HowToCode7

COLLABORATORS			
	TITLE : HowToCode7		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		August 9, 2024	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

HowToCode7

1.1 HowToCode: Action Replay

```
                      Action Replay Cartridges
                      ------------------------

  1  Entering 'sysop mode'
  2  How Action Replay 2 works
```

1.2 Entering 'Sysop Mode'

```
Action Replay is great fun, even more so if you get into the
'sysop mode' (Allows disassembly of ram areas not previously
allowed by Action Replay, including non-autoconfig ram and
the cartridge rom!)

To get into sysop mode on Action Replay 1 type:

LORD OLAF

To get into sysop mode on Action Replay 2 type:

MAY
THE
FORCE
BE
WITH
YOU

To get into sysop mode on Action Replay 3 type the same as
Action Replay 2. After this you get a message
"Try a new one".
Then type in

NEW

and sysop powers are granted!
```

1.3 How ActionReplay 2 Works

```
The Action Replay 2, How it and the Amiga works, and why
--------------------------------------------------------
```

For all the hackers amongst you lot, especially those with one of Datel's excellent Action Replay Mk.][s (unlike the first one, which was useless), here is a little technical info about it, and how to protect against it.

1.4 The Cartridge Internals

```
The cart's internals
--------------------
```

The cartridge contains 128k of Rom (contained in two 27512 (64kx8bits) roms) and 32k of static ram. All the chips are surface-mount types, and are buggers to get off intact with a soldering iron. My advice is, don't bother! (There is also a scattering of custom PLAs and things, but they're not very interesting.) The Rom is addressable from $400000-$41ffff (and is repeated due to partial address decoding from $420000-$43ffff). The Ram is from $440000-$447fff, and is repeated loads of times for the same reason.

Normally, this Rom & Ram is totally invisible & undetectable. It is switched out of the address space, and there is *nothing* you can do to switch it in via software (there is an exception to this, see later).

Pressing the Button..

1.5 Pressing the Button

```
When you press the button
-------------------------
```

When you press the button, the cartridge generates a level 7 (non-maskable) interrupt, and then when the 68000 comes to get the vector (at $7c) for the interrupt, the cartridge whangs on the OVR* line on the expansion port, which has the effect of disabling all the amiga's internal chip memory!

The cart then makes its internal Rom appear instead of the chip ram from $00000-$80000. This Rom (and its Ram) simultaneously appears at their normal addresses at $400000/$440000. If this sounds like magic, it's actually incredibly simple to do in hardware!

What happens, and why..

1.6 What happens, and why...

A Quick explanation of what happens and why

As a matter of interest, the cart is doing something very similar to what the Amiga normally does on a reset, which is to switch all of Kickstart in from $00000-$80000, thereby providing the reset vector. That's why if you do a RESET instruction, then your code will most likely blow up instantly as all chip ram disappears! (The way to make thing return to normal is to set the CIA OVL* (OVerLay) signal back to 0, but if your code is dead, this is easier said than done.. however it can be done by some deeply sneaky programming.. see the Double Dragon][/Shinobi protection that I coded a while back)

Anyway, I digress. This new level 7 vector from the cartridge is $400088, a routine in the Rom proper which first restores the chip ram,etc to normal. Then it treks off into the bowels of the cartridge software.. you can follow it through and ReSource it if you like!

This is, however, only one of the ways to get into the cartridge. There are two others..

Two ways to get into the cart without pressing the button

1.7 Two ways to get into the cart without pressing the button

Two ways to get into the cart without pressing the button

(Or: 'Futility- an object lesson'!)

1)
This first, is the way the cart boots up on a reset (displaying that little piccy). The way this works is indecently sneaky! Instead of (as I had expected) either intercepting the reset wholesale, or appearing to the Kickstart reset code as an autoboot cartridge (which can be done by making your Rom appear at $F00000 on reset with $1111 as the first word - see $fc00d2 in the Rom), the AR2 goes for a MUCH sneakier route!

It detects processor accesses to location $0000008, and when one happens, it effectively 'presses' it's own button. (i.e. does the level 7 business) Now, it just so happens that the Kickstart Rom does a MOVE to location $8 very early on, when it sets up the exception vectors, so voila! There it goes into the cart, returning later when it feels like it! At this point, I feel I should deeply disappoint those of you with a more technical bent (oo-er!)... You may be thinking that if you use the processor to generate a reset (with the RESET instruction), and then access location $8, then the cart will reveal it's presence, and thereby you can protect against it!

No way jose!

The designer of the cart was clever enough to put in a circuit that can actually tell the difference between you plonking your fingers on the 3-keys of doom, and the processor doing a RESET.. Not easy to do! (It actually times the reset pulse, and whereas the processor RESET instruction makes a

really short one, the keyboard reset lasts for about ½ a second, and only responds to a location 8 access after a genuine reset.)

Smart eh?

2)
The second way is a bit simpler. When you set a breakpoint with the cart, or set the exceptions with SETEXCEPT, what the cart does is to put a set of TST.B $BFE001 instructions at location $100, with the vectors in question pointing there. What then happens, is then when you get an exception, the TST.B $BFE001 is executed, the cart detects it happening, and does the Level 7 thang. Now. This TST.B can't just be anywhere. It has to be from $100 to $120ish.

"Ah HA!", you think,"I'll just have a few 'TST.B $BFE001's executed at $100 before my game loads!!"

....Oh no you won't! This whole thing is only enabled after the user does a SETEXCEPT or sets up a breakpoint! Normally the above has NO EFFECT AT ALL! Who's a clever little cart designer then???

How does it know what's going on with the custom chip registers?

1.8 How does it know what's going on with the custom chip registers?

How does it know what's going on with the custom chip registers?

I knew you would ask me that one! Ok, get ready for some interesting (?) technical info.

As you may well know, all the custom chip registers at $DFF000-$DFF200 are EITHER read-only OR write-only, never, ever, both. This is for a good design reason (take my word for it). Some of the more interesting registers have separate read registers (e.g. DMACONR) so you can tell what the register itself contains. Most don't...

Oh dear!

It is therefore simply NOT POSSIBLE to tell what was last written to, say, COLOUR00 ($DFF180), without either; A) Asking the person looking at the screen to describe the border colour, or B) Extra hardware.

Not entirely suprisingly, the AR2 goes for the latter solution. First, a quickie lesson in how the amiga's custom chips communicate internally...

The custom chip 'registers' are not really actual memory at all. What that area ($Dff000-$dff200) actually is is a 'window' into the internal custom-chip computer system. This system consists of a 'bus' (i.e. a channel for data consisting of several physical connections grouped together, each one carrying one bit of the data that it being passed) that is connected to all the custom chips inside the Amiga. (See the pins labeled 'RGA1-8' which contains the number $000-$200 of the custom register, and 'D0-D15' which contain the 16 bits of data being transferred, that are on all of the main custom chips)

Using this, all the various registers (which in physical reality are located scattered about in various of the custom chips, according to their usage. I.e. DIWSTRT is in Denise, which generates the display, and BLTSIZE is in Fat Agnus, which contains the blitter amongst other things) can be read or written.

These registers do not exist solely for the purpose of being read/written by the 68000. Certainly most of them have no purpose than to be set up by the main processor in order to tell the appropriate custom chip what to do, but others are really of no use to the programmer whatsoever. For example, the xxxxxxDAT registers (more on them later).

Apart from the 68000 using this bus to communicate with the custom chip registers, the chips themselves use it all the time too!

For example, how the screen display is generated...

Fat Agnus does all the DMA handling; i.e. it is the chip that, when a bit of data (screen,disk,audio,etc) has to be transferred to/from chip memory, actually does the donkey work of read/writing the value to the appropriate main memory location. (Main memory is a separate system to this internal custom chip world, and Agnus is the interface)

For screen DMA, the data actually is destined for the Denise chip, which is a separate lump of silicon, so how does it get from Agnus (that has just fetched it from Ram) to Denise? Via this internal bus! This is where those registers that no-one seems to know about come in. I mean the xxxxDAT registers!

BPL1DAT (for example) is not really meant to be accessed by the user at all! It is there for internal usage.. i.e. Agnus reads a word of the screen memory from the outside world, and writes it into BPL1DAT, which is a physical register inside the Denise chip.

Now, this operation is functionally the same as you doing a MOVE.W into BPL1DAT, but it is done by Agnus, with no help from the processor. This is DMA. If you read from VHPOSR, then the read request from the 68000 would be passed to Agnus, which would then consult the internal bus, and then deliver the value back.

Ok, basically the processor is just a spectator on this internal register bus. (Am I being too patronising? Sorry.)

Right, so now you know that the custom chip registers are not really part of main memory at all (that's why the Copper can only work within this small world of registers and not with all of Ram), and you know/already knew that there are many registers that you cannot read at all.

Back on the actual subject in hand.. i.e. how the AR2 knows what these internal registers contain, when it is simply not possible to read them....!

Answer: It doesn't.

What it actually does, is use an idea stolen from Romantic Robot (a company that made the first decent freezer cart for the Amstrad CPC, which also had write-only registers) which is to make a bit of sneaky hardware that

effectively sits there and watches the 68000 like a hawk. Whenever the 68000 does a write operation to a chip register, it makes an internal copy of the value being written! What this amounts to is $200 bytes of totally invisible (invisible, that is, until you whack the button, when it appears - to be read - in another area of memory) Ram that contains all the values most recently written by the processor to any custom chip register.

Handy eh? So the action replay has all those values you wrote to COLOUR00, DSKSYNC,etc,etc copied down in its' own bit of ram!

Now, here's where the major catch comes...

You may have noticed two things....
A) Only the value last written is kept, and
B) Only VALUES WRITTEN BY THE PROCESSOR can be kept.

The first argument is not really that important, but the second one IS!

How to protect against Action Replay Mk.][

1.9 How to protect against Action Replay Mk.][

3 example ways to protect against Action Replay Mk.][

A)
There is NO POSSIBLE WAY that the AR2 can EVER TELL WHAT THE COPPER HAS WRITTEN TO A CUSTOM CHIP REGISTER.

A fundimental flaw, and one which it is not possible to solve in hardware without having a set of leads connected inside your amiga. The chip register bus is not available on the explansion port.

"Hang on!", I hear you cry (?),"how come it knew about that DIWSTRT that I only wrote in my copper list?"

Ah ha. Well, this is where the sneaky programming in the cart comes in. It knows where your copper list is (after all, you must have written to COP1LC at some point.. with the processor) and so what it does is to follow your copper list through and it actually updates its little internal copy of the custom registers to do the changes that the copper must have done!

I.e. if you did a CLR $DFF180 to clear colour0, and then in your copper list you had a MOVE $FFFF,$0180 to there, the AR2 would only have your processor CLR stored by the hardware, but the software would see the copper instruction and alter its record. In fact, it has no way of actually knowing whether this copper instruction was ever run or not, it just guesses!

So, the main form of protection is to set up all the important chip registers using a temporary one-off copper list, one that, once it has run, is dumped and a new copper list (the proper in-game one) is started instead.

The AR2 will never know about any of the chip writes done in the first copper list, if you press the button when the final one is running.

This is better than it seems, because the AR2 changes loads of things when it runs it's own monitor (i.e. resolution, # of bitplanes, pointers,etc,etc) and relies on knowing what to put back into the registers when it returns to your code. If it has a totally false idea of what was in, for example, BLTCON0, or DSKSYNC, then there is no way your program will ever run again!

That's the only practical way to protect against the cart, and as you can see, it is naff in that it doesn't either a) detect the cart before the program runs, or b) stop you going into the monitor.

The other ways are....
("The Dog-In-A-Manger Approch".. if I can't.. neither can you!)

B)
Point the Supervisor stack to an ODD address, and run your program in user mode, with NO INTERRUPTS! When you get an interrupt, the processor always enters supervisor mode, switches over to the supervisor stack, and pushes on the address to return to after the interrupt and the current Status Register value. If the address that it tries to push these to is odd...? Kapooof. Not just an address error, but the address error itself also tries to push words onto the odd-stack, and you get a double-exception.. i.e. total 68000 lock-up. It will not recover until you do a hardware reset.

This is absolutely the best way to fuck ANY cart up. Press the button when this has been done and the entire computer crashes totally. But... Can you write a game without using interrupts?

(The 'Say kids.. what time is it?' approach)
C) Use the CIA Time-of-day alarm. This is semi-complex..

Each of the 2 CIAs have 'Time of Day' clocks. These are clocks that run on the conventional hour/minute/second scale, and are driven by the system clock. They have to be set to the right time after every reset, so are almost never used for their intended purpose. Thing is, the clocks also have an alarm facility, whereby you can get an interrupt from the CIA when the current time=the preset alarm time. There are 3 registers (hours/minutes/seconds) that if read, contain the current time, if written in mode 1 (mode is set by a register bit), will change the current time, and if written in mode 2, will change the alarm time.

This alarm time cannot ever be read. So.. what you do is...

Set your alarm time for, say, 00:00:10, then set the current time to 00:00:00, and enable the interrupt. Start your game going. When you get the alarm interrupt, set the current time back to 00:00:00, and in another 10 seconds you will get another interrupt, and so on. If, however, you notice that the time has ever gone past 00:00:10 without you getting an interrupt, or that the alarm occurs at the wrong time, then you know that someone has tampered with the program and didn't set the right alarm time! I know it sounds complicated, but if you use a weird alarm time, then noone will ever know what the correct value to set it to is, and so the 'freezer' can never produce a copy of the game that will unfreeze and work.

Both these approches use features of the Amiga/68000 that cannot be got around. The first one can be fixed by having loads of internal connections into the Amiga, but no-one will want to do that. Unfortunately, at time of writing (31st Jan'91), I can't think of another way to stop the cart getting into the monitor. Like I said, it's quite well designed!

If you can find a bug in the software to exploit, all well and good, but remember bugs can be fixed!

P.P.P.P.S. Thanx to Bob & Jim for the help.

Brought to you by
GREMLIN of MAYHEM (finished 5:50am 31st Jan 1991)