

HowToCode7

COLLABORATORS

	<i>TITLE :</i> HowToCode7	
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>
WRITTEN BY		November 28, 2024
<i>SIGNATURE</i>		

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

1	HowToCode7	1
1.1	HowToCode: Reading C	1
1.2	EnConn BBS	1
1.3	Introduction	2
1.4	Braces and Brackets	2
1.5	Define, UBYTE, USHORT, etc..	5
1.6	GetOffset (->), Index (.) Address (&)	8
1.7	<< >> = == ! && > < . + ++ +=	10
1.8	Loops	12
1.9	Which INCLUDE files?	12
1.10	What if the C code uses amiga.lib routines!?	13
1.11	Complete example - Bitmap Scaling using the OS	13

Chapter 1

HowToCode7

1.1 HowToCode: Reading C

READING C FOR ASSEMBLER PROGRAMMERS

This file has been left intact as I received it for HTC (apart from AmigaGuide Conversion) There are a few errors in it, but in general it's pretty good. The example programs referred to are in source/codemanual/ (but don't expect them to work!) - CJ

-----=====The EnConn BBS Better Programming Guide=====-----

Introduction: Read C, write assembler?

- 1 Braces {} and brackets () and square brackets []
- 2 Define, UBYTE, APTR, USHORT, STRUCT, BPTR, etc..
- 3 << >> == || | && > < . + ++ +=
- 4 GetPointer (->), Index (.), Address (&), Pointers (*) etc
- 5 C loops
- 6 Which INCLUDE file to include?
- 7 What if the C code uses Amiga.lib routines!?
- 8 A Complete Example converting a C source to ASM
- 9 The Example (using 2.04 bitmap scaling)

1.2 EnConn BBS

Call EnConn, The Engineering Connection on +612 524-1584

ASM coding our speciality. C coders welcome.

Home of coders such as:

Piranha, Coz, BigMac, Drizzt, Questa, Punisher, Wildfire, Ziggy.

Enconn BBS is Real-Names-Only and Legal-Only .

Running MaxsBBS V1.52 Enhanced :)

Fidonet # 3:712/613

AmigaNet # 41:200/584

MaxsNet # 3:30000/99

1.3 Introduction

INTRODUCTION

Some of the most useful utilities have come from the demo coders scene. Mostly they serve the purpose they were designed for but lack one thing. A graphical user interface or even just a friendly interface. How many ST module rippers have you seen that just use keyboard input from a cli window? Lame aren't they! All this builtin code going to waste. Instead they get out their Abacus machine language book & just copy the asm code to get keyboard input in a simple console window! Totally lame.

Ok, heres the reason.. Where else is there for an asm coder to get some examples!! Nowhere much. Theres all those ROM KERNAL MANUALS but all the examples are in bloody C code (also known as 'line noise'.) But there are HEAPS of them in there. If only you could read them & learn how easy it is to convert these into assembler. Well I started reading the C examples and after a while you get to know exactly what is going on & can easily convert it to assembler. Just treat the code as example pseudo-code!

Please note, I had NEVER read any C books when I worked out how to convert it so don't be afraid to give it a go. I think the fact that I learned it this way makes me a pretty good choice for writing this article. To all the C coders that are likely to pick up a lot of mistakes.. Well please do but just remember, this is all the knowledge I ever need to READ C, WRITE ASSEMBLER. I gave this text to a certain Asm coder who remains anonymous, he has now started writing system friendly code & reading the RKM's without any of the previous fears!! 8-)

=====

PLEASE NOTE: After writing this text I asked Andrew Patterson to read it & correct any OBVIOUS mistakes. Andrew is a competent C and Asm programmer unlike myself (I only know Asm really). His additions are marked with a "!" character at the begining of the line.

Thanks Andrew!!

=====

1.4 Braces and Brackets

CHAPTER #1 BRACES and BRACKETS

OK, first off lets take a look at what these things {} (BRACES) do.

```

/*****C-CODE***** LINE NUMBERS USED ONLY FOR DEMONSTRATION
1      {
2      intbase = OpenLibrary("intuition.library",0)
3      }
*****/

```

Easy! its just a routine to open a library.
They just mark the start & end!!!

This would convert to:

```

;-----ASM-CODE-----
GetIB:
11     lea     intname,a0
12     moveq  #0,d0
13     movea.l (4).w,a6
14     jsr     _LV0OpenLibrary(a6)
15     move.l  d0,intbase

intname:
        dc.b   'intuition.library',0
        even
;-----

```

Line1 is the start of the routine.

Line2 says "Open the intuition library & put the library base in "intbase"
Sometimes its easier to read the lines back to front to make sense of them.
Ok, look up OpenLibrary, it will tell you this:

OpenLibrary(libname,version) (a0,d0) and that the result is returned in D0.
Meaning that it wants the address of the text "intuition.library" in A0 & a
version number in D0.

Line11 we put the address into A0

Line12 we tell it version 0 (ANY version)

Line13 OpenLibrary is an exec routine. So put execbase in A6.

Line14 we call the OpenLibrary routine

Line15 we store the result in intbase just like the C code

Most of the time you will see this though

```

/*****/
1     {
2     if (intbase = OpenLibrary("intuition.library",0))
3         {
4             if (wd = OpenWindow(newwin))
5                 {
6                     Do tricky stuff etc. etc.
7                 }
8             CloseWindow(newwin)
9         }
10    CloseLibrary(intbase)
11    }
12
13 }
/*****/

```

Line2 says "Open intuition library, any version, store it in intbase"

The IF means "if we get it continue with the code in the next braces, otherwise,
skip past the code in braces (to line 12)" The IF works on the statement
within the brackets. It could just say IF (A = B).

If we got intbase, go to the code starting at Line3. It does the same thing
except it is opening a window. If it gets the window, it continues at Line5
and if not, it skips past the matching brace where it will close the intbase
(Line10) and then exit. Notice how it always skips to the next matching brace.
Here it is in Assembler:

```

;-----
Start:                                ;{
        movea.l (4).w,a6
        lea     intbase,a0
        moveq  #0,d0

```

```

        jsr      _LVOpenLibrary(a6)
        move.l   d0,intbase      ;if (intbase = OpenLibrary("intuition.library",0))
        beq     exit
OpenWd:                                ; {
        move.l   d0,a6
        lea     newwin,a0
        jsr     _LVOpenWindow(a6)
        move.l   d0,wd          ; if (wd = OpenWindow(newwin))
        beq     CloseInt
                                ; {
        ;tricky stuff          ;tricky stuff
        ;etc. etc.             ;etc. etc.

CloseWin:
        move.l   intbase,a6
        move.l   newwin,a0
        jsr     _LVOCloseWindow(a6) ;CloseWindow(newwin)
                                ; }

CloseInt:
        movea.l (4).w,a6
        move.l   intbase,a1
        jsr     _LVOCloseLibrary(a6) ;CloseLibrary(intbase)
                                ; }

exit:
        moveq   #0,d0
        rts
                                ;}
;-----

```

Brackets ()

These are use to pass arguments to fuctions. Such as library routines or our own subroutines. Heres how they pass arguments.

/* Here i made up a routine called multiply that takes 2 numbers & multiplies them & then returns */

```

/*****/
1  int i1 = 2;
2  int i2 = 5;
3  ulong leftedge = 0;
4
5  leftedge = multiply(i1,i2); /* call the subroutine */
6  etc. etc. exit.
7/* a subroutine goes like this.. Return-value-type name(parameters)*/
8/*So this one returns an integer & wants 2 parameters
9
10int multiply(int m1, int m2)
11 {
12     int r; /*set aside an integer sized variable*/
13
14     r = m1 * m2; /* multiply them*/
15     return r; /* return the result */
16 }
/*****/

```

Line 1 makes an integer sized variable called i1 and gives it the value "2"

Line 2 & 3 do the same things

Line 5 calles the subroutine "multiply" giving it the 2 parameters "i2 & i1" and puts the result in the variable called "leftedge"

Line10 is our subroutine. It takes the 2 parameters & now puts them in new variable names (m1 & m2)

Line12 gets a variable for the result

Line14 does the big multiply

Line15 sets the return result & then RTS's

In Assembler. This is a pretty simple one. A trickier one might have a 32bit multiply routine (for the 68000) here.

```

;-----
        move.w  #2,d0
        move.w  #5,d1
        bsr    multiply
        move.w  d1,leftedge
        etc..

multiply:      ;value1 in d0,  value2 in d1,  result in d1
        mulu.w  d0,d1
        rts

```

leftedge dc.w 0

```

;-----
If the subroutine says this: void multiply(int m1, int m2)
The word VOID just means that it does not return any value.
=====

```

Square Brackets.[] These normally signify an array. You might see this

UBYTE data[10]; This means define an array of 10 unsigned bytes.

Which is the same as

```

data:
        dcb.b  10,0

```

The data in the array can then be accessed like this:

```

i = 2;
data[i] = 50 /* moves 50 into the data+2 */

```

which is like

```

moveq  #2,d0
lea    data,a0
move.b #50,0(a0,d0.w) ;moves 50 into data+2

```

or just

```

                data[2] = 50
converts to
        lea data,a0
        move.b #50,2(a0)

```

1.5 Define, UBYTE, USHORT, etc..

CHAPTER #2 DEFINE (#define), UBYTE, USHORT etc

 These are easy.

```
#define LEFT 10    is the same as    LEFT = 10
UBYTE           means an unsigned byte (like dc.b 0)
USHORT          means an unsigned word (like dc.w 0)
ULONG           means an unsigned long (like dc.l 0)
```

If they don't have the U (BYTE SHORT LONG) then they are using the most significant bit as the sign bit. A UBYTE can be any value from 0 - 255 but a signed BYTE can be -127 to +127.

INT This is just an integer. IMPORTANT! Some compilers treat INT as a word, others treat INT as a longword.

APTR this is A POINTER. The contents points to somewhere else. Its like this.

```
screenptr:
    dc.l myscreen
.....
myscreen:
    dc.w 10,10,12,etc
```

* A star in front of a variable name means its a pointer as well

```
!-----
! A star in front of a pointer (ie something that is already defined as
! a pointer) refers to what the pointer is pointing at.
!
! ie char *string;    // Defines 'string' as the address of some characters
!
! *string now means the character that 'string' points at.
! myfunctionthatwantsacharacter(*string);
!
! move.l string,a0          ; a0 has the string address
! move.b (a0),d0            ; d0 has the character ie *string
! jsr myfunctionthatwantsa..
!
!-----
!
! an & in front of a variable name means give me the address of this item
! ie
!     wd = OpenWindow(&newwindow);
!
! means like
!
!     lea newwindow,a0
!     .. call OpenWindow etc
!
! newwindow:
!     dcb.b    the size of a newwindow struct
```

BPTR this is a bcpl pointer. These are supposedly left over from a long lost

language called BCPL which was used to code the dos.library before WB2.

BPTRs have to be multiplied (when reading) or divided (writing) by 4.

eg.

```
move.l ThisTask,a0
move.l pr_CurrentDir(a0),a0 ;prCurrentDir is a BPTR
add.l a0,a0
add.l a0,a0 ;we have to multiply it by 4 to get the address it points to
```

STRUCT This normally defines a preset structure & fills in the variables accordingly. Such as this:

```
struct Gadget mygad =
{
0,10,10,50,12,0,0,0,0,0,"Mygad",0,0,0
}
```

OK, the C compiler sees this & looks up what a Gadget structure looks like.

```
STRUCT Gadget
{
struct Gadget *NextGadget;
WORD LeftEdge, TopEdge;
WORD Width, Height;
UWORD Flags;
UWORD Activation;
UWORD GadgetType;
APTR GadgetRender;
APTR SelectRender;
struct IntuiText *GadgetText;
LONG MutualExclude;
APTR SpecialInfo;
WORD GadgetID;
APTR UserData;
}
```

```
struct Gadget mygad =
{
0,10,10,50,12,0,0,0,0,0,"Mygad",0,0,0,0
}
```

One by one it fills it in like this.

```
mygad:
struct Gadget *NextGadget;      dc.l 0          ;a pointer to the next gadget
WORD LeftEdge, TopEdge;        dc.w 10,10     ;left top
WORD Width, Height;           dc.w 50,12     ;width height
UWORD Flags;                   dc.w 0         ;flags
UWORD Activation;              dc.w 0         ;activation
UWORD GadgetType;              dc.w 0         ;gadgettype
APTR GadgetRender;             dc.l 0         ;gadgetrender (none)
APTR SelectRender;             dc.l 0         ;selectrender
struct IntuiText *GadgetText;  dc.l Mygad.txt ;gadget text
LONG MutualExclude;            dc.l 0
APTR SpecialInfo;              dc.l 0
```



```

rsreset
rs.l wd_NextWindow      ; for the linked list of a Screen
rs.w wd_LeftEdge       ; screen dimensions
rs.w wd_TopEdge        ; screen dimensions
rs.w wd_Width          ; screen dimensions
etc. etc.
rs.l wd_MenuStrip      ; first in a list of menu headers
rs.l wd_Title          ; title text for the Window
rs.l wd_FirstRequest   ; first in linked list of active Requesters
etc. etc. etc.
rs.l wd_RPort          ; this Window's very own RastPort
and so on....

```

So what (win->RPort) is doing is this in assembler..

```

movea.l win,a0 ;window in a0
movea.l wd_RPort(a0),a0 ; like 50(a0) approximately

```

It gets the pointer from wd_RPort(a0) & puts it in the register

So now A0 has our rastport just where PrintIText wants it.

Next it says.. "&myIttext" for A1. The & sign means ADDRESS

Which is just like writing.. lea myIText,a1

next it says LEFT, TOP for D0 & D1 which is the same as

```

move.l #LEFT,d0
move.l #TOP,d1

```

So the full thing in assembler is

```

;-----PrintIText(win->RPort,&myIttext,LEFT, TOP)

```

```

LEFT = 10

```

```

TOP = 12

```

```

movea.l win,a0
move.l wd_RPort(a0),a0
lea myIText,a1
move.l #LEFT,d0
move.l #TOP,d1
movea.l intbase,a6
jsr _LVOPrintIText(a6)

```

```

;-----

```

The other one to watch is the INDEX symbol (.)

You will see code like this at times.

```

win.RPort

```

This works similar to

```

win->RPort

```

but heres where its different

```
win->RPort will do this  movea.l win,a0
                        movea.l wd_RPort(a0),a0
```

```
win.RPort  will do this  movea.l win,a0
                        lea    wd_Rport(a0),a0
```

```
or
                        movea.l win,a0
                        addi.l #wd_RPort,a0
```

It just points us to that location in the window structure while the GETOFFSET one (->) gives us the contents of that location.

You should take care not to confuse the two of these because nothing will work if you get them wrong.

1.7 << >> == ! || | && > < . + ++ +=

CHAPTER #3 << >> == ! || | && > < . + ++ +=

Math symbols

```
-----
= equals          a = 2          move.l #2,d0
+ add            a = a + 1       addq.l #1,d0
* multiply       a = a * 5       mulu.w #5,d0
/ divide        a = a / 2       divu.w #2,d0
% modulo        a = a % 3       divu.w #3,d0
                        swap d0

| OR            a = a | 2       OR.L #2,d0
& AND          a = a & 1       AND.L #1,d0
^ EOR          a = a ^ 1       EOR.L #1,d0
~ NOT          a = ~a          NOT.L d0
<< ASL         a = a << 2       ASL.L #2,d0
>> ASR         a = a >> 2       ASR.L #2,d0
```

```
!-----
! These can all be abbreviated if the variable to the left is the same as
! the variable to the right.
```

```
!
! ie a += 1 is equivalent to a = a+1
!    a |= 2 is equivalent to a = a|2
!
```

eg. Wait((1L<<win->userport->mp_sigbit));

Converts to..

```
movea.l (4).w,a6    ;exebase
movea.l win,a0      ;get our window
movea.l wd_UserPort(a0),a0 ;get our windows userport
movea.l MP_SIGBIT(a0),d1 ;get our windows userports signal bit no.
moveq #1,d0
asl.l d1,d0        ;Shift it left into its position.
```

```
jsr    _LVOWait(a6)      ;Wait for the signal to arrive.
```

Conditional code

== Is exactly equal. They use this in C when they compare something.

```
eg.  if(a == 2)      cmp.l #2,d0
     {              beq is2
     etc.  not2:
     }              is2:
```

In english, IF a is exactly equal to 2 then continue.

```
!= is NOT equal      eg.  if(a != 2) cmp.l #2,d0
                       { }      bne not2
                       is2:
                       not2:
```

```
> < >= =<  greater than, less than, less or equal, greater or equal.
           BGT          BLT          BLE          BGE
```

? Seems to mean if true do this, if false do that.

```
such as  b = (a == 2) ? 10 : 0;
```

its saying.. if a is exactly 2 then b = 10. If not, b = 0

```
        cmp.l #2,d0    ;is d0 2
        beq  do0      ;if so then d1 is 0
        move.l #10,d1 ;otherwise d1 is 10
        bra  done
do0:    move.l #0,d1
done:
```

&& Think of this as being the same as AND in english!!

```
if (a = 2 && b = 4)
```

will continue if both the statements are true

|| Think of this as being the same as OR in english!!

```
if (a = 2 || b = 4)
```

will continue if either of the statements are true

Misc

====

```
++ = Increment      a++ means add 1 to a  addq.l #1,d0 or move.w (a0)+,d0
-- = Decrement      a-- means subtract 1  subq.l #1,d0 or move.w -(a0),d0
8) a standard smiley
```

```
/* and */ These mark the start & end of a comment. Anything inbetween is
ignored.
```


fast! We did one assembly in Macro68 on BigMacs 040. It took 3 SECS for a 900k source file. Devpac3 took 2 MINUTES!!!! NO JOKE!!!

1.10 What if the C code uses amiga.lib routines!?

CHAPTER #7 What if the C code uses amiga.lib routines!?

This is a real PAIN IN THE ASS!! What happens is that amiga.lib contains some ready made routines that sometimes just didn't make it into ROM. Then in the examples in the RKMs they use these "amiga.lib MACROS" without any mention of how to do the same thing in assembler. What you have to do is find your RKM(includes & autodocs) and look up AMIGA.LIB in the linker libraries section. Here you'll find the source code for each routine (IN C or ASM)!!! If its C its up to you to convert this back to assembler or at least work out what happens.

Another one is a C macro that just appears in the include files (such as graphics/gfxmacros.h)

Heres one of the macros in gfxmacros.h

```
#define ON_VBLANK           custom.intena = BITSET|INTF_VERTB;
```

This macro Turns on the VBlank interrupt

In assembler its..

```
    lea _custom,a5
    move.w #BITSET|INTF_VERTB,intena(a5)
```

or in democoders asm

```
    move.w #$8020,$dff09a
```

1.11 Complete example - Bitmap Scaling using the OS

CHAPTER #8 Complete example BITMAP SCALING USING THE OS

*****THIS CODE REQUIRES 2.04+ TO RUN*****
Two source codes are included with this text. Scale.C & Scale.asm

Both written by ME.Please forgive my C!! 8) Heres whats going on.

First, the includes. We just include all the same ones. Plus we need the _lib.i includes for gfx, dos & intuition because the have the vales for _IWOOpenWindow etc. etc. C compilers know them all I think or use those .fd files.

```
!-----
! Thats what the pragma and proto files do.
!
```

Next it defines the variables Intuitionbase & Gfxbase. They have to be those longer names for the C compiler. It tells them that they are pointers (*)

Next it goes straight into void main(). This generally is the standard startup code which opens dos.library & handles WB loading etc. Since this is a CLI only utility all I'll do is open the dos.library.
(for a demo or something you should include standard startup code.
See the howtocode.txt for example source)

Then it defines a few other things like a taglist for openwindow & a bitmap structure etc. In assembler we prefer this stuff at the bottom (have a look.)

Next it opens the intuition library version 37.
If it gets that it opens the graphics library v37
If it gets that it opens the Window.
Next it Initializes a BitMap structure. What we have to do is to just look up what a bitmap structure looks like & save some space for it in our data area which we do with `bm: dc.b bm_SIZEOF.`

Next `InitBitMap(&bm,3,400,200);`
Looking at the InitBitmap call it wants the bitmap structure in A0
The number of planes in d0
and the width & height in d0 & d1.

```
lea MyBitMap,a0
moveq #3,d0
move.l #400,d1
move.l #200,d2
```

Next it initializes a structure called BitScaleArgs. This can be found in graphics/scale.i. What you have to do is set aside the bytes for it, fill in the details, & then put the address in A0 & call BitMapScale.

What we have planned is that our Bitmap that we have initialised with InitBitMap will be scaled to the sizes specified and then copied to our window's bitmap (The copy is done by BitMapScale). Since this is graphics data & BitMapScale uses the BLITTER, our bitmaps planedata must be in CHIP ram.

```
Where it says:      bsa.SrcX = 0;
                   bsa.SrcY = 0;
                   bsa.SrcWidth = 400;
```

```
We convert to:
                   lea    bsa,a0
                   move.w #0,bsa_SrcX(a0)
                   move.w #0,bsa_SrcY(a0)
                   move.w #400,bsa_SrcWidth(a0)
```

etc. etc. Till all the arguments are filled.

```
Then we call it    movea.l gfxbase,a6
                   jsr    _LVOBitMapScale(a6)
```

```
Then it delays for 1 second:
                   Delay(50)
```

```
which is          move.l #50,d1
                  move.l dosbase,a6
                  jsr      _LVODelay(a6)
```

Next it just repeats the scaling a few times using different scaling values. I have not done a direct copy of this code. I did about 10 different scalings in the assembler version to try it all out.

The BitMap scaling code is very flexible. You can scale things to any ratio within about 1:16000 (Scale it 16000 times its size). Examples 1:2 (double) 3:1 (one 3rd) 295:761 etc. etc. Its not made for doing demos or anything so don't think you can code a scaling demo with it.

After this it just frees the resources one at a time in the reverse order that we opened them. We do the same of course. 8-)

Well I hope this Volume of the Enconn Coders Manual has taught you something about reading C example code and I look forward to some system friendly demos from the demo coders scene in the near future! 8-)