Programming in machine code on the Amiga

A. Forness & N. A. Holten

Issue 7

Content

DATA SCHOOL
Postbox 62
Nordengen 18
2980 Kokkedal

Phone:	49 18 00 77
Postgiro:	7 24 23 44

BLITTER III

In this chapter, we look at scrolling. In the program examples MC0701 and MC0702, we will demonstrate scrolling of a text. Let us first explain how it works in theory.

A character set which contains the alphabet, plus some other characters (period, comma, parentheses, etc.) is often called a font. A font is required to make a scroller - or in other words: You need a complete set of characters – an alphabet - to create a scroller.

To scroll, e.g., the word "Amiga" across the screen you shall proceed as follows:
Put the character "A" (from your own or any other finished font) to the right screen edge. Notice that we must insert the character at the right side. Starting from the left - which of course also can be done technically, you can not read the word until all the characters are scrolled into the screen. Furthermore it is difficult to read a text scrolling left to right. Scrolling from the right to the left side is the normal order in which text is read and the characters appear in the correct sequence.

The scroller passes the entire screen (all display data) by 1 pixel (BIT) from right to the left. This is repeated as many times as characters are wide, e.g., 16 times if the "A" is 16 pixel wide.

After scrolling the character its width to the left, there is again a free space at the right screen edge and we can go on with the next character. This continues character by character until the entire text disappeared into the left screen edge. That way you get a regular text-scroller across the screen.

You can imagine it as an assembly-line in a factory where you put characters on. When it is transported far enough to the left the next character is added. When a character reached the end of assembly-line, the characters are removed so that the characters do not pile up.

This also explains the technique although a bit simplified. Before we continue scrolling, we have another look at a concept in the data world, i.e. ASCII codes. In connection with computers it is a well-known expression. When you press a key down on the keyboard, it becomes registered and transformed into a so-called ASCII codes inside Amiga OS.

ASCII is an acronym for "**A**merican **St**andard **C**ode for **I**nformation **I**nterchange". Let us show some examples. A (capital) "A" has the value (ASCII code) 65 (decimal). The space key has the value (ASCII code) 32. Have a look at the table at the end of the issue which contains all the ASCII codes for a complete alphabet.

The font, which is used in the program examples, does not cover all ASCII characters. Instead of replacing the lacking characters with white-space, we have created another sequence of characters in the font for these ASCII-codes.

Here is how the characters in our FONT are set up:

A	B	C	D	E	F	G	H	I	J	K	L	M
0	1	2	3	4	5	6	7	8	9	10	11	12
N	O	P	Q	R	S	T	U	V	W	X	Y	Z
13	14	15	16	17	18	19	20	21	22	23	24	25
[	\	]	,	.								
26	27	28	29	30	31							

The order and values have changed from the standard (or normal) ASCII code, so we must convert ASCII codes for the values we use in our font. The character "C" is 67 in ASCII code, and it must change to 3 in our font.

We are begin with the program example MC0701. The code is not printed in tthis issue because of lack of space. You can find it on the course disk # 1 in the directory 'BREV07'.

Line 1:	Turn off all interrupts.
Line 3-4:	Stops the floppy motor.
Line 6:	Turn off the bitplane, copper and sprite-DMA.
Line 8-17:	Set up a 352 *256 pixel screen with a BITPLANE and with a value 2 for the MODULO. The modulo for bitplanes and how it works is explained later in this issue.
Lines 19-25:	Moves the address of the bitplanes into the copper-list.

Lines 27-28:	Moves the copper-list address into the copper register
Line 29:	Starts bitplane and copper-DMA again
Line 31:	Here begins the main loop
Line 32-36:	Waiting for the electron beam to reach screen line number 300
Line 38:	Branch to sub-routine "scroll"
Lines 40-41:	Check whether left mouse button is pressed, if not, the program branches back to "main loop"
Line 43:	Turns off copper-DMA
Line 45-47:	Moves the original address of the system copper-list into copper-register
Line 49:	Starts copper and sprite-DMA
Line 51:	Enable all interrrupts again
Line 54-58:	Here we reserved a word to be used as a scroll-counter
Line 60:	Here begins the scroll- routine
Line 61:	Loads the effective address of "scrollcnt" into A1
Line 62:	Compares "scrollcnt" with value 8
Line 63:	If the value of "scrollcnt" is not equal to 8 branch to label "nochar"
Line 65:	Clears the "scrollcnt" (set to 0)
Line 67:	Load the effective address of "charcnt" to A1
Line 68:	Move the value A1 points to ("charcnt") to D1
Line 69:	Increase "charcnt" by 1; note that the value 1 is directly copied into the address A1 points to and not in D1!
Line 71:	Loads the effective address of the "text" into A2.
Line 72:	Clears D2 (sets to 0)

Line 73: Moves the content, the address in A2 + D1 points to, into D2. This instruction retrieves an ASCII code from your own text ("text"), where OFFSET to the text is in D1. So if D1 contains 0, the first letter will be placed in D2. If D1 contains 1, it will move the second letter, etc.

Line 75: Check if D2 is equal to 42 ASCII code 42 is a "*". This "*" must be the last character in the text to indicate that the text is finished.

Line 76: If we have NOT reached the last character, then branch to the label "notend".

Lines 78-79: If we reached the last character in the text, we reset "charcnt" to 0 and move an value of 32 into D2 (ASCII code for space = 32).

Line 82: Loads the effective address of the conversion-table "convtab" to A1. This table contains the letters (characters) position in our font.

Line 83: Moves the content, the address in Al + D2 points to, into register D2. The old value of D2 (which contained ASCII code for the character) is used here as an offset for the table. So: If D2 contains 32 (a space), the 33rd byte (remember counts from 0) from the table is moved to D2. In this case $1F or 31 decimal was the offset. We now have created the ASCII code for our own font and used only a single instruction!

Line 84: The contents of D2 shifted one bit to the left. This is the same as a multiplication by 2. It must be made because a character in FONT is 16 pixels (2 bytes) wide.

Line 86: Loads the effective address of the "font" to A1

Line 87: Add the value of D2 to the address in A1 – now A1 points to the first character which in turn will be copied on the screen.

Line 89: Loads the effective address of the display ("screen") into the A2

Line 90: Add a constant value of 6944 to A2. As mentioned, the screen 352 pixels wide (including 16 pixel overscan on each side) and 2 of modulo. So a screen line has 46 bytes (352 / 8 + 2). With bitplanes modulo works the same way as with the blitter. Because the visible screen width set to 352 pixels (the same as 44 bytes), there will be two bytes left over per screen line. When we define modulo of 2 the BITPLANE-DMA jumps over 2 bytes at the end of each line. This results in that we have 2 bytes of invisible display data on each line. In this invisible area, we copy the characters which must be scrolled. When we do it this way, it seems as if the characters scroll out of the edge of the screen. So: When we add a constant of 6944 to the screen address we reach line 150 and the byte position 44 (150 * 46 +44 = 6944).

Line 92: Moves quickly the constant value of 19 into D0. D0 is used as a counter to count the height of the character to be added into the screen (display memory). The font we use here has characters which are 20 pixels high.

Line 94: Here begins loop to put the characters from the font into screen memory.

Line 95: Move the value, A1 points to (position in the font area), to the address A2 points to (certain position in the screen).

Line 96: Adds 64 to the address A1. Let A1 point to next line in font (32 characters * 16 pixels = 512; 512 / 8 = 64).

Line 97: Adds 46 to the address in A2 (certain position in the screen). A2 points now on the next line in our video memory.

Line 98: Decrement D0 by 1, check if D0 is -1, if not branch back to the label "putcharloop".

Line 101-102: Waiting until the blitter is available.

Line 104: Loads the effective address of the screen ("screen") to A1

Line 105: Add the constant value of 7820 to the address in Al. Al will now point to the bottom of the line of the scroller (46 * (150 +20) = 7820). It is customary to point to the upper left corner of blitt, but in this case we must point to the bottom (last) line, which must be blitted. This is because we are using the descending mode (i.e. to run blitter "backwards" so that it counts downwards in memory). If we would use the blitter in normal modus the text would scroll the wrong way.

Line 107: Moves the address from A1 into the blitter A-channel register

Line 108: Moves the address from A1 into the blitter D-channel register

Line 109: Sets A-modulo to 0

Line 110: Sets also D-modulo to 0

Line 111: Sets A-mask to $FFFFFFFF

Line 112: Sets the logical operation D = A, giving 2 bits Shift on the A channel. It is this value containing the number of pixels, to be scrolled at a time. In this case we scroll pixel pairs. If you want to scroll slower, you can set it to 1, and change the number from 8 in the program line 62 to 16

Line 113: Bit 1 in BLTCON1 register is set to "1" to indicate the descending mode

Line 114: The value of $5017 is moved into the BLTSIZE register (width = 23 word and height = 20 lines). The value is calculated from following: 20 * 64 +23 = $5017). This instruction starts the blitter automatically

Line 116: Load the effective address of "scrollcnt" to A1.

Line 117: Increase "scrollcnt" by 1

Line 119: Branch to the calling instance – here to line 38, and continue the program execution there

Line 122-135: The copper list is declared here

Line 138: Here we have reserved memory for the screen

Line 141: Here memory for the font is reserved

Line 144-237: Here is the conversion table used to change from the ASCII code of characters to position in our font

Line 240: Here is the scroll text which of course can be edited as you like. Add noticed that we have used special characters. We have done so because it can be difficult to get these letters to all keyboards. So, when you need an "Æ" type "@" you will have a "0" use "#" and "A" is a "$" sign. Do not forget to write a "*" character to finish the text.

To run this program you must first assemble it. Then attach the file "font" to the label "FONT" as follows:

```
Seka> a
OPTIONS>
No errors
Seka> ri
FILNAME> font
BEGIN> font
END> (just press RETURN)
Seka> j
```

BLITTER IV

Our next program example - MC0702 - is quite similar to the first (MC0701) except that we have added a so-called color-cycling or rather a copper-cycling effect to it.

This effect creates a colorful pattern within the scrolling text. This is done by getting the copper to wait on a special screen line, and then set text color to red for example.

Then let the copper wait on the next screen line to put a new color into the same color register which is used by the text. This is repeated 20 times (the font of our text is 20 pixel high). In this way we can produce a nice color pattern for our text.

The color cycling is produced by updating the copper list all the time. The update of the copper list is combined with a shift in the color table, which gives a nice effect on the screen.

We review now the cycle routine for the program example MC0702. The rest of the program is similar to the MC0701 so there is need for further explanations.

Line 40:	This instruction is added to the main routine to update color-cycling
Line 124:	Here we have declared a word which holds the number of cycles for the cycle-routine
Line 126:	Here begins the cycle routine
Line 127:	Loads the effective address of "cyclecnt" to A1
Line 128:	Moves the value of "cyclecnt", A1 points to, to D1
Line 130:	Adds value to 2 to the value of "cyclecnt", A1 points to
Line 132:	Compares D1 with the value 96
Line 133:	If D1 is NOT EQUAL to 96, then BRANCH to the label "notround"
Line 135:	Clears "cyclecnt" (sets it to 0)
Line 136:	Clears D1 (sets it to 0)
Line 139:	Load the effective address of the "cycle table" into A1. The "cycletable" is the table of colors which are to be rolled in text.
Line 140:	Loads the effective address of "cyclecop" into A3. This is the beginning of the copper-instructions which change the contents of the color register. Read copper-list. (line 153-208) to understand how the following lines of code work
Line 142:	Moves the constant value of 19 quickly into D0, which serves as a counter for the number of color lines which has to be changed.
Line 144:	Here begins the loop which copied the color data into the copper-list.
Line 145:	Move the value where the address of A1 + D1 points to, to the address A3 +6 points to. Study the color table ("cycletable") and the copper-list carefully and try to understand how the values are fetched and stored to the copper list
Line 146:	Adds 2 to D1 quickly. Next time the program executes the line 145, it will start with the next color from the color table. We add the value of 2 since each color in the table is defined as a word (2 bytes)
Line 147:	Adds 8 to A3. A3 will now point to the next space in the copper-list where the next color must be inserted.

Line 148: Decrement D0 by 1, checks if -1 and if not, branch back to the "cycleloop". Note that this loop (line 145-148) is performed 20 times.

Line 150: Branched back to the calling instance – here the main routine

Line 203: Here is the definition of the copper-instructions which makes the color-scroll possible. These copper-instructions can be explained as follows:
- Wait until the electron beam has reached Line $C2 (194)
- Move color "xxx" (what color at this moment since we change it all the time) into COLOR01 register ($ DFF182). This is our text color therefore it is visible wherever a character of the text is on the screen.
- Wait until the electron beam reached line $C3 (195) ... and so on and so forth.
This is repeated 20 times in copper-list and results in 2 different colored lines on the screen.

Line 313-322: Here the color table is declared.

FONTS

In this chapter we will look at Fonts. A font is another name of a character set - even if you made one yourself. As you know, we need a FONT for the scroller.

In the programs MC0701 and MC0702, we use a font that is 16 pixels wide and 20 pixels high, but it is perfectly acceptable to create a font that is either smaller or larger. In our code example we only have one bitplane per Font. This makes the font simple and we only need one color register.

In the example program MC0702, we used only one bitplane - but we "decorated" the font by changing the colors with the copper. If you make a font with more colors, you must of course remember to scroll all bitplanes synchronously (i.e. with the same speed).

Let us begin by describing how to create your own font. When you draw your own font you can use, e.g., Deluxe Paint, which is a widely used graphics program for the Amiga). Before you draw, you should begin by finding out what special characters you want to have (period, comma, hyphen, question mark, etc.).

Then you must not forget the space-characters (which can be easily forgotten). It is just a blank "square".

When you are finished drawing, you must put the characters in an appropriate order. In our font we have 32 characters (the entire alphabet plus commas, periods and spaces). We chose to put all characters in one horizontal row. So, "A" placed at the top left corner of the screen. "B" placed 16 pixels to the right of "A", etc.

The image size we used was calculated as follows: 32 (characters) * 16 (pixels) = 512 pixels wide and 20 pixels high. We now calculate the font's memory footprint (512 / 8) * 20 = 1280 bytes of memory.

Because we put all the characters in a row, we only need to know the x-position in font when it is converted from ASCII code to the "font-position". If you draw small and capital as well as special characters in your font, it will be difficult to get all the letters in a row. You must make several rows. This means you must use both the x and y position when you convert the ASCII code to your "font-position".

A common problem when you make a FONT is that some letters are wider than others (e.g., a "W" is broader than an "I").

This means that there will be varying distances between the letters when scrolling the screen (it does not look very nice).

One way to avoid this is to draw all characters with the same width. A second and better (i.e. nicer) method is that you "press" them together scrolling the screen. This is called proportional spacing. Let us show this with an example.

- The characters in the font is 16 pixels wide.

- Assume that the "W" takes 14 pixels wide.

- Assume that the "I" takes 8 pixels wide.

First scroll the characters "WWWWW" across the screen. This is done as follows:

Copy the character "W" to the right edge of the screen and scroll the screen 16 pixels to the left. Put the next "W" to the screen and scroll 16 more pixels.

Thus, you continue as long as you want.

When we scrolled 16 pixels between each "W" there was 2 pixels between the characters (which is quite appropriate).

Let us now scroll letters "IIIII" across the screen.

To get the "I" s to look prettier, we need only scroll 10 pixels (this value is valid for our font) between each character (8 pixels wide, plus 2 pixels between characters). So, copy an "I" on the screen, scroll 10 pixels, set the next "I" at screen, etc. If we now want SCROLL initials "IW" across the screen we will take the proportionality into account:

Copy the "I", SCROLL 10 pixels, set the "W" and SCROLL 16 pixels.

It looks like the machine must know how many pixels each single character must be scrolled. These values can be put in the table, which is used to convert the ASCII codes, so that you get both the position and the width of the character as converted. Try to make your own font and a program that makes use of proportional spacing.

We do not say it is easy, but it is surely possible if you try to keep your code in order have a good overview.

Use many sub-routines which makes the program more structured and thus easier to read.

MACHINE CODE VI

We start this chapter on machine code on the AMIGA to explain what a signet value is and how it behaves when you need these numbers in programming.

You've already heard of unsigned and signed values, but what are these exactly? Let us explain this with the help of a byte (8 bits, as usual). A byte is unsigned and can contain values from 0 to 255 These values can not be considered either positive or negative – they indicate only a number within the given limits.

A signed byte can contain both positive and negative values. It can contain values from -128 to 127, so from MINUS 128 to PLUS 127th (total 256 values).

Let us first show some examples of values (we explain more then):

binary	hexadecimal	unsigned	signed
00000000	$00	0	
00010111	$17	23	
11111111	$FF	255	-1
11111110	$FE	254	-2
10000000	$80	128	-128
10000001	$81	129	-127
10000010	$82	130	-126
01111111	$7F	127	
01111110	$7E	126	

You probably understand how it works already? So: the most significant bit – the first far left in the bit group which in this case is BIT 7, determines whether the value must be positive or negative.

If it is "0", the value will be positive or zero (0 to 127). However, if it is "1", the value will be negative (-1 to -128). It is perhaps not so easy to understand the seven other bits.

Let us study this a little closer.
If the value is positive (BIT 7 is "0") the remaining BIT 0-6 show directly the value.
If the value is negative (BIT 7 is "1"), you need to make a little trick for that we need to see what the value actually is.

Let us take the value 11101001 (= 233). We see immediately that it is negative but the negative value it not immediately visible has (perhaps some experienced programmers can see it). This can be seen in the following example:

11101001 = 233

First invert all bits ("1" becomes "0" and vice versa), after that our byte looks like this:

00010110 = 22

Then we add 1 to the Byte and it looks as follows:

00010111 = 23

We now have the value 23, so 11101001 is equal to -3, simple, right? One just inverts the bits and add 1 to the outcome… Let's try to go the other way:

00010111 = 23

Invert:

11101000 = 232   Add 1 to the value...

11101001 = 233

As we see, the value is equal to the one we started with.

The method of inverting and adding 1 to is called the "TWO-Compliment". All computers are using the 2-compliment.

There is also another method for sign numbers which is not used by computers, the so-called "ONE-complement" (1-compliment). Let's quickly have a look at this method:

In 1-compliment inverted value, but there is nothing added to.

We get as follows:

00000001 = 1

Inverted it becomes:

11111110 = -1

An example perhaps?

00000000 = 0

inverted, it will be ...

11111111 = -0

What happened? When we inverted 00000001, the result was 11111110 as it must be -1. So far so good.

But what happens when we invert the bits of 00000000? Yes, we get 11111111 as a result, which shall then be -0! Now you probably understand why computers do not use the "1-compliment" (ONE-complement).

We keep going with the 2-compliment.

We have now seen that a signed byte can contain values from -128 To 127 (minus 128 to plus 127), but what can one signed word (16 bit) contain?

We've created a small table that illustrates this. Remember it is always the leftmost bit which indicates whether the value should be regarded as positive or negative (in a word bit No. 15 determines the sign).

TYPE	#BITS	SIGNED RANGE	UNSIGNED RANGE
BYTE	8 bits	-128 to 127	0 to 255
WORD	16 bits	-32768 to 32767	0 to 65535
L0NGW0RD	32 bits	-2147483648 to 2147483647	0 to 4294967295

We've created a small program example that you can experiment with:

```
        CLR.L       D0
        M0VE.B      #-50, D0
        ADD.B       #70, D0
        SUB.B       #5, D0
        .....
        RTS
```

Experiment also using different data types (word and longwords - ".w" and ".l"), to see what happens with D0's content

You probably wonder how the machine knows if it will treat numbers as signed or Unsigned? Instructions ADD and SUB will always deal with numbers/ data as a signed value. The rule is that all instructions that perform mathematical calculations treat the values as signed values.

Two exceptions are instructions MULU and DIVU. These instructions mean: MULTIPLY unsigned (MULtiply Unsigned) and DIVIDE unsigned (DIVide Unsigned). The signed versions are MULS and DIVS. We will come back with a more thorough explanation of mulu, muls, divu and divs in a later issue.

So when it comes to multiplication and division you can choose whether you want to work signed or unsigned, while addition and subtraction are always interpret the values as signed.

This was the end of the machine code section of this letter. Rehearse this until the next issue comes out - sooner or later you will use it in your own programs. Good luck!

SOLUTIONS FOR TASKS IN ISSUE VI

Task 0601: The word MODULO is difficult to explain with one word, a fairly translation is: appendix, skip, offset or remainder.

Task 0602: A blitt with BLTSIZE of $la69 is 105 Pixels (lines) high and 656 Pixels (BITS) wide.

Task 0603: "0" will be inverted (logical NOT, "0" becomes "1" and vice versa).

Task 0604: The minimum number of pixels horizontally that can be defined in BLTSIZE is 16 (Remember: A WORD, 16 bits or 16 pixels).

TASKS FOR ISSUE VII

Task 0701: Try to change the scroll-speed, color in color table, etc. So, experiment with the scroll program!

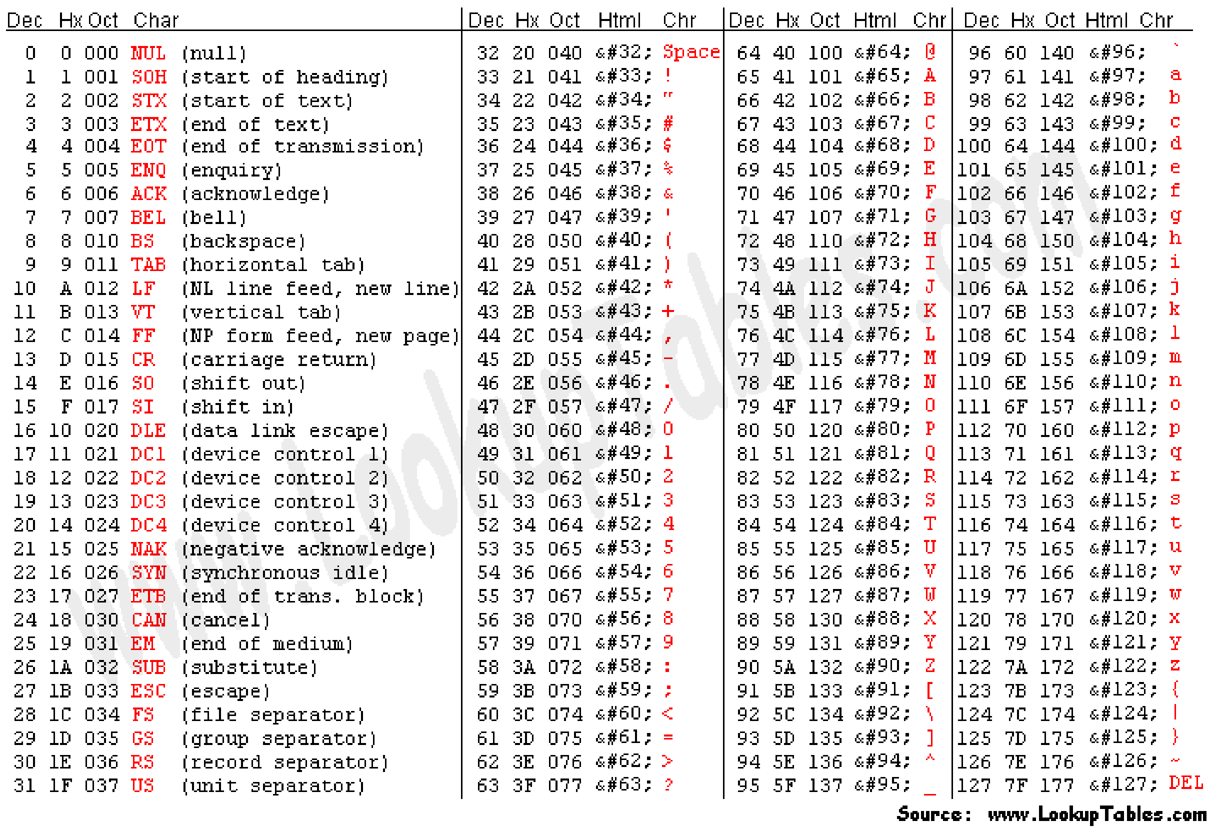
Task 0702: What are these binary numbers (BYTE length) Decimal, signed and unsigned respectively: % 01101001, %1011110, %11110001, %00101101, %10001011 and %11010011

Task 0703: What is the highest and the lowest value, a signed bit group of 20 bits can represent?

Task 0704: Try to write a program that converts a positive to a negative value (signed value) and vice versa (without using "muls" or "not").

DATA DICTIONARY

SIGNED	A bit-group which is signed can represent positive and negative values.
UNSIGNED	A bit-group which is unsigned can represent positive values only - value 0 is also included here.
OFFSET	Indicates a distance between two points in your program (actually the distance between two addresses in memory).
HAM	“Hold and modify” a color register
MODULO	A modulo is the remainder, skip or appendix. It is used to obtain a proper distance to the next data position, e.g., for the blitter.
STACK	Can be translated as pile. It is a memory area used by computers to store data temporarily.
INVERT	Changing from 1 to 0 or vice versa.
MASK	With a "mask" a bit masked is meant here. It is used to identify one or more bits in a bit group (BYTE, WORD, LONG WORD) to which are left unchanged depending on the chosen instruction.
SUBROUTINE	A small (or large) part of the program, that performs a specific task.



Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040	&#32;	Space	64	40	100	&#64;	@	96	60	140	&#96;	`
1	1	001	SOH	(start of heading)	33	21	041	&#33;	!	65	41	101	&#65;	A	97	61	141	&#97;	a
2	2	002	STX	(start of text)	34	22	042	&#34;	"	66	42	102	&#66;	B	98	62	142	&#98;	b
3	3	003	ETX	(end of text)	35	23	043	&#35;	#	67	43	103	&#67;	C	99	63	143	&#99;	c
4	4	004	EOT	(end of transmission)	36	24	044	&#36;	$	68	44	104	&#68;	D	100	64	144	&#100;	d
5	5	005	ENQ	(enquiry)	37	25	045	&#37;	%	69	45	105	&#69;	E	101	65	145	&#101;	e
6	6	006	ACK	(acknowledge)	38	26	046	&#38;	&	70	46	106	&#70;	F	102	66	146	&#102;	f
7	7	007	BEL	(bell)	39	27	047	&#39;	'	71	47	107	&#71;	G	103	67	147	&#103;	g
8	8	010	BS	(backspace)	40	28	050	&#40;	(	72	48	110	&#72;	H	104	68	150	&#104;	h
9	9	011	TAB	(horizontal tab)	41	29	051	&#41;	)	73	49	111	&#73;	I	105	69	151	&#105;	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	&#42;	*	74	4A	112	&#74;	J	106	6A	152	&#106;	j
11	B	013	VT	(vertical tab)	43	2B	053	&#43;	+	75	4B	113	&#75;	K	107	6B	153	&#107;	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	&#44;	,	76	4C	114	&#76;	L	108	6C	154	&#108;	l
13	D	015	CR	(carriage return)	45	2D	055	&#45;	-	77	4D	115	&#77;	M	109	6D	155	&#109;	m
14	E	016	SO	(shift out)	46	2E	056	&#46;	.	78	4E	116	&#78;	N	110	6E	156	&#110;	n
15	F	017	SI	(shift in)	47	2F	057	&#47;	/	79	4F	117	&#79;	O	111	6F	157	&#111;	o
16	10	020	DLE	(data link escape)	48	30	060	&#48;	0	80	50	120	&#80;	P	112	70	160	&#112;	p
17	11	021	DC1	(device control 1)	49	31	061	&#49;	1	81	51	121	&#81;	Q	113	71	161	&#113;	q
18	12	022	DC2	(device control 2)	50	32	062	&#50;	2	82	52	122	&#82;	R	114	72	162	&#114;	r
19	13	023	DC3	(device control 3)	51	33	063	&#51;	3	83	53	123	&#83;	S	115	73	163	&#115;	s
20	14	024	DC4	(device control 4)	52	34	064	&#52;	4	84	54	124	&#84;	T	116	74	164	&#116;	t
21	15	025	NAK	(negative acknowledge)	53	35	065	&#53;	5	85	55	125	&#85;	U	117	75	165	&#117;	u
22	16	026	SYN	(synchronous idle)	54	36	066	&#54;	6	86	56	126	&#86;	V	118	76	166	&#118;	v
23	17	027	ETB	(end of trans. block)	55	37	067	&#55;	7	87	57	127	&#87;	W	119	77	167	&#119;	w
24	18	030	CAN	(cancel)	56	38	070	&#56;	8	88	58	130	&#88;	X	120	78	170	&#120;	x
25	19	031	EM	(end of medium)	57	39	071	&#57;	9	89	59	131	&#89;	Y	121	79	171	&#121;	y
26	1A	032	SUB	(substitute)	58	3A	072	&#58;	:	90	5A	132	&#90;	Z	122	7A	172	&#122;	z
27	1B	033	ESC	(escape)	59	3B	073	&#59;	;	91	5B	133	&#91;	[	123	7B	173	&#123;	{
28	1C	034	FS	(file separator)	60	3C	074	&#60;	<	92	5C	134	&#92;	\	124	7C	174	&#124;	|
29	1D	035	GS	(group separator)	61	3D	075	&#61;	=	93	5D	135	&#93;	]	125	7D	175	&#125;	}
30	1E	036	RS	(record separator)	62	3E	076	&#62;	>	94	5E	136	&#94;	^	126	7E	176	&#126;	~
31	1F	037	US	(unit separator)	63	3F	077	&#63;	?	95	5F	137	&#95;	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com