8:000Eh, no matter what the value of this was, 4AD8:000Eh would be stored on the stack.

As it happens, that would look like 0E 00 D8 4A on the stack, remembering that the least significant nibble is stored first.

FUN FACT: Value Parameters actually put the value on the stack, Reference Parameters store the address. :)

Okay, now I've got you well and truly confused, it gets a little worse!

To reference these parameters in your code, you have to use the stack pointer, SP. Trouble is, you aren't allowed to play with SP directly, you have to push BP, and move SP into it. This now adds another two bytes to the stack. Lets say BP was equal to 0A45h. Before pushing BP, the stack would look like this:

```
   4C EF 43 12 0F 00 14 00 0A 00

   ^^^^^^^^^^^ ^^^^^ ^^^^^ ^^^^^
      CS:IP    Color   Y     X
```

After pushing BP, you get:

```
 45 0A 4C EF 43 12 0F 00 14 00 0A 00

  ^^^^ ^^^^^^^^^^^ ^^^^^ ^^^^^ ^^^^^
   BP     CS:IP    Color   Y     X
```

Now we've finally got over all that, we can actually access the damn things! What you'd do after calling -- PutPixel(10, 20, 15); -- to access the Color value - 15 - is this:

```
   PUSH  BP
   MOV   BP, SP

   MOV   AX, [BP+6]   ; Now we have Color
```

We can access X and Y like this:

```
   MOV   BX, [BP+8]   ; Now we have Y

   MOV   CX, [BP+10]  ; Now we have X
```

And now we restore BP:

```
   POP   BP
```

Now we return from a FAR call, and remove the six bytes of data we put on the stack:

```
   RETF  6
```

And that's it!

Now, let's put the PutPixel, InitMCGA and Init80x25 into some Assembler code. You end up with something like this:

```
CODE SEGMENT WORD PUBLIC
     ASSUME  CS:CODE, DS:DATA

     PUBLIC PutPixel        ; Declare the public procedures
     PUBLIC InitMCGA
     PUBLIC Init80x25

.386                        ; Let's use some 386 registers

; ----------------------------------------------------------------------------

;
; Procedure PutPixel(X, Y : Integer; Color : Byte);
;


PutPixel PROC FAR           ; Declare a FAR procedure

   PUSH  BP
   MOV   BP, SP             ; Set up the stack

   MOV   BX, [BP+10]        ; BX = X
   MOV   DX, [BP+08]        ; DX = Y
   XCHG  DH, DL             ; As Y will always have a value of less than 200,
   MOV   AL, [BP+06]        ; this is 320x200 don't forget, saying XCHG DH,DL
   MOV   DI, DX             ; is an ingenious way of saying SHL DX, 8
   SHR   DI, 2
   ADD   DI, DX
   ADD   DI, BX             ; Now we have the offset, so...
   MOV   FS:[DI], AL        ; ...plot it at FS:DI

   POP   BP
   RETF  6

PutPixel ENDP

; ----------------------------------------------------------------------------

;
; Procedure InitMCGA;
;

InitMCGA PROC FAR

   MOV   AX, 0A000H         ; Point AX to the VGA
   MOV   FS, AX             ; Why not FS?
   MOV   AH, 00H
   MOV   AL, 13H
   INT   10H
   RETF

InitMCGA ENDP

; ----------------------------------------------------------------------------

;
; Procedure Init80x25;
;

Init80x25 PROC FAR
```

```
   MOV   AH, 00H
   MOV   AL, 03H
   INT   10H
   RETF

Init80x25 ENDP

CODE    ENDS
        END
```

And that's it. I'm sorry if I made the whole thing a bit of a confusing exercise, but that's the fun of computers! :)

Oh, by the way, you can use the above code in Pascal by assembling it with TASM, or MASM. <shudder> Next, include it in your code as follows:

```
{$L WHATEVERYOUCALLEDIT.OBJ}
{$F+}
Procedure PutPixel(X, Y : Integer; Color : Byte);   External;
Procedure InitMCGA;                                 External;
Procedure Init80x25;                                External;
{$F-}

Begin
   InitMCGA;
   PutPixel(100, 100, 100);
   ReadLn;
   Init80x25;
End.
```

FUNCTIONS AND FURTHER OPTIMIZATION

You can get your Assembler routines to return values which you can use in your high-level language if you wish. The table below contains all the necessary information you'll need to know:

Type to Return	Register(s) to Use
Byte	AL
Word	AX
LongInt	DX:AX
Pointer	DX:AX
Real	DX:BX:AX

Now that you've seen how to write external code, you'll probably want to know how you can tweak it to get the full performance that external code can deliver.

Some points for you to work with are as follows:

- You can't use SP directly, but you CAN use ESP. And no, I don't mean use your mental powers to get the parameter you want. :)

- That'll do away with the slow pushing/popping of BP.

- Remember that you'll need to change [xx+6] to [xx+4] for the last, (first), parameter - as BP is now no longer on the stack.

Have a fiddle, and see what you can do with it. It is possible through tweaking to make the code faster than the inline routine featured in MODE13H.ZIP version 1 - (available from my homepage).

Note: I plan to further develop the MODE13H library, adding fonts and other cool features. It will be eventually coded in standalone Assembler, AND be callable from C and Pascal.

Standalone code also has a hefty speed increase. Today I tested the PutPixel routine in the MODE13H library and a standalone PutPixel, (practically identical), and saw an amazing speed difference.

On a 486SX 25 with 4MB of RAM and a 16-bit VGA card, it took only 5 hundredths of a second for the standalone routine to plot 65,536 pixels in the middle of the screen, as opposed to 31 hundredths of a second for the other routine. Big difference, huh?

OPTIMIZATION

As speedy as Assembler may be, you can always speed things up further. I'm going to give some coverage on how you can speed your code up on the 80486, and the 80386, to some extent.

I'm not going to worry too much about the Pentium for now, as the tricks to use on the Pentium certainly ARE tricky, and would take quite a while to explain. Also, you should avoid Pentium specific code, (though this is slowly changing).

The AGI (Address Generation Interlock):

What the hell is that?, you ask. An AGI occurs when a register that is currently being used as a base or index was the destination of a previous instruction. AGI's are bad, and chew up clock ticks.

```
EG:   MOV   ECX, 3
      MOV   FS, ECX
```

This can be avoided by performing another instruction between the two MOVes, as an AGI can only occur on adjacent instructions. (On the 486 anyway.) On the Pentium, an AGI can occur anywhere between THREE instructions.

Use 32-bit Instructions/Registers:

Using 32-bit registers tends to be faster than using their 16-bit counterparts. (Particularly EAX, as many instructions actually become one byte shorter when this register is used. Using DS instead of ES is also faster for a similar reason.)

Other things to try:

- Avoid LOOPing. Try using just a DEC, or INC following by a JZ or similar instruction. This can make a big difference.
- When zeroing out registers, use XOR rather than MOV xx, 0. Believe it or not, this is actually faster.
- Make use of TEST when you are checking to see if a register is equal to zero. By ANDing the operands together, no time is wasted in farting around with a destination register. TEST EAX, EAX is a good way of checking to see if EAX = 0.
- USE SHIFTS! Don't use multiplication to work out even the simplest of sums. The CPU can move a few ones and zeros left or right much faster than it can do the multiplication/division.
- Make cunning use of LEA. One instruction is all it takes to perform an integer multiply and store the result in a register. This is a useful alternative to SHL/SHR. (I know, I know... I said multiplication was bad. But an LEA can sometimes be useful as it can save several instructions.)

  ```
  EG: LEA ECX, [EDX+EDX*4]   ; ECX = EDX x 5
  ```

- Avoid MOVing into segment registers often. If you are going to be working with a value that doesn't change, such as A000h, then load it into, say, FS and use FS from then on.
- Believe it or not, string instructions, (LODSx, MOVSx, STOSx) are much faster on a 386 than they are in a 486. If working with 486+, then use other, more simple instructions instead.
- When moving 32-bit chunks, REP STOSD is a lot faster than using a loop to accomplish the same thing.

Well, now you've seen how you can write external code, declare procedures in Assembler and optimize your routines. Next week I'm _finally_ going to draw all that you've learnt together, and see if we can make some sense out of it all. I'm also going to include a stand alone Assembler example - a better starfield with palette control, to demonstrate INs and OUTs, program control, procedures and TESTing.

Next week's tutorial will include:

- A review of all you've learnt - finally(sorry!);
- Declaring sub-procedures in Assembler;
- A nifty example; :)
- Some other great topic.

If you wish to see a topic discussed in a future tutorial, then mail me, and I'll see what I can do.

Don't miss out!!! Download next week's tutorial from my homepage at:

- http://www.faroc.com.au/~blackcat

See you next week!

- Adam.

Just a little joke I found on a local BBS, which I thought quite amusing. I guess those with a UNIX background will understand it more...

Micro was a real-time operator and dedicated multi-user. His broad-band protocol made it easy for him to interface with numerous input/output devices, even if it meant time-sharing.

One evening he arrived home just as the Sun was crashing, and had parked his Motorola 68040 in the main drive (he had missed the 5100 bus that morning), when he noticed an elegant piece of liveware admiring the daisy wheels in his garden. He thought to himself, "She looks user-friendly. I'll see if she'd like an update tonight."

Mini was her name, and she was delightfully engineered with eyes like COBOL and a PR1ME mainframe architecture that set Micro's peripherals networking all over the place.

He browsed over to her casually, admiring the power of her twin, 32-bit floating point processors and enquired "How are you, Honeywell?".

"Yes, I am well", she responded, batting her optical fibers engagingly and smoothing her console over her curvilinear functions.

Micro settled for a straight line approximation. "I'm stand-alone tonight", he said, "How about computing a vector to my base address? I'll output a byte to eat, and maybe we could get offset later on."

Mini ran a priority process for 2.6 milliseconds, then transmitted 8 K. "I've been dumped myself recently, and a new page is just what I need to refresh my disks. I'll park my machine cycle in your background and meet you inside." She walked off, leaving Micro admiring her solenoids and thinking, "Wow, what a global variable, I wonder if she'd like my firmware?"

They sat down at the process table to top of form feed of fiche and chips and a bucket of baudot. Mini was in conversation mode and expanded on ambiguous arguments while Micro gave the occassional acknowledgements, although, in reality, he was analyzing the shortest and least critical path to her entry point. He finally settled on the old 'Would you like to see my benchmark routine', but Mini was again one step ahead.

Suddenly she was up and stripping off her parity bits to reveal the full functionality of her operating system software. "Let's get BASIC, you RAM", she said. Micro was loaded by this; his hardware was in danger of overflowing its output buffer, a hang-up that Micro had consulted his analyst about. "Core", was all he could say, as she prepared to log him off.

Micro soon recovered, however, when Mini went down on the DEC and opened her divide files to reveal her data set ready.
He accessed his fully packed root device and was just about to start pushing into her CPU stack, when she attempted an escape sequence.

"No, no!", she cried, "You're not shielded!"

Adam's Assembler Tutorial 1.0

PART VIII

Revision : 1.4
Date : 28-06-1996
Contact : blackcat@faroc.com.au
http://www.faroc.com.au/~blackcat

Well, welcome back assembler coders. This tutorial is _really_ late, and would have been a lot later were it not for Björn Svensson, and many others like him, who thanks to their determination to get Tutorial 8, persuaded me to get this thing written. Of, course, this means I've probably failed all my exams over the past two weeks, but such is life. :)

Okay, this week we're really going to learn something. We're going to take a much closer look at how we can declare variables, and delve into the world of structures. You'll learn how to create arrays in Assembler, and this concept is reinforced with the demo program I included - a fire routine!

DATA STRUCTURES IN ASSEMBLER

Okay, by now you should know that you can use the DB, (Declare Byte) and DW, (Declare Word) to create variables. However, up until now we have been using them as you would use the Const declaration in Pascal. That is, we have been using it to assign a byte or word with a value.

EG:

```
MyByte DB 10  --  which is the same as  --  Const MyByte : Byte = 10;
```

However, we could just have easily said:

```
MyByte DB ?
```

...and then later on said:

```
MOV   MyByte, 10
```

In fact DB is very powerful indeed. Several tutorials ago when you were learning to put strings on the screen, you saw something along the lines of this:

```
MyString DB 10, 13 "This is a string$"
```

Now the more inquisitive of you would have probably said to yourselves, "Hang on... that tutorial guy said that DB declares a BYTE. How can DB declare a string then?" Well, DB has the ability to reserve space for multiple byte values - from 1 to as many bytes as you need.

You may also have wondered what the 10 and 13 before the text stood for. Well, dig out your ASCII chart and have a look at character 10 and character 13. You'll notice that 10 is Line Feed and 13 is Carriage Return. Basically, it's just like saying:

```
MyString := #10 + #13 + 'This is a string';
```

in Pascal.

Okay, so you've seen how to create variables properly. But what about constants? Well, in Assembler, constants are known as Equates. Equates make Assembler coding much more easy, and can simplify things greatly. For instance, if I were to have used the following in previous tutorials:

```
LF   EQU 10
CR   EQU 13

DB   LF, CR "This is a string$"
```

...people would have got the 10, 13 thing straight away. However, just to make things a little more complicated, there is yet another way that you can assign values to identifiers. You can do things just like you would in BASIC:

```
Population  = 4Ch
Magnitude   = 0
```

Basically, you can bear the following points in mind:

- Once you use EQU to assign a value to an identifier, you can not change it.
- EQU can be used to define just about any type - including strings. You cannot, however, do this when you use a '='. An '=' can only define numeric values.
- You can use EQU almost anywhere in your program.
- Values defined with '=' can be changed.

And now on with one of the trickier aspects of Assembler coding - structures. Structures are not variables themselves, they are a TYPE - basically a schematic for a variable.

As an example, if you had the following in Pascal:

```
Type
   Date      = Record;
      Day    : Byte;
      Month  : Byte;
      Year   : Word;
   End;    { Record }
```

You could represent this in Assembler as follows:

```
Date         STRUC
   Day       DB ?
   Month     DB ?
   Year      DW ?
Date         ENDS
```

However, one of the advantages of Assembler is that you can initialize all or some of the fields of the structure before you even refer to the structure in your code segment.

That structure above could easily be written as:

```
Date          STRUC
   Day        DB ?
   Month      DB 6
   Year       DW 1996
Date          ENDS
```

Some important points to remember are as follows:

- You can declare a structure anywhere in your code, although for good program design, you should really put them in the data segment, unless they will only be used by a subroutine.

- Defining a structure does not reserve any bytes of memory. It is only when you declare a structured variable that memory is allocated.

REFERENCING DATA STRUCTURES IN ASSEMBLER

Well, you've seen how to define structures, but how do you actually refer to them in your code?

All you have to do, is place a few lines like the ones below somewhere in your program - preferably in the data segment.

```
Date          STRUC
   Day        DB 19
   Month      DB 6
   Year       DW 1996
Date          ENDS

Date_I_Passed_Physics   Date <>   ; I hope!
```

At this point in time, Date_I_Passed_Physics has all three of its fields assigned. Day is set to 19, Month to 6 and Year to 1996. Now, what are those brackets, "<>", doing after date you ask?

The brackets present us with yet another chance to alter the contents of the variable's fields. If I had written this:

```
Date_I_Passed_Physics   Date <10,10,1900>
```

...then the fields would have been changed to the values in the brackets. Alternatively, it would have been possible to do this:

```
Date_I_Passed_Physics   Date <,10,>   ;
```

And now only the Month field has been changed. Note that in this example, the second comma was not needed as we did not go on to change further fields.

It is your choice, (and the compiler's!), whether to leave the second comma in.

Now all this is very well, but how do you use these values in your code? It is simply a matter of saying:

```
MOV   AX, [Date_I_Passed_Physics.Month]    ; or something like

MOV   [Date_I_Passed_Physics.Day], 5       ; or maybe even

CMP   [Date_I_Passed_Physics.Year], 1996
```

Simple, huh?

CREATING ARRAYS IN ASSEMBLER

Okay, arrays are pretty easy to implement. As an example, let's say you had the following array structure in Pascal:

```
Var
   MyArray : Array[0..19] Of Word;
```

To create a similar array in Assembler, you must use the DUP operator. DUP, or DUPlicate Variable, has the following syntax:

- <label>   <directive> <count> DUP (expression)

Where (expression) is an optional value to initialize the array to.

Basically, that Pascal array would look like this:

```
MyArray    DW 20 DUP (?)
```

Or, if you wanted to initialize each value to zero, then you could say this:

```
MyArray    DW 20 DUP (0)
```

And, as another example of just how flexible Assembler is, you could say something along the lines of:

```
MyArray    DB  0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

..to create a 10 byte array with all ten elements initialized to 1, 2, 3...

INDEXING ARRAYS IN ASSEMBLER

Well, now you've seen how to create arrays, I guess you are going to want to know how to reference individual elements. Well, let's say you had the following array:

```
   AnotherArray   DB  50 DUP (?)
```

If you wanted to move element 24 into, say, BL, then you could do this:

```
   MOV   BL, [AnotherArray + 23]   ; Or, it would be possible to say:

   MOV   AX, 23
   MOV   BL, [AnotherArray + AX]
```

NOTE: Do not forget that all arrays start at element ZERO. High-level languages like C and Pascal make you forget this due to the way they let you reference arrays.

Now that was easy, but what if AnotherArray was 50 WORDS, not BYTES?

```
   AnotherArray   DW  50 DUP (?)   ; like this.
```

Well, to access element 24, you would have multiply the index value by two, and then add that to AnotherArray to get the desired element.

```
   MOV   AX, 23                    ; Access element 24
   SHL   AX, 1                     ; Multiply AX by two
   MOV   BX, [AnotherArray + AX]   ; Get element 24 in BX
```

Not all that hard, no? However, this method gets a little tricky when you don't have nice neat little calculations to do when the index is not a power of two.

Let's say that you had an array that had an element size of 5 bytes.
If we wanted to check the seventh element, we'd have to do something like this:

```
   MOV   AX, 6                        ; Get the seventh element
   MOV   BX, 5                        ; Each element is five bytes big
   MUL   BX                           ; AX = 6 x 5
   MOV   DX, [YetAnotherArray + AX]   ; Get element 7 in DX
```

However, as I have stressed before, MUL is not a very efficient way of coding, so replacing the MUL with a SHL 2 and an ADD would be the order of the day.

Just before we press on with something else, I guess I'd better take the time to mention floating point numbers. Now, floating point numbers can get awkward to manipulate in Assembler, so don't go and write that spreadsheet program you've always wanted in machine code! However, when working with texture mapping, circles and other more complicated functions, it is inevitable that you'll need something to declare floating point numbers.

Let's say we wanted to store Pi. To declare Pi, we need to use the DT directive. You could declare Pi like this:

```
Pi   DT 3.14
```

DT actually reserves ten bytes of memory, so it would be possible to declare Pi to a greater number of decimal places.

I'm not going to go into the specifics of floating point numbers in this tutorial. When we need them later on, I'll cover them.

Okay, in the last tutorial I said I'd give some sort of summary of what we've covered over the last four months. (Hey - that's roughly a tutorial every two weeks, so maybe they haven't been so wildly erratic after all!)

Anyway, as it happens I'm going to go over getting and setting individual bits in a register, because this is an important topic that I should have covered a long time ago.

LOGICAL OPERATORS

Okay, way back in Tutorial Five, I gave the three truth tables for AND, OR and XOR.

(By the way, in one edition of Tutorial Five, I messed up the table for XOR, kindly pointed out by Keith Weatherby, so if you don't have the most up-to-date version, (currently V 1.3), then get it now. Please, although I try my best to weed out any mistakes from the Tutorials, some do get through, so if you spot any, please let me know.

Make sure you have the most recent editions of the tutorials before you do this though!)

Okay, enough of my mistakes. Those tables looked like these:

```
            AND              OR              XOR

        0 AND 0 = 0     0 OR 0 = 0     0 XOR 0 = 0
```

```
                  0 AND 1 = 0     0 OR 1 = 1     0 XOR 1 = 1
                  1 AND 0 = 0     1 OR 0 = 1     1 XOR 0 = 1
                  1 AND 1 = 1     1 OR 1 = 1     1 XOR 1 = 0
```

This is all very well, but what use can these be to us? Well, first of all, lets have a look at what AND can do. We can use AND to mask bits in a register or variable, and thus set and reset individual bits.

As an example, we will use AND to test a value of a single bit. Look at the following examples, and see how you can use AND for your own ends. A good use for AND would be to check if a character read from the keyboard is either a capital letter or not. (You can do this, because the only difference between a capital letter and a lowercase letter is one bit.

```
   EG:  'A' =  65   = 01000001
        'a' =  97   = 01100001

        'S' =  83   = 01010011
        's' =  115  = 01110011)
```

So, in the same way that you can AND the following binary numbers together, you could use a similar approach to write a routine that checks whether a character is upper or lower case.

```
   EG:         0101 0011                       0111 0011
           AND 0010 0000                   AND 0010 0000

             = 0000 0000                     = 0010 0000

      ^^^ This is upper case ^^^           ^^^ This is lower case ^^^
```

Now, what about OR? OR is most often used after an AND, but does not have to be. You can use OR to change individual bits in a register or variable without changing any of the other bits. You could use OR to write a routine to make a character uppercase if it is not already, or perhaps lower case if it was previously upper.

```
   EG:                            0101 0011
                               OR 0010 0000

                               =  0111 0011

            ^^^ Capital S has now been changed to lower case s ^^^
```

The AND/OR combination is one of the most often used tricks of the trade of Assembler, so make sure you have a good grip on the concept. You will often see me using them, taking advantage of the speed of the instructions.

Finally, what about XOR? Well, eXclusive OR can be very useful at times. XOR can be useful in toggling individual bits on and off without having to know what the contents of each bit was beforehand. Remember, as with OR, a zero mask allows the original bit to pass through.

```
   EG:                           1010 0010
```

```
XOR 1110 1011

  = 0100 1001
```

Make some attempt to learn these binary operators, and what they do. They are an invaluable tool when working with binary numbers.

NOTE: For simplicity, Turbo Assembler allows you to use binary numbers in your code. EG, it would be possible to say, AND AX, 0001000b instead of AND AX, 8h to test bit 3 of AX. This can possibly make things easier for you when coding.

THE DEMO PROGRAM

Okay, enough of the boring stuff - on to the demo program I included! I thought it was time to write another demo - a proper 100% Assembler one this time, and had a go at a fire routine. Fire routines can look pretty effective, and are surprisingly easy to make, so why not I thought...

Now, the principles of a fire routine are quite simple. You basically do the following:

- Create a buffer to work with

  This buffer may be almost any size, though the smaller you make it, the faster your program will be, and the larger you make it, the more well defined the fire will be. You need to strike a balance between clarity and speed.

  My routine is a little slow, and this is partly due to the clarity of the fire. I chose 320 x 104 as my buffer size, so I made a compromise. The horizontal resolution is good - 1 pixel per array element, but the vertical resolution is a little low - 2 pixels per array element.

  However, I've seen routines where an 80 x 50 buffer is used, meaning there is both 4 pixels per element for the horizontal and vertical axis. It's fast, but grainy.

- Make a nice palette

  It would be good idea to have color 0 as black, (0, 0, 0) and color 255 as white - (63, 63, 63). Everything in between should be a reddish-yellow flamey mix. I guess you could have green flames if you wanted, but we'll stick to the flames we know for now. :)

Now the main loop begins. In the loop you must:

■ Create a random bottom line, or two bottom lines

Basically, you have a loop like:

```
For X := 1 To Xmax Do
 Begin
    Temp := Random(256);
    Buffer[X, Ymax - 1] := Temp;
    Buffer[X, Ymax]     := Temp;
 End;
```

Code that in the language of your choice, and you're in business.

■ Soften the array

Now this is the only tricky bit. What you have to do, is as follows:

* Start from the second row down of the buffer.
* Move down, and for each pixel:
  * Add up the values of four pixels that surround the pixel.
  * Divide the total by four to get an average.
  * Take one from the average.
  * Put the average - 1 back into the array DIRECTLY ABOVE where the old pixel used to be. (You can alter this, and say, put it above and to the right, and then it will look like the flame is being blown by the wind.)
* Do this till you get to the last row.

■ Copy the array to the screen

If your array is 320 x 200, then you can copy element-for-pixel. If it isn't, then things are harder. What I had to do was copy an array row to the screen, move down a screen row, copy the same array row to the screen, and then go onto a different row in the array and screen.

This way, I spread the fire out a bit.

You will of course, wonder exactly why my array is 320 x 104 and not 320 x 100. Well, the reason for this is fairly simple. If I had used 320 x 100 as my array dimensions, and then copied that to the screen, the last four or so rows would have looked pretty weird. They would not have been softened properly, and the end result would not be at all flamey. So, I just copied up to row 100 to the screen, and left the rest.

As an experiment, try changing the third line below in the DrawScreen procedure to   MOV BX, BufferY   and changing the dimensions to 320x100 and see what happens.

```
MOV   SI, OFFSET Buffer          ; Point SI to the start of the buffer
XOR   DI, DI                     ; Start drawing at 0, 0
MOV   BX, BufferY - 4            ; Miss the last four lines from the
                                 ; buffer.  These lines will not look
                                 ; fire-like at all
```

■ Loop back to the top.

Well, no matter how well I explained all that, it's very hard to actually see what's going on without looking at some code. So now we'll step through the program, following what's going on.

Well, first of all, you have the header.

```
   .MODEL SMALL   ; Data segment < 64K, code segment < 64K
   .STACK 200H    ; Set up 512 bytes of stack space
   .386
```

Here, I have said that the program will have a code segment and data segment total of less than 128K. I go onto to give the program a 512 byte stack, and then allow 386 instructions.

```
  .DATA

CR        EQU 13
LF        EQU 10
```

The data segment begins, and I give CR and LF the carriage return and line feed values.

```
BufferX   EQU 320                        ; Width of screen buffer
BufferY   EQU 104                        ; Height of screen buffer

AllDone   DB CR, LF, "That was:"
          DB CR, LF
          DB CR, LF, "         FFFFFFFFF    IIIIIII     RRRRRRRRR    ..."
          DB CR, LF, "          FFF           III        RRR   RRR   ..."
          DB CR, LF, "          FFF           III        RRR   RRR   ..."
          DB CR, LF, "          FFF           III        RRRRRRRR    ..."
          DB CR, LF, "          FFFFFFF       III        RRRRRRRR    ..."
          DB CR, LF, "          FFF           III        RRR  RRR    ..."
          DB CR, LF, "          FFF           III        RRR   RRR   ..."
          DB CR, LF, "          FFF           III        RRR    RRR  ..."
          DB CR, LF, "         FFFFF        IIIIIII     RRRR    RRRR ..."
          DB CR, LF
          DB CR, LF
          DB CR, LF, "   The demo program from Assembler Tutorial 8. ..."
          DB CR, LF, "   author, Adam Hyde, at: ", CR, LF
          DB CR, LF, "     ■ blackcat@faroc.com.au"
          DB CR, LF, "     ■ http://www.faroc.com.au/~blackcat", CR, LF, "$"

Buffer    DB BufferX * BufferY DUP (?) ; The screen buffer

Seed      DW 3749h                      ; The seed value, and half of my
                                        ; phone number - not in hex though. :)

INCLUDE PALETTE.DAT                     ; The palette, generated with
                                        ; Autodesk Animator, and a simple
                                        ; Pascal program.
```

Now, at the end, I declare the array and declare a SEED VALUE for the Random procedure that follows. The seed is just a number that is necessary to start the Random procedure off, and can be anything you want it to.

I have also saved some space and put the data for the palette into an external file which is included during assembly. Have a look inside the file. Being able to use INCLUDE can save a lot of space and confusion.

I've skipped through some procedures that are fairly self-explanatory, and moved onto the DrawScreen procedure.

```
DrawScreen PROC
   MOV   SI, OFFSET Buffer             ; Point SI to the start of the buffer
   XOR   DI, DI                        ; Start drawing at 0, 0
   MOV   BX, BufferY - 4               ; Miss the last four lines from the
                                       ; buffer.  These lines will not look
                                       ; fire-like at all
Row:
   MOV   CX, BufferX SHR 1             ; 160 WORDS
   REP   MOVSW                         ; Move them
   SUB   SI, 320                       ; Go back to the start of the array row
   MOV   CX, BufferX SHR 1             ; 160 WORDS
   REP   MOVSW                         ; Move them
   DEC   BX                            ; Decrease the number of VGA rows left
   JNZ   Row                           ; Are we finished?
   RET
DrawScreen ENDP
```

This is also easy to follow, and takes advantage of MOVSW, using it to move data between DS:SI and ES:DI.

```
AveragePixels PROC
   MOV   CX, BufferX * BufferY - BufferX * 2  ; Alter all of the buffer,
                                              ; except for the first row and
                                              ; last row
   MOV   SI, OFFSET Buffer + 320              ; Start from the second row

Alter:
   XOR   AX, AX                        ; Zero out AX
   MOV   AL, DS:[SI]                   ; Get the value of the current pixel
   ADD   AL, DS:[SI+1]                 ; Get the value of pixel to the right
   ADC   AH, 0
   ADD   AL, DS:[SI-1]                 ; Get the value of pixel to the left
   ADC   AH, 0
   ADD   AL, DS:[SI+BufferX]           ; Get the value of the pixel underneath
   ADC   AH, 0
   SHR   AX, 2                         ; Divide the total by four

   JZ    NextPixel                     ; Is the result zero?
   DEC   AX                            ; No, so decrement it by one
```

NOTE: ONE is the decay value. If you were to change the line above to, say

```
       SUB AX, 2  you would find that the fire would not reach so high.
       Experiment...be creative!  :)


NextPixel:
   MOV   DS:[SI-BufferX], AL           ; Put the new value into the array
   INC   SI                            ; Next pixel
   DEC   CX                            ; One less to do
   JNZ   Alter                         ; Have we done them all?
   RET
AveragePixels ENDP
```

Now we've seen the procedure that does all the softening. Basically, we just have a loop that adds up the color values of the pixels around it, carrying the values of the pixels before. When it has the lot, the total - held in AX, is divided by four to get an average. The average is then plotted directly above the current pixel.

For more information regarding the ADC instruction, look it up in Tutorial 5, and look at the programs below:

```
   Var                                      Var
      W : Word;                                W : Word;

   Begin                                    Begin
      Asm                                      Asm
         MOV  AL, 255                             MOV   AL, 255
         ADD  AL, 1                               ADD   AL, 1
         MOV  AH, 0                               MOV   W, AX
         ADC  AH, 0                            End;
         MOV  W, AX
      End;                                     Write(W);
                                            End;
      Write(W);
   End;

 ^^^ This program returns 256             ^^^ This program returns 0
```

Remember that ADC is used to make sure that when a register or variable is not big enough to hold a result, the result is not lost.

Okay, after skipping a few more irrelevant procedures, we come to the main body, which goes something like this:

```
Start:
   MOV   AX, @DATA
   MOV   DS, AX                        ; DS now points to the data segment.
```

We firstly point DS to the data segment, so we can access all our variables.

```
   CALL  InitializeMCGA
   CALL  SetUpPalette

MainLoop:
```

```
   CALL  AveragePixels

   MOV   SI, OFFSET Buffer + BufferX * BufferY - BufferX SHL 1
   ; SI now points to the start of the second last row
   MOV   CX, BufferX SHL 1             ; Prepare to get BufferX x 2 random #s

BottomLine:
   CALL   Random                       ; Get a random number
   MOV    DS:[SI], DL                  ; Use only the low byte of DX - ie,
   INC    SI                           ; the number will be 0 --> 255
   DEC    CX                           ; One less pixel to do
   JNZ    BottomLine                   ; Are we done yet?
```

Here, a new bottom line is calculated. The random procedure - many thanks to it's unknown USENET author - returns a very high value in DX:AX. However, we only require a number from 0 to 255, so by using only DL, we have such a number.

```
   CALL  DrawScreen                    ; Copy the buffer to the VGA

   MOV   AH, 01H                       ; Check for keypress
   INT   16H                           ; Is a key waiting in the buffer?
   JZ    MainLoop                      ; No, keep on going

   MOV   AH, 00H                       ; Yes, so get the key
   INT   16H

   CALL  TextMode
   MOV   AH, 4CH
   MOV   AL, 00H
   INT   21H                           ; Return to DOS
END Start
```

And I think the end part is also pretty easy to understand. I've tried to comment the source as much as I can, perhaps a little too heavily in some parts, but I hope by now everyone has an idea of how a fire routine works.

Anyway, the goal was not to teach you how to make a fire routine, but how to use arrays, so if you got the fire routine stuff too, then that's an added bonus. I referred to my arrays slightly differently to how I explained in this tutorial, but the theory is still the same, and it shows you other ways of doing things. If you didn't get how to use arrays from that, then maybe you never will, at least not with my tutorials anyway. Hey, go buy a $50 book! :)

Next week's tutorial will include:

- File I/O
- Using Assembler with C/C++
- Lookup tables?
- Macros.

If you wish to see a topic discussed in a future tutorial, then mail me, and I'll see what I can do.

Don't miss out!!! Download next week's tutorial from my homepage at:

- http://www.faroc.com.au/~blackcat

See you next week!

- Adam.

Yet another joke I grabbed off a local BBS:

If God Was A Computer Programmer:

Some important theological questions can best be answered by thinking of God as a computer programmer.

Q: Did God really create the world in seven days?
A: He did it in six days and nights while living on cola and candy bars.
On the seventh day he went home and found out his girlfriend had left him.

Q: What causes God to intervene in earthly affairs?
A: If a critical error occurs, the system pages him automatically and he logs on from home to try to bring it up. Otherwise, things can wait until tomorrow.

Q: How come the Age of Miracles ended?
A: That was the development phase of the project.
Now we're in the maintenance phase.

Q: Who is Satan?
A: Satan is an MIS director who takes credit for more powers than he actually possesses, so nonprogrammers become scared of him. God thinks he's irritating but irrelevant.

Q: Why does God allow evil to happen?
A: God thought he eliminated evil in one of the earlier revs.

Q: How can I protect myself from evil?
A: Change your password every month and don't make it a name, a common word, or a date like your birthday.

Q: If I pray to God, will he listen?
A: You can waste his time telling him what to do, or you can just get off his back and let him program.

Q: Some people claim they hear the voice of God. Is this true?
A: They are much more likely to receive email.