HowToCode7

COLLABORATORS

	TITLE : HowToCode7		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		March 28, 2025	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

1 HowToCode7 **1**

1.1 HowToCode: General Guidelines 1

1.2 Read the Commodore Manuals 1

1.3 Self-Modifying Code 2

1.4 Write Relocatable Code 2

1.5 32Bit Addresses 3

1.6 Using Packers/Crunchers 3

1.7 exec.library/CacheClearU 5

1.8 exec.library/CacheControl 5

1.9 Don't use the hardware if you can avoid it! 6

1.10 Opening Libraries Properly 6

1.11 Nothing is fixed - Almost 7

1.12 Version Numbers 7

1.13 System Private values/functions 8

Chapter 1

HowToCode7

1.1 HowToCode: General Guidelines

```
        General Guidelines for Amiga Demo Programming
        ---------------------------------------------

   1. Read the Manuals
   2. Self-Modifying code
   3. Use Relocatable code
   4. All addresses are 32bit
   5. Packers/Crunchers
   6. Avoid unnecessary hardware access
   7. Opening libraries properly
   8. Nothing is fixed - almost!
   9. Version Numbers
  10. System private structures/functions
```

1.2 Read the Commodore Manuals

```
Read the Commodore manuals. All of them. Borrow them off friends or from
your local public library if you have to.

Read the "General Amiga Development Guidelines" in the dark grey (2.04)
Hardware Reference Manual and follow them TO THE LETTER.
If it says "Leave this bit cleared" then don't set it!

The official manuals are currently (up to 2.04 revision)

Addison Wesley
--------------

Amiga Rom Kernal Manual - Libraries
Amiga Rom Kernal Manual - Devices
Amiga Rom Kernal Manual - Includes and Autodocs
Amiga Hardware Reference Manual

 (All the above are available on CD from CATS USA)
```

```
Amiga Style Guide

Bantam
------

AmigaDOS Manual 3rd Edition

A V39/V40 Kickstart update manual should be available shortly.
```

1.3 Self-Modifying Code

Don’t use self-modifying code. Processors with cache ram cannot
handle self-modifying code at all well. They grab a large number
of instructions from ram in one go, and execute them from cache ram. If
these instructions alter themselves the changes are not made to
the copy in cache ram, so the code can crash. The larger the cache
the more likely this is to happen, even when you think you will be
safe, so the best strategy is to either

a) Disable caches (and suffer a large speed-loss penalty)

b) Avoid using self modifying code.

1.4 Write Relocatable Code

If you write demos that run from a fixed address you should be shot.
NEVER EVER DO THIS. It’s stupid and completely unnecessary.
Now with so many versions of the OS, different processors, memory
configurations and third party peripherals it’s impossible to
say any particular area of ram will be free to just take and
use.

Remember that writing relocatable code does *NOT* mean you
have to write PC relative code. Your assembler and AmigaDOS
handle all the relocation for you at load time.

It’s not as though allocating ram legally is dificult. If you
can’t handle it then perhaps you should give up coding and
take up graphics or something :-)

If you require bitplanes to be on a 64Kb boundary then try the
following (in pseudo-code because I’m still too lazy to write it
in asm for you):

```
        for c=65536 to (top of chip ram) step 65536

        if AllocAbs(c,NUMBER_OF_BYTES_YOU_WANT) == TRUE then goto ok:

  next c:

        print "sorry. No free ram. Close down something and retry demo!"
        stop
```

ok: Run_Outrageous_demo with mem at c

Keep your code in multiple sections. Several small sections are better than one large section, they will more easily fit in and run on a system with fragmented memory. Lots of calls across sections are slower than with a single section, so keep all your relevent code together. Keep code in a public memory section:

```
        section mycode,code
```

Keep graphics, copperlists and similar in a chip ram section:

```
        section mydata,data_c
```

Never use code_f,data_f or bss_f as these will fail on a chipram only machine.

And one final thing, I think many demo coders have realised this now, but $C00000 memory does not exist on any production machines now, so stop using it!!!

1.5 32Bit Addresses

Always treat *ALL* addresses as 32-bit values.

"Oh look" says clever programmer. "If I access $dcdff180 I can access the colour0 hardware register, but it confuses people hacking my code!".

Oh no you can’t. On a machine with a 32-bit address bus (any accelerated Amiga) this doesn’t work. And all us hackers know this trick now anyway :-)

Always pad out 24-bit addresses (eg $123456) with ZEROs in the high byte ($00123456). Do not use the upper byte for data, for storing your IQ, for scrolly messages or for anything else.

Similarly, on non ECS machines the bottom 512k of memory was paged four times on the address bus, eg:

```
  move.l #$12345678,$0

  move.l  $80000,d0 ; d0 = $12345678
  move.l  $100000,d1  ; d1 = $12345678
  move.l  $180000,d2  ; d2 = $12345678
```

This does not work on ECS and upwards!!!! You will get meaningless results if you try this, so PLEASE do not do it!

1.6 Using Packers/Crunchers

Don’t ever use Tetrapack or Bytekiller based packers. They are crap.
Many more demos fall over due to being packed with crap packers than
anything else. If you are spreading your demo by electronic means
(which most people do now, the days of the SAE Demodisks are long
gone!) then assemble your code, and use LHARC to archive it, you
will get better compression with LHARC than with most runtime
packers.

If you *have* to pack your demos, then use Powerpacker 4+, Turbo
Imploder or Titanics Cruncher, which I’ve had no problems with myself.

(found in the documentation to IMPLODER 4.0)

```
>** 68040 Cache Coherency **
>
>With the advent of the 68040 processor, programs that diddle with code which is
>subsequently executed will be prone to some problems. I don’t mean the usual
>self-modifying code causing the code cached in the data cache to no longer
>be as the algorithm expects. This is something the Imploder never had a
>problem with, indeed the Imploder has always worked fine with anything
>upto and including an 68030.
>
>The reason the 68040 is different is that it has a "copyback" mode. In this
>mode (which WILL be used by people because it increases speed dramatically)
>writes get cached and aren’t guaranteed to be written out to main memory
>immediately. Thus 4 subsequent byte writes will require only one longword
>main memory write access. Now you might have heard that the 68040 does
>bus-snooping. The odd thing is that it doesn’t snoop the internal cache
>buses!
>
>Thus if you stuff some code into memory and try to execute it, chances are
>some of it will still be in the data cache. The code cache won’t know about
>this and won’t be notified when it caches from main memory those locations
>which do not yet contain code still to be written out from the data caches.
>This problem is amplified by the absolutely huge size of the caches.
>
>So programs that move code, like the explosion algorithms, need to do a
>cache flush after being done. As of version 4.0, the appended decompression
>algorithms as well as the explode.library flush the cache, but only onder OS
>2.0. The reason for this is that only OS 2.0 has calls for cache-flushing.
>
>This is yet another reason not to distribute imploded programs; they might
>just cross the path of a proud ’40 owner still running under 1.3.
```

I doubt it! Only a complete *IDIOT* would run an ’040 under KS1.3. They
deserve to have their software crash!!

```
>It will be interesting to see how many other applications will run into
>trouble once the ’40 comes into common use among Amiga owners. The problem
>explained above is something that could not have been easily anticipated
>by developers. It is known that the startup code shipped with certain
>compilers does copy bits of code, so it might very well be a large problem.
```

You can use the following exec.library functions to solve the problem.

```
   CacheClearU()  and  CacheControl()

Both functions are available with Kickstart 2.0 and above.

I strongly disadvise trying to 'protect' code by encrypting
parts of it, it's very easy for your code to fail on >68000 if you
do. What's the point anyway? Lamers will still use Action Replay
to get at your code.

I never learnt anything by disassembling anyones demo. It's usually far
more difficult to try and understand someone elses (uncommented)
code than to write your own code from scratch.
```

1.7 exec.library/CacheClearU

```
CacheClearU - User callable simple cache clearing (V37)

CacheClearU()
    -636

void CacheClearU(void);

Flush out the contents of any CPU instruction and data caches.
If dirty data cache lines are present, push them to memory first.

Caches must be cleared after *any* operation that could cause
invalid or stale data.  The most common cases are DMA and modifying
instructions using the processor.

Some examples of when the cache needs clearing:
      Self modifying code
      Building Jump tables
      Run-time code patches
      Relocating code for use at different addresses.
      Loading code from disk
```

1.8 exec.library/CacheControl

```
CacheControl - Instruction & data cache control

oldBits = CacheControl(cacheBits,cacheMask)
  D0         -648          D0        D1

ULONG CacheControl(ULONG,ULONG);

This function provides global control of any instruction or data
caches that may be connected to the system.  All settings are
global -- per task control is not provided.

The action taken by this function will depend on the type of
CPU installed.  This function may be patched to support external
caches, or different cache architectures.  In all cases the function
```

will attempt to best emulate the provided settings. Use of this
function may save state specific to the caches involved.

The list of supported settings is provided in the exec/execbase.i
include file. The bits currently defined map directly to the Motorola
68030 CPU CACR register. Alternate cache solutions may patch into
the Exec cache functions. Where possible, bits will be interpreted to
have the same meaning on the installed cache.

```
IN:
   cacheBits - new values for the bits specified in cacheMask.

   cacheMask - a mask with ones for all bits to be changed.

OUT:
   oldBits   - the complete prior values for all settings.
```

As a side effect, this function clears all caches.

1.9 Don’t use the hardware if you can avoid it!

This one is aimed particularly at utility authors. I’ve seen some
awfully written utilities, for example (although I don’t want
to single them out as there are plenty of others) the Kefrens
IFF converter.

There is NO REASON why this has to have it’s own copperlist. A standard
OS-friendly version opening it’s own screen works perfectly (I
still use the original SCA IFF-Converter), and multitasks properly.

If you want to write good utilities, learn C.

1.10 Opening Libraries Properly

This has got to be one of the worst pieces of code I’ve
ever seen! Don’t ever do this, Michel!

```
   move.l   4.w,a0          ; get execbase
   move.l   (a0),a0         ; wandering down the library list...
   move.l   (a0),a0         ; right. I think this is graphics.library

   ; now goes ahead and uses a0 as gfxbase...
```

Oh yes, graphics.library is always going to be second down the chain from
Execbase? No way!

(Note by Michel: I’m sorry... I’ll never do it again :))

If you want to access gfxbase (or any other library base) OPEN the
library. Do not wander down the library chain, either by guesswork or
by manually checking for "graphics.library" in the library base name.
OpenLibrary() will do this for you.

Here is the only official way to open a library.

```
   MOVEA.L  4,a6
   LEA.L    gfxname(PC),a1
   MOVE.L   #39,d0               ; version required (here V39)
   JSR      _LVOOpenLibrary(a6)  ; resolved by linking with amiga.lib
                                 ; or by include "exec/exec_lib.i"
   TST.L d0
   BEQ.S OpenFailed
; use the base value in d0 as the a6 for calling graphics functions
; remember d0/d1/a0/a1 are scratch registers for system calls

gfxname  DC.B  'graphics.library',0
```

Don't use OldOpenLibrary! Always open libraries with a version, at least V33. V33 is equal to Kickstart 1.2. And DON'T forget to check the result returned in d0 (and nothing else).

OldOpenLibrary saves no cycles. All it does is

```
   moveq.l  #0,d0
   JMP      _LVOOpenLibrary(a6)
```

1.11 Nothing is fixed - Almost

Unlike inferior machines, almost everything on the Amiga can and will change. Chipsets change, the OS changes, the processor changes. This means you can never assume that something that is set to a particular value on your machine will be set to the same value on other machines. The one exception is the pointer to EXECBASE ($0000004). This long word will always point to the ExecBase structure, which you can use to access every other system level function on the Amiga.

At the hardware level make sure that every hardware register is initialised to the value you require before you start (do not assume that bitplane modulos, for example, are always set to zero by workbench. Under V39 OS they are not!

If you are using AGA hardware registers, make sure you check for AGA and AGA only, AAA (when it is released) is *NOT* AGA harware compatible.

Even the processor isn't necessarily final. It is strongly rumoured that the Motorola MC68060 is the final member of the 68000 series, and may not even come out. Expect Amigas in 2-3 years to come with RISC chip processors running 680x0 emulation.

1.12 Version Numbers

```
Put version numbers in your code. This allows the CLI version command
to determine easily the version of both your source and executable
files. Some directory utilities allow version number checking too (so
you can't accidentally copy a newer version of your source over
an older one, for example). Of course, if you pack your files the
version numbers get hidden. Leaving version numbers unpacked
was going to be added to PowerPacker, but I don't know if this is
done yet.

A version number string is in the format

$VER: howtocode6      7.0 (13.06.92)
^          ^          ^Version number (date is optional)
|          |
|          | File Name
|
| Identifier

The Version command searches for $VER and prints the string it finds
following it.

For example, adding the line to the begining of your source file

; $VER: MyFunDemo.s 4.0 (21.01.93)

and somewhere in your code

  dc.b  "$VER: MyFunDemo 4.0 (21.01.93)",0

means if you do VERSION MyFunDemo.s you will get:

MyFunDemo.s 4.0 (21.01.93)

and if you assemble and do Version MyFunDemo, you'll get

MyFunDemo 4.0 (21.01.93)

This can be very useful for those stupid demo compilations
where everything gets renamed to 1, 2, 3, etc...

Just do version 1 to get the full filename (and real date)
```

1.13 System Private values/functions

```
System-Private
--------------

  If anywhere in the manuals, includes or autodocs it says that
  this or that is  PRIVATE or RESERVED or INTERNAL (or something
  similar) then

    . don't read this entry
    . never ever WRITE something to it
    . if it's a function, then DON'T use it (*)
    . don’t check it for anything

  Private system points can be changed without reason, or without
  writing it into any documentation !

  (Thanks Arno!)

 And to add to that, if a system structure member has a routine that
allows you to alter it (for example, SetAPen() alters the Pen
value in the RastPort. It is currently possible to alter the pen
by poking the structure) then USE IT! Do not Poke system structures
unless there is no other way to alter the value.
```