USER MANUAL

DATEL TRAINER TOOLKIT
For NINTENDO® DS™

V1.00 US/UK

Index

9. Technical Support & Customer Services

1. Introduction

Congratulations on your purchase of Datel's Trainer Toolkit for Nintendo DS.

Trainer Toolkit is a revolutionary product designed exclusively for serious DS gamers, amateur programmers and aspiring game hackers. It provides you with all the hardware and software necessary to create your own game-busting Action Replay codes.

This manual is your induction into the underground world of game hacking.

This manual includes all the information you need to create your first codes. It also gives you a good grounding in code hacking, so you can create many more. Some codes are found easily, but others require perseverance, dedication and skill. But one thing's for sure; with Trailer Toolkit in the right hands, nothing is impossible.

2. Getting Started

2.a. Pack Contents

Your Trainer Toolkit is made up of the following parts; please check that everything is included before proceeding. If anything is missing, contact Datel's customer service department (see Section 8).

- 1 x Trainer Board
- 1 x Trainer Software PC CD-ROM
- 1 x USB to Mini USB Cable
- 1 x User Manual

To use Trainer Toolkit you also need an Action Replay DS running compatible firmware (v1.52 or newer). Your Action Replay's firmware can be updated through the Action Replay Code Manager software that came with your Action Replay. Use the 'Software Upgrade' button to upgrade to the latest PC software and cartridge firmware.

Trainer Toolkit's *Trainer Board* can also be updated with new firmware. To update the Trainer Board's firmware use the Trainer Toolkit PC application. See Section 4.g for more details.

2.b. System Requirements

Because Trailer Toolkit uses PC-based 'training' software (connected to your DS), in order to use the product you need a PC that meets the following minimum system requirements:

MINIMUM SYSTEM REQUIREMENTS	
Computer / Processor	Personal Computer with one free USB port. USB 2.0 recommended for higher-speed data transfer.
Operating System	Windows 2000 or XP. Windows XP recommended.
Communication	Internet access required for software updates.
Hard Drive	<10Mb free hard drive space for software installation.

2.c. Installing the Trainer Software

Insert the Trainer Software CD into your drive and wait for the installation to auto-run. If auto-run is not enabled on your PC, browse to the CD and double-click the SETUP file.

Follow the onscreen prompts to install the Trainer Software on your PC. Once set-up is complete, you have a new program folder, ‘Datel > Trainer Toolkit’. If opted for during installation, a program icon for ‘Trainer Toolkit’ appears on your desktop.

Driver Installation

The drivers for the Trainer Board are included on the installation CD, but are not automatically installed by Windows.

When you connect the Trainer Board to your PC, you’re prompted to install the drivers. To do so, direct Windows to the Trainer Software CD, which you should leave in your drive during this process.

2.d. Connecting to the Trainer Board

The Trainer Board is a special piece of hardware that allows your PC real-time access to the contents of your DS’s memory.

Snap the Trainer Board into the GBA cartridge slot on your DS, and connect it to your PC via the supplied mini-USB to USB lead.

When Windows detects the Trainer Board, must direct it towards your Trainer Software CD for the necessary USB drivers.

Specify that Windows uses the driver file: **DSTRAINER.SYS** in the root of the Trainer Software CD.

Can’t Talk to Trainer Board Errors

When using Trainer Toolkit, from time to time you may receive the error ‘Can’t talk to trainer board!’. This message means that USB communication between the Trainer Board and the PC application has failed. This may be due to the USB cable being nudged or removed, or the result of a crash during training.

There are a number of ways to re-establish USB communication; different methods work for different situations:

If you nudge or disconnect the cable	Reconnect the cable (if necessary) and open the Hex viewer in Trainer Toolkit. Communication should be re-established.
If your DS powers down	Power your DS back up (booting using the Action Replay Trainer cartridge) and open the Hex viewer to establish/test communication.
If you crash the trainer	Reboot the DS and try opening the Hex viewer to re-establish communication. If this doesn’t work, try disconnecting the USB lead from the Trainer Board and then removing the Trainer Board from the DS. This will remove power from the board and the USB controller. Once you reconnect the Trainer Board and reboot, you should be able to re-establish communication.

2.e. The Action Replay Trainer Cartridge

Trainer Toolkit also uses a modified version of Action Replay DS running in the DS cartridge slot to test codes as you develop them.

Snap the Action Replay Trainer cartridge into the DS cartridge slot on your console before you begin to train.

3. Introduction to Game Training (Hacking)

3.a. How Action Replay Codes Work

Before you begin creating your own Action Replay codes, it's important to first understand how Action Replay codes themselves work.

Action Replay DS codes can only be used by someone with an Action Replay DS and the game in question. The codes cannot be entered without an Action Replay DS, and they're of no use to someone who doesn't own the game to which they refer.

When a user enters a new code into their Action Replay, and then enables the code and runs the game, Action Replay loads the codes into the console's memory and runs the codes.

In the case of simple codes like 'Infinite Health', this involves constantly writing a value to a location in memory, where both the memory address and the value are included in the Action Replay code. By constantly writing a value such as '99' to the address that stores the value of how many lives you have, whenever the game tries to change the value at that location (when you lose a life) Action Replay immediately puts it back up to '99' again!

3.b. Anatomy of an Action Replay Code

Action Replay codes generally are made up of a code type, a memory location and a value for that memory location.

Here is the Action Replay code 'Infinite Lives' for Super Mario 64 DS:

```
020973EC 00000063
```

Let's break down the code:

Code type	Memory location	Value
`0`	`20973EC`	`00000063`
32bit write-to location.	`20973EC` is the address in the console's RAM where the value is written to.	Writes the value '99' (which is HEX 63 in decimal).

In plain English this code equates to:

"Keep writing the 32bit value '99' to the memory location `20773EC`".

While 'training' the game we found the memory location `20973EC` is where Super Mario 64 DS stores the value of how many lives you have. By constantly writing the value of `00000063` to that location, you always have 99 lives, which effectively gives you 'Infinite Lives'.

Multi-line codes

In the previous example, we looked at a very simple Action Replay code, ‘Infinite Lives’ for Super Mario 64 DS. When you first start creating your own codes, it‘s likely most will be simple one-line ‘infinite….’ codes like this.

Not all codes are only one line long, though. You’re probably familiar with entering some very long codes into Action Replay. So why are some codes many lines long, and what is contained within them?

Let’s take a look at a more complicated code by breaking it down line by line to understand how it works.

Here’s the Action Replay code, ‘Press Y For Moon Jump’ for Metroid Prime: Hunters:

```
923fffa8 00002400
020da74c 00000398
d0000000 00000000
```

Again, let’s break down the code:

Code type	Memory location	Value
`9`	`23fffa8`	`00002400`
16bit "if equal" instruction.	The memory location where the button press states for this game are stored.	The value when ‘Y’ alone is pressed.
`0`	`20da74c`	`00000398`
32bit write	The memory location where the value for gravity is stored.	Write the value of 920 (HEX 398 in decimal)
`d`	`0000000`	`00000000`
End If		

In plain English, this code is:

If the value for the ‘Y’ button being pressed is true (ie. the player pressed ‘Y’), write a value of 920 to the memory location that stores the current gravity value.

In this example, you can see Action Replay codes are actually a programming language in themselves. Included in the Action Replay Code Engine are 27 different ‘code-types’ whose functions can be combined and nested to achieve amazing results, even when a game seems determined to make life hard for a hacker!

A full list of all the Action Replay code-types is included towards the back of this manual.

3.c. How Games are Trained

Sometimes referred to as ‘hacking’, ‘game training’ is the more descriptive name given to the process of using third party hardware/software to analyze the way a game works and modify its behaviour in various ways.

Training is generally done by sequentially comparing blocks of memory dumped from the console’s RAM at slightly different points during a game. The blocks of memory are compared using logical ‘operators’ such as Greater Than, Less Than, Equal To, etc. The results of these comparisons then provide a list of memory locations where the desired value (like the value that stores how many bullets or lives you have left) *may* reside. This process is called ‘searching’.

Once you have a list of only a few possible memory locations where the value want might be stored, you can use Trainer Toolkit features like 'Watched list' to watch the values at a list of particular memory locations. By keeping an eye on the values at different locations, you can easily spot when a value changes at exactly the time you do a certain thing in the game. This information can then be used to rule certain locations in or out of a particular search.

Many dumps and comparisons using different operators may be necessary in order to reduce the number of possible memory locations to a manageable number (maybe 20 or less).

Where the memory dumps are made depends on what code you're trying to create.

To find:	Compare memory dumps:	Look for:
Infinite time	Sequentially as the timer goes up	Values greater then previous dump
Infinite lives	Before losing a life, after losing a life, after losing another life...	Values less than previous dump or equal to a specific value
Have a number of stars	Before collecting a star, after collecting a star, after collecting another star…	Values greater than previous dump or equal to a specific value
Infinite ammo	Before firing a bullet, after firing a bullet, after firing another bullet…	Values less that previous dump or equal to a specific value

Let's look at a couple of examples of searches to put this idea into context.

Example 1: Searching for Infinite Ammo

1. Load the game and get it to a point where we're in the actual game itself (not a menu, loading screen, etc.)

2. Now perform a 'new search' which 'dumps' the entire contents of the DS's memory into Trainer Toolkit.

3. Wait a few seconds and then search again for values that have stayed the same (because we haven't fired any bullets). This would rule out thousands of memory locations that *have* changed like locations storing the coordinates of sprites, the timer and any graphics that are changing, etc.

4. Now fire the gun and search again, this time asking Trainer Toolkit to only show locations where the value is *less than* before (because the number of bullets we have has gone down). Chances are the value has gone down for several memory locations, not just the location that stores the amount of bullets you have – we're unlikely to get it first time. At this stage, it's normal to perform several more searches. Simply fire another bullet and look for values that have gone down, and do searches without firing a bullet first and look for values that have stayed the same.

5. You could also try firing all the bullets in the clip, and when the game provides us with a new clip, we could search for a value that has gone up.

Through this process of searching and then searching again only within the results of the last search, we narrow down the number of possible locations for the value we want to find until we have only a manageable number to check. Less than ten is perfect.

False Positives

In the above example, it's likely the number of possible memory locations never falls below two, because when you fire your gun there's usually two things that happen. The actual place in the memory that records the number of bullets you have goes down, and the place in memory storing the graphic that tell you how many bullets you have changes.

It’s quite possible by watching the memory locations change as you fire the bullets to think that you have found the place where the number of bullets you have is stored when in fact you’re actually looking only at the location that controls the graphics that tell you how many bullets you have.

If you tried to create a code for infinite bullets but accidentally wrote a value to the location that controls the graphics for the number of bullets and not the location of the number of bullets itself, when you fire your gun the number of bullets would appear to stay the same, but eventually you would run out of bullets anyway!

It’s things like this you need to look out for when you test your codes.

Example 2: Searching for Freeze Timer

1. In this example, we have little control over time itself in the game, other than for as long as the game is not paused, the timer will certainly be going up (if it measures how long you’re taking to complete a level), or down (if you’re playing against a timer). Start with a New Search and get a dump of the memory at the start of the game. Let’s assume we’re dealing with the former, maybe a racing game where your lap times are recorded by a timer which (of course) increases as you play.

2. Without moving or doing anything other than letting a few seconds pass, we then search again, asking Trainer Toolkit to show us only the values that have *increased* since the last search.

3. We can then continually repeat this process as we try to eliminate other spurious memory locations from our search.

4. We could also try pausing the game and asking Trainer Toolkit to provide us with a list of locations that have *not* changed since the last search.

As with the previous example, we should be mindful that there might be values that are always changing when the timer is running. The first is the timer itself; the second is the area of memory that controls the graphics for the timer display.

3.d. Finding the Right Address

Once you’ve used the search techniques described in the previous section to identify some possible memory locations, you can use other tools within Trainer Toolkit to identify the correct address from the list of possible locations you have found.

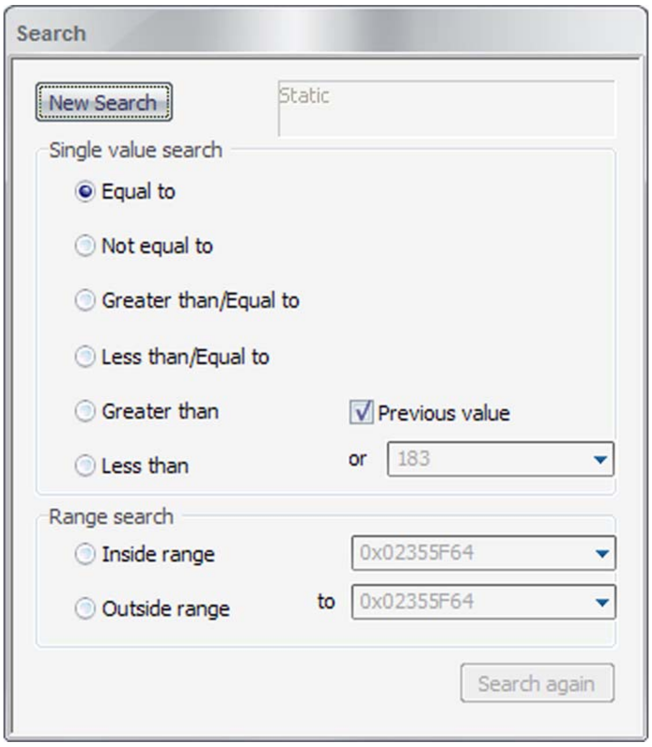
Trainer Toolkit has a variety of functions (explained in detail in the next section) which can help you identify the correct locations. Some of these functions allow you to watch selected memory locations real-time and others allow you to actually modify the value at a location and examine the result.

Try using the ‘Hex View’ feature to look at a memory location you are interested in. Enable ‘Auto refresh’ and then watch to see if the value to changes in a way consistent with what you are looking for.

If you have a number of different possible memory locations, try adding them to your ‘Watched Locations’ list. This acts as a clip-board of possible memory locations from where you can try ‘Poking’ different addresses with a chosen value and looking at the result. As you rule out locations, remove them from your Watched Locations list.

4. Overview of the Trainer Toolkit Software

4.a. Search



The *Search* panel is the starting point for all your game training projects. Not until you have performed a number of search operations and narrowed down your options will you know where to look in the DS's memory for the function you are trying to modify.

When you start a new training project you will need to perform a 'New Search'. The 'Search again' button is used to search within the results of the previous search.

Single Value Search

Use the radio buttons to select the operator appropriate to your search requirements.

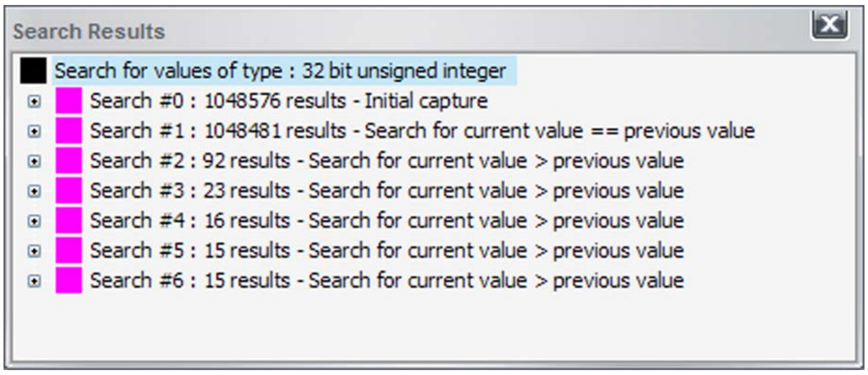
If the 'Previous value' box is ticked the comparison will be made with the values of previous searches. If the box is unchecked then the comparison will be made with the value entered in the text box.

Range Search

If you know you are looking for a specific value stored somewhere in your DS's memory then you can perform what is called a 'Range Search'. Use the radio buttons to control whether you are looking for a specific value (or range of values) or excluding a value (or range of values).

For example, to perform a search for all memory locations containing a value between 1000 and 2000 you would use a range search 'Inside range' from `1000` to `2000` (or if you wanted to use hex you could type from `0x000003E8` to `0x000007D0`).

4.b. Search Results



```
Search for values of type : 32 bit unsigned integer
  Search #0 : 1048576 results - Initial capture
  Search #1 : 1048481 results - Search for current value == previous value
  Search #2 : 92 results - Search for current value > previous value
  Search #3 : 23 results - Search for current value > previous value
  Search #4 : 16 results - Search for current value > previous value
  Search #5 : 15 results - Search for current value > previous value
  Search #6 : 15 results - Search for current value > previous value
```

The *Search Results* panel displays the results from each search you have performed (see 4.a.). Each search is shown sequentially alongside the number of results for a given search and the type of search that was performed. Results can be collapsed and expanded using the [+] symbol.

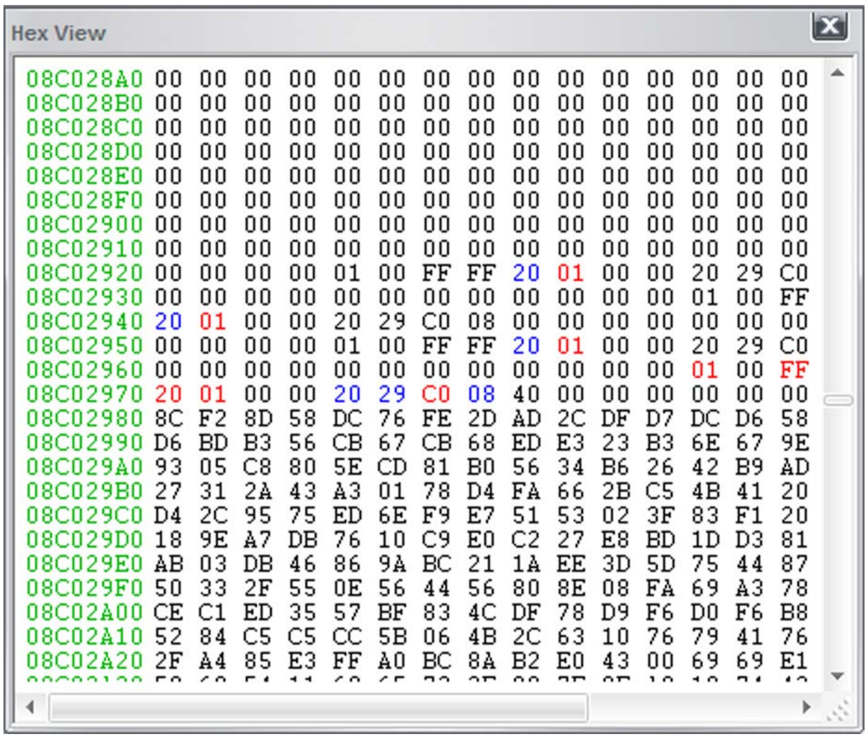
To see the actual results themselves for the search, double click on a search in the *Search Results* window (if there are lots of results they will be limited to 1000) and the results will be launched in a new window.

Rolling back

One of the most useful features of the *Search results* window is the ability to ‘roll back’ to the results set of any one of the searches in the results window. Rolling back means that any future search comparisons will apply to the results of the search that is rolled back to (not the latest set of results) and can be very useful if you have mistakenly ruled out the actual value you are looking for. You’ll know this has happened if you end up with no results!

To roll back to a previous results set, highlight the search you would like to roll back to (in the Search Results window and right click. Click ‘Roll Back’.

4.c. Hex View



```
08C028A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
08C028B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
08C028C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
08C028D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
08C028E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
08C028F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
08C02900 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
08C02910 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
08C02920 00 00 00 00 01 00 FF FF 20 01 00 00 20 29 C0
08C02930 00 00 00 00 00 00 00 00 00 00 00 00 01 00 FF
08C02940 20 01 00 00 20 29 C0 08 00 00 00 00 00 00 00
08C02950 00 00 00 00 01 00 FF FF 20 01 00 00 20 29 C0
08C02960 00 00 00 00 00 00 00 00 00 00 00 00 01 00 FF
08C02970 20 01 00 00 20 29 C0 08 40 00 00 00 00 00 00
08C02980 8C F2 8D 58 DC 76 FE 2D AD 2C DF D7 DC D6 58
08C02990 D6 BD B3 56 CB 67 CB 68 ED E3 23 B3 6E 67 9E
08C029A0 93 05 C8 80 5E CD 81 B0 56 34 B6 26 42 B9 AD
08C029B0 27 31 2A 43 A3 01 78 D4 FA 66 2B C5 4B 41 20
08C029C0 D4 2C 95 75 ED 6E F9 E7 51 53 02 3F 83 F1 20
08C029D0 18 9E A7 DB 76 10 C9 E0 C2 27 E8 BD 1D D3 81
08C029E0 AB 03 DB 46 86 9A BC 21 1A EE 3D 5D 75 44 87
08C029F0 50 33 2F 55 0E 56 44 56 80 8E 08 FA 69 A3 78
08C02A00 CE C1 ED 35 57 BF 83 4C DF 78 D9 F6 D0 F6 B8
08C02A10 52 84 C5 C5 CC 5B 06 4B 2C 63 10 76 79 41 76
08C02A20 2F A4 85 E3 FF A0 BC 8A B2 E0 43 00 69 69 E1
```

The *Hex View* is a really useful aspect of Trainer Toolkit and has a variety of uses. Open *Hex View* by choosing 'Tools > Show Hex View' from the main menu in Trainer Toolkit.

The *Hex View* shows the memory address down the left hand side (in green) and then the values of the memory locations adjacent. The *Hex view* will colourise values that have just changed and can be set to auto-refresh (see below) to provide a virtually real-time view of your DS's memory.

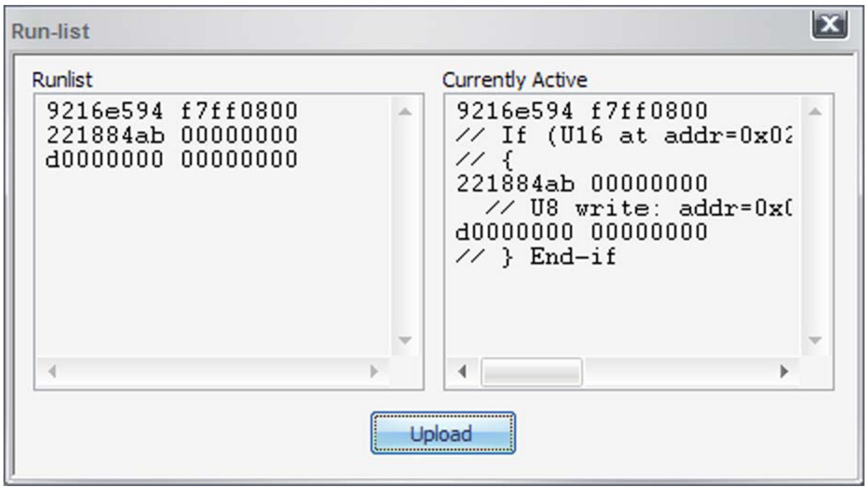
Hex View is NOT read only! - You can actually make modifications directly to the memory locations you are looking at by selecting them and typing in new values.

Functions

When you right click anywhere in the *Hex view* you will be presented with a menu with the following function.

Refresh	When you do not have 'Auto refresh' enabled, use this menu option to refresh the *Hex view*.
Auto-refresh	Enable this option to provide a virtually real-time hex view of your DS's memory. When enabled, the hex view will constantly update, showing the current values at the visible memory locations.
Goto address	Because the *Hex view* displays the entire contents of the DS's memory, you will normally need a means of jumping to a particular location. Right click anywhere in the *Hex view* and choose 'Goto Address'. You will be prompted to enter the memory address to goto, enter an address (using the 0x prefix) and hit 'Goto'.
Search/replace	Use search and replace to make changes to values in the DS's memory.
Fit to window	Control how the data is displayed in the *Hex view*
Load from file	You can load a complete or partial memory dump using this option in the *Hex view*.
Save to file	You can save out a complete or partial memory dump using this option is the *Hex view*. This functionality is covered in section 7.d.

4.d. Run-list



```
9216e594 f7ff0800
221884ab 00000000
d0000000 00000000
```

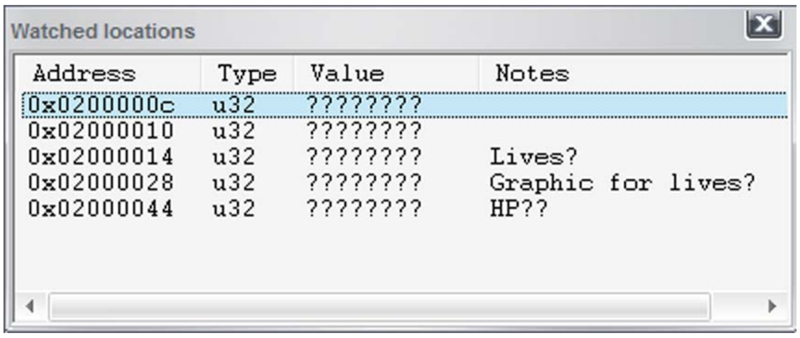
```
9216e594 f7ff0800
// If (U16 at addr=0x02
// {
221884ab 00000000
  // U8 write: addr=0x(
d0000000 00000000
// } End-if
```

The *Run-list* is another core aspect of Trainer Toolkit and is a real-time list of what Action Replay codes are currently enabled. When the 'Currently Active' window is empty, no Action Replay codes are enabled.

Use the *Run-list* to test codes whilst you are working on them. Paste your codes into the Runlist window (make sure you remove any 0x's from the start of your codes) and then hit the 'Upload' button to send them activate them in game. To turn off any codes simply clear the Runlist window and click 'Upload'.

The *Run-list* can be used to test any combination of Action Replay codes, included nested conditional codes, in fact, anything you can run on an Action Replay!

4.e. Watched Locations


Watched locations

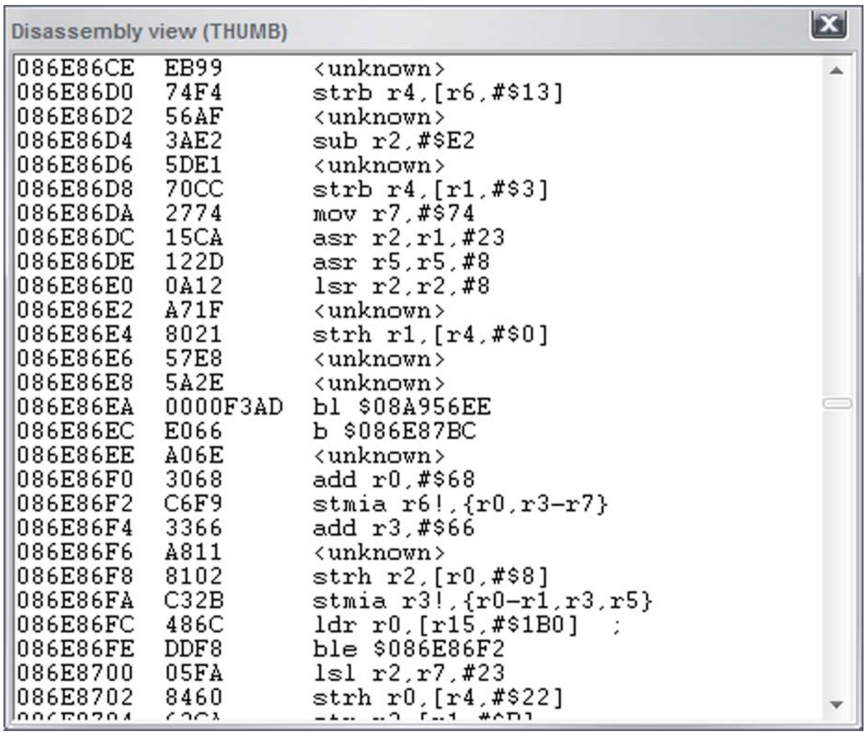
Address	Type	Value	Notes
0x0200000c	u32	????????	
0x02000010	u32	????????	
0x02000014	u32	????????	Lives?
0x02000028	u32	????????	Graphic for lives?
0x02000044	u32	????????	HP??

The *Watched Locations* panel is like a clip-board you can use to store a list of memory locations you are interested in. Click 'Tools > Watched Locations' to open the panel.

It's most likely that you'll add locations to the *Watched Locations* panel by right click on a memory location in the Results for a particular search (double click a search in the Search Results panel to open the results panel). In the popup menu, choose 'Add to watchlist', you will even be given the option to add notes to accompany the watched location.

Once a location has been added to the Watched Locations panel you will be able to monitor the value at the location and also 'Poke' a value to the location by double clicking it. This allows you to perform a one-time write of a new value to the location. This is a very quick way to test a location.

4.f. Disassembly View



```
086E86CE  EB99      <unknown>
086E86D0  74F4      strb r4,[r6,#$13]
086E86D2  56AF      <unknown>
086E86D4  3AE2      sub r2,#$E2
086E86D6  5DE1      <unknown>
086E86D8  70CC      strb r4,[r1,#$3]
086E86DA  2774      mov r7,#$74
086E86DC  15CA      asr r2,r1,#23
086E86DE  122D      asr r5,r5,#8
086E86E0  0A12      lsr r2,r2,#8
086E86E2  A71F      <unknown>
086E86E4  8021      strh r1,[r4,#$0]
086E86E6  57E8      <unknown>
086E86E8  5A2E      <unknown>
086E86EA  0000F3AD  bl $08A956EE
086E86EC  E066      b $086E87BC
086E86EE  A06E      <unknown>
086E86F0  3068      add r0,#$68
086E86F2  C6F9      stmia r6!,{r0,r3-r7}
086E86F4  3366      add r3,#$66
086E86F6  A811      <unknown>
086E86F8  8102      strh r2,[r0,#$8]
086E86FA  C32B      stmia r3!,{r0-r1,r3,r5}
086E86FC  486C      ldr r0,[r15,#$1B0]  ;
086E86FE  DDF8      ble $086E86F2
086E8700  05FA      lsl r2,r7,#23
086E8702  8460      strh r0,[r4,#$22]
```

Included in Trainer Toolkit is a basic *Disassembly View* which can be accessed by choosing 'Show Assembler View' in the Tools menu.

The *Disassembly* view will not provide a complete disassembly of the code as this is beyond the scope of Trainer Toolkit but can provide some useful insights for the trained eye. For those seeking full disassembly we would suggest a professional package such as IDA.

4.g. Updating your Trainer Board Firmware

The firmware on your Trainer Board (the GBA port cartridge) can be updated, should a firmware update become available, by choosing 'Tools > Update Trainer Firmware' in Trainer Toolkit.

Use the browse dialogue to select a firmware file and choose OK.

5. Finding Your Own Cheat Codes

5.a. Set-up Checklist

Before you hack your first code, follow this checklist to make sure you have everything in place, ready to go:

- The Trainer Board is connected to the GBA cartridge slot on your DS.
- The Trainer Board is connected to your PC using the supplied mini-USB to USB cable.
- The Action Replay Trainer cartridge is in the DS cartridge slot on your DS.
- The Trainer Software is installed on your PC.
- The Trainer Board drivers have been installed on your PC.
- You have a DS game ready to hack!

If you've completed the above steps, you're ready to power up your DS console and begin hacking your first code!

5.b. Hacking Examples – Step-by-Step

The easiest way to learn how to use trainer toolkit is to follow one or more of the examples below using your own copy of the game to mirror each step in the example.

Choose an example which uses a game you own or can borrow. If you don't have access to any of the games used in the examples, don't worry. Just read through them and try to follow what's happening in each step before you try your own games.

5.b.i. Castlevania: Dawn of Sorrow

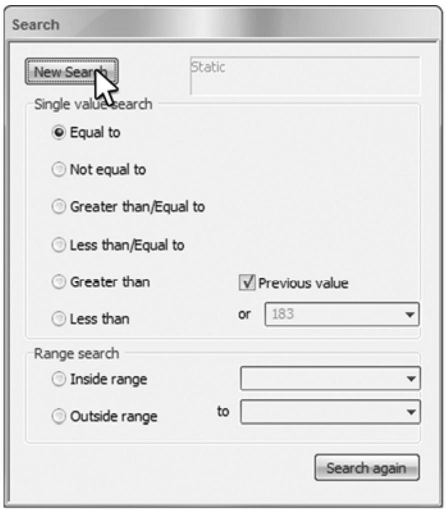
Code to find:	**Infinite Health (HP)**
Region:	USA
Difficulty:	[✓ 1] 2 3 4 5

Code Introduction

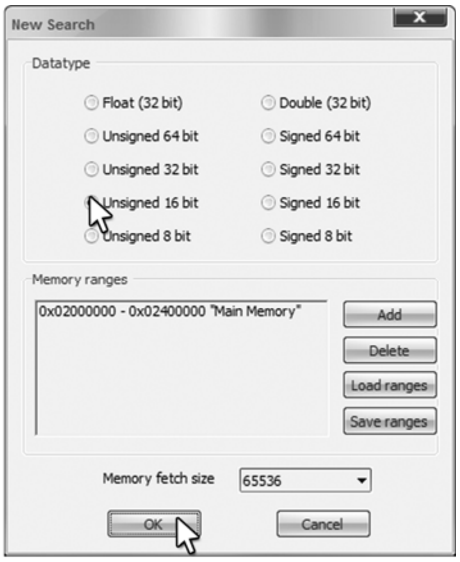
In this example, we try to find the memory location that stores the amount of health (HP) our character has. We will perform a number of searches, each time after having been 'hit' by the skeleton at the start of the level (reducing our HP), each time looking for values that have gone down. Once we find the correct location, we will write the value '99' to it, effectively giving us infinite health.

Step by Step

Load 'Castlevania: Dawn of Sorrow' and start a new game. When the game loads and you're standing on the street, don't move. Just click the 'Start' button to pause the game.



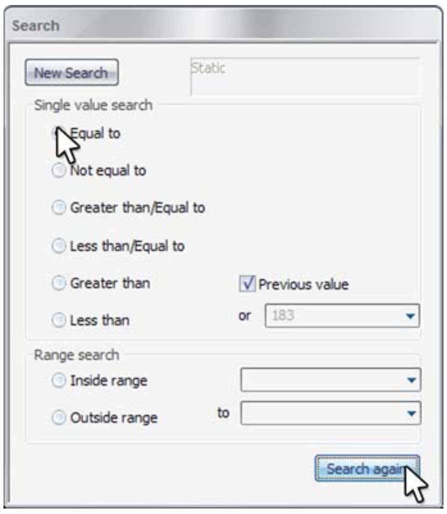
Equal to [✓]
Previous value [✓]
or 183



0x02000000 - 0x02400000 "Main Memory"

Memory fetch size 65536

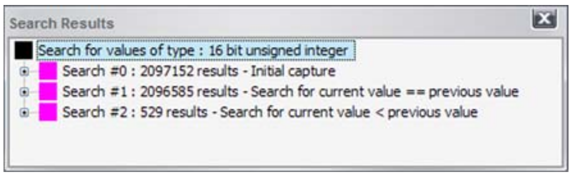
1. In the Search window, click 'New Search' and when prompted, choose 'Unsigned 16bit' and click OK. This starts the first dump of the DS's memory to your trainer.



[✓] Previous value
or 183

2. Now we'll let a few seconds pass before setting the 'Single search value' to 'Equal to' and clicking 'Search again', to search within the previous results. This is because we know our health hasn't changed, so we can narrow down the search to only values that haven't changed since the last search.

3. Now walk left slightly towards the skeleton. He'll react by throwing something at you. Let it hit you so that your HP goes down slightly, then pause the game once more. Set the 'Single search value' to 'Less than' (since you know your HP has just gone down) and click 'Search again'.

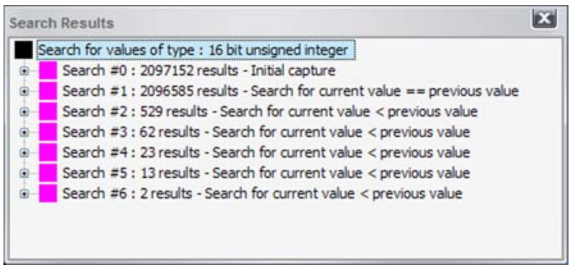


```
Search for values of type : 16 bit unsigned integer
  Search #0 : 2097152 results - Initial capture
  Search #1 : 2096585 results - Search for current value == previous value
  Search #2 : 529 results - Search for current value < previous value
```

4. Your search results window should now look similar to one shown above. Notice you've narrowed your possible locations down from 2,097,152 to 529. Let's do a few more searches to bring that number down even further.

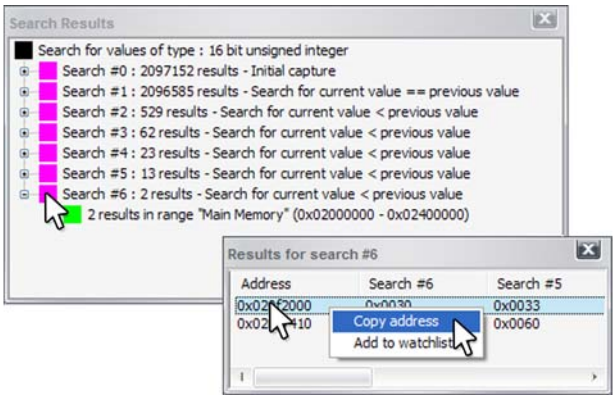
5. Unpause, and let the skeleton hit you again before pausing again. Leave the 'Single search value' set to 'Less than', and click the 'Search again' button.

   Repeat the above step three more times, and you should be left with only two possible results. Further searches won't reduce this number any further (try if you like, it won't do any harm).



```
Search for values of type : 16 bit unsigned integer
  Search #0 : 2097152 results - Initial capture
  Search #1 : 2096585 results - Search for current value == previous value
  Search #2 : 529 results - Search for current value < previous value
  Search #3 : 62 results - Search for current value < previous value
  Search #4 : 23 results - Search for current value < previous value
  Search #5 : 13 results - Search for current value < previous value
  Search #6 : 2 results - Search for current value < previous value
```

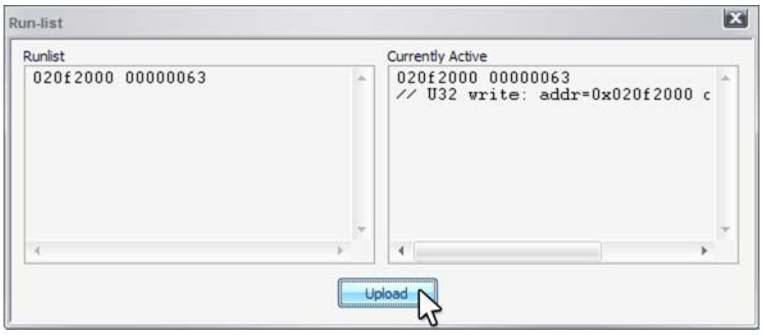
6. Now we have only two possible memory locations where HP is being stored, we need to find out which is the correct one. With so few options, the quickest thing to do in this case is try a memory location and see if it works.



Search for values of type : 16 bit unsigned integer
Search #0 : 2097152 results - Initial capture
Search #1 : 2096585 results - Search for current value == previous value
Search #2 : 529 results - Search for current value < previous value
Search #3 : 62 results - Search for current value < previous value
Search #4 : 23 results - Search for current value < previous value
Search #5 : 13 results - Search for current value < previous value
Search #6 : 2 results - Search for current value < previous value
2 results in range "Main Memory" (0x02000000 - 0x02400000)

Address	Search #6	Search #5
0x020f2000	0x0030	0x0033
0x02 410		0x0060

7. Double-click on the last search in the ‘Search Results’ list so the ‘Results for search…’ window opens. In the new window, right-click on the first address “`0x020F2000`” and choose ‘Copy address’.



Runlist

```
020f2000 00000063
```

Currently Active

```
020f2000 00000063
// U32 write: addr=0x020f2000 c
```

8. To test the address, we need to create a code and make it active within the game. Click on ‘Tools’ and then ‘Run-list’. The Run-list is a sort of real-time Action Replay where you can enable and disable codes at any time. Right-click in the left-hand Run-list window, and choose ‘Paste’ to paste the memory location copied from the Results window.

   After you’ve pasted the address, delete the “`0x`” from its start and then add a space after it, to separate it from the value you add next. Let’s test the code by trying to write a value of ‘99’ (63 in hex), which we do by adding the following to the right of the address in the Run-list: ‘`00000063`’. The finished code looks like this:

   ```
   020F2000 00000063
   ```

   Click the ‘Upload’ button to activate this code in your game now.

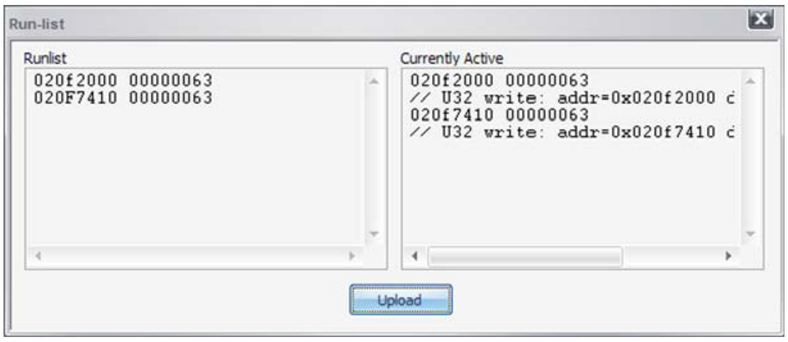
9. Now that we are constantly writing the value of ‘99’ to the memory location “`020F2000`” we can un-pause the game and see if we have given ourselves infinite health. After un-pausing, wait a couple of seconds to be hit again by the skeleton. Unfortunately, when you get hit you notice your health (HP) still goes down. This must be because we picked the wrong one of the two values. Let’s pause the game (so we don’t die!) and look again at the addresses in our search results. Right-click on the second address “`020F7410`” and choose ‘Copy address’.

   Return to the Run-list window and delete the first address you tried (the `020F2000` bit) and replace it with “`020F7410`” so that your code looks like:

   ```
   020F7410 00000063
   ```

   Click the ‘Upload’ button again to write the value ‘99’ to the new memory location. Un-pause the game and let the skeleton hit you again to see if this new location works. Congratulations! You should now find your HP rating never changes from ‘99’.

Incidentally, the other memory location is used to control the on-screen graphic showing how many HP you have. You could happily write ‘99’ to both memory locations like this:



```
020f2000 00000063
020F7410 00000063
```

```
020f2000 00000063
// U32 write: addr=0x020f2000 c
020f7410 00000063
// U32 write: addr=0x020f7410 c
```

```
020F2000 00000063
020F7410 00000063
```

Now the onscreen HP bar always stays full too (even though this doesn’t effect anything in game).

5.b.ii. New Super Mario Bros

Code to find:	**Have 99 coins**
Region:	USA
Difficulty:	[✓ 1] 2 3 4 5

Having completed the steps in the checklist (Section 4.a.), switch on your DS console and when prompted, swap your Action Replay Trainer Cartridge for your copy of New Super Mario Bros. Tap the screen to launch the game.

Code Introduction

In this example we’re looking for the memory location that stores the number of coins you’ve collected. We do this by performing sequential searches after collecting coins one at a time, and looking for memory locations that store values equal to the number of coins. Once we find the location, we’ll create a code that writes the value ‘99’ to it.

Step by Step

When the game loads, choose to play a ‘Mario Game’ and then select a blank save file to start a new game. Let the game start, run right and press ‘A’ to begin world 1-1.

The number of coins we have collected is shown in the top-left corner of the screen. We must now find where in the DS’s memory the current number of coins is stored, so we can write our own value to it.

1. As you start the level, open your Trainer Toolkit PC software and click the ‘New Search’ button. When prompted, choose ‘Unsigned 8-bit’ (the reasons for this are explained in a later chapter).

2. Once the dump is complete, you see the ‘Search Results’ window displayed in the trainer software. Return to the Search window and in the ‘Single search value’ section, un-tick the box marked ‘Previous value’ and set the search type to ‘Equal to’. Now enter the number ‘0’ in the box underneath, marked ‘Previous value’. Click

‘Search again’.

3. Now collect a single coin. Once you have done this, click the ‘Start’ button to pause the game and change the value ‘0’ to the value ‘1’ in the box underneath ‘Previous value’ in the search window before clicking ‘Search again’. Previously, you looked for the value ‘0’, as you had zero coins. Now you’re looking for ‘1’, as you have one coin. A value which read ‘0’ after the previous search but reads ‘1’ now might be the one that stores the number of coins…

4. Un-pause the game, collect one more coin and then pause again. Change the value in the search window to ‘2’ and click ‘Search again’. See what we’re doing? Each search looks for the current number of coins you hold. Any location which didn’t read ‘0’ when you had no coins, ‘1’ when you had a single coin and ‘2’ now you have two coins can’t possibly be the value which stores the number of coins, and is eliminated by our searches.

   You should notice that the ‘Search results’ window is displaying the results of each search along with the number of results after each search. If you have followed the above steps you should now have only two results.

5. Repeat the process one more time, un-pausing the game, collecting a third coin, pausing, changing the search value to ‘3’ and clicking ‘Search again’. This produces a search with only one result; you’ve now found the place where New Super Mario Bros stores the number of coins you hold!

6. The next step is to check we have the right location by trying to write our own value to it. Double-click on Search #4 to look at the actual results for the search. In the window that opens, you see one address listed: “`0x0208B37C`”. Right-click on the address and choose ‘Copy Address’.

7. Click ‘Tools’ in the menu, and choose ‘Show Run-list’. The Run-list is a sort of real-time Action Replay where you can enable and disable codes at any time. Right-click in the left-hand ‘Run-list’ window and choose ‘Paste’ to paste the memory location copied from the Results window. Next, remove the “`0x`” from the start of the memory location to leave “`0208B37C`”.

8. Now we need to pick a value to write to this location. Let’s choose ‘99’. First, we need to convert ‘99’ into hexadecimal (since all values used by Action Replay are in hex). Using the conversion table at the back of this manual, we see this value is ‘63’. Values are entered in eight-character blocks, so we enter ‘63’ as “`00000063`”. This value is typed along side the memory location in the Run-list window (separated from the memory location with a space), so the finished code reads: “`0208B37C 00000063`”.

9. It’s now time to activate our code. Click the ‘Upload’ button to send the code to the ‘Currently Active’ list, which is the Action Replay codes that are enabled in your game. When you do so, you see the number of stars in the top-left corner of the screen change to ‘99’.

   Congratulations, you have created your Action Replay code!

   Note: If you received the message ‘Failed to parse the runlist’, it’s probably because you have entered an incorrect number of characters as the value or have forgotten to take the “`0x`” off the address. Check the code against the one listed above and try again.

5.b.iii. Super Mario Kart DS

Code to find:	**Freeze Timer**
Region:	USA
Difficulty:	1 2 [✓ 3] 4 5

Code Introduction

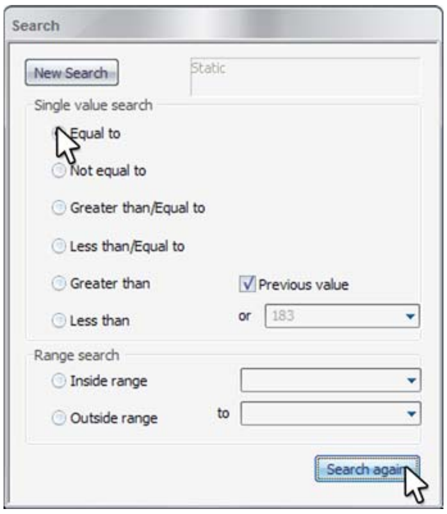
In this example, we're looking for a way to freeze the timer in Super Mario Kart DS. This code is more difficult to find than the two previous examples, because the memory location where the timer is stored varies from level to level. This means if we make a code that freezes the timer in the first Time Trial level, it won't work in a different Time Trial level, in a Multiplayer or Grand Prix level.

Because the location where the timer is stored changes, to create this code we need to find another memory location too, the location that stores what is called a 'pointer' to the timer's memory location.

Step by Step

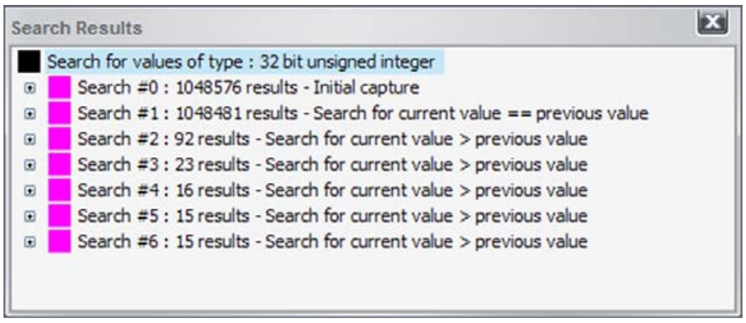
Load Trainer Toolkit and then swap your Action Replay Trainer Cartridge for Mario Kart DS. When Mario Kart DS loads; choose 'Single Player' and then 'Time Trials' from the menu. Choose the default character, kart and course. Turn off 'ghost data' if asked, then click 'OK'.

1. Once the race starts, pause the game without having moved at all. In Trainer Toolkit, perform a 'New Search' and select 'Unsigned 32-bit'.



2. When the search is completed, leave the game paused and in the Search window, set the Single value search to 'Equal to', tick the box for 'Previous value' and click 'Search Again' (since we know the timer value will not have changed).

3. Next, un-pause the game but don't move the kart in any way. Let a couple of seconds pass, and then pause again. In the Search box, change the 'Single search value' to 'Greater than' and click 'Search again'. You notice this search returns around 92 results.

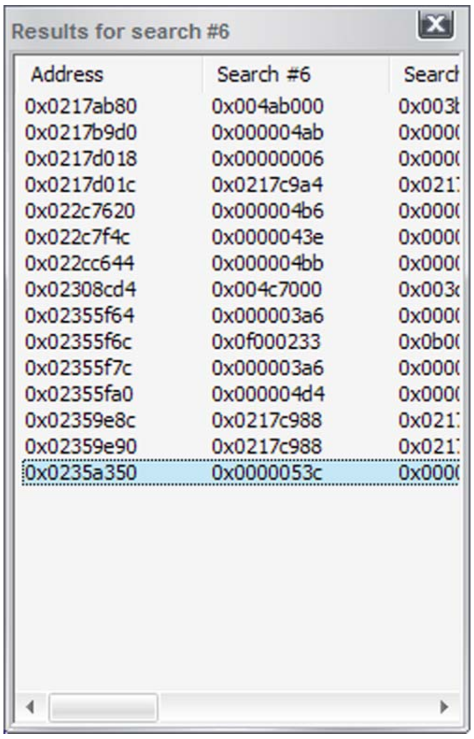


```
Search Results
■ Search for values of type : 32 bit unsigned integer
  Search #0 : 1048576 results - Initial capture
  Search #1 : 1048481 results - Search for current value == previous value
  Search #2 : 92 results - Search for current value > previous value
  Search #3 : 23 results - Search for current value > previous value
  Search #4 : 16 results - Search for current value > previous value
  Search #5 : 15 results - Search for current value > previous value
  Search #6 : 15 results - Search for current value > previous value
```

4. Let us repeat the previous step several times to reduce the number of memory locations in our results. By un-pausing, re-pausing and searching again using the 'Greater than' operator, we can reduce the number of possible locations to around 15.

5. With only 15 locations (we don't seem to be able to reduce this number much further through searches), we can try each of these memory locations in turn to see if we can identify the right one. Double-click on the last search in your 'Search results' window to view the actual results themselves.

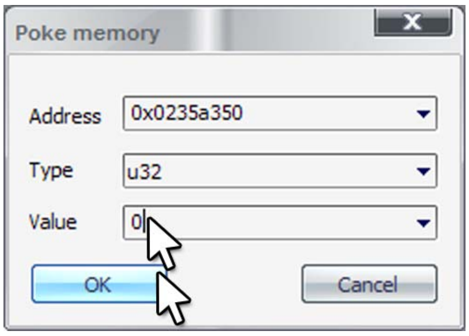
   We will now use a feature called 'Poke memory' to quickly try out our different locations. Your results window should look something like this:



Results for search #6

Address	Search #6	Searcl
0x0217ab80	0x004ab000	0x003
0x0217b9d0	0x000004ab	0x000
0x0217d018	0x00000006	0x000
0x0217d01c	0x0217c9a4	0x021
0x022c7620	0x000004b6	0x000
0x022c7f4c	0x0000043e	0x000
0x022cc644	0x000004bb	0x000
0x02308cd4	0x004c7000	0x003
0x02355f64	0x000003a6	0x000
0x02355f6c	0x0f000233	0x0b0
0x02355f7c	0x000003a6	0x000
0x02355fa0	0x000004d4	0x000
0x02359e8c	0x0217c988	0x021
0x02359e90	0x0217c988	0x021
0x0235a350	0x0000053c	0x000

   We can 'poke' a memory location by simply double-clicking on it in the results window. Try double-clicking the last entry in the list of results. You should see a window open like this:



Address 0x0235a350

Type u32

Value 0

'Poking memory' is simply a way of performing a 'one time' write of a value to a memory location. In this case, we will try poking the value '0' to the first memory location we try. We then look in the game and see if the time jumps down to '0', and then starts counting up again, since poking a memory location only performs a one-time write, not a continuous write.

Un-pause the game so it's running, and in the 'Poke memory' panel, type '0' in the 'Value' box and click 'OK'. Take a look at the timer in-game and see if it suddenly restarts. If you poked the address "`0x0235A350`", you find the timer doesn't change. We clearly need to try some more addresses.

6. If we're going to try poking a number of locations to find the right one, we can start by choosing the most likely locations. In this example, because we know the time starts at '0' and counts up, it's likely to still be quite a low number (maybe it counts in milliseconds), so we can look down the list of possible locations and identify those that look quite low.

   Working up, from the bottom of our list (yours should be very similar), we might choose to try:

   `0x0235A350`, `0x02355FA0`, `0x02355F7C`, `0x02355F64`, `0x022CC644`, `0x022CF4C`, `0x022C7620`, `0x0217B9D0`.

   Since all these locations have values like:

   `0x0000053C`, `0x000004D4`, `0x000003A6` etc, which all look like they could be timer values.

   Try poking, one at a time, each of the memory locations listed above with the value '0' to see which one resets the timer. After several tries, you should find the location "`0x02355F64`" holds the value for the timer. As soon as you poke that location with a '0', the timer is reset.

7. Once you have found the memory location that stores the timer for this level, it's now worth testing this location to see if it works for other levels or types of race. One way we can do this is to add this memory location to our Run-list using right-click, 'Copy location' and then pasting it into the left-hand window of the Run-list panel, remembering to remove the '0x' from the start. Add the value "`00000000`" to the right of the memory location (to write the value '0' to the location). You should have a code in the Run-list like this:

   ```
   02355F64 00000000
   ```

   Click the 'Upload' button to start the constant write of the value '0' to the memory location. If you un-pause the game, you should see that the timer is now frozen at 0:00 (the milliseconds digit is flickering). This is because the code you have created and run in the Run-list constantly writes the value of '0' to the memory location, unlike when you 'poke' a value and it's written only once.

With this code running, we can quit this race and try a different race. Press 'Pause' then choose 'Quit'. Now start a new race choosing 'Single Player', 'Grand Prix', 50cc and then choose the default character, kart and circuit.

When the race starts you notice that the timer is running! This shows us our 'Freeze Timer' code doesn't work on all levels. In fact in this case, it only works on the race we created it for. Clearly, we're not done yet with this code!

8. What we need to do now is find out whether the memory location used to store the timer varies just from track to track, or whether it moves location each time a new race is started. The way we do this is by going back to the original track and seeing if our code is still working. Quit the existing race and start a new one, choosing: 'Single Player', 'Time Trials', 'Time Trials' and then the default character, kart and circuit (the same as we chose originally). Again, choose no ghost data.

   When the race starts, you notice the timer isn't changing. This shows the memory location storing the timer does not change every time a race starts. Instead, it most likely just uses a different location for each race.

   Since we know the game needs to know where to look for the timer too, there must be a memory location that stores the *address* of the timer for that level. This location is called a 'pointer'.

   There might be a pointer specifically for the timer, or there might be a pointer that points to the start of a number of locations that store other values that move from race to race as well as the timer. Only further investigation will reveal which way this game works.

   Let's look for the memory location that stores the pointer to the timer. For simplicity, we start by looking for a memory location storing the address of the location we found in the original search.

   Since we know the address of the timer on the 'Time Trails' first level to be "`02355F64`", we can perform a new search looking for a memory location storing that value.

   In the 'Search' window, click on 'New search' to clear the previous search and get a fresh dump of the DS's memory. Once complete, click on 'Inside range' under 'Range search' in the Search window, and set the range to be from `0x02355F64` to `0x02355F64`. This will return all addresses storing exactly the value of the address of the timer for this race.

   As it happens, this returns a result showing us there's a pointer that points specifically to the timer location. In other games, this may have returned no results, meaning the pointer may point only to the start of a memory location that stores many values. If this was the case, we would have needed to be less narrow in our 'Range search' and found the address near `0x02355F64` that the pointer points to and then use an 'offset' to identify the actual address of the timer.

9. Looking at the only result for the search, we can see the pointer for the timer is stored at `0x021755FC`. Knowing this location, we can build our Action Replay code using the 'Load offset register' code-type.

   Looking at the back of this manual, we can see the 'Load offset register' code type uses the format:

   ```
   BXXXXXXX ????????
   ```

   Where `XXXXXXX` is the address of the pointer and `????????` is the offset from the pointer to the value we want (our offset is ‘0’ since the pointer points directly to the address storing the timer). Therefore, we use the code in the following way:

```
B21755FC 00000000
```

With the pointer address loaded into the offset register, the value we now write is written to the address given by the pointer. Since we don’t need to give an address, we provide the value ‘0’ as follows:

```
B21755FC 00000000
00000000 00000000
```

We now have a code that writes the value ‘0’ to the address indicated by the pointer at address `0x21755FC`, which is location of the timer for any level.

10. All we need to do now is add a condition to the code to make sure that it doesn’t crash the game if the pointer hasn’t been set (like when you first start the game). We do this by adding a ‘32bit If not equal’ code to the start. We will simply use this code to check whether the value of the pointer we identified in the previous section has been set to anything (so is *not* equal to `00000000`).

    We can find the code-type for a ‘32bit If not equal’ code at the back of the manual:

    ```
    6XXXXXXX YYYYYYYY
    ```

    Where ‘x’ is the address and ‘y’ is the value. This gives us:

    ```
    621755FC 00000000
    ```

    Which in English means “If the value at memory location 621755 is NOT equal to “`00000000`” then….”

    So now we can add this condition to the start of our code:

    ```
    621755FC 00000000
    B21755FC 00000000
    00000000 00000000
    ```

    Finally, we simply need to end the ‘If’ which we do with a D2 code-type, meaning our finished code looks like:

    ```
    621755FC 00000000
    B21755FC 00000000
    00000000 00000000
    d2000000 00000000
    ```

    We now have a code that freezes the timer on any type of race and that is stable because it only becomes active once the pointer it uses has been set.

6. Publishing your Action Replay Codes

Once you’ve created your own first killer codes, no doubt you’ll want to share them with the rest of the Action Replay community. There are many ways to do this and no doubt you’ll become more sophisticated in how you share you codes as the codes you create grow in popularity.

6.a. Are Your Codes Ready To Be Published?

The most important thing to ask yourself before you share your codes with the World is “are your codes ready to be published?”

Action Replay gamers around the World are going to search out your codes, spend time entering them in or copying them to their Action Replay and then excitedly loading their game and expecting your codes to do just what they’re supposed to. You owe it to them to ensure that your codes do what they’re supposed to, that they don’t crash a couple of levels in and that they don’t do anything unexpected. The way you achieve this is by thoroughly testing your codes and resolving any issues you encounter.

The very best codes will be easy and fun to use, well tested and totally stable.

Tip: If you’re not sure that a code is 100%, why not upload it to your website or chosen forum as ‘BETA’, that way other users can test the code and provide you with feedback without expecting the code to be perfect first time.

6.b. Share Codes Through Online Forums or Your Website

One of the easiest ways to share your codes is to simply post them on a discussion forum dedicated to new Action Replay DS codes or on your own website.

When you post new codes, be sure to present them clearly, with all the necessary information.

Always include:

- The game’s full name
- The game’s cartridge ID (see below)
- The region of the game (is it USA, European or Japanese?)
- The code’s name (be descriptive)
- The codes themselves, displayed clearly using a fixed width font (if possible).

If you’re not sure that your code is 100% stable, call it a BETA code. This will warn people that the code may crash and encourage them to provide you with feedback.

Game IDs

It is important that you know the *cartridge ID* of a game in order to publish codes for it. Cartridge IDs are codes that are unique to each different version of a game (for example if it has been re-released following a change) and help to ensure that a set of codes was created for the exact version of a game that a user has.

The cartridge ID of the game you are training is clearly displayed on the trainers home screen (on your DS console) when you insert a game cartridge, before you tap to run it.

6.c. Create an XML Code Feed for Action Replay Code Manager

The most sophisticated way to share your codes is by creating an XML feed designed for Action Replay Code Manager (which is included with Action Replay). By creating an unofficial XML *Code Feed* users will be able to subscribe to your feed for instant access to your codes as soon as you add them to your feed.

In order to provide an XML Code Feed you will need somewhere to host your feed, like your own web server or space on a shared server. Once you have created your feed you upload it to your server and provide people with the address of the feed so that they can subscribe.

Users can then add your feed to their subscriptions in Action Replay Code Manager and will have instant access to your codes from within the Code Manager.

Who Should Create an XML Code Feed?

XML Code Feeds were designed with prolific hackers, groups of hackers and large games websites in mind. Although the XML Code Feed is easy to understand and put together by hand, it is ideally suited to being automatically generated by dynamic pages like PHP or ASP, accessing a database of codes.

XML Code Feed Schema

If you would like to build your own XML Code Feed, you should be able to get an understanding of the schema by looking at the official Codejunkies.com XML Code Feeds pre-configured in Action Replay Code Manager.

You will see that the XML is structured as follows (data is for example only):

```
<?xml version="1.0" encoding="UTF-8"?>
<codelist>
       <name>Codejunkies EU Games</name>
       <game>
              <name>Advance Wars Dual Strike</name>
              <gameid>AWRP 1bd98037</gameid>
              <date>2006/08/01 15:08</date>
              <cheat>
                     <name>Infinite Funds In Battle (Press L+R)</name>
                     <codes>94000130 fdff0000 021a7794 05f5e0ff d2000000
00000000 </codes>
              </cheat>

       </game>
</codelist>
```

General methods for creating XML feeds for your server technology will be well documented elsewhere and are beyond the scope of this document.

7. Advanced Techniques

7.a. Using Masks

Masking 16-bit conditional code-types

Masks are traditionally used to 'hide' bits that are not relevant to a condition we're interested in, and can be very useful in a range of circumstances. An example where a mask is often employed is on a memory location that stores which buttons are pressed on your DS at any given time.

Let's say you want to activate a code when button 'Y' is pressed. The value at that memory location is `0000` when no buttons are pushed, and is set to `0002` when 'Y' is pressed. This memory location is set to a different value depending on the button or button combination you press:

'UP'+'Y' is 0012, 'DOWN'+'Y' is `0042`, etc. so the condition 9XXXXXXX `00000002` will only be true when you press button 'Y' alone, and stops working if you press 'DOWN'+'Y' or 'A'+'Y'.

You can use a mask with a 16-bit condition code-type to hide non-relevant bits, meaning that whatever button is pressed in addition to 'Y', the condition is still true and the code is activated.

The 16-bit value `0x0002` equals `0000000000000010` in binary. This means when looking for the button-press 'Y', your condition is only interested in the second of the 16 bits. You can create a mask to hide the 15 other bits by setting them up to '1' and leaving the bit you're interested in set on ‘0’. In this case, mask would be `1111111111111101` in binary, which we convert to `0xFFFD` in Hex.

If you use the condition `9XXXXXXX FFFD0002`, it will always activate when the value of the second bit is ‘1’ at address `XXXXXXXX`.

With this mask in place, the following values would have the following effects:

CODE ACTIVE		CODE IN-ACTIVE	
Hex	Binary	Hex	Binary
`0x0002`	`00000000000000`**`1`**`0`	`0xFFF0`	`11111111111100`**`0`**`0`
`0x0003`	`00000000000000`**`1`**`1`	`0x0001`	`00000000000000`**`0`**`1`
`0xFFFF`	`11111111111111`**`1`**`1`		
`0XFFF2`	`11111111111100`**`1`**`0`		

The values in the left column activate the code because they all have the second bit from the right set to ‘1’, regardless of what other bits are set to. The values in the right column will not activate the code because they all have the second bit to the right set to ‘0’, again, regardless of what other bits are set to.

7.b. Button Press Locations

Sophisticated Action Replay codes often use button presses to activate or deactivate them or to control them in some way (like cycling through items using ‘LEFT’ or ‘RIGHT’). In order to use button presses in our codes, we need a way to know their state.

In the previous section, we looked at using a mask on a 16-bit conditional code type and used the example of the location that stores the current pressed state of the DS’s buttons. If you have not done so, read Section 6.a. for an understanding of how masks can give you better precision when working with button-press states.

To set up a condition on a button press in your own codes, you can find a generic memory location in the hardware registers at address `0x4000130`.

Use the Hex viewer to jump to that address and enable auto-refresh. Try pressing each button on your DS console and watch the value at that address change.

Button(s)	Value	Button(s)	Value	Button(s)	Value
A	FE	A + SELECT	FA	START + UP	B7
B	FD	A + UP	BE	START + DOWN	77
X	n/a	A + DOWN	7E	START + LEFT	D7
Y	n/a	A + LEFT	DE	START + RIGHT	E7
START	F7	A + RIGHT	EE	SELECT + UP	BB
SELECT	FB	B + START	F5	SELECT + DOWN	7B
UP	BF	B + SELECT	F9	SELECT + LEFT	DB
DOWN	7F	B + UP	BD	SELECT + RIGHT	EB
LEFT	DF	B + DOWN	7D	UP + LEFT	9F
RIGHT	EF	B + LEFT	DD	UP + RIGHT	AF
A + B	FC	B + RIGHT	ED	DOWN + LEFT	5F
A + START	F6	START + SELECT	F3	DOWN + RIGHT	6F

The above table is by no means an exhaustive list of all button combinations and their values. If you need values for any other button combinations, just use the Hex viewer to look at the address and note down the value for the button combination you are after.

The hardware register at `0x4000130` actually only maps the original GBA buttons and therefore does not include the states for the 'X' and 'Y' buttons.

7.c. Code-types and Advanced Nesting

With the exception of the 'memcopy' function, code-types can be nested indefinitely for extra precision and advanced hacking.

Here's three examples:

Example 1
Pointer Type – Super Mario 64 - Invincibility

Checks that a pointer has been set, if it has loads the value of the pointer into the stored register and then writes a value of `0x1C` to the stored register location (plus an offset).

Code	Explanation
`6209B450 00000000`	32bit 'If not equal' instruction. Code below will only activate when the value at address `0x0209B450` differs from '0' (this is to prevent the code from being activated before the pointer has been set up at that memory location).
`B209B450 00000000`	Load offset register. Loads the offset register with the data at address `0x0209B450`. In this example it will be a pointer which points to another memory location.
`200006A0 0000001C`	8bit write of `0x1C` to the location set by the offset register + `0x06A0`.
`D2000000 00000000`	End-code instruction. Ends the current repeat block (if any), then performs 'End-if' on any outstanding conditional statements. Also sets 'offset' and 'stored' to zero.

Example 2
Repeat Type - Age of Empires - All Levels Unlocked

Store the value `00030201` at address `0238203C`, then at `02382090`, and keep repeating for another 27 times.

Code	Explanation
`D3000000 0238203C`	Loads the offset register with the value `0238203C`
`D5000000 00030201`	Sets the Stored register with value `00030201`
`C0000000 0000001C`	Set the repeat to `0x1C` (28 in decimal)
`D6000000 00000000`	32bit store and increment. Saves all 32 bits of 'stored' register to address (XXXXXXXX + 'offset'). Post-increments 'offset' by 4.
`DC000000 00000050`	adds `0x0050` to offset register
`D2000000 00000000`	End all Condition & clear up all registers

Example 3
Advanced Nesting - Resident Evil Deadly Silence - Super Item Modifier Slot 1

On inventory menu window, press ‘SELECT’+’L’ or ‘SELECT’+’R’ to modify the object found in the highlighted slot.

Item modifier is at address `0x0213BF2E`. Each time you modify the value at this address, it turns the item in Slot 1 into another item.

Code	Explanation
Increase at address `0213BF2E` when ‘SELECT’+’L’ is pressed…	
`9213BDC2 00000000`	16bit ‘If equal’ instruction. Used here to make sure that the code activates only when you are on the item menu.
`74000100 FF00000C`	16bit ‘If less-than’ instruction. At the address `0x04000100` there is a generic counter. This slows down the code's execution so items appear one by one instead of scrolling through ten items in less than a second.
`7213BF2E FF00004E`	16bit ‘If less-than’ instruction. Makes sure the item count doesn't increase above `0x4e`, which is the last available item (otherwise the game would crash).
`94000130 FDFB0000`	16bit ‘If equal’ + mask. This means the code will activate if button-pressed state equals ‘SELECT’+’L’.
`DA000000 0213BF2E`	16bit load from address `0x213BF2E` into stored register.
`D4000000 00000001`	Add `0x0001` to the value in stored register.
`D7000000 0213BF2E`	16bit store to address `0x213BF2E` the stored register.
`D2000000 00000000`	End all condition & clear up all registers.
Decrease at address `0213BF2E` when ‘SELECT’+’R’ is pressed…	
`9213BDC2 00000000`	
`74000100 FF00000C`	
`8213BF2E FF000001`	
`94000130 FEFB0000`	
`DA000000 0213caee`	
`D4000000 FFFFFFFF`	
`D7000000 0213CAEE`	
`D2000000 00000000`	

7.d. Saving out a Binary File

Another technique used by advanced hackers is dumping out the complete memory as a binary file and then analysing the binary in a disassembler such as IDA*. When disassembled, it’s possible to look for ASCII text such as ‘debug menu’ or ASCII names for items/costumes/characters within the game.

To save out a binary file using Trainer Toolkit for later disassembly, open the Hex View and right click. Choose ‘Save to file’ from the popup menu. On the range panel that appears type `0x02000000` as the ‘From address’ and `0x02400000` as the ‘To address’. You’re asked to choose a destination and name to save the binary file to.

There is also an alternative memory location between `0x037F8000` and `0x03810000` that you may wish to consider though, to date, nothing of interest has been found in this memory range.

Please note: This is very much an advanced technique; you’re on your own!

*IDA is a commercial disassembler and debugger aimed at professional reverse-engineers.

8. Useful Information

8.a. Decimal to Hexadecimal Conversion Table

Here are the hexadecimal equivalents of some decimal values you might use frequently when developing new codes.

Decimal	Hex	Decimal	Hex	Decimal	Hex
0	0	15	F	200	C8
1	1	16	10	300	12C
2	2	17	11	400	190
3	3	18	12	500	1F4
4	4	19	13	600	258
5	5	20	14	700	2BC
6	6	30	1E	800	320
7	7	40	28	900	384
8	8	50	32	999	3E7
9	9	60	3C	1,000	3E8
10	A	70	46	9,999	270F
11	B	80	50	10,000	2710
12	C	90	5A	99,999	1869F
13	D	99	63	100,000	186A0
14	E	100	64	1,000,000	F4240

Tip: Use Windows' built-in calculator (in Scientific Mode) to quickly convert any decimal value you may need into hexadecimal if the number you need isn't in the table above.

8.b. Complete List of Action Replay Engine Code-types

For explanation of symbols used in the code-type table, see key below table.

Code	Function
`0XXXXXXX YYYYYYYY`	32bit write of YYYYYYYY to location: (xxxxxxx + 'offset')
`1XXXXXXX ????YYYY`	16bit write of YYYY to location: (XXXXXX + 'offset')
`2XXXXXXX ??????YY`	8bit write of YY to location: (XXXXXXX + 'offset')
`3XXXXXXX YYYYYYYY`	**32bit 'If less-than' instruction.** If the value at (XXXXXXX or 'offset' when address is 0) < YYYYYYYY then execute the following block of instructions. Conditional instructions can be nested.
`4XXXXXXX YYYYYYYY`	**32bit 'If greater-than' instruction.**

	If the value at (XXXXXXX or ‘offset’ when address is 0) > YYYYYYYY then execute the following block of instructions. Conditional instructions can be nested.
`5XXXXXXX YYYYYYYY`	**32bit ‘If equal’ instruction.** If the value at (XXXXXXX or ‘offset’ when address is 0) == YYYYYYYY then execute the following block of instructions. Conditional instructions can be nested.
`6XXXXXXX YYYYYYYY`	**32bit ‘If not equal’ instruction.** If the value at (XXXXXXX or ‘offset’ when address is 0) != YYYYYYYY then execute the following block of instructions. Conditional instructions can be nested.
`7XXXXXXX ZZZZYYYY`	**16bit ‘If less-than’ instruction.** If the value at (XXXXXXX or ‘offset’ when address is 0) masked by ZZZZ < YYYY then execute the following block of instructions. Conditional instructions can be nested.
`8XXXXXXX ZZZZYYYY`	**16bit ‘If greater-than’ instruction.** If the value at (XXXXXXX or ‘offset’ when address is 0) masked by ZZZZ > YYYY then execute the following block of instructions. Conditional instructions can be nested.
`9XXXXXXX ZZZZYYYY`	**16bit ‘If equal’ instruction.** If the value at (XXXXXXX or ‘offset’ when address is 0) masked by ZZZZ == YYYY then execute the following block of instructions. Conditional instructions can be nested.
`AXXXXXXX ZZZZYYYY`	**16bit ‘If not equal’ instruction.** If the value at (XXXXXXX or ‘offset’ when address is 0) masked by ZZZZ != YYYY then execute the following block of instructions. Conditional instructions can be nested.
`BXXXXXXX ????????`	**Load offset register.** Loads the offset register with the data at address (XXXXXXX + ‘offset’) Used to perform pointer relative operations.
`C??????? NNNNNNNN`	**Repeat operation.** Repeats a block of codes for NNNNNNNN times. The block can include conditional instructions. Repeats blocks cannot contain further repeats.
`D0?????? ????????`	**End-if instruction.** Ends the most recent conditional block.
`D1?????? ????????`	**End-repeat instruction.** Ends the current repeat block. Also implicitly ends any conditional instructions inside the repeat block.
`D2?????? ????????`	**End-code instruction.** Ends the current repeat block (if any), then End-if's any further outstanding

	conditional statements. Also sets ‘offset’ and ‘stored’ to zero.
`D3?????? YYYYYYYY`	**Set offset register.** Loads the offset register with the value YYYYYYYY.
`D4?????? YYYYYYYY`	**Add to ‘stored’.** Adds YYYYYYYY to the ‘stored’ register.
`d5?????? YYYYYYYY`	**Set ‘stored’.** Loads the value YYYYYYYY into the ‘stored’ register.
`D6?????? XXXXXXXX`	**32bit store and increment.** Saves all 32 bits of ‘stored’ register to address (XXXXXXXX + ‘offset’). Post-increments ‘offset’ by 4.
`D7?????? XXXXXXXX`	**16bit store and increment.** Saves bottom 16 bits of ‘stored’ register to address (XXXXXXXX + ‘offset’). Post-increments ‘offset’ by 2.
`D8?????? XXXXXXXX`	**32bit store and increment.** Saves bottom 8 bits of ‘stored’ register to address (XXXXXXXX + ‘offset’). Post-increments ‘offset’ by 1.
`D9?????? XXXXXXXX`	**32bit load "stored" from address.** Loads ‘stored’ with the 32bit value at address (XXXXXXXX + ‘offset’)
`DA?????? XXXXXXXX`	**16bit load ‘stored’ from address.** Loads ‘stored’ with the 16bit value at address (XXXXXXXX + ‘offset’)
`DB?????? XXXXXXXX`	**8bit load "stored" from address.** Loads ‘stored’ with the 8bit value at address (XXXXXXXX + ‘offset’)
`EXXXXXXX NNNNNNNN VVVVVVVV VVVVVVVV * ((NNNNNNNN+7)/8)`	**Direct memory write.** Writes NNNNNNNN bytes from the list of values VVVVVVVV to the addresses starting at (XXXXXXX + ‘offset’)
`FXXXXXXX NNNNNNNN`	**Memory copy.** Copies NNNNNNNN bytes from addresses starting at the ‘offset’ register to addresses starting at XXXXXXXX.

Key to symbols used in the code-type table

Symbol	Meaning
????	Values here don’t matter
xxxx	Address
yyyy	Data
zzzz	Mask
nnnn	Count
vvvv	Direct values
“Offset”	A code-engine register used to hold an address offset

“Stored”	A code engine register used to store data

9. Technical Support & Customer Services

Trainer Toolkit is a technical product and is, by its nature, aimed at technically minded individuals with a working knowledge or interest in computers, programming, games and the Internet.

All available information relating to starting out in game hacking has been compiled to produce this manual. Datel are unable to provide technical support on specific hacking problems or general hacking techniques.

Of course, if you’re experiencing problems with your Trainer Toolkit hardware itself; we’re here to help. Before contact Datel’s technical support or Customer Services staff though, please ensure you have read and understood the content’s of this User Manual.

Contacting Us

When you contact Datel customer services or technical support department, please have ready the version number of the software you are using (normally found on the inner ring on the underside of the software disc) along with when and where you purchased the product.

DATEL CUSTOMER SERVICES EUROPE:

Customers Services,
Datel Ltd,
Stafford Road,
Stone,
STAFFS
ST15 0DG
UNITED KINGDOM

Email: support@datel.co.uk
Web: www.codejunkies.com

UK Technical Support Hotline:

0906 550 1236*

*Calls cost £1 per minute. Lines open Monday-Friday 9am-5pm and 9am-3pm Saturday, excluding Bank Holidays and other National holidays. You will be asked to confirm that you are aged 18 or over and have the permission of the bill payer. UK residents only.

DATEL CUSTOMER SERVICES USA:

ATTN: Customer Services,
Datel Design & Development
15500 Lightwave Drive, Suite 101,
ClearwaterFL 33760

Email: support@dateldesign.com
Web: www.codejunkies.com