



Infinite Selection Time Off  
 Kill Boss with 1 hit Now! Set  
 Enemy never fires at you! Off  
 Infinite Fire Bomb Shot Time Off  
 Always have 1000000 bonus Off  
 Select Level to follow Level 1 ◀Level 6  
 Kill Boss with 1 hit Now! Set  
 Infinite Lives PL1 Off  
 Infinite Bombs PL1 Off  
 Maximum Shot PL1 Off  
 Maximum Combo PL1 Off  
 Invincibility PL1 Off

Cycles 3196  
 Beam 5  
 Beam 240  
 Frame 1027  
 Flags .ZY.X..C  
 PC 0898  
 SP 8000  
 AF 1869  
 BC 180C  
 DE A593  
 HL AC18  
 IX 300A  
 IY FFFF  
 AF2 8384  
 BC2 0000  
 DE2 0001  
 HL2 0000  
 WZ A984  
 R 08

0895 1d a,(\$A9B3)  
 0896 1d 1,a  
 0897 1d a,(h1)  
 089A r1ca  
 089E 3p c,\$0890  
 089F 1d c,(h1)  
 08A1 inc h1  
 08A2 1d b,(h1)  
 08A3 1d (h1),\$FF  
 08A4 inc h1

MAME debugger version  
 Currently targeting sp

```
<cheat desc="Infinite Loops">
<script state="run">
<action>maincpu.pb@E12B=11
</script>
</cheat>

<cheat desc="Select next level">
<parameter min="1" max="15">
<script state="change">
<action>maincpu.pb@E450=(p
</script>
```

```
A5 D0 00 37 54 48 40
40 77 78 79 40 00 D0
52 20 20 20 40 79 D1
2E 2E 7E 40 00 D0 00
2F 40 87 D0 40 7A 7B
20 20 20 20 20 43 4F
49 4F 4E 40 94 D0 00
54 20 44 55 54 59 20
96 D0 00 20 20 20 20
4C 41 54 49 4F 4E 40
56 45 52 59 20 44 55
45 44 40 B2 00 E0 D3
82 03 03 94 EE 22 A6
85 CD D0 85 8D 03 D8
C2 F6 29 63 77 A3 F6
77 8D A2 A6 E0 11 97
54 92 08 0C 23 31 80
```

```
Ha...6THE...
..@ghi...@wxy
.|..TIMER
...@jklmno@
|}~.@@...
NGRATULATION
YOUR FIRST DU
FINISHED...
CONGRATULAT
...YOUR EVERY
TY FINISHED@
...n...
...[...
...cw...1.cw...
```



## GREETINGS

I would like to take this very opportunity and send my greetings to:

- \* MAME Team, for providing best Arcade Emulator ever.
- \* Pugsy, for maintaining XML cheat collection archive and Emulator Cheats forum.
- \* People who contributed to XML cheat collection and Emulator Cheats forum members.
- \* My teammates in Commodore64 cracking group Nostalgia, of which I'm very proud to be a member.
- \* All active sceners of Commodore64, for keeping the scene alive.

Erhan Alparslan

a.k.a. syndromtr of Emulator Cheats forum ([www.mamecheat.co.uk/forums/](http://www.mamecheat.co.uk/forums/))

syndromtr AT gmail.com

First edition, 28.07.2014.

Copyright (C) 2014 Erhan Alparslan.

All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law.

Cover page's background image was taken with permission from "The Most Gorgeous Arcade Games wallpaper", (C) 2012-2014 by SolidAlexei of deviantART.

MAME (C) Nicola Salmoria and the MAME team. MAME is a registered trademark of Nicola Salmoria. All rights reserved.

For some arcade games in Section 2 (In-depth cheat finding), info gathered from history.dat for MAME, its source is <http://www.arcade-history.com> (C) Alexis Bousiges.

All arcade games are used for informational purposes and are copyrighted by their respective copyright holders.

# TABLE OF CONTENTS

<b>INTRODUCTION</b>	v
<b>1. MAME CHEAT FILE BASICS</b>	1
Script states, memory types and sizes	1
Parameter and condition tags	1
Now! cheats and "on" state	2
ROM cheats and reverse engineering	2
Double condition usage	2
Using "frame" variable	4
Bitwise operations	4
Output and argument tags	4
Testing a single cheat in MAME	5
<b>2. IN-DEPTH CHEAT FINDING</b>	6
Football Champ (World) (C) 1991 TAITO	6
Unlimited Super Shoots cheat	6
Time Pilot (C) 1982 KONAMI	9
Select Starting Stage cheats	9
Rapid Fire cheat	12
Miss World '96 Nude (C) 1996 COMAD	14
Invincibility cheat	14
Ultimate Mortal Kombat 3 (C) 1995 MIDWAY	16
MAME cheat xml address < > TMS34010 address exchange	16
Auto fatality cheats	17
DoDonPachi Dai-Fukkatsu (C) 1998 CAVE	20
Infinite Lives cheats	20
Infinite Bombs cheats	21
Unlock Strong Style cheat	21
Infinite Hyper Counter cheats	22
Invincibility cheats	23
Select Area to follow Area 1 cheat	23
Kill Bosses with 1 hit cheat	26
Vendetta (C) 1991 KONAMI	29
Select Starting Stage cheat	29
Weapons do not disappear cheat	30
See End Sequence Now! cheat	31
Mortal Kombat II (C) 1993 MIDWAY	32
Fight VS Smoke easily cheat	32
Johnny Kage always performs Red Shadow Kick cheat	34
Kombat Tomb stage auto fall from spikes cheat	34
Fatal Fury Special (C) 1993 SNK	36
Always have Easy Supers cheat	36
See End Sequence Now! cheats	41

<b>3. OTHER CHEAT EXAMPLES</b>	42
Space Harrier (C) 1985 SEGA	42
Rapid Fire cheat	42
Phoenix (C) 1980 AMSTAR	42
Rapid Fire cheat	42
Art of Fighting (C) 1992 SNK	43
Desperation move cheat	43
Ninja Spirit (C) 1988 IREM	43
Select Starting Stage cheats	43
Finish Current Stage Now! cheats	43
Batman (C) 1991 ATARI	45
Select Starting Level cheat	45
See End Sequence cheat	45
DoDonPachi (C) 1997 ATLUS	45
Select Level to follow Level 1 cheat	45
Kill Boss with 1 hit Now! cheat	46
Lethal Enforces II (C) 1994 KONAMI	47
Finish Current Section Now! cheat	47
See End Sequence Now! cheat	47
Shadow Dancer (C) 1989 SEGA	47
Finish Current Section Now! cheat	47
Finish Bonus Stage Successfully Now! cheat	48
Final Fight (C) 1990 CAPCOM	48
Knife stab ability for Guy and Haggar cheat	48
Never drop weapons cheat	49
The Gladiator (C) 2003 IGS	50
One Hit Kills cheat	50
One button codes for Mode Selection screen cheat	51
<b>4. TIPS AND TRICKS</b>	52
Essential instruction codes	52
Branches, jumps and jump to subroutines	52
Branch forward - backward and how it's calculated	52
Exercising with MAME debugger	52
Disassembling to get instruction codes and operation codes	52
Examining previously executed code, upper level routines	52
Extended wp command	52
Extended bp command	53
Extended bp command to poke	53
Selective use of find command	53
Trace file, remove duplicate lines	53
Trace file, getting unique instructions	53
<b>5. LINKS</b>	54

## INTRODUCTION

This guide has been developed as cheat reference source for those of you who want to learn how MAME (Multiple Arcade Machine Emulator) cheats are found, how they're installed and what MAME cheat file basics are.

This manual contains all the information you need to examine present cheats, adapting them to other games, using powerful built-in MAME debugger features efficiently to monitor code, memory region, registers, data and how to modify code (if needed), in terms of optimization.

In first section, all structures and components of MAME xml cheat file are introduced with all available tags and examples of their usage. For this purpose, existing cheats from current MAME cheat database are chosen. Finally, testing an xml cheat file in MAME emulator is briefly explained; in accordance with current MAME cheat file structure, to assist you in testing your own cheats in MAME emulator.

In second section, from approach to finalized cheats as a result, how several cheats for 8 games were found is briefly explained with all single steps. Conclusions and final words are emphasized to specialize.

Third section consists of other cheat examples to provide assistance to previous section, as cheats in this section don't include all single steps. Only point of view and guidelines are supplied, referring to the cheats in previous section.

To simplify searching, these tags below can be used in sections two and three:

[#easteregg](#) [#reverseengineer](#) [#ci&cn](#) [#adaptexisting](#) [#linearlevel](#) [#non-linearlevel](#)  
[#hook&customcode](#) [#detectautomate](#) [#rapidfire](#) [#movementcodes](#) [#endsequence](#) [#boss](#)

Section four includes all tips and tricks used for creating this manual. Mastering these skills mentioned in this section will result in automation of future cheat investigations.

Last section is a collection of World Wide Web links, in four categories.

What you can do with this guide is to find valuable machine code and assembler language information for different kinds of CPUs in several architectures. This will help the reader to understand how instruction code progress is executed inside CPU and how programming logic works. If you are beginner in programming, 8-bit 6502 instruction reference will probably be a good start to learn basics of machine language or to improve your skills. That reference can be found in web links section. From another point of view, more experienced programmers will find all information needed to master several code flows for various CPU systems.

Also it's strongly recommended that, for your WIP (work in progress) or finalized MAME cheats; taking notes including memory locations, conditions found, short disassembly of original code and patched code with comments in a particular document file in your computer is very essential to save precious time and easy accessing. Keep in mind that origin of this guide you are reading is a text document consisting of MAME cheats found by the author.

## MAME Cheat file basics

```
<cheat desc="Infinite Lives">
  <script state="run">
    <action>maincpu.pb@E101=06</action>
  </script>
</cheat>
```

Script state can be ON – OFF – CHANGE – RUN and it defines how or when the action will be executed. In example above, “run” means it’s always enabled, running each frame.

maincpu states the cheat is for main CPU unit. There may be other CPUs as soundcpu, cart or prot.  
Example rom: [goldnaxe] Golden Axe, after entering MAME debugger, debugger window title is:  
M68000 maincpu Pressing F6 in MAME debugger changes CPU, so  
Z80 soundcpu is the second CPU of this game.

.pb after maincpu states these two options:  
p letter states memory type (program write - RAM) of incoming address after @  
b letter states a BYTE will be poked into the address after @

### Memory types:

p	program write	(RAM = Random Access Memory)
m	region write	(ROM = Read Only Memory)
r	RAM write	(for ROM cheats when “m” does not work or for RAM cheats when “p” does not work)
o	opcode write	(for ROM cheats when “m” or “r” does not work – often for encrypted memory)
d	data write	
i	i/o write	
3	SPACE3 write	

### Memory sizes:

b	byte	0x00	8 bits
w	word	0x0000	16 bits
d	double	0x00000000	32 bits
q	quad	0x0000000000000000	64 bits

In this example, memory address is E101. It’s always in hexadecimal format and the number after = will be poked into this address. Current value of this address (E101) can be checked with MAME debugger by pressing Ctrl + M when MAME debugger is active and entering E101 in that new program space memory window.

After “=” that’s the memory size (byte in this example) to be poked. And value is 06 (in hexadecimal format).

Result is, byte 0x06 is poked into E101 memory location, type of memory write is program, on main CPU, and this poke process is executed in every frame. This cheat is taken from 1942 game, it states that game code stores number of lives in memory location E101. Its value is 0x02 when game starts (without cheat of course) and decreases when a life is lost. When this cheat is enabled, that byte is always 0x06, it’s also displayed at lower left corner, with ship icons, game code will decrease it to 0x05 when a life is lost, but as the cheat is always executed (“run”), that byte will be 0x06 again and will be poked into E101.

After this basic cheat above, here come more complex cheats and descriptions:

```
<cheat desc="Select Starting Level">
  <parameter min="1" max="32" step="1"/>
  <script state="run">
    <action condition="(maincpu.pb@E103==00)">
      maincpu.pb@E103=(param-1)</action>
    </script>
</cheat>
```

From same game, 1942; this is a typical example of “Select Starting Level” cheats. Parameter tag defines value of (param) variable. (param) will be between 1 and 32 (decimal). Script state is “run”, so this cheat is also executed every frame, will always be active after cheat is turned on. There’s a condition between quotes. That condition is also checked every time, for main CPU, RAM area, byte value of memory location E103 is checked. If it’s equal to 0x00, the command after > is executed. Like this, POKE main CPU’s RAM area E103 memory location, (param-1).

(param) series was mentioned above, as 1 to 32. As “param” can be used with mathematical expressions, here, (param-1) series becomes 0 to 31. It means, if player selected to start from Level 8, 8 is visible in cheat menu, param-1 will be 7 and 0x07 will be poked into E103, when its value becomes 0x00. (When a new game starts, mostly, game code resets memory location of level number, usually 0x00 or 0x01; here, cheat uses this feature and installs itself when this memory location is set to default.)

```
<cheat desc="Select Perm. Weapon">
  <parameter>
    <item value="0x02">None</item>
    <item value="0x00">Batarang</item>
    <item value="0x01">Bat Rope</item>
    <item value="0x04">Gas Grenades</item>
  </parameter>
  <script state="run">
    <action>maincpu.pb@103BED=param</action>
  </script>
</cheat>
<cheat desc="Select Temp. Weapon">
  <parameter>
    <item value="0x02">None</item>
    <item value="0x00">Batarang</item>
    <item value="0x01">Bat Rope</item>
    <item value="0x04">Gas Grenades</item>
  </parameter>
  <script state="change">
    <action>maincpu.pb@103BED=param</action>
  </script>
</cheat>
```

Above, there are two cheats taken from [batman]. To use non-linear variables, this special kind of <parameter> tag can be used. 4 parameters are defined for each cheat. For the first cheat, script state is “run”, so one of these hexadecimal values (0x02, 0x00, 0x01 and 0x04) is poked into 103BED memory location in every frame. (Cheat’s name is also “Permanent”.) This memory location is also on main CPU and it’s a RAM cheat.

Second cheat is same but it has “change” state. What does it mean? It’s executed only ONCE when player selects one of the choices under “Select Temp. Weapon” list and choose one of the 4 options and press ENTER. If “Gas Grenades” is chosen, byte 0x04 will be poked into memory location 103BED, only once. That’s why cheat description says “Temporary”.

```
<cheat desc="Drain All Energy Now! PL1">
  <script state="on">
    <action>maincpu.pb@1092CD=01</action>
  </script>
</cheat>
```

This is a typical example of Now! cheats. They’re displayed in MAME Cheat menu as “Set” instead of “On” or “Off”. Script state “on” states that this cheat will be executed only once, when player selects this entry and presses ENTER over it. Byte 0x01 will be poked into memory location 1092CD only once as above example.

```
<cheat desc="Invincibility">
  <comment>You can still be hurt by other players or by falling down
  holes</comment>
  <script state="on">
    <action>temp0 =maincpu.mb@00ADCA</action>
  </script>
  <script state="run">
    <action>maincpu.mb@00ADCA=60</action>
  </script>
  <script state="off">
    <action>maincpu.mb@00ADCA=temp0 </action>
  </script>
</cheat>
```



Here is the first example of a ROM cheat. While dealing with RAM cheats before, only plain values were stored to appropriate RAM memory locations (values responsible for number of lives, timer value, energy value etc). For ROM cheats, mostly, operation code (hexadecimal equivalent of instruction code) itself or part of it, or a code modification (modified branch address), or part of game code to be added are poked into ROM area.

ROM cheat should provide a handy backup-restore function because most games have RAM/ROM checks just after game boots. There, RAM/ROM regions are checked with hash control routines (as CRC or modern day MD5 algorithm) and result is compared with checksum(s) to avoid code or data modification. If checksums do not match, RAM/ROM or write error message is displayed on screen and game boot operation halts.

So, above cheat is examined in three phases:

- 1) Script state "on" runs only once, just after this cheat is turned ON, on switching phase. It gets the byte value of memory 00ADCA in ROM region and stores the byte into temp0 variable (backup operation).
- 2) Script state "run" will be executed every frame, and will poke 0x60 to memory location 00ADCA in ROM region.
- 3) Script state "off" will run only once, but opposite of first step, when this cheat is ON and it's turned OFF afterwards, another switching phase, it stores value of temp0 variable into 00ADCA (restore operation).

This is an ideal use of ROM cheat, in 3 steps; backup, every frame poking progress and restore. But what's the byte 0x60 refers to? It's not a plain variable. To understand its function, disassembly of game code around 00ADCA is needed. To get the disassembled code via MAME debugger:

- While MAME debugger is active with **Debug: goldnaxe – M68000 ":maincpu"** title,
- Pressing Ctrl + D will open new disassembly window. On that window, typing 00ADCA (onto curpc) and pressing ENTER will display game code starting from 00ADBC. Here is debugger command to get disassembled code into text file:

- **dasm d.txt,adba,20** (disassembly from 00ADBA to 00ADBA+0x20 is dumped into file d.txt)
- Here is disassembly: (00ADBA is selected as start address to check previous code and after ; Instruction code comments are added)

```
00ADBA: 206E 0028      movea.l ($28,A6), A0      ;read (A6+$28) and store to A0 in 32 bits
00ADBE: 1028 FFFB      move.b (-$5,A0), D0      ;read (A0-$05) and store to D0 in byte
00ADC2: 6B00 00E8      bmi $aeac               ;if result is negative, branch AEAC
00ADC6: B02E 0023      cmp.b ($23,A6), D0      ;compare byte in (A6+$28) with D0
00ADCA: 6600 00E0      bne $aeac               ;if they're not equal, branch AEAC
00ADCE: 6100 FE80      bsr $ac50               ;branch to subroutine at AC50, and return
00ADD2: 7000          moveq #$0, D0           ;here and go on...
00ADD4: 7600          moveq #$0, D3
00ADD6: 4BF9 FFFF C000 lea $ffffc000.1, A5
```

What does poking 0x60 to 00ADCA mean? Here is the result:

```
00ADCA: 6600 00E0      bne $aeac               ;if value in (A6+$23) != D0, branch AEAC
00ADCA: 6000 00E0      bra $aeac               ;always branch to AEAC
```

ROM cheat modified game code, BNE instruction became BRA (Branch Always). So, when cheat is on, branch happens at 00ADCA and game code starting from 00ADCE and forward is omitted. From coder's view, it makes sense that game code after 00ADD2 has energy lose routine. With that BRA operation code, that routine is bypassed and invincibility is provided. Also, this is the first example of reverse engineering an existing cheat.

```
<cheat desc="Quick Charge Shield PL1">
  <script state="run">
    <action condition="(maincpu.pw@FFAE74 GT 0FC0) AND
      (maincpu.pw@FFAE74 LT 1000)">maincpu.pw@FFAE74=0000</action>
  </script>
</cheat>
```

Here is an example of double condition checks before installing a cheat, from [gigawing]. If word (2 bytes) value of FFAE74 memory location in RAM area is greater than 0x0FC0 AND lower than 0x1000 [0FC0 < (FFAE74) < 1000] 0x0000 will be poked into FFAE74. More conditions can be used as GE (greater or equal), LE (lower or equal), != (not equal). This command in MAME debugger will provide further info about them: **help expression**

Also, BASIC equivalent of the cheat above is:

IF (PEEK(FFAE74)>\$0FC0) AND (PEEK(FFAE74)<\$1000) THEN POKE FFAE74,\$0000

```

<cheat desc="Infinite Time">
  <script state="on">
    <action>temp0 =maincpu.pb@E004</action>
  </script>
  <script state="run">
    <action condition="(frame % 60 == 0) AND
      (maincpu.pb@E004!=temp0 )">maincpu.pb@E004=05</action>
  </script>
</cheat>

```

From [kungfum], this is the example of frame variable. Game runs at 55 Hz. Firstly, byte value of E004 is backed up to temp0 variable. "frame" variable starts from 0 and is increased one by one for each frame is displayed. So, first condition is frame MOD 60 what means, for frames 0, 60, 120, 180 ... this value will be 00 (Each 60 frame ~ 1 seconds). Other condition check is, getting new value of E004 and comparing it with temp0 (backup value). Result is, if each 60 frames passed (delay occurred) AND current value of E004 is not equal to its backup value (it proves E004 has changed), 0x05 is stored to E004. This poke progress is repeated after 60 new frames.

```

<cheat desc="Drop A Bomb Now!">
  <script state="on">
    <action>maincpu.pb@FF0EC0=20|(maincpu.pb@FF0EC0 BAND ~20)</action>
  </script>
</cheat>

```

Cheat is from [3wonders]. When this cheat is executed ("on" means one hit cheat), 0x20 (BITWISE OR) (value of FF0EC0 BINARY AND 0xDF) is poked back to FF0EC0. Description below:

- | sign is for BITWISE OR (for more, in MAME debugger, [help expression](#) command).
- BAND is for bitwise AND. As all numbers are in hexadecimal format here, performed operation is in BINARY origin.
- ~ means binary opposite, NOT, ~20 is 0xEF in hexadecimal or 11011111 in decimal.
- Also, BASIC equivalent of the cheat above is:  
POKE FF0EC0, (\$20 OR (PEEK(FF0EC0) AND %11011111))
- Result is, with bitwise operations, bit 5 of FF0EC0 will be always 1, other all bits will keep their original values.

These bitwise cheats are mainly used for rapid fire; changing some movements those are stored in bits, changing game button actions etc.

```

<cheat desc="Watch ALL 14 tiles - Player">
  <script state="run">
    <output format="%02X %02X %02X %02X %02X %02X %02X %02X %02X %02X %02X %02X %02X %02X %02X">
      <argument count="14">maincpu.pb@(F406 + argindex)</argument>
    </output>
  </script>
</cheat>

```

Above cheat is from [apparel]. <output> and <argument> tags are used for displaying values from memory locations with a defined output format. Mostly used in quiz and mahjong games to display solutions or tiles from these games. %02X means all output format will be in 2 digits hexadecimal, argument count states end value is 14 (decimal) and when this cheat is turned ON, memory values from F406 to F406+0E will be displayed as hexadecimal numbers in upper left corner, each frame, when values change, their display on screen will also be updated.

### Testing a single cheat in MAME emulator:

To test a single cheat (or single xml cheat file) inside MAME emulator, these steps need to be performed:

- Finalize xml file with <mamecheat> and </mamecheat> tags.
- Save xml as <gamenam>.xml and then compress xml file to a 7z file with file compression software which supports 7zip format.
- Rename compressed 7z file to cheat.7z and copy to MAME root folder where mame.exe is (Backup or rename existing cheat.7z file in MAME root folder, it may be full cheat database).
- For MAME command-line binary: MAME needs to be executed this way: mame.exe -cheat <gamenam>
- For MAME derivatives which support GUI: Options > Default Game Options > Miscellaneous > Check "Enable game cheats" and then click apply and then OK.
- When <gamenam> is started, "Cheat" will be active in main menu, which can be displayed by Tab key.

Here is a test cheat file, finalized with "mamecheat" tags, saved as 1942.xml and compressed with 7zip, 7z file renamed to cheat.7z and copied to the folder of mame.exe, and game started with: mame.exe -cheat 1942

```
<mamecheat version="1">
  <cheat desc="Infinite Lives">
    <script state="run">
      <action>maincpu.pb@E101=06</action>
    </script>
  </cheat>
</mamecheat>
```

## [footchmp] Football Champ (World) (C) 1991 TAITO

Goal: As explained in this video (<http://www.youtube.com/watch?v=wsKTK2LxXh0>), this game has a hidden feature that, when some conditions are met, player can perform a “Super Shoot”.

Known facts: Current MAME cheats have Timer and Score entries, so they can be used as a start.



Here is the Super Shoot in progress... After certain criteria are all met, screen gets black and captain player performs a “Super Shoot” which will always result with a goal. This feature can definitely be called an easter egg because few players & people were aware of it. In arcade systems I've never noticed this happen, I accidentally run across this Super Shoot only once while playing with MAME emulator, 10 years ago, and it was really memorable and shocking. I promised myself to investigate & find a proper cheat for this after I noticed MAME debugger and got familiar with it, so my first cheat attempt was this game, directly dealing with game code instead of built-in cheat searching.

#easteregg #adaptexisting

OK. As a start, I check current cheats like this below (Infinite time cheat):

```
<action>maincpu.pb@102929=3C</action>
<action>maincpu.pb@10A335=3C</action>
<action>maincpu.pb@10A334=63</action>
```

After checking values from memory locations 10A334-10A335-102929 with MAME debugger, I notice that, word value in 10A334 always have the current timer value in hexadecimal format, like examples below:

Timer is 1:56                      10A334 = 0138                      -                      Timer is 1:49                      10A334 = 0131

Another helpful entry is PL1 score. Its value is at 10A332. Also, PL2 score is at 10A333. To find the responsible routine for Super Shoot, I need to perform same steps what's described in the youtube link above. First I start a 1UP game, after game starts, I start checking 10A332 with: **wp 10a332,1,r** command in MAME debugger. While some playing, that watchpoint is not triggered so far. When I perform these, let's see what happens:

- My captain player (number 8 – I chose him after game had started) has the ball, he has the ball in certain field as described, and i give high pass to my teammate with Button B. Last step: I again give a high pass with Button B, to my captain this time...
- MAME debugger stops with this message: **“Stopped at watchpoint 1 reading byte from 10A332 (PC=1F166)”**
- So, game code around 1F166 is probably responsible for Super Shoot. Let's check further with disassembly:

01F150:	4A6D	2312	tst.w	(\$2312,A5)	;A5 = 1080000 and A4 = 10460E
01F154:	6A7A		bpl	\$1f1d0	;test word at 10A312
01F156:	4A2D	2342	tst.b	(\$2342,A5)	;if result is plus, go end
01F15A:	6674		bne	\$1f1d0	;test byte at 10A342
01F15C:	4A2C	0028	tst.b	(\$28,A4)	;if not equal to zero, go end
01F160:	676E		beq	\$1f1d0	;test byte at 104636 (A4+\$28)
01F162:	102D	2332	move.b	(\$2332,A5), D0	;if it's zero, go end
01F166:	6768		beq	\$1f1d0	;move byte 10A332 (PL1 score) to D0
01F168:	B02D	2333	cmp.b	(\$2333,A5), D0	;if it's zero (PL1 current score) go end
01F16C:	6662		bne	\$1f1d0	;Compare it with PL2 score?
01F16E:	302D	2334	move.w	(\$2334,A5), D0	;if PL1 score != PL2 score go end
01F172:	0C40	001E	cmpi.w	#\$1e, D0	;move word 10A334 (timer) to D0 register
01F176:	6E58		bgt	\$1f1d0	;compare it with \$1e = 30
01F178:	0C40	000A	cmpi.w	#\$a, D0	;if greater than 30 seconds, go end
01F17C:	6D52		blt	\$1f1d0	;compare it with \$0a = 10
01F17E:	3B7C	0006 B8E6	move.w	#\$6, (-\$471a,A5)	;if lower than 10 seconds, go end
					;probably Super Shoot code starts here

With first disassembly, here is what we've learned:

- 10A312, 10A342 and 104636 are unknown. Their values are checked, but we don't know why.
- PL1's score must be at least 1 to perform a Super Shoot.
- Also score must be a tie (1-1, 2-2, 3-3 ...).
- Timer must be between 10 and 30 seconds (remaining time).



To clarify those 3 unknown memory locations, I apply those: **wp 10a312,1,w** , **wp 10a342,1,w** , **wp 104636,1,w**

After watchpoint analysis for those 3 locations, I get:

- 10A312 is FFFF when ball is uncontrolled, 01xx when ball is on other team's control. When its value is 0X (05 for example), our player but player number 6 has the control of the ball (+1). This also proves that ball control should be uncontrolled, high pass from our teammate should start, to our chosen captain player.
- 10A342 is 00.
- 104636 is 01. After further investigate with this in MAME debugger, **bp 1f150** (start of Super Shoot checks) when I try to perform a Super Shoot with another player (e.g. number 11) and debugger stops at 1f150 but A4 register is different = 104A8E, so 104A8E+\$28 is checked as byte, it's 104AB6 and it's value is 00. That's why Super Shoot routine is terminated. It means, 01F15C: **tst.b (\$28,A4)** is checking captain player. It means its value is 0x01 only when our chosen captain player is about to take the high pass given with Button B.

To finalize what 10A342 is for, I use a breakpoint trick to poke a specific value into ROM and execute Super Shoot routine:

- First, **wp 10a342,1,w** for detecting write to that memory.
- Special breakpoint command, **bp 1f150,1,{maincpu.mw@1f154=6028;g}**
- This bp command breaks at 1F150 but pokes 6028 value into 1F154, and goes on execution. So 1F154 becomes **bra \$1F17E**. It's the start of Super Shoot routine (60XX is the instruction code for BRA).
- What happens after above? I go on playing game and apply criteria of Super Shoots, after a while,

10A342 becomes 0xFF (PC=184C8) and Super Shoot begins.

It seems 10A342 value is set 0xFF so this Super Shoot is probably happens only once in each game. If you investigate further for 10A342, you'll notice that 10A342 won't be reset to 0x00 until a new coin is inserted after game over happens. Result is, by default, Super Shoot can be performed only once for each coin.

#### Construction of MAME cheat:

- 10A342 check should be removed/bypassed, so Super Shoot will be available more than once in each game.
- **tst.b (\$28,A4)** should be preserved. Because, only captain players will be able to perform Super Shoot.
- PL1 – PL2 score checks should be removed / bypassed.
- Timer values should be adjusted. Super Shoot will be available every time, but as Super Shoot sequence takes around 5 seconds, it's better to deny Super Shoot if less than 5 seconds are remaining, to avoid graphics init bugs.

Original code:

01F156: 4A2D 2342	tst.b (\$2342,A5)	;test byte at 10A342
01F15A: 6674	bne \$f1d0	;if not equal to zero, go end
01F15C: 4A2C 0028	tst.b (\$28,A4)	;test byte at 104636 (A4+\$28)
01F160: 676E	beq \$f1d0	;if it's zero, go end
01F162: 102D 2332	move.b (\$2332,A5), D0	;move byte 10A332 (PL1 score) to D0
01F166: 6768	beq \$f1d0	;if it's zero (PL1 current score) go end
01F168: B02D 2333	cmp.b (\$2333,A5), D0	;Compare it with PL2 score?
01F16C: 6662	bne \$f1d0	;if PL1 score != PL2 score go end
01F16E: 302D 2334	move.w (\$2334,A5), D0	;move word 10A334 (timer) to D0 register
01F172: 0C40 001E	cmpi.w #\$1e, D0	;compare it with \$1e = 30
01F176: 6E58	bgt \$f1d0	;if greater than 30 seconds, go end
01F178: 0C40 000A	cmpi.w #\$a, D0	;compare it with \$0a = 10
01F17C: 6D52	blt \$f1d0	;if lower than 10 seconds, go end

Here is game code after cheats are applied: (Changes are in red color)

01F156: 4A79 2342 6674	tst.w \$23426674.1	;4A79 is null test.w with 6 bytes,
01F15C: 4A2C 0028	tst.b (\$28,A4)	;( tst.b (\$2342,A5) bne \$f1d0) is now
01F160: 676E	beq \$f1d0	;disabled with a smart instruction code
01F162: 102D 2332	move.b (\$2332,A5), D0	
01F166: 6006	bra \$1f16e	;we bypass PL1 and PL2 score checks and
01F168: B02D 2333	cmp.b (\$2333,A5), D0	;branch further to 1F16E (timer check)
01F16C: 6662	bne \$f1d0	
01F16E: 302D 2334	move.w (\$2334,A5), D0	;below, 0C78 null cmpi.w with 6 bytes:
01F172: 0C78 001E 6E58	cmpi.w #\$1e, \$6e58.w	;(cmpi.w #\$1e, D0 bgt \$f1d0) disabled
01F178: 0C40 0005	cmpi.w #\$5, D0	;comparing with 0x05 instead of 0x0A
01F17C: 6D52	blt \$f1d0	;if less than 5 seconds, no super shoot

MAME cheat in xml format:

```
<cheat desc="Unlimited Super Shoots (All players)">
  <comment>Each captain of any 4 players can perform Super Shoot without timer-
  score restrictions and infinitely in a game.</comment>
  <script state="on">
    <action>temp0 =maincpu.mb@1F157</action>
    <action>temp1 =maincpu.mw@1F166</action>
    <action>temp2 =maincpu.mb@1F173</action>
    <action>temp3 =maincpu.mb@1F17B</action>
  </script>
  <script state="run">
    <action>maincpu.mb@1F157=79</action>
    <action>maincpu.mw@1F166=6006</action>
    <action>maincpu.mb@1F173=78</action>
    <action>maincpu.mb@1F17B=05</action>
  </script>
  <script state="off">
    <action>maincpu.mb@1F157=temp0 </action>
    <action>maincpu.mw@1F166=temp1 </action>
    <action>maincpu.mb@1F173=temp2 </action>
    <action>maincpu.mb@1F17B=temp3 </action>
  </script>
</cheat>
```

Final words: Now it's time to test the cheat (very detailed) and make sure that cheat does not destroy other parameters as captain player's location on field, to perform Super Shoot. (Confirmed, OK) Next step is, applying same cheat to clones of this game. ([euroch92] and [hthero]) You'll notice that [euroch92] code is slightly different, by default; it allows Super Shoots when remaining time is between 10 and 36 seconds, and if score is 0-0, Super Shoot is possible.

As code was examined in every way and all requirements of Super Shoot were totally clear then, I contacted authors of <http://www.arcade-history.com> (Team supplying history.dat file for MAME) and contributed this info about requirements of Super Shoot in Football Champ game and for its clones. This info is already included.

## [timeplt] Time Pilot (C) 1982 KONAMI



"Time Pilot was released in November 1982.

Yoshiki was told to design a driving game. When he learned of the game's concept, he balked at making it and started on Time Pilot. As development continued, Okamoto showed his boss design docs for the driving game, all the while working on Time Pilot. Although his boss told him to do the driving game instead, he tried to take the credit for Time Pilot. Okamoto decided not to disgrace his boss and let the episode go!! The background moves in the opposite direction to the player's plane, rather than the other way around; the player's plane always remains in the center.

Stage 1 (A.D. 1910) and Stage 5 (A.D. 2001) are never played in the attract mode."

#ci&cn #linearlevel #rapidfire

Goal: To find "Start from Stage" and "Rapid Fire" cheats.

Known facts: According to history.dat:

"Time Pilot consists of 5 different stages of play which are as follows

...

The next, Stage 6, is identical with Stage 1, but the number of planes attacking you, their speed and number of shots and grenades are gradually increased."

As we start, first thing to investigate is, trying to get memory location where current stage number is stored. We can assume that, stage number increases linearly, so for our first attempt, we'll try this method.

MAME debugger has a powerful cheat search engine, so here is some info about it:

- **cheatinit** or (**ci**) command initializes a cheat search, if no parameters are used, it will start monitoring whole changeable memory of main CPU. Default width will be unsigned bytes.
- **cheatnext** or (**cn**) command will make comparisons with the last matches. For example, after **ci** command, **cn +,1** command picks memory locations where its value is increased by one and start to monitor them further. It means all memory locations that were not increased by one are eliminated; they're out of cheat searching.

So, what to do is:

- Starting a 1UP game, and when we're on stage 1, entering MAME debugger and initializing a cheat search by **ci** command. It says 2560 cheats initialized.
- We have to progress further in game, to reach incoming stages, so existing cheats can be used like "Invincibility" and "Put Stage Guardian On Now!".
- When we reach stage 2, entering MAME debugger again and checking increased value with **cn +,1** command. It says around 25 cheats found. So, as you see, results are narrowed. After some playing, as you're still at stage 2, you can also try **cn eq** command as we're in same stage, eq means equal (to narrow down results).
- More game play, and as we reach stage 3, **cn +,1** command again, and it says 11 cheats found. We have two options here, as results are very low:
  - 1) **cheatlist** command to see all those 11 memory locations in MAME debugger console. (It's recommended to note them down for further analysis – only one of them has the current stage number value and proper for final cheat.) Also **cheatlist <filename>** command can be used to save the results in basic xml format to <filename>.
  - 2) Go on playing, reaching next stage and repeating **cn +,1** command again. Result is, still 9 cheats, it's time to list them and investigate (Some entries start with value 5C so they're omitted, as we expect stage number starts from 0 or 1):

AD01 (01 to 04)  
AD04 (00 to 03)  
AD0C (01 to 04)  
AD11 (01 to 04)

AD14 (00 to 03)  
AD1C (01 to 04)  
AFF0 (01 to 04)

Now, it's time to change these values from MAME debugger memory windows that we open with Ctrl + M. Here is the result of changing those bytes (Before, it's better to take a snapshot with Shift + F7 when current stage is 4 – And apply each step below, after loading snapshot again):

AD01	was 04,	poking 01	:	Nothing special happens.
AD04	was 03,	poking 00	:	Enemy planes transform into Stage 1 planes!
AD0C	was 04,	poking 01	:	Background changes, same background color as Stage 1?
AD11	was 04,	poking 01	:	Nothing special happens.
AD14	was 03,	poking 00	:	Nothing special happens.
AD1C	was 04,	poking 01	:	Nothing special happens.
AFF0	was 04,	poking 01	:	Nothing special happens.

According to tests, AD04 seems only data for stage info for now. It needs to be investigated further, when poke operation happens, so, here are the steps:

- Reset game with F3 key, after reset, in MAME debugger, **wp ad04,1,w** to monitor write operations.
- Insert coin(s) and start a new game, it will stop at that watchpoint.
- Same time, Ctrl + M for monitor window and display AD04 memory location and change its value with 04.
- Close debugger and return to game. We tested same thing again, but here is another result, "Stage 1" appeared as stage number, but enemies from stage 4 started to appear.

Probably, there are two memory locations responsible for current stage. First one might be real number, i.e. if current stage is 18, number 18 to display it on screen, and other location is AD04, responsible for one of 5 different stages, it seems its value can be 00 to 04 to determine stage characteristics.

So, more AD04 analysis, from code now:

When **wp ad04,1,w** is active, resetting and starting a new game again, debugger stops at PC=4C8C. Afterwards, a new disassembly window with Ctrl + D, changing "curpc" entry with "curpc-10", to start from backwards:

4C78:	3A 32 AD	ld	a,(\$AD32)	;get value of \$AD32 and store in A register
4C7B:	A7	and	a	;and operation
4C7C:	21 10 AD	ld	hl,\$AD10	;store \$AD10 value into HL register
4C7F:	28 03	jr	z,\$4C84	;if zero flag is 1, branch 4C84
4C81:	21 20 AD	ld	hl,\$AD20	;store \$AD20 value into HL register
4C84:	11 00 AD	ld	de,\$AD00	;store \$AD00 value into DE register
4C87:	01 10 00	ld	bc,\$0010	;store 0x10 into BC register
4C8A:	ED B0	ldir		;load, increment, repeat = ;copy 0x10 bytes starting from HL into DE=\$AD00

To understand how this routine works, here is last step. Remove that watchpoint and this time create a breakpoint as **bp 4c78**, very start of this routine. When it stops there, step into with F11 key; execute instruction codes one by one. Result is, \$AD32 is zero, so HL=\$AD10, DE=\$AD00, BC=\$10, ldir command copies bytes in region \$AD10 to \$AD20 into \$AD00. So, original value comes from \$AD14, for that \$AD04. Afterwards, in game, \$AD04 is in use. Also, this is worth a try: As we know, this game supports 2 player mode, so, probably, \$AD32 is second player byte, and according to above routine, \$AD20 to \$AD30 area might be default 2UP data? Let's check it. As **bp 4c78** is still active, after resetting game and starting a 2UP game this time, result is:

Stopped at 4C78 first time, \$AD32 is 00 and it's Player 1's turn. After losing a life, Stopped at 4C78 again, \$AD32 is 01. Applying F11 as single step, HL became \$AD20 and (\$AD20-\$AD30) block copied to AD00. It's Player 2's turn now.

So, for now: Stage character value first appearance is at \$AD14 for PL1, at \$AD24 for PL2. (It's copied to \$AD04 afterwards.)

Now, last step to find "displayed" stage number, it should be one of the bytes in \$AD00-\$AD10 region. You remember previous cheat search?

AD01	was 04,	poking 01	:	Nothing special happens.
------	---------	-----------	---	--------------------------

Time to test this value, this time, at the end of a level, so, start a new 1UP game and again use "Invincibility" and "Put Stage Guardian On Now!" cheats, when stage 1 boss is killed, manually enter MAME debugger and poke 0x05 into AD01. Yes, "Stage 6" appeared for next stage, as \$AD04 is same, enemies are from stage 2 though. So, here is combined last test for 1UP game:

Start a new game and enter **wp ad11,1,w** and **wp ad14,1,w** to monitor writes to these locations. After two watchpoint triggering, manually poke 0x04 into AD11 and 0x03 into AD14. (Stage 4 and characteristic values.)



It successfully started from Stage 4. What is the last stage number? Is it unlimited or ends at 100? Let's see.

Same watchpoints above, this time let's poke 0x63 into AD11 and 0x04 into AD14. (99 in decimal for stage number, and 0x04 for characteristics, as 0x04 is its maximum value.) Stage 99 started! Time to enable invincibility cheat and try to proceed to next stage. Also we notice that stage number is displayed with different blocks just below high score line. "Stage 101" is not displayed below, and that number is not reset to one, also difficulty goes on increasing as new stage didn't start in default Stage 1 difficulty. So, we'll decide how to install the cheat. It seems in cheat xml, with parameters, limiting maximum starting stage to 100 will be fine.

This cheat can be installed now (all required data collected), but here is an important info for these kinds of cheats:

- For "Start from Level/Stage" cheats, best way is to use a RAM cheat, as below example:

```
<action condition="(extra conditions) AND (maincpu.pb@[level]==00)">maincpu.pb@[level]=(param-1)</action>
```

Extra conditions: They will be explained in further cheat examples, as there should be one or more extra conditions if game has a demo mode (mostly some time after insert coin screen – AI controlled demo mode is displayed for most games). With these extra checks, we can use those extra conditions to poke the level number only when a real game (after coin inserted) starts, so the cheat won't affect demo game or so.

As we know that this game also has a demo mode (demo games from stages 2, 3 and 4). So, applying the cheat without (extra conditions) will probably cause demo game misbehaviors. To finalize cheat, we need to find a parameter (memory location) for demo game. Examining \$ADXX might help.

Here is method to investigate, **wp ad11,1,w** to detect stage number writes, for demo game and for normal 1UP game, same routine is executed below:

```
27EA: 32 11 AD      ld    ($AD11),a      ;A=1 here, store to $AD11
27ED: 32 21 AD      ld    ($AD21),a      ;store to $AD21
27F0: 32 1E AD      ld    ($AD1E),a      ;store to $AD1E
27F3: 32 2E AD      ld    ($AD2E),a      ;store to $AD2E
27F6: 3A 30 AD      ld    a,($AD30)      ;get value of $AD30 and store to A register
27F9: A7           and    a              ;and operation
27FA: 28 39        jr     z,$2835      ;branch or not (due to z flag)
```

This \$AD30 might be special, let's check with step into F11 key:

- When demo game starts executing this code, \$AD30 is read and it's 0x00.
- When normal 1UP or 2UP game starts executing this code, \$AD30 is read and it's 0xFF.

So, we can finalize cheat, we have (extra condition) that we'll allow the cheat only if \$AD30 is 0xFF.

```
<cheat desc="Select Starting Stage PL1">
  <parameter min="1" max="100" step="1"/>
  <script state="run">
    <action condition="(maincpu.pb@AD30 == FF) AND
      (maincpu.pb@AD11 == 01)">maincpu.pb@AD11=param</action>
    <action condition="(maincpu.pb@AD30 == FF) AND
      (maincpu.pb@AD11 == param)">maincpu.pb@AD14=(param-1)%5</action>
  </script>
</cheat>
```

```
<cheat desc="Select Starting Stage PL2">
  <parameter min="1" max="100" step="1"/>
  <script state="run">
    <action condition="(maincpu.pb@AD30 == FF) AND
      (maincpu.pb@AD21 == 01)">maincpu.pb@AD21=param</action>
    <action condition="(maincpu.pb@AD30 == FF) AND
      (maincpu.pb@AD21 == param)">maincpu.pb@AD24=(param-1)%5</action>
  </script>
</cheat>
```

Condition checks if \$AD30 is 0xFF and \$AD11 is 0x00 for a 1UP game. So, when default stage value (01) is poked into \$AD11, cheat also checks \$AD30, if conditions are met, first, pokes selected stage number into \$AD11. Second line of the cheat checks same conditions, plus, checks if \$AD11 is equal to (param), and then pokes [(param-1) in MOD 5] into \$AD24. (% means MOD arithmetic operation, in xml file.) So, both required memory locations responsible of stage are poked with correct values from selected parameters, 1 to 100. As we test this briefly in all modes (demo game – 1UP – 2UP) in different combinations, cheat works as it's constructed, without any problems.

#### Rapid Fire cheat:

In most shooting games, game code checks fire button with some special routines that pressing & holding fire button is detected and when it happens, shooting is disabled. Or, some games count number of bullets on screen and there's maximum limit for it. Or, first fired bullet should reach some coordinate that new bullet is allowed, this is another type of these checks. For button value check, XOR (exclusive or) commands are mostly used. So, first approach to find these routines is, finding memory location of fire button and check for read/writes and finally finding that XOR or hold button detection routine.

To investigate this routine for this game,

- Start a new game and when you're not firing, enter MAME debugger and start a trace: **trace <filename>** (Make sure that on MAME debugger, maincpu is displayed, so trace will start for maincpu, all instruction codes executed will be written to specified file.)
- Quit MAME debugger and immediately press Button A (fire) and then immediately enter MAME debugger again and stop the trace with: **trace off**
- Check end of this document, Tips and Tricks section, "[Trace file, getting unique instructions](#)" and apply it to main trace file.
- Examine unique instructions (Instruction codes those were executed only once for that trace session – saved to new file for example), after you see RET (return from subroutine – end of subroutine) or there is large space between two addresses, add breakpoints for those routine start addresses in MAME debugger like these: **bp 0f1a , bp 2407 , bp 31e8 , bp 33b8 ...**
- Check when those breakpoints are triggered, and remove unwanted ones by **bpclear <number>** command.
- As you notice, for above example, last 2 breakpoints are triggered periodically, even we're not pressing Button A. So it's time to clear them. Also first breakpoint is triggered without firing, time to clear it also.
- Now, only **bp 2407** is active, if we check it, as soon as we press Button A, it's triggered. Now it's time to investigate memory around 2407:

```

23F4: 07          rlca          ;rotate left (with carry) A register 1 bit
23F5: 07          rlca
23F6: 07          rlca
23F7: 07          rlca          ;4 bits rotated left
23F8: 21 8E A9   ld  hl,$A98E   ;store $A98E into HL register
23FB: CB 16     rl  (hl)    ;rotate left $A98E value
23FD: 7E         ld  a,(hl)   ;get $A98E value and store into A register
23FE: E6 03     and $03     ;and with 0x03
2400: FE 01     cp  $01     ;is it 0x01?
2402: 21 81 AA   ld  hl,$AA81 ;store $AA81 into HL register
2405: 20 02     jr  nz,$2409 ;if ($A98E) and 0x03 != 0x01 then branch 2409
2407: 36 03     ld  (hl),$03   ;store 0x03 into $AA81
2409: 3A 30 AD   ld  a,($AD30)  ;$AD30 is read, probably checking it's a
240C: A7         and  a        ;demo game or not

```

So when we remove that breakpoint and start examining from \$23F4 by **bp 23f4** command, it will stop there every time, what about **bp 2407**? It seems it's only executed if branch at \$2405 do not happens. Yes, it makes sense; it stops at \$2407 only fire button is pressed, for the first time, if fire button is hold down, no stop at \$2407. So, we need to find a way to always execute \$2407. Above routine checks 3 rightmost bits of \$A98E with and \$03 operation. If result is \$01, it means, if value of \$A98E is xxxxxx01 in binary format, fire is allowed. As we monitor \$A98E value periodically, here is example:

1F – 3F – 7E – FC – F8 – F0 – E0 – C0 – 80 – 00 and then always 00.

This above is due to rotate left operations, Carry flag is also important for this, for above example, only first two entries, 1F and 3F, if AND 0x03 is applied, result is 0x01 and fire happens. Afterwards, no fire. Example of a smart routine to prevent rapid fire, apart from XOR routine.

What if we compare with 0x00 after AND 0x03? We shouldn't break current case, normal fire should be allowed, additionally, fire button hold down should be taken as firing. Testing with **bp 23f8,1,{maincpu.mb@2401=00;g}**

As we suspected, result is AUTO firing instead of rapid fire, fire button is not pressed but auto firing in progress. So, last step might be needed for that branch: Using JR Z instead of JR NZ, if Zero flag is 1, branch to \$2409. So, resetting game and starting a new game, **bp 23f8,1,{maincpu.mb@2401=00;maincpu.mb@2405=28;g}**

This time code compares it with 0x00 and uses JR Z (0x28 is its operation code). Result is perfect, rapid fire is always active, for both players, when fire button is hold down, rapid fire happens, after it's released and button is in idle, no auto firing observed. Here is original code VS cheat installed code:

```

23FD: 7E          ld  a,(hl)          ;get $A98E value and store into A register
23FE: E6 03      and $03            ;and with 0x03
2400: FE 01      cp  $01            ;is it 0x01?
2402: 21 81 AA    ld  hl,$AA81        ;store $AA81 into HL register
2405: 20 02      jr  nz,$2409    ;if ($A98E) and 0x03 != 0x01 then branch 2409

2400: FE 00      cp  $00            ;is it 0x00?
2405: 28 02      jr  z,$2409    ;if ($A98E) and 0x03 is 0x00 then branch 2409

```

```

<cheat desc="Rapid Fire">
  <script state="on">
    <action>temp0 =maincpu.mb@2401</action>
    <action>temp1 =maincpu.mb@2405</action>
  </script>
  <script state="run">
    <action>maincpu.mb@2401=00</action>
    <action>maincpu.mb@2405=28</action>
  </script>
  <script state="off">
    <action>maincpu.mb@2401=temp0 </action>
    <action>maincpu.mb@2405=temp1 </action>
  </script>
</cheat>

```

Final words: Start from stage cheat works through levels 1 to 100, demo game is not affected. Rapid fire is always active when Button A is pressed and hold down. Applying same procedure to [timeplta], [timepltc] and [spaceplt]; clones and bootleg of this game is also required to finalize.

## [missw96] Miss World '96 Nude (C) 1996 COMAD



"Rip-off of 'Gals Panic!', featuring 'adult' photographs instead of the cartoon imagery.

...

Like many of Comad's games, the music is comprised of short MSM5205 samples and loops, which are bootlegs ripped from various well-known songs without the approval from the original music authors.

Levels contain samples and loops from these music tracks :

'Oxygene Part IV' by Jean-Michel Jarre, 'Tonight Is The Night' by La Bouche, 'Let's All Chant' by Michael Zager Band (a modern 90's remake of this song)"

#reverseengineer #adaptexisting

Goal: To find "Invincibility" cheat for this game.

Known facts: [galsnew] has invincibility cheat found and installed, as game engines are similar, code might be similar, too. Worth a try.

Here is [galsnew] Gals Panic! Invincibility cheat (only RUN section):

```
<action>maincpu.mb@00311E=60</action>
<action>maincpu.mb@005BAE=60</action>
<action>maincpu.mb@00601D=02</action>
```

Now it's time to start [galsnew], enter MAME debugger and partially disassemble those 3 memory locations:

003118: 082E 0000 0005	btst	#\$0, (\$5,A6)	
00311E: 6700 00D8	beq	\$31f8	;this becomes 60 = BRA
003122: 2248	movea.l	A0, A1	
005BA6: 0839 0000 00C8 211D	btst	#\$0, \$c8211d.1	
005BAE: 6700 0006	beq	\$5bb6	;this becomes 60 = BRA
005BB2: 6100 0014	bsr	\$5bc8	
006008: 0839 0002 00E0 0004	btst	#\$2, \$e00004.1	
006010: 6600 00C6	bne	\$60d8	
006014: B439 00C8 266F	cmp.b	\$c8266f.1, D2	
00601A: 6700 00B0	beq	\$60cc	;branch counter is changed to 02 so
00601E: B439 00C8 2670	cmp.b	\$c82670.1, D2	;it branches here instead of 60CC
006024: 6700 00AC	beq	\$60d2	

What to do is easy, after those notes above, we need to start [missw96] and perform some memory searches. If we're lucky, we can find same or equivalent routines to apply the cheat.

For first part of the cheat, let's search significant bytes as above: In MAME debugger,

- find 0,10000,w.082e,0000 (search 0x082E followed by 0x0000 in memory region 0 – 10000)
- Found at 2AF8 – 6C30 - 6C5E. After disassembling those in MAME debugger with Ctrl + D, we notice that routine at 2AF8 has the same structure; this is probably the routine to be patched.

Second part of the cheat, a new search (6700 0006 is a short branch so it might be used very few):

- find 0,10000,w.6700,0006,6100,0014
- Found at 5900. After examining, it's also identical routine.

Third part of the cheat, last search:

- find 0,10000,w.266f,6700
- Found at several places, but if we check 5D66, that's the identical routine.

So it's time to check these 3 again and construct the cheat, of course based on [galsnew]. That cheat should be checked by playing, of course.



[missw96] original code:

```
002AF8: 082E 0000 0015      btst    #$0, ($15,A6)
002AFE: 67B6                    beq     $2ab6           ;this should be 60, as in [galsnew]
002B00: 2248                    movea.l A0, A1

0058F8: 0839 0000 00C0 211D      btst    #$0, $c0211d.1
005900: 6700 0006                beq     $5908           ;this should be 60, as in [galsnew]
005904: 6100 0014                bsr     $591a

005D58: B439 00C0 5001            cmp.b   $c05001.1, D2
005D5E: 6700 00A6                beq     $5e06
005D62: B439 00C0 266F            cmp.b   $c0266f.1, D2
005D68: 6700 0090                beq     $5dfa           ;branch counter should be 02,
005D6C: B439 00C0 2670            cmp.b   $c02670.1, D2   ;as in [galsnew]
005D72: 6700 008C                beq     $5e00
```

```
<cheat desc="Invincibility">
  <script state="on">
    <action>temp0=maincpu.mb@2AFE</action>
    <action>temp1=maincpu.mb@5900</action>
    <action>temp2=maincpu.mb@5D6B</action>
  </script>
  <script state="run">
    <action>maincpu.mb@2AFE=60</action>
    <action>maincpu.mb@5900=60</action>
    <action>maincpu.mb@5D6B=02</action>
  </script>
  <script state="off">
    <action>maincpu.mb@2AFE=temp0</action>
    <action>maincpu.mb@5900=temp1</action>
    <action>maincpu.mb@5D6B=temp2</action>
  </script>
</cheat>
```

Final words: Cheat tested, as it's supposed to be, in every circumstance invincibility is preserved. So, as this game has 3 clones ([missmw96], [missw96a] and [missw96b]); it will be an easy approach to apply this cheat on them, with identical routine search method.

## [umk3] Ultimate Mortal Kombat 3 (C) 1995 MIDWAY



“An update of "Mortal Kombat 3", with new characters added to the original MK3 cast. There are lots of other additions and goodies in the game (see Updates section (REV. 1.0)).

- TECHNICAL -

Midway Wolf Unit Hardware

Main CPU : TMS34010 (@ 6.25 Mhz)

Sound CPU : ADSP2105 (@ 10 Mhz) Sound

Chips : DMA-driven (@ 10 Mhz)

Players : 2 Control : 8-Way Joystick Buttons

: 6 => [1] High Punch, [2] Block, [3] High

Kick => [4] Low Punch, [5] Low Kick, [6]

Run”

#reverseengineer #adaptexisting

Goal: To adapt “Auto Fatalities” (CPU or Player finishes with fatality or Stage fatality) from [mk2].

Known facts: [mk2] has these 3 cheats; with some luck, they can be transferred to [umk3] successfully.

First, some info about TMS34010 processor, most MIDWAY games used this processor as main CPU. Its ROM addressing is quite weird, RAM addressing is also different. Examples:

- MAME cheat xml address < > TMS34010 address exchanging:

```
<action>maincpu.mw@0003FD8=0300</action> ;[mk2] Always have Secret Game Over Screen
```

You won't notice a “3FD8” address in ROM region of TMS34010, so, here is the formula to convert it to TMS34010 address:

$(\text{xml ROM address}) * 8 + \text{FF800000} = (\text{TMS34010 real address})$

So,  $3\text{FD8} * 8 + \text{FF800000}$  is FF81FEC0. Let's see,

FF81FE40:	09E8 3BA0 FF82	MOVI	FF823BA0h,A8	;store FF823BA0 value to A8 register
FF81FE70:	09C0 0014	MOVI	14h,A0	;store 0x14 to A0 register
FF81FE90:	0D5F EC20 FF80	CALLA	FF80EC20h	;call routine at FF80EC20 and return
FF81FEC0:	C904	JRNC	FF81FF10h	;if carry flag is 0, branch FF81FF10

So, this cheat stores 0x0300 into FF81FEC0, result is:

FF81FEC0:	0300	NOP	;No Operation
-----------	------	-----	---------------

As you can easily guess, reverse calculation is needed to get xml ROM address, like this:

$[(\text{TMS34010 real address}) - \text{FF800000}] / 8 = (\text{xml ROM address})$

- RAM cheat example:

```
<action>maincpu.pb@10602F0=01</action> ;[mk2] Drain all Energy Now! PL1
```

This is real address in RAM area, but that's how you'll see in MAME debugger (Ctrl + M memory window):

1060280:	000000000000000001	0079FFFC00000000
	(1060280 to 10602C0)	(10602C0 to 1060300)
	(reverse order)	(reverse order)

Example of memory locations and their values:

1060280 is 01	10602E8 is FF	10602F0 is 79	(all in hexadecimal)
---------------	---------------	---------------	----------------------

As RAM – ROM handling of TMS34010 is clear now, here are existing cheats for [mk2]:

```
<action>maincpu.mw@00428D2=temp0 </action>
<action>maincpu.mb@003677F=temp1 </action>
<action condition="param==01 OR param==03">maincpu.mw@00428D2=1002</action>
<action condition="param==01 OR param==02">maincpu.mb@003677F=C0</action>
```

First entry, 428D2, is converted into real address:  $428D2 * 8 + FF800000 = FFA14690$

Second entry, 3677F, is converted:  $3677F * 8 + FF800000 = FF9B3BF8$

Disassembly of both entries:

FFA14640: 5601	XOR	A0,A1	
FFA14650: 4820	CMP	A1,A0	
FFA14660: C800 033C	JRC	FFA17A40h	
FFA14680: B5A4 0210	MOVE	*A13(210h),A4,0	;move value in (A13+\$0210) to A4
FFA146A0: 0BA4 0002 0000	ORI	2h,A4	
FFA146D0: B08D 0210	MOVE	A4,*A13(210h),0	
FF9B3B90: 09C0 0022	MOVI	22h,A0	
FF9B3BB0: 1942	MOVK	Ah,A2	
FF9B3BC0: 0D5F 80D0 FF83	CALLA	FF8380D0h	
FF9B3BF0: C85A	JRC	FF9B41A0h	;if Carry is 1, branch FF9B41A0

When cheats are on:

FFA14680: B5A4 1002	MOVE	*A13(1002h),A4,0	;new index, move (A13+\$1002) to A4
FF9B3BF0: C05A	JR	FF9B41A0h	;branch FF9B41A0

As we investigate, especially `MOVE *A13(1002h),A4,0` instruction code, A4 should be zero, its pointer is changed to \$1002 to read zero from further memory. So, plan for [umk3] cheat is:

- Find first entry, same or similar routine, and force A4 register to get value 00. (We'll improve the cheat – first entry of [mk2] cheat is not reliable, the one we'll use in [umk3] will be stable.)
- Find second entry, same or similar routine, and remove conditional branch, apply direct branch.

When we run [umk3] and perform these searches from MAME debugger:

- **find ff800000,800000,w.5601,4820** ;(search 0x5601 followed by 0x4820 in memory region FF800000 - end)
- Found at FFB5D3B0. Routine is examined, equivalent of above.
- **find ff800000,800000,w.1942,0d5f**
- Found at FFB29D50. Again, routine is equivalent.

So, here is original [umk3] code:

FFB5D3B0: 5601	XOR	A0,A1	
FFB5D3C0: 4820	CMP	A1,A0	
FFB5D3D0: C843	JRC	FFB5D810h	
FFB5D3E0: B5A4 0210	MOVE	*A13(210h),A4,0	;A4 should be zero here
FFB5D400: 0BA4 0002 0000	ORI	2h,A4	;0x00 OR 0x02 = 0x02 to A4
FFB5D430: B08D 0210	MOVE	A4,*A13(210h),0	
FFB29D30: 09C0 002B	MOVI	2Bh,A0	
FFB29D50: 1942	MOVK	Ah,A2	
FFB29D60: 0D5F F580 FF9A	CALLA	FF9AF580h	
FFB29D90: C800 00A5	JRC	FFB2A800h	;we'll remove conditional branch

Here is cheat applied code:

FFB5D400: 09E4 0002 0000	MOVI	2h,A4	;move (immediate) 0x02 to A4
FFB29D90: C000 00A5	JR	FFB2A800h	;always branch to FFB2A800

OK. It seems we found equivalents of two different routines responsible for auto fatality, from [mk2] and adapted them for [umk3]. But when it comes to testing, if you construct a xml cheat from these and start testing, this won't work 100% as it should, because other game's behaviors will differ: [umk3] has "animalities" feature. As a result, when you test this cheat in this state, you'll notice:

- 1) Stage fatalities will have some bugs. (There won't be auto Stage fatalities 100%.)
- 2) Auto fatality – CPU fatality will sometimes result in animality.

So, from now on, we need to investigate those two routines very detailed to fix those issues. It seems first routine (we patch with A4=0x02) is responsible for automatic Player / CPU movement on screen before fatality starts, so, second routine is probably the one responsible of black screen and start of fatality.

What to do:

- Take some snapshots when the cheat is on, and Player / CPU is about to die, on last round with Shift + F7.
- After you take several snapshots; you'll be able to catch the bugs, will be able to take snapshots of:

Stage fatality enabled, but ended with no Stage fatality (cheat failed).

Auto or CPU fatality enabled, but ended with animality (side effect).

Now it's time to trace the code and find the branch changes. For normal and buggy snapshots, load them with F7 in MAME and for start; give **bp ffb29d30** command for a breakpoint just before fatality starts. When that breakpoint is triggered, **trace <filename>** command and then after operation is over, **trace off** command. You need to take 4 trace logs, two for 100% success fatalities (normal and stage) and two for unwanted results (stage fatality failed and animality occurred).

If you compare those traces, two by two, here is the result for stage fatality fail:

(Of course, you can compare them manually, or with file compare tools – Where to focus is first changes from very first start.)

Stage fatality OK VS Stage fatality failed:

FFB29DB0: 09C0 01F4	MOVI	1F4h,A0	
FFB29DD0: 0D5F A770 FF96	CALLA	FF96A770h	
FFB29E00: C90D	JRNC	FFB29EE0h	;in OK trace, no branch
FFB29E10: 05A1 2260 0106	MOVE	@1062260h,A1,0	;in buggy trace, branch occurs

Auto fatality OK VS animality happens:

FFB2A8D0: 09C0 0032	MOVI	32h,A0	
FFB2A8F0: 0D5F A770 FF96	CALLA	FF96A770h	
FFB2A920: C8CE	JRC	FFB2A610h	;in OK trace, no branch
FFB2A930: 09E0 B7D0 FFB2	MOVI	FFB2B7D0h,A0	;if branch happens, animality

So, we had 2 lines of cheat before, converted from [mk2] and finally added 2 more lines to avoid the bugs, so here are all of them listed (original code):

FFB5D400: 0BA4 0002 0000	ORI	2h,A4	;0x00 OR 0x02 = 0x02
FFB29D90: C800 00A5	JRC	FFB2A800h	;we'll remove conditional branch
FFB29E00: C90D	JRNC	FFB29EE0h	;in OK trace, no branch
FFB2A920: C8CE	JRC	FFB2A610h	;in OK trace, no branch

Here are in cheat applied format:

FFB5D400: 09E4 0002 0000	MOVI	2h,A4	;move (immediate) 0x02 to A4
FFB29D90: C000 00A5	JR	FFB2A800h	;always branch to FFB2A800
FFB29E00: 0300	NOP		;conditional branch removal
FFB2A920: 0300	NOP		;conditional branch removal

To test them, we're converting these real addresses to xml cheat format with the formula:

$[(\text{TMS34010 real address}) - \text{FF800000}] / 8 = (\text{xml ROM address})$



```

<cheat desc="Select Perm Fatality Type">
  <comment>Automatic Stage Fatality (Use in Kombat Zones - The Subway, Bell
  Tower, The Pit III, Scorpion's Lair)</comment>
  <parameter>
    <item value="0x02">CPU always finishes with a Fatality</item>
    <item value="0x01">Automatic Fatality</item>
    <item value="0x03">Automatic Stage Fatality</item>
  </parameter>
  <script state="on">
    <action>temp0 =maincpu.mw@006BA80</action>
    <action>temp1 =maincpu.mb@00653B3</action>
    <action>temp2 =maincpu.mw@00653C0</action>
    <action>temp3 =maincpu.mw@0065524</action>
  </script>
  <script state="run">
    <action>maincpu.mw@006BA80=temp0 </action>
    <action>maincpu.mb@00653B3=temp1 </action>
    <action>maincpu.mw@00653C0=temp2 </action>
    <action>maincpu.mw@0065524=temp3 </action>
    <action condition="param==01 OR param==03">
    maincpu.mw@006BA80=09E4</action>
    <action condition="param==01 OR param==02">
    maincpu.mb@00653B3=C0</action>
    <action condition="param==01 OR param==03">
    maincpu.mw@00653C0=0300</action>
    <action condition="param==01 OR param==02">
    maincpu.mw@0065524=0300</action>
  </script>
  <script state="off">
    <action>maincpu.mw@006BA80=temp0 </action>
    <action>maincpu.mb@00653B3=temp1 </action>
    <action>maincpu.mw@00653C0=temp2 </action>
    <action>maincpu.mw@0065524=temp3 </action>
  </script>
</cheat>

```

Final words: As we test it very detailed, in every ways and combinations, we notice that it's working like a charm. But as you might guess, there are still more tasks to do. Firstly, clones of [umk3]. But attentive research needed, because this one and similar games have different clone game structure! If you check history.dat:

In Updates section, what's different in REV1.0 – REV1.1 – REV1.2 is mentioned. It seems in all versions; this cheat is supposed to work, so now it's time to prepare cheats for the clones ([umk3r10] and [umk3r11]).

Another game in same series, [mk3]: Now for the last step, we need to adapt the cheat above to [mk3]. Again, version (revision) check needed, so we check history.dat:

In Updates section, revisions are: PROTO4.0 – REV1.0 – REV2.0 – REV2.1  
When we examine all those revision history briefly, here is the result:

- For [mk3] and [mk3r20] we adapt above and apply the cheat with 4 lines:  
(movement – fatality start – stage fix – animality fix)
- For [mk3r10] and [mk3p40] we adapt above and apply the cheat with 3 lines (Animality not implemented):  
(movement – fatality start – stage fix)

## [ddpdfk] DoDonPachi Dai-Fukkatsu (C) 1998 CAVE



### “- TECHNICAL -

Cave CV1000D Hardware

CPU : Hitachi SH-3 CPU, 133 Mhz Clock Sound Chip : Yamaha YMZ770C-F Chip U1 : IS42S32400D 128Mb SDRAM @ 166mhz (standard SH3 has MT48LC2M32 64Mb SDRAM) Chip U4 : 32Mbit Spansion S29JL032H (standard SH3 has a 16Mbit Spansion S29AL016D).

### - TRIVIA -

DoDonPachi Dai-Fukkatsu was showed fully playable at AOU 2008, which took place 15-16th February at the Makuhari Messe, Japan. The game was then released on May 22, 2008 in Japan.

### - SERIES -

1. DonPachi (1995) 2. DoDonPachi (1997) 3. Bee Storm - DoDonPachi II (2001) 4. DoDonPachi Dai-Ou-Jou (2002) 5. DoDonPachi Dai-Fukkatsu (2008) 6. DoDonPachi Saidaioujou (2012) 7. DoDonPachi Maximum (2012) [Apple App Store]”

#easteregg #ci&cn #linearlevel #hook&customcode #boss

Goal: As this game became fully emulated with MAME 0.153, no cheats are present (yet) and at least, preliminary cheats need to be found and installed.

Known facts: Game is supposed to have same gameplay features like DoDonPachi, also after searching for this game on internet (both arcade and console port versions), these can be gathered:

- There's a hidden style, where “Style Select” is displayed, only “Bomb” and “Power” styles are selectable, but hidden style is named “Strong”.
- There's a hyper gauge, horizontal gauge just above planes remaining, at upper left corner. When it's full, hyper shoot becomes active and it's the most powerful shooting in game.
- It seems, as DoDonPachi game, this game also has Second Loop feature. First loop has 5 Areas (levels), after completing them with meeting certain requirements; Second Loop starts from Area 1. But after Second Loop is completed, special Area 6 is reached (special final boss).

So, let's start.

For infinite lives research, stock **ci**, **cn -1** method in MAME debugger will be fine. But, **ci** command causes MAME debugger halt, so we'll start with **ci uw** (word search – instead of byte). After starting a 1UP game and **ci uw** command at first, and then **cn -1** when a life is lost, finally, we find the memory location, it's C5C3703. It's for PL1.

Here is a hint to find the value for PL2 (or all other players available): **wp c5c3703,1,w** command in MAME debugger, get PC address (PC=C1F350E), now disable that watchpoint, create a breakpoint for that PC address as **bp c1f350e** and wait for that breakpoint is triggered, but for PL2 this time (when PL2 loses a life). If you check previous instruction code, from MAME debugger window, it's C1F350C: MOV.W R2,@R4 (move word value of R2 into R4). R2 is 0x0002 then, R4 is C5C3A0E. When this move happens, C5C3A0F becomes 02, number of lives for PL2 (in byte format). So here is the cheat (0x07 is poked because 6 ships will cover that whole area):

```
<cheat desc="Infinite Lives PL1">
  <script state="run">
    <action>maincpu.pb@C5C3703=07</action>
  </script>
</cheat>

<cheat desc="Infinite Lives PL2">
  <script state="run">
    <action>maincpu.pb@C5C3A0F=07</action>
  </script>
</cheat>
```

Now similar procedure for infinite bombs, this time I can search with bytes like this, as I can assume that PL1 parameters (lives, bombs, energy ...) are not in further locations in whole memory. So,

**ci ub,c5c0000,10000** (search for bytes, between c5c0000 and c5d0000) and **cn -,1** after a bomb is used:

C5C3706 is the memory location for number of bombs for PL1. But when you play game for some time, you'll notice that number of bombs can be max. 6 and current bomb display slot allows max. 5. So, next byte after C5C3706 can be that slot? Probably, because C5C3707 is 05. So we need to poke 0x0606 into C5C3706 for the bombs of PL1. Above method, getting breakpoint, applying it for PL2 and using a bomb, gives PL2 bomb location, it's C5C3A12.

```
<cheat desc="Infinite Bombs PL1">
  <script state="run">
    <action>maincpu.pw@C5C3706=0606</action>
  </script>
</cheat>
```

```
<cheat desc="Infinite Bombs PL2">
  <script state="run">
    <action>maincpu.pw@C5C3A12=0606</action>
  </script>
</cheat>
```

How to find & unlock Strong Style? Here is a tip again, after resetting game and reaching "Style Select" screen, only "Bomb" and "Power" is visible & selectable. So, there should be one or more memory locations that control what is selected there, e.g. 0x01 for Bomb, 0x02 for Power, maybe 0x03 for hidden Strong style? So, what to do is, starting a new cheat search with **ci uw** when default "Bomb" is selected, and move cursor to "Power" and in debugger **cn +,1** (index increased) and move cursor again to "Bomb" and in debugger **cn -,1** (index decreased). After a few tries, it's displayed that 6 cheats found. I'll give it a shot for C88D01E; it was the first entry when **cheatlist** command was given. **wp c88d01e,1,w** command to monitor write operations, we notice that each time moving cursor, it's triggered. So, it's time to be careful and examine the routine that triggers the watchpoint. When moving cursor from "Bomb" to "Power", PC is C2A8264. After applying step into (F11) to execute code one by one, this is interesting:

0C2A826A: 5638	MOV.L	@(\$20,R3),R6	;move R3+\$20 = C88D070 value into R6, it's 02
0C2A826C: 3263	CMP/GE	R6,R2	;compare if R2 <= R6. R2 is 01, "Power"
0C2A826E: 8FAF	BFS	\$0C2A81D0	;branch backwards

This routine probably checks navigation between "Bomb" "Power" and the hidden "Strong", in forward mode, moving cursor down. So, if manually change 02 value of C88D070 to 03 in MAME debugger memory window, something very interesting happens: We can navigate down from "Power", there's no "Strong" writing there but if we move the cursor below "Power" and press button, STRONG Style is selected!

So, last steps to finalize this cheat are, setting a special watchpoint for C88D070 (to understand how its initial value is read by game code).

**wp c88d070,1,w,wpdata==2** ;(Watchpoint will only be triggered if stored data is equal to 0x02)

After player is selected, before Style screen, this watchpoint is triggered. PC=C2A69C6. Here is disassembly:

0C2A69B6: D018	MOV.L	@(\$0060,PC),R0	[0C2A6A18]
0C2A69B8: 400B	JSR	R0	
0C2A69BA: 787C	ADD	#\$7C,R8	
0C2A69BC: 2008	TST	R0,R0	;R0 and R0. T flag is 1 if result is 0 (0 and 0)
0C2A69BE: 8F02	BFS	\$0C2A69C6	;if T (true) is 1, delayed branch to C2A69C6
0C2A69C0: 1808	MOV.L	R0, @(\$20,R8)	;this is executed, 0x02 to C88D070 and jumps
0C2A69C2: E503	MOV	#\$03,R5	;here can only be executed if above R0 is 0x00
0C2A69C4: 1858	MOV.L	R5, @(\$20,R8)	
0C2A69C6: 5188	MOV.L	@(\$20,R8),R1	
0C2A69C8: E00C	MOV	#\$0C,R0	
0C2A69CA: 0107	MUL.L	R0,R1	
0C2A69CC: 1819	MOV.L	R1, @(\$24,R8)	

(On delayed branches, the instruction code after branch is executed, and then branch happens.)

We're about to find it, so, R0 should be zero at C2A69BC. To investigate it, remove all watchpoints and breakpoints and then this command: **bp c2a69b6** to understand where R0 is set. Now it's time to wait for breakpoint to be triggered, and then execute commands one by one with F11 key from MAME debugger. Here it is:

0C1EA802:	6020	MOV.B	@R2,R0	;copy byte from R2 (C5C1E1D currently) into R0
0C1EA804:	2008	TST	R0,R0	;it's zero by default
0C1EA806:	2FE6	MOV.L	R14,@-R15	
0C1EA808:	0029	MOVT	R0	;move T bit into R0. T bit is 1 so R0 gets 01
0C1EA80A:	2008	TST	R0,R0	;after this test operation, T bit is 0
0C1EA80C:	0029	MOVT	R0	;move T to R0. R0 is again 00
0C1EDF4E:	2008	TST	R0,R0	;00 and 00. T bit is 1
0C1EDF50:	8900	BT	\$0C1EDF54	;branch C1EDF54 if T=1. (non-delayed branch)
0C1EDF52:	E803	MOV	#\$03,R8	
0C1EDF54:	6083	MOV	R8,R0	;R8 was 02, stored to R0. R0 is finally 02

And after performing more “step into” operations with F11 key, we’re sure that R0 value is not changing until execution of this subroutine is completed. Above is history of R0 register in this subroutine. So, probably C5C1E1D address responsible of normal or extended modes. When we check memory, its value is 0x00 by default. What if we change its value to 0x01? Let’s see...

After game reset, manually poking 0x01 into C5C1E1D: When you reach Style select screen, “Strong” style is displayed, and selectable! Of course, detailed tests should be performed as, starting a game with both players. (As C5C1E1D cannot reset itself after the cheat is turned on and then off, to reset its value, script state=“off” needs to be added as below.)

```
<cheat desc="Unlock Strong Style (Both Players)">
  <comment>Should be activated before STYLE SELECT screen.</comment>
  <script state="run">
    <action>maincpu.pb@C5C1E1D=01</action>
  </script>
  <script state="off">
    <action>maincpu.pb@C5C1E1D=00</action>
  </script>
</cheat>
```

Now it’s time to seek for Hyper Gauge memory location, to find and maximize it for Hyper Shoot. In most games, value of a gauge (life bar, boss energy indicator, level progress bar etc.) is stored as bytes, words or double words in specific RAM area. So, what to do is simple: Initializing a new cheat search with **ci**, when visual display on the gauge increases, **cn +** command and repeating this **cn +** several times until memory location is found.

So here is the result applied to the game: After 1UP game has started, Hyper Gauge is empty. Especially in modern games, these gauge – progress bar values are stored in words or double words than bytes, we need to start a new search by **ci uw** command. And, entering **cn +** command when we see progress on Hyper Gauge. After repeating this several times, until Hyper Gauge is full, cheat is found, C5C392E, 0x0000 to 0x0200, as we guessed, word value. Setting a watchpoint for C5C392E for write operation, getting PC value and setting a breakpoint for that PC address, and starting a 2UP game will give PL2 Hyper Gauge address, as C5C3C3A. So, here is the cheat (to avoid permanently Hyper Shooting, current Hyper Gauge value is backedup, it will be restored when cheat is turned off):

```
<cheat desc="Infinite Hyper Counter PL1">
  <comment>Activate with Button D (Control Type A) or Button C (Control Type B) .
  Provides maximum firepower.</comment>
  <script state="on">
    <action>temp0=maincpu.pw@0C5C392E</action>
  </script>
  <script state="run">
    <action>maincpu.pw@C5C392E=0200</action>
  </script>
  <script state="off">
    <action>maincpu.pw@0C5C392E=temp0</action>
  </script>
</cheat>
```

```

<cheat desc="Infinite Hyper Counter PL2">
  <comment>Activate with Button D (Control Type A) or Button C (Control Type B) .
  Provides maximum firepower.</comment>
  <script state="on">
    <action>temp0=maincpu.pw@0C5C3C3A</action>
  </script>
  <script state="run">
    <action>maincpu.pw@C5C3C3A=0200</action>
  </script>
  <script state="off">
    <action>maincpu.pw@0C5C3C3A=temp0</action>
  </script>
</cheat>

```

Invincibility cheat, it's a vital cheat and most games support it partially. Especially, after you lose one life in most of the games, incoming life starts with partial invincibility (i.e. our hero in fighting games or our ship in shooter games appear blurred – flashing or in a darker color to indicate invulnerability for some seconds). It means there's mostly a countdown timer in memory for that invulnerability period. When timer has not reached to zero, player won't get hits – wounds. Sometimes additionally there may be a byte, related with that countdown timer address, to store invulnerability status (i.e. 0x01 for on, 0x00 for off).

So, a simple check and then research is enough, we hope. When we check gameplay, at very first start or after losing a life, new ship appears with a shield and that shield disappears slowly. Then only thing to do is that shield's countdown timer research. As time is limited in this search (shield disappears in a few seconds), we can use a special cheat init method as below:

In 1UP game, at very first start or after losing a life, immediately, **ci uw** to initialize cheat search in words, **gv** command to wait for new VBLANK happens, **cn -,1** to mention countdown timer decreased by one, and then **gv** and **cn -,1** repeatedly, and **cheatlist** when number of cheats are low, results are displayed. One cheat memory location starts from C5C3 (in same memory region as above found cheats). It might be what we're looking for. Let's check that, C5C3886. Yes, after checking its value from monitor window, it starts from 0x0100 and decreases in time and reset to 0x0000, that time shield is completely gone. There may be a byte (invulnerability) to check if this value is above zero, to find it, starting a new game again and after our ship is displayed with its shield, **wp c5c3886,1,r** command to monitor reading value from this memory location. Read happens from 2 PC addresses, but one of them, after we trace all routines with F11 key in MAME debugger, unfortunately nothing special happens when C5C3886 becomes 0x0000, no other bytes control its progress (shield on or off, same time checking invulnerability), so we need to construct invincibility cheat based on this shield value.

As we test what fixed value to poke into C5C3887, as byte, 0x02 will be fine, because even it's updated every frame, if 0x01 is selected, by game code it becomes to 0x00 with decrease and invincibility is removed. So, with choosing 0x02, game code can only decrease it to 0x01 and invincibility is still on, cheat will update it again to 0x02, also shield is not visible with this value. Also with breakpoint method, we find shield's memory location for PL2, it's C5C3B93 as byte.

```

<cheat desc="Invincibility PL1">
  <script state="run">
    <action>maincpu.pb@C5C3887=02</action>
  </script>
</cheat>

<cheat desc="Invincibility PL2">
  <script state="run">
    <action>maincpu.pb@C5C3B93=02</action>
  </script>
</cheat>

```

Now, it's time for the Area cheat (ability to select start area). We need to get memory location of current Area at first, but it will take some time because we need to complete some Areas. As cheat init in byte format is problematic for this game, let's initialize a new cheat for Area number. After starting a new game, **ci uw** to initialize, above Invincibility cheat may help a lot, after turning it on, we go on playing and when Area 2 appears, **cn +,1** and repeating this in every next Area. Also, taking MAME snapshot with Shift + F7 will be fine, after defeating Area 1 boss, because after it, Area 2 will start, so the byte we'll find later can be monitored with loading that snapshot instead of playing from the beginning. So, **cheatlist** reported 6 addresses. 3 of them start with C5C1.... So one of them is probably holding current Area, because the cheats we found were starting with C5C3....



So, it's time to load that snapshot when Area 1 boss is defeated, and add these watchpoints for write operations:

**wp c5c1e82,1,w      wp c5c1f2e,1,w      wp c5c1f32,1,w**

As we investigate all code around where write operations happened for these 3 above, the result is:

C5C1F33 value is physical Area value, but it's increased by one after boss is defeated. When you defeat boss in Area 1, this byte becomes 0x01. When next Area starts (2), its value is read and poked into C5C1F2F. So, C5C1F2F is responsible for current Area progress, incoming enemies, mid-bosses etc. while playing.

So, first test is, starting directly from Area 2 is possible? Let's check. After starting a new game, **wp c5c1f2f,1,w** command, when it stops, manually changing that byte into 0x01, result is, graphics distorted. We know that normal game starts with an animation, it seems that animation and Area 1 start is "forced" so it's not possible to start from desired Area. Then, what to do? As in DoDonPachi game, we can adapt this cheat as "Select Area to follow Area 1" so, after Area 1 is finished, custom code can force desired Area number and game goes on that way.

Then, we need to investigate C5C1F2F store progress, more detailed this time. Because, there might be another byte controlling Second Loop. (DoDonPachi game has that feature, normal areas, and Second Loop for the same areas, and a special last area.) C5C1F33 is set 0x01 at PC=C1F0104, so here is trace log of executed code just before that address:

```

0C1EF9F4: 5048      MOV.L   @($20,R4),R0      ;(C5C1F2F) into R0. (0x00)
0C1EF9F6: 6EF3      MOV     R15,R14
0C1EF9F8: 6FE3      MOV     R14,R15
0C1EF9FA: 000B      RTS

...
0C2024DA: 7001      ADD     #$01,R0          ;R0 is 0x01 now.
0C2024DC: 6503      MOV     R0,R5           ;copy R0 register to R5 also

...
0C2024E4: D019      MOV.L   @($0064,PC),R0   ;PC+$0064 is 0C20254C here, store it into R0

...
0C2024F6: 402B      JMP     R0               ;jump to R0. (Its value is 0C1F00B0 by default)

...
0C1F00BA: 6953      MOV     R5,R9           ;copy R5 register to R9. R9 is 0x01 now

...
0C1F00F6: E305      MOV     #$05,R3
0C1F00F8: 3936      CMP/HI  R3,R9            ;compare R9 with 0x05 (last Area value).
0C1F00FA: 8911      BT      $0C1F0120

...
0C1F0100: 110C      MOV.L   R0,@($30,R1)
0C1F0102: 1199      MOV.L   R9,@($24,R1)    ;poke 0x01 into (C5C1F33)

```

So, for the last step, we need to complete all areas (invincibility cheat will help) and start Second Loop and check memory locations just after C5C1F33. We notice that C5C1F37 and C5C1F3B locations become 0x01 when Second Loop is active. Here is the table for all available areas:

C5C1F37 & C5C1F3B value	C5C1F2F value	Current Area
00	00	1
00	01	2
00	02	3
00	03	4
00	04	5
01	00	1 (Second Loop)
01	01	2 (Second Loop)
01	02	3 (Second Loop)
01	03	4 (Second Loop)
01	04	5 (Second Loop)
01	05	6 (Special-Final Boss)

Very likely, it's impossible to find free memory in main routines area, to insert our custom code, so hook method should be tried for this. First, just before this value increment (area value) happens, a proper JSR or JMP needs to be found, and then, redirecting JMP/JSR branch address to our custom code should be the first step. But most important part is: Our custom code should not destroy currently used register values! If register values are limited, backup – restore of registers can be applied.

Here is proper way applied as hook & custom code for this game:

```

0C2024E4: D019      MOV.L    @($0064,PC),R0      ;PC+$0064 is 0C20254C here, store it into R0
0C2024F6: 402B      JMP      R0                  ;jump to R0. (Its value is 0C1F00B0 by default)

```

Memory location 0C20254C will be the hook point. After examining memory area with Ctrl + M command, we notice that 0C670000 and afterwards are free, all filled with 0xFF. So, custom code will be installed there.

1) Store double word value of 0C670000 into memory location 0C20254C.

2) Create custom code starting from 0C670000:

```

0C670000      MOV R4,R3          6343 ;backup R4 register to R3 = C5C1ECC
              ADD #$60,R3      7360 ;add $60 to R3 so it becomes C5C1F2C
              MOV.L @($08,R3),R0 5032 ;get C5C1F37 (byte) value to R0. (second loop)
              CMP/PL R0         4015 ;is it plus (>0) second loop active?
              BT end            890F ;if second loop is active, do not apply, go end
              MOV.L @($00,R3),R0 5030 ;current area value (C5C1F2F) into R0
              CMP/PL R0         4015 ;current area >1 ?
              BT end            890C ;if it's not 1, terminate cheat. Go end
              MOV #01,R0        E001 ;here, our selected param is stored into R0
              MOV #05,R2        E205 ;max. Area number 05 into R2
              CMP/HI R2,R0       3026 ;if param >5 T=1.
              BT lev6           8902 ;and apply Area 6 parameters.
x             MOV R0,R5         6503 ;R5 is set, (original code's Area nr register)
              BRA end           A006 ;restore default jmp address and go on.
              NOP              0009
lev6          MOV #$01,R0       E001 ;R0 is 0x01
              MOV.L R0,@(08,R3) 1302 ;store it into C5C1F37 (2nd loop memory loc)
              MOV.L R0,@(0C,R3) 1303 ;store it into C5C1F3B (2nd loop memory loc)
              MOV #05,R0        E005 ;R0 is set to 0x05. (Area 6 will appear)
              BRA x             AFF7 ;store it as current Area and go on
              NOP              0009
end           MOV.L @($0001,PC),R0 D001 ;read original R0 value (original jmp value)
              JMP R0            402B ;go on executing original routine (0C1F00B0)
              NOP              0009
              0C1F ;original jmp address
              00B0 ;original jmp address

```

Above routine checks current Area byte and Second Loop byte at first, current Area byte should be 0x00 (Area 1) and Second Loop byte must be 0x00 (Second Loop not active) for the cheat to operate (cheat only works after Area 1 is cleared & no Second Loop), when these conditions are met, according to selected "param", this code prepares values to be poked into C5C1F2F – C5C1F37 – C5C1F3B, with conditional checks.

Here is finalized cheat, its only "run" state:

```

<action>maincpu.pd@0C20254C=0C670000</action>
<action>maincpu.pq@0C670000=6343736050324015</action>
<action>maincpu.pq@0C670008=890F50304015890C</action>
<action>maincpu.pb@0C670010=E0</action>
<action>maincpu.pb@0C670011=param</action>
<action>maincpu.pq@0C670012=E205302689026503</action>
<action>maincpu.pq@0C67001A=A0060009E0011302</action>
<action>maincpu.pq@0C670022=1303E005AFF70009</action>
<action>maincpu.pq@0C67002A=D001402B00090C1F</action>
<action>maincpu.pw@0C670032=00B0</action>

```

First line shows changed hook address, all other lines have instruction codes of our custom code, but code is split into two parts because "param" should be stored as a byte. That parameter is selected from cheat menu by player.

On next page, full cheat is displayed with all states and its comment:

```

<cheat desc="Select Area to follow Area 1">
  <comment>If you select Second Loop: (harder game) After a cutscene, choose YES.
  Area 6 is final stage after Second Loop. </comment>
  <parameter>
    <item value="0x01">Area 2</item>
    <item value="0x02">Area 3</item>
    <item value="0x03">Area 4</item>
    <item value="0x04">Area 5</item>
    <item value="0x05">Second Loop</item>
    <item value="0x06">Area 6 (Special)</item>
  </parameter>
  <script state="on">
    <action>temp0 =maincpu.pd@0C20254C</action>
    <action>temp1 =maincpu.pq@0C670000</action>
    <action>temp2 =maincpu.pq@0C670008</action>
    <action>temp3 =maincpu.pq@0C670010</action>
    <action>temp4 =maincpu.pq@0C670018</action>
    <action>temp5 =maincpu.pq@0C670020</action>
    <action>temp6 =maincpu.pq@0C670028</action>
    <action>temp7 =maincpu.pd@0C670030</action>
  </script>
  <script state="run">
    <action>maincpu.pd@0C20254C=0C670000</action>
    <action>maincpu.pq@0C670000=6343736050324015</action>
    <action>maincpu.pq@0C670008=890F50304015890C</action>
    <action>maincpu.pb@0C670010=E0</action>
    <action>maincpu.pb@0C670011=param</action>
    <action>maincpu.pq@0C670012=E205302689026503</action>
    <action>maincpu.pq@0C67001A=A0060009E0011302</action>
    <action>maincpu.pq@0C670022=1303E005AFF70009</action>
    <action>maincpu.pq@0C67002A=D001402B00090C1F</action>
    <action>maincpu.pw@0C670032=00B0</action>
  </script>
  <script state="off">
    <action>maincpu.pd@0C20254C=temp0 </action>
    <action>maincpu.pq@0C670000=temp1 </action>
    <action>maincpu.pq@0C670008=temp2 </action>
    <action>maincpu.pq@0C670010=temp3 </action>
    <action>maincpu.pq@0C670018=temp4 </action>
    <action>maincpu.pq@0C670020=temp5 </action>
    <action>maincpu.pq@0C670028=temp6 </action>
    <action>maincpu.pd@0C670030=temp7 </action>
  </script>
</cheat>

```

Last cheat to be found is, "Kill Bosses with 1 hit" cheat. As we know, while "Select Area to follow Area 1" cheat research, we've noticed some bosses in game, middle bosses and Area bosses at the end of each area. After first area, other bosses were complex as they transformed to other bosses when boss energy gauge reached to certain positions. So, those will be tough to investigate. But the aim is, for each boss, there should be static memory locations that hold default energy gauge value, and their transform. Those memory locations can be transferred to other locations e.g. Area 2 boss is displayed. So, we'll monitor boss gauge with **ci** and **cn** - commands, but we need to find where that value comes from, initially. So if we change initial values, boss gauge indicator will appear as the value we changed. For all of these, 6 snapshots need to be taken in MAME, for all Areas (Select Area to follow Area 1 cheat can be used to start from Area 6). But it's better those snapshots are taken just before "Warning" is displayed before bosses appears, because just before boss display, their energy gauge can be initied.

So, starting from Area 1 boss, loading that snapshot, when boss energy gauge is filled to the maximum, **ci ud** to initialize cheat (double word chosen because boss energy gauges are very big, they may be hold in double words). And then, shooting at the boss and after some time, **cn** - command and **cn eq** command when no firing at boss, result is: C7A3128 and C89322C (initial value 00078140) is found. (These memory locations can be different each time loading snapshot and searching cheat with this method, because as explained before, they're copied from fixed location but their location currently may vary.) So, we need to find this initial 78140 value, where it comes from at first. When we search briefly, with loading snapshot again and **wp c7a3128,1,w**, it stops at C2071A4, checking code just before that:

```

0C207194: 5391      MOV.L    @($04,R9),R3      ;R9=C52DEE4, copy long C52DEE8 to R3 (00078140)
...
0C2071A2: 1639      MOV.L    R3,@($24,R6)     ;copy R3 (00078140) to C7A3128. (dynamic)

```

Same wp **c7a3128,1,w**, after hitting boss:

```

0C20744C: 1929      MOV.L    R2,@($24,R9)     ;copy R2 (current boss energy) to C7A3104+$24
0C20744E: 53A6      MOV.L    @($18,R10),R3    ;copy long C7A31F4+$18 to R3 (00000000)
0C207450: 3237      CMP/GT   R3,R2            ;compare if R2 > R3,
0C207452: 890D      BT       $0C207470        ;it will branch to C207470, boss is still alive
0C207454: 6C83      MOV      R8,R12          ;here is executed if boss energy has run out
0C207456: 7C4C      ADD      #$4C,R12

```

So, we need to investigate where this C7A31F4+\$18=C7A320C is poked.

```

0C20713E: 5934      MOV.L    @($10,R3),R9     ;R3=C52DEF8, copy long C52DF08 to R9 (00000000)
0C207140: 1296      MOV.L    R9,@($18,R2)     ;copy R9 (00000000) to C7A320C. (dynamic)

```

Now Area 1 boss parameters are clear, let's check original values:

```

C52DEE8:    00078140      ;it seems this is the "gauge display" value, not related with boss health.
C52DF08:    00000000      ;this value is read and compared, above routine.

```

So, for Area 1 boss, we need to change C52DF08 with 00078140 (max value), so when boss is hit, lower value is compared with this static 00078140 in PC=C207450 and it will go on from C207454 (boss dead).

Same needs to be applied for other Area bosses. But you'll notice that, other bosses have more parameters for second gauge (one boss dies and transforms into another boss in same Area), after some hard work here is all of boss tables revealed and ready for cheat format:

```

<action>maincpu.pd@0C52DF08=00078140</action> ;A1 boss max. power (1)      00000000
<action>maincpu.pd@0C52FE94=00083680</action> ;A2 boss max. power (1)      00024400
<action>maincpu.pd@0C52FEA4=00000000</action> ;A2 boss max. power (2) *    00024400
<action>maincpu.pd@0C52FE8C=00000001</action> ;A2 boss 1 hit kill cnt      00000000
<action>maincpu.pd@0C534A78=00091AE0</action> ;A3 boss max. power (1)      0007D100
<action>maincpu.pd@0C534A88=00000000</action> ;A3 boss max. power (2) *    00057E80
<action>maincpu.pd@0C534A9C=00000000</action> ;A3 boss max. power (3) *    00025280
<action>maincpu.pd@0C534A70=00000002</action> ;A3 boss 1 hit kill cnt      00000000
<action>maincpu.pd@0C5377C4=000A2380</action> ;A4 boss max. power (1)      0004A500
<action>maincpu.pd@0C5377D4=00000000</action> ;A4 boss max. power (2) *    0002C680
<action>maincpu.pd@0C5377E8=00000000</action> ;A4 boss max. power (3) *    0001DE80
<action>maincpu.pd@0C5377BC=00000002</action> ;A4 boss 1 hit kill cnt      00000000
<action>maincpu.pd@0C53B7C0=00100780</action> ;A5 boss max. power (1)      0007D100
<action>maincpu.pd@0C53B7D0=00000000</action> ;A5 boss max. power (2) *    00050A80
<action>maincpu.pd@0C53B7E4=00000000</action> ;A5 boss max. power (3) *    0002C680
<action>maincpu.pd@0C53B7B8=00000002</action> ;A5 boss 1 hit kill cnt      00000000
<action>maincpu.pd@0C53F600=00122E80</action> ;A6 boss max. power (1)      00000000

```

(\* means progress bar value and rightmost values after ";" are their default values.)

Short explanation, i.e. Area 4 boss: Boss maximum health is at 000A2380 (read from 0C5377A4) and stored to 0C5377C4 (compare value taken from here). Two sub-bosses exist, their progress bar values found and cleared (0C5377D4 and 0C5377E8). There's a counter for these bosses (including sub-bosses that appear with transform) and its default value is 00, increased by one and when a sub-boss dies. So, maximum number of bosses for that Area is stored to 0C5377BC. It means, when cheat is on, after one single hit, first compare will result in "boss dead" in routine above, and at second compare, sub-bosses check, as counter is modified to its maximum value, Area will be clear immediately!

On next page, full cheat is displayed. Notice that, if temp variables are more than 10, a special tag should be used as **tempvariables="XX"**:

```

<cheat desc="Kill Bosses with 1 hit" tempvariables="17">
  <script state="on">
    <action>temp0 =maincpu.pd@0C52DF08</action>
    <action>temp1 =maincpu.pd@0C52FE94</action>
    <action>temp2 =maincpu.pd@0C52FEA4</action>
    <action>temp3 =maincpu.pd@0C52FE8C</action>
    <action>temp4 =maincpu.pd@0C534A78</action>
    <action>temp5 =maincpu.pd@0C534A88</action>
    <action>temp6 =maincpu.pd@0C534A9C</action>
    <action>temp7 =maincpu.pd@0C534A70</action>
    <action>temp8 =maincpu.pd@0C5377C4</action>
    <action>temp9 =maincpu.pd@0C5377D4</action>
    <action>temp10=maincpu.pd@0C5377E8</action>
    <action>temp11=maincpu.pd@0C5377BC</action>
    <action>temp12=maincpu.pd@0C53B7C0</action>
    <action>temp13=maincpu.pd@0C53B7D0</action>
    <action>temp14=maincpu.pd@0C53B7E4</action>
    <action>temp15=maincpu.pd@0C53B7B8</action>
    <action>temp16=maincpu.pd@0C53F600</action>
  </script>
  <script state="run">
    <action>maincpu.pd@0C52DF08=00078140</action>
    <action>maincpu.pd@0C52FE94=00083680</action>
    <action>maincpu.pd@0C52FEA4=00000000</action>
    <action>maincpu.pd@0C52FE8C=00000001</action>
    <action>maincpu.pd@0C534A78=00091AE0</action>
    <action>maincpu.pd@0C534A88=00000000</action>
    <action>maincpu.pd@0C534A9C=00000000</action>
    <action>maincpu.pd@0C534A70=00000002</action>
    <action>maincpu.pd@0C5377C4=000A2380</action>
    <action>maincpu.pd@0C5377D4=00000000</action>
    <action>maincpu.pd@0C5377E8=00000000</action>
    <action>maincpu.pd@0C5377BC=00000002</action>
    <action>maincpu.pd@0C53B7C0=00100780</action>
    <action>maincpu.pd@0C53B7D0=00000000</action>
    <action>maincpu.pd@0C53B7E4=00000000</action>
    <action>maincpu.pd@0C53B7B8=00000002</action>
    <action>maincpu.pd@0C53F600=00122E80</action>
  </script>
  <script state="off">
    <action>maincpu.pd@0C52DF08=temp0 </action>
    <action>maincpu.pd@0C52FE94=temp1 </action>
    <action>maincpu.pd@0C52FEA4=temp2 </action>
    <action>maincpu.pd@0C52FE8C=temp3 </action>
    <action>maincpu.pd@0C534A78=temp4 </action>
    <action>maincpu.pd@0C534A88=temp5 </action>
    <action>maincpu.pd@0C534A9C=temp6 </action>
    <action>maincpu.pd@0C534A70=temp7 </action>
    <action>maincpu.pd@0C5377C4=temp8 </action>
    <action>maincpu.pd@0C5377D4=temp9 </action>
    <action>maincpu.pd@0C5377E8=temp10</action>
    <action>maincpu.pd@0C5377BC=temp11</action>
    <action>maincpu.pd@0C53B7C0=temp12</action>
    <action>maincpu.pd@0C53B7D0=temp13</action>
    <action>maincpu.pd@0C53B7E4=temp14</action>
    <action>maincpu.pd@0C53B7B8=temp15</action>
    <action>maincpu.pd@0C53F600=temp16</action>
  </script>
</cheat>

```

**Final words:** As an introduction, 7 cheats found and installed. It seems this game has more extra features and hidden content, they can be examined later, using same methods explained here. Also, adapting these cheats for previous revision [ddpdfk10] needed, but revision history should be checked, there might be some features missing for previous release.



## [vendetta] Vendetta (C) 1991 KONAMI



“Evil is lurking in Dead End City. In a turf war, the Dead End Gang has kidnapped Kute Kate from the rivals, the Cobras. The Cobras set out to save her and stop the expansion of the Dead End Gang in Vendetta.

The Dead End Gang is looking to expand their territory, so the Cobras must infiltrate every area around the city. First, they must gain entrance to Dead End City. Relying on their fighting skills, the Cobras face a multitude of enemies. ... Each player character resembles a real-life personality. Hawk looks like 'Hulk Hogan', Sledge looks like 'Mr. T', Boomer looks like 'Jean-Claude Van Damme' and Blood looks like 'Mike Tyson'.”

#ci&cn #non-linearlevel #detectautomate  
#endsequence

Goal: To find “Start from Stage” cheat, also, if player holds a weapon and then gets hit by enemies afterwards, weapon is dropped on floor and vanishes in time, examine routines to avoid vanishing. See Endsequence cheat may also be investigated.

Known facts: Stage numbers can be linear, as there are 5 main stages + sixth stage where all bosses appear. But, as “Start from Stage” cheat is missing in cheat database, maybe it's a bit complex. Weapon vanishing can be investigated with built-in MAME debugger cheat search, as that vanishing progress period seems constant. For Endsequence cheat, at first “Start from Stage” should be finalized and then extra parameters need to be collected.

OK, starting with stage value analysis, so after starting game, standart **ci** and **cn +,1** method applied. After you reach Stage 3, cheat is found as 2848 (Start 01 – Current 03). Time to check this memory location, after reset, checking write operations with **wp 2848,1,w** and it will stop twice writing 0x01 to 2848, when we manually poke 0x03 into 2848 twice, what we notice is, only blinking “Stage 1” text became “Stage 3” on the screen, when stage started. So, 2848 probably only keeps stage number to be printed on screen. So, there should be another byte responsible of current stage, enemies of that stage, shortly, plot of each stage. This info can help us, while playing Stage 1, once, player enters through a door and starts inside of the building, these also happen in each stages, so there are sub-stages, that's why, very first start of each stages (1 to 6) may have non-linear values. Then, it's time to start a new game and start a new cheat search with **ci** and **cn +** method this time, **cn +** after each stage starts (1 to 6 – not sub-stages) and result is, 2846 (Start 00 – Current 0B) when we reach Stage 4. This is worth a try, this time, after reset, **wp 2846,1,w** and before character select screen appears, it's triggered. Let's manually poke 0x0B into 2846 with MAME debugger monitor window and see the result: Blinking “Stage 1” appeared at first, but game directly started from Stage 4! As a result, two memory locations found, one is linear (2848) and responsible for stage number display, other one is non-linear (2846) and responsible for real stage data. Below table gathered by playing all stages and noting down values from 2846 and 2848:

Stage 1:	2846 is 00	2848 is 00	Stage 4:	2846 is 0B	2848 is 03
Stage 2:	2846 is 03	2848 is 01	Stage 5:	2846 is 0D	2848 is 04
Stage 3:	2846 is 07	2848 is 02	Stage 6:	2846 is 10	2848 is 05

To finalize this cheat, some more research needed, as game starts and no coins are inserted, demo mode starts after a minute, from Stage 1, with “GAME OVER” on screen to indicate it's a demo game. So, directly poking into 2846 and 2848 when their values are 00 probably won't work because when demo game starts, conditions are the same, demo game will also start from what's selected in cheat menu. So, real game specific parameters need to be found. This may help:

While examining memory, from 4A00 and forward, it changes when current screen changes, so kind of tiles or graphic data is displayed there. So, taking two bytes from there and using as conditions will help, so when demo game starts, 4A00 and forward memory will be different so cheat won't be applied. Choosing two bytes as:

4A50 = 0x25    and    4A6A = 0xE6    (when character select screen is displayed)

So, on next page final cheat can be examined. These two memory locations are used for conditions, all 6 stage parameters are defined from 2846 non-linear values, and when this condition is true, cheat pokes proper value into 2846. Last step, checks condition again with (param) and pokes linear value into 2848. Also tested when demo game starts, does not affect demo game at all.

```

<cheat desc="Select Starting Stage">
  <comment>This cheat must be turned on before main character select
  screen.</comment>
  <parameter>
    <item value="0x00">Stage 1</item>
    <item value="0x03">Stage 2</item>
    <item value="0x07">Stage 3</item>
    <item value="0x0B">Stage 4</item>
    <item value="0x0D">Stage 5</item>
    <item value="0x10">Stage 6</item>
  </parameter>
  <script state="run">
    <action condition="(maincpu.pb@4A50 == 25) AND
    (maincpu.pb@4A6A == E6)">maincpu.pb@2846=param</action>
    <action condition="(maincpu.pb@4A50 == 25) AND
    (maincpu.pb@4A6A == E6) AND (param == 01)">maincpu.pb@2848=00</action>
    <action condition="(maincpu.pb@4A50 == 25) AND
    (maincpu.pb@4A6A == E6) AND (param == 03)">maincpu.pb@2848=01</action>
    <action condition="(maincpu.pb@4A50 == 25) AND
    (maincpu.pb@4A6A == E6) AND (param == 07)">maincpu.pb@2848=02</action>
    <action condition="(maincpu.pb@4A50 == 25) AND
    (maincpu.pb@4A6A == E6) AND (param == 0B)">maincpu.pb@2848=03</action>
    <action condition="(maincpu.pb@4A50 == 25) AND
    (maincpu.pb@4A6A == E6) AND (param == 0D)">maincpu.pb@2848=04</action>
    <action condition="(maincpu.pb@4A50 == 25) AND
    (maincpu.pb@4A6A == E6) AND (param == 10)">maincpu.pb@2848=05</action>
  </script>
</cheat>

```

Next cheat to be found is preventing weapons vanish in time. While holding a weapon, allowing opponents to hit us, and when weapon falls onto the floor, **ci** and **cn** - method: Around 6 cheats found (These may vary because they're mostly variable):

2E0F – 3286 – 328F – 32EF – 340F

We need to monitor their values, where they reach zero, and on screen, weapon should be completely vanished same time. Above, all of them loops, it means, decreases to zero and becomes 0xFF and decreases again. So they're eliminated. But when we examine 3286, it started from 0x12 and ended with 0x00 when we investigate to the end. Afterwards, weapon is vanished. So, routine which causes 3286 value decrease needs to be found, to avoid vanishing. Here is the routine, found by **wp 3286,1,w**

1106: 12 24 F6	lda	#\$-0a,x	;x=3290, so loads A register with value of 3286
1109: 73 28	beq	\$1133 (40)	;if it's zero, branch forward
110B: 13 C4 3A	ldb	\$3a	;load B register with 283A, it's 0x4B now
110E: 29 0F	bitb	#\$0f	;bit B register with 0x0F value
1110: 63 01	bne	\$1113 (1)	;if not equal to zero, omit below & branch 1113
1112: 8C	deca		;decrease A register's value by one
1113: 3A 24 F6	sta	#\$-0a,x	;store it back to 3286

So, this "deca" instruction needs to be changed with NOP to avoid countdown.

1112: 8C	deca	;decrease A register's value by one
1112: <b>AE</b>	nop	

But, if you try to install this cheat with <action>maincpu.mb@1112=AE</action>, it won't work because, this game uses updated code, it means, main ROM area is updated/changed several times with different routines. Here is the proof: **bp 1112** in MAME debugger, you'll notice the code at 1112 is periodically changing (at least two different routines are present), when debugging breaks at 1112 address. So, here is the solution: Adding "4" before 1112 address. After testing the cheat, we notice that this countdown is always disabled and dropped weapons do not vanish, also this does not cause side-effects.

```

<cheat desc="Weapons do not disappear">
  <comment>Weapons on the floor never vanish.</comment>
  <script state="on">
    <action>temp0 =maincpu.mb@41112</action>
  </script>
  <script state="run">
    <action>maincpu.mb@41112=AE</action>
  </script>
  <script state="off">
    <action>maincpu.mb@41112=temp0 </action>
  </script>
</cheat>

```

Now, as "Select Starting Stage" is found, we may take a chance, for jumping to End Sequence. For this purpose, this info may help.

As valid for some games, there are special points in game, where an automatic progress is executed, as player's joystick controls might be disabled by code and automatic progress starts with automatic player movement, e.g. to enter somewhere and proceed to next section. These are of course decided with conditional checks, for example, checking if all enemies are dead in current section, or scrolling has ended and end of level is reached. If these parameters can be found, by changing one or a few memory locations, this automated progress can be forced, same time, level/stage parameters can be changed and skipping current level/stage or even directly jumping to end sequence will be possible.

As this game already has this option, at Stage 1, after defeating first boss, players automatically enter through the door and next section begins, we need to investigate exactly that point. If we can learn how it's triggered, we can manually trigger it with the cheat, same time, changing current stage parameters inside same cheat and we can directly jump to end sequence.

So, at Stage 1, we reach that first boss (masked creature with green pants) and when boss has little energy, taking a snapshot with Shift + F7. With some luck, you can catch this value:

- When we're controlling our player, **ci ub,2800,100** to initialize cheats for 2800 – 2900 area.
- After we kill the boss, and our movement is CPU controlled; **cn +**
- When new section starts, we're controlling our player again; **cn -**

Found address is 2855. It becomes 0x01 when current section is cleared, automated progress starts and its value is 0x00 when new section starts. Let's check with snapshot, load that snapshot with F7, when normal game play is active, poke 0x01 into 2855. You'll notice that section will automatically be cleared!

But we need to find some more conditions to finalize this cheat, because it should only work in-game, should not be executed in other screens as character select, demo game etc. So, 28XX region is checked with monitor, in all conditions; normal game, demo game, character select... Here are special memory locations:

- 2803 (Its value is 0x02 only for normal game; otherwise its value is different.)  
 283C (Its value is 0x01 for normal game, in demo mode it's 0x00.)

```

<cheat desc="See End Sequence Now!">
  <comment>Set this cheat ingame, during play.</comment>
  <script state="on">
    <action condition="(maincpu.pb@283C == 01) AND (maincpu.pb@2855 == 00)
    AND (maincpu.pb@2803 == 02)">temp0=01</action>
    <action condition="(temp0 == 01)">maincpu.pb@2855=01</action>
    <action condition="(temp0 == 01)">maincpu.pb@2846=10</action>
  </script>
</cheat>

```

Above is the constructed cheat with those condition checks. It's a Now! (one hit) cheat. If 283C is 0x01 and 2855 is 0x00 (player controlled) and 2803 is 0x02, when executed, temp0 variable is set to 1. And according to it is 1, it means conditions are met, end sequence is ready to be executed with poking 0x01 into 2855 (auto movement – current section finished) and 0x10 into 2846 (last Stage's value).

Final words: After detailed testing all cheats above, applying cheats to clones [vendettaj], [vendetta2pd], [vendetta2pu], [vendetta2p] and [vendettar] will be the last step.

## [mk2] Mortal Kombat II (C) 1993 MIDWAY



"Mortal Kombat 2 is a direct sequel to the 1992 original featuring new fighters (increasing the character roster from 7 to 12) and improved graphics. As with its predecessor, MKII's matches are divided into rounds, with victory awarded to the first player to win two rounds. At the point of defeat, the losing fighter will become dazed and the winner is given the opportunity to execute a 'finishing move'. Some of the standard fighting moves (moves shared by all characters) have now been expanded or enhanced."

#easteregg #adaptexisting

Goal: As requested, investigate these 3 issues:

- From [http://mortalkombat.wikia.com/wiki/Mortal\\_Kombat\\_II](http://mortalkombat.wikia.com/wiki/Mortal_Kombat_II):

"After landing a strong uppercut against the opponent, the face of sound designer Dan Forden would appear in the lower-right corner of the screen and shout, "Toasty!" ... In the Portal stage, if the player very quickly held down and hit the start button before Dan's head left the screen, they would then instantly begin a new stage against a secret character named Smoke, a grey recolor of Scorpion."

So, we'll investigate the code and try to find an easy way to display Dan Forden's face and start fighting VS Smoke afterwards.

- Johnny Cage rarely performs a red shadow kick; we can find the code responsible for it and cause Jonny Cage always performing a red shadow kick.

- From same website above,

"On the Kombat Tomb stage, if the player holds down on both joysticks immediately after knocking the victim into the spikes, the victim will gradually slide down the spikes."

We can find the code and automate it, knocked opponent will automatically slip off the spikes to the ground, instead of manually holding down both joysticks.

Known facts: It seems fighting VS Smoke has certain conditions to be fulfilled, so current [mk2] cheats, especially RAM cheats can be examined and starting with monitoring those addresses might be a good start.

First one, Smoke battle analysis, it's mentioned that current stage should be Portal Stage. This is from [mk2] select Kombat Zone cheat:

```
<action>maincpu.pb@1061AC0=param</action> ;0x07 for Portal
```

So, let's start a new 1UP game and check 1061AC0 with **wp 1061ac0,8,r**

(Taking width as 8 because each byte in TMS34010 CPU start from xxxxx00, xxxxx08 etc.)

It's good that debugging stops at this watchpoint, after uppercutting opponents. But which routine is the one we're looking for? If we go on searching, execution of code stops at FF8DF040. Also we notice in MAME window, next instruction is `cmpi 7h, A0` (compare with 0x07) so it's probably the routine we're looking for. Here is disassembly:

FF8DEFB0:	05A0 01D0 0106	MOVE	@10601D0h,A0,0	
FF8DEFE0:	0B40 FFFE	CMPI	1h,A0	
FF8DF000:	CA27	JREQ	FF8DF280h	
FF8DF010:	05A0 1AC0 0106	MOVE	@1061AC0h,A0,0	;get current stage(1061AC0)&poke into A0
FF8DF040:	0B40 FFF8	CMPI	7h,A0	;is it 0x07 Portal?
FF8DF060:	CB06	JRNE	FF8DF0D0h	;if not equal to 0x07 branch FF8DF0D0
FF8DF070:	0D5F E740 FF81	CALLA	FF81E740h	;call this routine
FF8DF0A0:	0B40 FFFD	CMPI	2h,A0	;after return, compare A0 with 0x02
FF8DF0C0:	CB08	JRNE	FF8DF150h	;if A0 != 0x02, branch further
FF8DF0D0:	09C0 0028	MOVI	28h,A0	;poke 0x28 into A0
FF8DF0F0:	0D5F EC20 FF80	CALLA	FF80EC20h	;call this routine
FF8DF120:	C880 BA10 FFA0	JAC	FFA0BA10h	;if Carry flag is 1, jump FFA0BA10
FF8DF150:	05A0 1AC0 0106	MOVE	@1061AC0h,A0,0	;when PC reaches here = nothing special

As we examined the code, the routine starting a fight VS Smoke should be at FFA0BA10. Here is another routine called from above: (We can manually poke 0x07 into 1061AC0 and apply **bp ff8df070** to trace the subroutines called after that.)



```

FF81E740: 5600          CLR      A0          ;clear A0 register
FF81E750: 05A1 01F0 0106 MOVE    @10601F0h,A1,0 ;get value of 106101F0, store into A1
FF81E780: 0D3F 0003      CALLR   FF81E7D0h      ;call routine below
FF81E7A0: 05A1 0DC0 0106 MOVE    @1060DC0h,A1,0 ;get value of 1060DC0, store into A1
FF81E7D0: 0B41 FFFE      CMPI    1h,A1      ;is it 0x01?
FF81E7F0: CB01          JRNE    FF81E810h      ;if not 0x01, return
FF81E800: 1020          INC      A0          ;increase A0 register
FF81E810: 0960          RETS      ;return back

```

Everything is clear now. Above routine should return with A0=0x02. For that, both 10601F0 and 1060DC0 should be 0x01. 10601F0 is already 0x01, but when we check 1060DC0, it's 0x00. It needs to be investigated. Here is interesting result:

1060DC0 is 0x00 for an active 1UP game, 0x01 for an active 2UP game. So, by examining game code, we also had this valuable info, it seems, fighting VS Smoke is only available in a 2UP game!

So, let's check main code on previous page, apply manual poke (0x07 into 1061AC0) and start a 2UP game from now on, and try **bp ffa0ba10** to monitor execution of this routine. It will take much time, when PC reaches there, guess what? Stopped at breakpoint, and then immediately Dan's face appeared! So, thinking of codewise, the subroutine at FF80EC20 is probably a randomizer routine or kind of minimizing chance routine, when it's executed, it's like rolling dices and waiting for 6:6 to appear! So, if we directly jump to FFA0BA10, we'll always notice Dan's face after an uppercut. So, original code is:

```

FF8DEFB0: 05A0 01D0 0106 MOVE    @10601D0h,A0,0
FF8DEFE0: 0B40 FFFE      CMPI    1h,A0
FF8DF000: CA27          JREQ     FF8DF280h
FF8DF010: 05A0 1AC0 0106 MOVE    @1061AC0h,A0,0 ;get current stage(1061AC0)&poke into A0
FF8DF040: 0B40 FFF8      CMPI    7h,A0      ;is it 0x07 Portal?
FF8DF060: CB06          JRNE    FF8DF0D0h      ;if not equal to 0x07 branch FF8DF0D0
FF8DF070: 0D5F E740 FF81 CALLA   FF81E740h      ;call this routine
FF8DF0A0: 0B40 FFFD      CMPI    2h,A0      ;after return, compare A0 with 0x02
FF8DF0C0: CB08          JRNE    FF8DF150h      ;if A0 != 0x02, branch further
FF8DF0D0: 09C0 0028      MOVI    28h,A0      ;poke 0x28 into A0.
FF8DF0F0: 0D5F EC20 FF80 CALLA   FF80EC20h      ;call this routine
FF8DF120: C880 BA10 FFA0 JAC      FFA0BA10h ;if Carry flag is 1, jump FFA0BA10

```

Cheat applied code:

```

FF8DF000: C011          JR      FF8DF120h      ;direct jump applied to FF8DF120 so
...                                     ;stage and 2UP game checks bypassed
FF8DF120: C080 BA10 FFA0 JA      FFA0BA10h ;conditional jump removed, direct jump

```

So, let's check the cheat as it is, when we temporarily check this, we'll notice that, Dan Forden's face is appearing every time with an uppercut, but immediately holding joystick down + pressing Start button afterwards, seems not responding, Smoke fight never starts. There might be another check? Let's see.

To install above temporarily, **bp FF8DEFB0,1,{maincpu.mw@1be00=c011;maincpu.mb@1be25=c0;g}**

We start a new 2UP game, and making sure that current stage is not the Portal stage, above command in debugger and then **trace <filename>** to start tracing, perform an uppercut, when Dan Forden's face appears, immediately joystick down + Start button, and **trace off** when all tasks are done. If we open trace file and search "cmpi 7h" text, to find where 0x07 is compared (with current stage) and here is part of code:

```

FFA1A110: 05A0 1AC0 0106 MOVE    @1061AC0h,A0,0 ;get current stage(1061AC0)&poke into A0
FFA1A140: 0B40 FFF8      CMPI    7h,A0      ;is it 0x07 Portal?
FFA1A160: CB00 0247      JRNE    FFA1C5F0h      ;if not equal to 0x07 branch end

```

So, there's another stage check just after Dan's face appears. We need to remove this condition check also:

```

FFA1A160: 09C0 0247      MOVI    247h,A0      ;changed JRNE with null MOVI so,
                                     ;branch never happens to the end

```

So, cheat is now finalized by changing 3 values:



```

<cheat desc="Fight VS Smoke easily">
  <comment>After an uppercut, Dan Forden's face will always appear, press
  DOWN+START to fight VS Smoke.</comment>
  <script state="on">
    <action>temp0=maincpu.mw@001BE00</action>
    <action>temp1=maincpu.mb@001BE25</action>
    <action>temp2=maincpu.mw@004342C</action>
  </script>
  <script state="run">
    <action>maincpu.mw@001BE00=C011</action>
    <action>maincpu.mb@001BE25=C0</action>
    <action>maincpu.mw@004342C=09C0</action>
  </script>
  <script state="off">
    <action>maincpu.mw@001BE00=temp0</action>
    <action>maincpu.mb@001BE25=temp1</action>
    <action>maincpu.mw@004342C=temp2</action>
  </script>
</cheat>

```

Now, second cheat search, about Johnny Cage's red shadow kick. We found that obstructive routine at FF80EC20 for Smoke cheat, maybe this same routine is executed to decide Johnny Cage's shadow kick (default in green color or very rarely, red)? So, it's time to start a new 2UP game for test simplicity, and selecting Johnny Cage as Player 1. **bp ff80ec20** command and performing shadow kick with Back, Forward, Low Kick. Yes, stopped at breakpoint, but we're inside of this subroutine. To reach main routine where this is called, pressing Shift + F11 to step out main routine. It's displayed on main debugger window now, as:

FF88F760: 56E7	CLR	A7	
FF88F770: 09C0 000A	MOVI	Ah,A0	
FF88F790: 0D5F EC20 FF80	CALLA	FF80EC20h	;obstructive routine
FF88F7C0: C903	JRNC	FF88F800h	;if Carry flag is 0, branch FF88F800.
FF88F7D0: 09C5 00C7	MOVI	C7h,A5	

We notice that, if FF88F800 is reached, it means it's always default yellow shadow kick. So, this JRNC should be replaced with NOP, to remove that branch and allow code to go on from FF88F7D0 to perform a red shadow kick every time:

```
FF88F7C0: 0300          NOP
```

This was easy, as we knew common routine and guessed it's executed also here.

```

<cheat desc="Johnny Cage always performs Red Shadow Kick">
  <script state="on">
    <action>temp0=maincpu.mw@0011EF8</action>
  </script>
  <script state="run">
    <action>maincpu.mw@0011EF8=0300</action>
  </script>
  <script state="off">
    <action>maincpu.mw@0011EF8=temp0</action>
  </script>
</cheat>

```

For last cheat investigation, stage fatality in Kombat Tomb Stage and holding joystick down for auto-fall from spikes, let's start a new 1UP game, we can use existing cheat to start from that stage, and take a snapshot at final round, when CPU opponent is about to die. OK, everything is ready, so we can kill CPU opponent and apply Stage Fatality. When CPU opponent hit the spikes, immediately, holding joystick down and waiting... No luck, auto-fall didn't happen. But after we load the snapshot and repeat these steps, after several tries, auto-fall from spikes happens! Maybe same obstructive routine is preventing auto-fall? It's worth a try. So, loading snapshot again, creating a breakpoint with **bp ff80ec20** and waiting it to be triggered. It may be triggered a few times, but most important moment is, when CPU opponent is on spikes. When this happens, let's check by pressing Shift + F11 to find from where this routine is called, it's FF8EB170. Here is the routine:

FF8EB150: 09C0 0032	MOVI	32h,A0	
FF8EB170: 0D5F EC20 FF80	CALLA	FF80EC20h	;obstructive routine called from here
FF8EB1A0: C925	JRNC	FF8EB400h	;if Carry flag is not 1, branch FF8EB400
FF8EB1B0: 09C0 0080	MOVI	80h,A0	

For final step, we press F11 (single step) when PC is at FF8EB1A0, it seems mostly Carry flag is 0 there and it jumps to FF8EB400. Now it's time to test what's in our mind: When PC is at FF8EB170, just before executing obstructive subroutine, poking 0x0300 (NOP) into FF8EB1A0 so that branch won't happen. Like this (we can remove previous breakpoint):

`bp ff8eb170,1,{maincpu.mw@1d634=0300:g}`

Yes, after Stage Fatality is performed in Kombat Tomb stage, knocked CPU opponent auto-falls from spikes!

So, above is original code. Cheat applied code is:

FF8EB1A0: 0300	NOP	;No Operation
----------------	-----	---------------

```
<cheat desc="Kombat Tomb stage auto fall from spikes">
  <comment>After performing a stage fatality in Kombat Tomb, knocked opponent
  will slip off the spikes to the ground.</comment>
  <script state="on">
    <action>temp0=maincpu.mw@001D634</action>
  </script>
  <script state="run">
    <action>maincpu.mw@001D634=0300</action>
  </script>
  <script state="off">
    <action>maincpu.mw@001D634=temp0</action>
  </script>
</cheat>
```

Final words: As it seems we've completed our task, it's time to test all these cheats briefly, in all game modes; after these tests, we're sure that for parent [mk2], these cheats are working perfectly. Now it's time to check revision history from history.dat:

"REV. 2.1 :

Smoke and Jade were added.

The Pit II/Kombat Tomb fatality was added."

According to this info, we won't apply Smoke and Kombat Tomb stage fatality cheat for [mk2r11], [mk2r14] and [mk2r20] as they are previous revisions, there's no support.

For last step, as fighting VS Smoke conditions were collected by examining code (it must be a 2UP game, otherwise; even in the Portal stage, holding joystick down and pressing Start button won't result in fighting VS Smoke), this info needs to be contributed, so, I again contacted authors of <http://www.arcade-history.com> and currently, this is in TIPS and TRICKS section of history.dat, for [mk2]:

"\* Fight against Smoke : in a 2UP game, press Down+Start when Dan says 'Toasty' on The Portal stage."

Last important note is, as you realized, this FF80EC20 "obstruction" routine in Mortal Kombat 2 has so much potential, probably there are many rare occasions, secrets are waiting to be revealed.

## [fatfursp] Fatal Fury Special (C) 1993 SNK



“Garou Densetsu Special was released in September 1993 in Japan.  
The title of this game translates from Japanese as 'Legend of Hungry Wolf Special'.  
This game is known outside Japan as 'Fatal Fury Special'.  
Things that changed in this particular game : \* The combo system has been improved, hit damage has been lessened & the game speed has been increased. \*  
Newcomers in the game are Duck King, Tung Fu Rue, Geese Howard & Ryo Sakazaki.”

#reverseengineer #adaptexisting #movementcodes  
#endsequence

Goal: As requested, explore player movement system (moves list) and exchange current “hard to perform” special moves with an “easy to perform” one, for all characters. Also, end sequence can be investigated?

Known facts: Fatal Fury 3 game has “Always have Easy Supers” cheat.

So, assuming that these “Fatal Fury” series used similar code in each game, best method right now is starting from [fatfury3] Easy Supers cheat. Here is cheat for [fatfury3]:

```
<action>maincpu.pb@10E001=01</action>
```

So much simple, we expected a huge list modifying all character’s super moves.

Probably, this is a test-byte, in some games; coders insert test bytes to perform some in-game operations directly, by controlling if a test byte is set or not. Test byte can execute a macro, a complex routine or combined routines, or change some settings. We all know that some of arcade games have DIP switches, right? As they’re hardware, of course their values (as bits or bytes) are also controlled from game’s RAM or ROM area. So, test bytes can be categorized as “Software DIP switches”.

So, after running [fatfury3], let’s start with **wp 10e001,1,r**

```
0839 000X 0010 E001      btst    #$X, $10e001.1      ;test xth bit of 10E001
082D 000X 6001          btst    #$X, ($6001,A5)      ;test xth bit of 10E001
```

As all stops are examined and it seems bit 0 is not checked, (we need to find the code checking rightmost bit, first bit) searching all code may help as:

```
find 0,80000,w.0839,0000,0010,e001      ;search above first example, for bit 0.
find 0,80000,w.082d,0000,6001          ;search above second example, for bit 0.
```

Found at 3164E and 44670. (When cheat is on, **bp c7b0** applied to get those values.)  
After examining both, the one at 44670 seems special one, can describe how movements are changed.

```
04466A: 47F9 0005 83DC      lea     $583dc.1, A3      ;A3=583DC
044670: 0839 0000 0010 E001    btst    #$0, $10e001.1   ;check bit 0 of 10E001
044678: 6700 0008              beq     $44682            ;if it's zero, go 44682
04467C: 47F9 0005 8414      lea     $58414.1, A3      ;bit 0 is 1. A3=58414
...
04468C: D7C1                  adda.l  D1, A3            ;A3 is read and written
04468E: 2653                  movea.l (A3), A3          ;afterwards
044690: 301B                  move.w  (A3)+, D0
```

It’s getting clear. If cheat is off, A3 doesn’t change, stays as 583DC. But if cheat is on, A3 value is changed and new value is 58414. Probably, these are start address of movement tables. Let’s check.

```
0583DC: 0000 0000 0005 844C 0005 84E0 0005 8574
0583EC: 0005 8600 0005 868C 0005 8718 0005 87C4
0583FC: 0005 885C 0005 88E0 0005 898C 0005 8A08
05840C: 0005 8A54 0005 8AC0
```

```

058414: 0000 0000 0005 8492 0005 8526 0005 85B6
058424: 0005 8642 0005 86CE 0005 876A 0005 880E
058434: 0005 889A 0005 8932 0005 89C6 0005 8A2A
058444: 0005 8A86 0005 8AFA

```

It seems they point some memory locations, 13 values for each. There might be 13 characters total? OK, we can learn it later. Let's pick one from list and compare data (chosen ones are in red color):

```

05844C: 0011 0005 8C4E 0005 8C60 0005 8C72 ;all normal tables start with 0011
058492: 0013 0005 8C4E 0005 8C60 0005 8C72 ;all cheat enabled tables start with 0013

```

Let's dump data starting from 58C4E, as they're static for both:

```

058C4E: 0000 0F02 000C 0F08 0C0F 040C F011 FF0B 0611
058C60: 0000 0F02 000C 0F08 0C0E 040C F00E FF0C 060E

```

Now it seems 0011 and 0013 may point number of entries on that table, cheat on one has 2 more tables, let's check what is added (We open memory window and compare 5844C and forward with 58492 and forward, here are +2 added entries):

```

0584C6: 0005 8C0E 0005 8C3A

```

Here are those tables, starting from 58C0E and 58C3A:

```

058C0E: 0000 0F00 000A 0F02 0A0F 000A 0F02 0CF0
058C1E: 09FF 0906 0000 0F03 000A 0F06 0A0F 020A
058C2E: 0F08 0A0E 040C F012 FF0A 0612

058C3A: 0000 0F00 000A 0F02 0A0F 000A 0F02 10F0
058C4A: 0BFF 0A06 0000 0F02 000C 0F08 0C0F 040C
058C5A: F011 FF0B 0611 0000 0F02 000C

```

Now let's check how they are read, as both start from 58C4E, **wp 58c4e,1,r** command: It stopped several times at "How to play" screen. Let's disable it and choose another fighter for PL1 (other than Terry) and check: If "any" player is not Terry, it's not triggered. So, this should be movement table for Terry. (As you noticed, Terry is default player in this game.)

Finally, checking table again, starting from 0011, now full table (no cheat):

```

05844C: 0011 0005 8C4E 0005 8C60 0005 8C72 0005 ;it seems this table is for movements of
05845C: 8C84 0005 8B66 0005 8B78 0005 8B8A 0005 ;first character, Terry.
05846C: 8B9C 0005 8BAE 0005 8BC0 0005 8BD2 0005
05847C: 8BE4 0005 8BF6 0005 8C22 0005 8B3C 0005
05848C: 8B4A 0005 8B58 0013 ;until 0013

```

So, Terry's movement data starts from 58B4A and ends just after 58C84 (mixed data on table). As first brief check of all those tables, we notice that, at the end of each table, there's a \$FF byte followed by another byte. That can be movement number, and \$FF probably points end of that movement, so here are two examples:

```

058B66: 0000 0F02 000A 0F08 0A0F 040A F009 FF01 ;first entry, $FF and $01.
058C78: 0000 0F02 000A 0F08 0A0F 040A F00B FF02 ;second entry, $FF and $02.

```

There are \$FF bytes on each, which might point end of command, we can assume these tables are read, but as they're sequential, if a movement is successfully performed, the counter should increase and read next byte, until that \$FF probably. First table is very small, maybe represents first special move of Terry? Let's check:

\$FF byte of first entry is at 58B74, so watchpoint for there: **wp 58b74,1,r**

Starting a new game with Terry as 1UP player, while this watchpoint is active, when we perform "Burn Knuckle" with D, DB, B and Punch button A, it's triggered. So this is first movement on the table.

\$FF byte of second entry is 58B86, so watchpoint is: **wp 58b86,1,r**

As second entry is same as first, except that F00B word, it should be same movement with another button probably. Yes, it's "Strong Burn Knuckle" with D, DB, B and Punch button C, and triggered.

So, checking \$FF byte for each character's entire movement tables, setting a watchpoint for that byte and performing their special moves one by one (in a 2UP game preferred, without annoyance) will result in finding all equivalents of joystick special movements in ROM memory.

Now we can investigate same movement tables for [fatfursp]. But our first aim is, checking if a test-byte as in [fatfury3] exists here, then, it will save us lots of work.

So, starting [fatfursp] and performing these searches, after game starts (identical words from [fatfury3]):

```
find 0,100000,w.0839,0000,0010,e001 ;btst #$0,$10e001.l
find 0,100000,w.6700,0008,47f9 ;beq forward + lea
```

No luck. It seems there's no test-byte present. So we need to find movement tables for each character and replace "hard to perform" super moves with easy ones. To not to waste time, with **wp 58c4e,1,r** command in [fatfury3], it stopped every time where Terry was present on screen. Checking again, here is the code causes stop there:

```
044698: 6100 0012      bsr      $446ac
04469C: 5889          addq.l   #4, A1
04469E: 51C8 FFF4      dbra     D0, $44694
0446A2: 4E75          rts
0446A4: 0010 8AAA      ori.b    #$aa, (A0)
0446A8: 0010 8B22      ori.b    #$22, (A0)
0446AC: 7200          moveq    #$0, D1
0446AE: 321E          move.w   (A6)+, D1 ;stops here
0446B0: D241          add.w    D1, D1
0446B2: D241          add.w    D1, D1
0446B4: 41FA 000A      lea      ($a,PC), A0; ($446c0)
0446B8: D1C1          adda.l   D1, A0
```

So, starting a new game in [fatfursp], when game starts, search is:

```
find 0,100000,w.d241,d241,41fa ;add + add + lea
```

Found at several places. When examining them, with breakpoints, no proper stop happened.

```
find 0,100000,w.5889 ;add #4
```

Yes, found at 3 places but the one at 2E8B0 is similar to above:

```
02E8B0: 5889          addq.l   #4, A1
02E8B2: 51CE FFE0      dbra     D6, $2e894
02E8B6: 4E75          rts
02E8B8: 0010 D062      ori.b    #$62, (A0)
02E8BC: 0010 D08A      ori.b    #$8a, (A0)
02E8C0: 4E71          nop
02E8C2: 4E71          nop
02E8C4: 381A          move.w   (A2)+, D4 ;breakpoint should be here
02E8C6: 0C44 0002      cmpi.w   #$2, D4
02E8CA: 6700 0014      beq      $2e8e0
```

After game starts and reaches, **bp 2e8c4** to check result, it stops several times, A2 register needs to be checked because movements are read from that register. As Terry VS Geese is fighting in "How to Play" mode, here are values of A2 for each trigger:

Terry: 38B1A to 38BB8      Geese: 3925C to 392D4

So, let's investigate Terry's two movements:

```
038B1A: 0000 0F00 0000 0C0F 0406 0D00 0C0F 04FF ;this should be first movement, ends with
038B2A: FF00 ;FF00. It's (D, DB, B, Button A)
...
038BB8: 0000 0F02 0000 0C0F 080C 0F04 0C0F 080C ;this seems the last movement, ends with
038BC8: 0F03 08F0 0FFF 0979 000 ;FF09. Super (desperation) move perhaps?
```

So, as we checked Fatal Fury Special game movement list, and characters:

- There are total 16 characters. (Ryo Sakazaki is hidden character.)
- Each character has only one Super Move, called "Desperation Move", can only be performed when life gauge is flashing. (20% or less life remaining)

When we test above with **wp 38b28,1,r** and **wp 38bcd,1,r** watchpoints (end of movements \$FF byte), first one is triggered after performing D, DB, B and Button A. Second one is desperation move, for sure, it's triggered after performing D, DB, B, DB, F and Buttons B+C (Power Geyser move).



So, we need to find all movement tables for each character (we'll use existing cheat to fight VS Ryo to get his table) and note down all memory location start points for 0000 xxxx xxxx xxxx .... FF09 sequence, because these are desperation move start addresses.

Afterwards, we need to decide a "simple" movement to replace with all those "hard" desperation moves. After examining all moves list for each character, Wolfgang's default desperation move is Hold B, UF and A+C button, so we can adjust it and change to B, F and A+C and apply for all characters. Here is result:

```
0F04 0000 0C0F 030C F00E FF09 ;code for Back, Forward and A+C
```

So, we'll poke this sequence, 16 times for each character, to their exact movement locations (their FF09 entries will be replaced). Of course, their current FF09 table lengths are all checked, as we're inserting a shorter sequence with 12 bytes above, it won't result in overflow. For cheat in xml format, "run" state, this 12 bytes long data is divided into two temp variables (temp32 and temp33) and they're poked to exact location where desperation move code starts for all 16 characters. Further tests performed and all characters are tested, their normal movements are OK and all their desperation moves are same, Back, Forward and A+C. Example for Terry's movement table:

```
038BBA: 0F02 0000 0C0F 080C 0F04 0C0F 080C 0F03 ;default Terry desperation move
038BCA: 08F0 0FFF 0979 0000 ;ends with FF09.

038BBA: 0F04 0000 0C0F 030C F00E FF09 ;code for Back, Forward and A+C
```

```
<cheat desc="Always have Easy Supers" tempvariables="34">
  <comment>For all characters; Back, Forward, A+C performs Desperation
  Move.</comment>
  <script state="on">
    <action>temp0 =maincpu.mq@038BBA</action>
    <action>temp1 =maincpu.md@038BC2</action>
    <action>temp2 =maincpu.mq@038C52</action>
    <action>temp3 =maincpu.md@038C5A</action>
    <action>temp4 =maincpu.mq@038D00</action>
    <action>temp5 =maincpu.md@038D08</action>
    <action>temp6 =maincpu.mq@038D78</action>
    <action>temp7 =maincpu.md@038D80</action>
    <action>temp8 =maincpu.mq@038DFC</action>
    <action>temp9 =maincpu.md@038E04</action>
    <action>temp10=maincpu.mq@038E66</action>
    <action>temp11=maincpu.md@038E6E</action>
    <action>temp12=maincpu.mq@038EEC</action>
    <action>temp13=maincpu.md@038EF4</action>
    <action>temp14=maincpu.mq@038F80</action>
    <action>temp15=maincpu.md@038F88</action>
    <action>temp16=maincpu.mq@039014</action>
    <action>temp17=maincpu.md@03901C</action>
    <action>temp18=maincpu.mq@0390A6</action>
    <action>temp19=maincpu.md@0390AE</action>
    <action>temp20=maincpu.mq@039142</action>
    <action>temp21=maincpu.md@03914A</action>
    <action>temp22=maincpu.mq@0391CA</action>
    <action>temp23=maincpu.md@0391D2</action>
    <action>temp24=maincpu.mq@03923E</action>
    <action>temp25=maincpu.md@039246</action>
    <action>temp26=maincpu.mq@0392D6</action>
    <action>temp27=maincpu.md@0392DE</action>
    <action>temp28=maincpu.mq@039390</action>
    <action>temp29=maincpu.md@039398</action>
    <action>temp30=maincpu.mq@039430</action>
    <action>temp31=maincpu.md@039438</action>
  </script>
  <script state="run">
    <action>temp32=0F0400000C0F030C</action>
    <action>temp33=F00EFF09</action>
    <action>maincpu.mq@038BBA=temp32</action>
    <action>maincpu.md@038BC2=temp33</action>
```

```

<action>maincpu.mq@038C52=temp32</action>
<action>maincpu.md@038C5A=temp33</action>
<action>maincpu.mq@038D00=temp32</action>
<action>maincpu.md@038D08=temp33</action>
<action>maincpu.mq@038D78=temp32</action>
<action>maincpu.md@038D80=temp33</action>
<action>maincpu.mq@038DFC=temp32</action>
<action>maincpu.md@038E04=temp33</action>
<action>maincpu.mq@038E66=temp32</action>
<action>maincpu.md@038E6E=temp33</action>
<action>maincpu.mq@038EEC=temp32</action>
<action>maincpu.md@038EF4=temp33</action>
<action>maincpu.mq@038F80=temp32</action>
<action>maincpu.md@038F88=temp33</action>
<action>maincpu.mq@039014=temp32</action>
<action>maincpu.md@03901C=temp33</action>
<action>maincpu.mq@0390A6=temp32</action>
<action>maincpu.md@0390AE=temp33</action>
<action>maincpu.mq@039142=temp32</action>
<action>maincpu.md@03914A=temp33</action>
<action>maincpu.mq@0391CA=temp32</action>
<action>maincpu.md@0391D2=temp33</action>
<action>maincpu.mq@03923E=temp32</action>
<action>maincpu.md@039246=temp33</action>
<action>maincpu.mq@0392D6=temp32</action>
<action>maincpu.md@0392DE=temp33</action>
<action>maincpu.mq@039390=temp32</action>
<action>maincpu.md@039398=temp33</action>
<action>maincpu.mq@039430=temp32</action>
<action>maincpu.md@039438=temp33</action>
</script>
<script state="off">
  <action>maincpu.mq@038BBA=temp0 </action>
  <action>maincpu.md@038BC2=temp1 </action>
  <action>maincpu.mq@038C52=temp2 </action>
  <action>maincpu.md@038C5A=temp3 </action>
  <action>maincpu.mq@038D00=temp4 </action>
  <action>maincpu.md@038D08=temp5 </action>
  <action>maincpu.mq@038D78=temp6 </action>
  <action>maincpu.md@038D80=temp7 </action>
  <action>maincpu.mq@038DFC=temp8 </action>
  <action>maincpu.md@038E04=temp9 </action>
  <action>maincpu.mq@038E66=temp10</action>
  <action>maincpu.md@038E6E=temp11</action>
  <action>maincpu.mq@038EEC=temp12</action>
  <action>maincpu.md@038EF4=temp13</action>
  <action>maincpu.mq@038F80=temp14</action>
  <action>maincpu.md@038F88=temp15</action>
  <action>maincpu.mq@039014=temp16</action>
  <action>maincpu.md@03901C=temp17</action>
  <action>maincpu.mq@0390A6=temp18</action>
  <action>maincpu.md@0390AE=temp19</action>
  <action>maincpu.mq@039142=temp20</action>
  <action>maincpu.md@03914A=temp21</action>
  <action>maincpu.mq@0391CA=temp22</action>
  <action>maincpu.md@0391D2=temp23</action>
  <action>maincpu.mq@03923E=temp24</action>
  <action>maincpu.md@039246=temp25</action>
  <action>maincpu.mq@0392D6=temp26</action>
  <action>maincpu.md@0392DE=temp27</action>
  <action>maincpu.mq@039390=temp28</action>
  <action>maincpu.md@039398=temp29</action>
  <action>maincpu.mq@039430=temp30</action>
  <action>maincpu.md@039438=temp31</action>
</script>
</cheat>

```

Also, we can investigate another cheat, reaching end sequence immediately? While playing, we notice that, in cutscenes, stage number is mentioned like: "The 2nd stage".

So, we need to check first, how many stage numbers are present? Shortly explaining, with **ci & cn +,1** method, we find that 10B2F3 holds current stage number. When we manually poke and check, when it becomes 0x11, end sequence is displayed, end of all stages. Now, as this game also has an attract mode (demo game), we need to find special memory addresses to use as conditions. We can narrow search area with 1000XXX. That block with 0x1000 bytes also covers existing RAM cheat memory locations. First, 100010 address is found. Its value is 49D8 for in-game (no demo game, no in "How to play" screen, no in character select menu ...). Also, we find 100034 address, and after testing, it seems its value can be 0x01, 0x02 and 0x03 due to current game mode (PL1 and/or PL2 active in game or not).

And last step is, as we'll install this cheat with automation method (selected player's number of already won round value will be 1, selected player's energy will be set more than other player's, and remaining timer will be set to a few milliseconds). So, these will simulate "End of current Round". Finally, current round number will be poked the maximum value, we need to find where number of rounds won are stored. Again with **ci & cn +,1** (also **cn -,1**) method, 100058 is found. It holds number of rounds won by PL1. After finding equivalent memory locations for PL2 also, here is what we have:

100010	should be 49D8	(in-game value)
100034	can be 01 – 02 – 03	(to decide which player is in game, which one is out)
10049A	is PL1's energy	(from existing cheat)
10059A	is PL2's energy	(from existing cheat)
10092A	is remaining time	(from existing cheat - will be set to 0x0001, 10 ms)
100058	has number of rounds won (PL1)	(will be set to 1)
100059	has number of rounds won (PL2)	(will be set to 1)
10B2F3	is current round number	(will be set to 0x10, will be 0x11 automatically)

Here is the cheat, finalized with above info (Cheat can only be executed if 100034=0x49D8 and first or second bit of 100010 is 1, for PL1 and PL2 respectively):

```
<cheat desc="See Endsequence Now! PL1">
  <comment>Set it ingame, in a Round.</comment>
  <script state="on">
    <action condition="(maincpu.pw@100010 == 49D8) AND
      ((maincpu.pb@100034 BAND 01) == 01)">temp0=01</action>
    <action condition="(temp0 == 01)">maincpu.pb@10049A=01</action>
    <action condition="(temp0 == 01)">maincpu.pb@10059A=00</action>
    <action condition="(temp0 == 01)">maincpu.pw@10092A=0001</action>
    <action condition="(temp0 == 01)">maincpu.pb@100058=01</action>
    <action condition="(temp0 == 01)">maincpu.pb@10B2F3=10</action>
  </script>
</cheat>
```

```
<cheat desc="See Endsequence Now! PL2">
  <comment>Set it ingame, in a Round.</comment>
  <script state="on">
    <action condition="(maincpu.pw@100010 == 49D8) AND
      ((maincpu.pb@100034 BAND 02) == 02)">temp0=01</action>
    <action condition="(temp0 == 01)">maincpu.pb@10059A=01</action>
    <action condition="(temp0 == 01)">maincpu.pb@10049A=00</action>
    <action condition="(temp0 == 01)">maincpu.pw@10092A=0001</action>
    <action condition="(temp0 == 01)">maincpu.pb@100059=01</action>
    <action condition="(temp0 == 01)">maincpu.pb@10B2F3=10</action>
  </script>
</cheat>
```

Final words: Movements in these kinds of games can be retrieved by this method, also; if there's any existing movement cheats for a game, it can be reverse engineered and movement tables can be revealed. Also, see end sequence cheat is a good example of simulating end of current stage; this method can be applied to similar "Fighter" games, to access end sequence directly.

## [sharrier] Space Harrier (C) 1985 SEGA

Rapid fire cheat:

[#rapidfire](#)

- Started a trace just after firing, and trace file examined with getting unique instructions.
- Firing routine found at 7AF4. Routine examined, it seems code is checking 4th bit, if it's set, goes end.
- As 1st bit of same memory location is always 1, one byte changed and allowed code to check 1st bit.
- This way, rapid fire became always active!

```
007ABE: 5279 0012 40E0      addq.w  #1, $1240e0.1      ;1240E0 is increased by 1
007AC4: 0C79 0010 0012 40E0    cmpi.w  #$10, $1240e0.1    ;compare it with 0x10
007ACC: 6E00 0026              bgt     $7af4              ;if > 0x10, branch 7AF4

007AF4: 3039 0012 40E0      move.w  $1240e0.1, D0      ;move 1204E0 value to D0 (>$10)
007AFA: 3200              move.w  D0, D1            ;move D0 to D1
007AFC: 5341              subq.w  #1, D1            ;decrease D1 by 1
007AFE: B340              eor.w   D1, D0            ;11 XOR 10 = D0 becomes 01
007B00: 0800 0004          btst    #$4, D0           ;test fourth bit of D0 (it's 0 now)
007B04: 6700 00AA          beq     $7bb0             ;if equal to zero, branch 7BB0 (RTS)
007B08: 41F9 0004 0800      lea     $40800.1, A0      ;here is our breakpoint (executed just
007B0E: 3039 0000 3290      move.w  $3290.1, D0      ;after fire pressed - displays fire)

007B00: 0800 0001          btst    #$1, D0           ;first bit of D0 will always be 01 so
                                           ;beq won't happen!
```

```
<cheat desc="Rapid Fire">
  <script state="on">
    <action>temp0 =maincpu.mb@7B03</action>
  </script>
  <script state="run">
    <action>maincpu.mb@7B03=01</action>
  </script>
  <script state="off">
    <action>maincpu.mb@7B03=temp0 </action>
  </script>
</cheat>
```

## [phoenix] Phoenix (C) 1980 AMSTAR

Rapid fire cheat:

[#rapidfire](#)

- Started a trace just after firing, and trace file examined with getting unique instructions.
- Firing subroutine found at 00BB. Routine examined, 43A0 and 43A1 is \$EF when fire is pressed.
- But with double check, one location locks another and rapid fire is not allowed this way. So, changed an "and" instruction code with different "and".
- Due to preventing A OR 43A1 operation, this lock never happened.

```
00BB: 21 A0 43      lxi    h,$43a0      ;store 43A0 into H register
00BE: 7E          mov     a,m        ;value of 43A0 poked into A register
00BF: 2F          cma         ;compare A register
00C0: A0          ana     b        ;A register value AND value of 43A0 operation
00C1: 2C          inr     l        ;increase H register, it becomes 43A1
00C2: A6          ana     m        ;A register value AND value of 43A1 operation
00C3: C9          ret

00C2: A0          ana     b        ;behaves like NOP. Above "and" operation repeated, but
                                           ;A register value AND value of 43A1 avoided.
```

```
<cheat desc="Rapid Fire">
  <script state="on">
    <action>temp0 =maincpu.mb@00C2</action>
  </script>
  <script state="run">
    <action>maincpu.mb@00C2=A0</action>
  </script>
  <script state="off">
    <action>maincpu.mb@00C2=temp0 </action>
  </script>
</cheat>
```

## [aof] Art of Fighting (C) 1992 SNK

Desperation move cheat:

[#adaptexisting](#)

- It's known that, for Ryo and Robert, desperation move can only be performed if energy is very low and spirit is near full.
- Used existing cheats and inserted 2 watchpoints to monitor reading energy and spirit values:
- [wp 1092cd,1,r](#) and [wp 109a4,1,r](#)
- After they had been triggered, examined code. And changed compare values, to allow desperation move all time.

```
0139A8: 0C2E 0060 0224      cmpi.b  #$60, ($224,A6)      ;compares 0x60 with spirit value
0139AE: 6500 0012              bcs     $139c2              ;if spirit >= 0x60 go end
0139B2: 302E 004C              move.w  ($4c,A6), D0        ;store energy into D0
0139B6: 0C40 0020              cmpi.w  #$20, D0            ;compare it with 0x20
0139BA: 6200 0006              bhi     $139c2              ;if > 0x20, go end
0139BE: 4EFA 00B4              jmp     ($b4,PC); ($13a74)  ;perform desperation move

0139A8: 0C2E 0000 0224      cmpi.b  #$00, ($224,A6)      ;compare with 0, any spirit value
                                ;accepted.
0139B6: 0C40 0080              cmpi.w  #$80, D0            ;0x80 is highest energy value, so
                                ;compare result won't be > 0x80.
```

```
<cheat desc="Desperation Move (Both Players)">
  <comment>Ryo or Robert's desperation move is always available. Perform
  desperation move with joystick D, DF, F and then key C followed by
  key A.</comment>
  <script state="on">
    <action>temp0 =maincpu.mb@0139AB</action>
    <action>temp1 =maincpu.mb@0139B9</action>
  </script>
  <script state="run">
    <action>maincpu.mb@0139AB=00</action>
    <action>maincpu.mb@0139B9=80</action>
  </script>
  <script state="off">
    <action>maincpu.mb@0139AB=temp0 </action>
    <action>maincpu.mb@0139B9=temp1 </action>
  </script>
</cheat>
```

## [nspirit] Ninja Spirit (C) 1988 IREM

Select starting stage cheat:

[#ci&cn](#) [#non-linearlevel](#)

- First, performed a linear cheat search with [ci](#) & [cn +,1](#) for each new stage, no luck.
- And then, a non-linear cheat search with [ci](#) & [cn +](#) for each new stage, found at A38EC with increasing values, but after examining from code, it seemed, stage number is stored as word, value starts from DF40 and memory location is A38ED, for PL1. All values for each stage noted down. Also PL2 stage value is at A38EF.
- Condition search performed, all demo games were checked. Luckily, none of demo games starts from very first stage, so there's no need for this condition.
- Special byte sequences found in A38EF. If it's 00FF, it means, no start animation for PL1 for very first start.
- Same byte sequence at A38FF. If it's FFFF, it means no start animation for PL2.
- So, for first line of the cheat, when poke operation into A38ED is detected (DF40 value), selected stage's value poked into A38ED.
- Afterwards, A38EF checked. If it's 0x0000 for PL1, 0x00FF stored to disable startup animation for PL1.

Finish current stage cheat:

[#detectautomate](#)

- End of stage investigated briefly, before it, trace started and after trace analysis, found that A38E6 becomes 0x00FF when stage ends (default value is 0x0000). Afterwards, stage number word is increased by 2 there.
- Found additional conditions: A391F is 0x0001 for PL1 active and 0x0101 for PL2 active. For demo game, A38E0 is responsible byte. It's 0xFF in demo games.
- All these conditions joined, and cheat created as:
- If A391F is 0001 and A38E0 not equal to 0xFF, poke FF byte into A38E6. Calculate value (next stage value-2) with temp variables, and store it as byte into A38ED (As all stages start from DF).



```

<cheat desc="Select Starting Stage PL1">
  <parameter>
    <item value="0x00">Stage 1</item>
    <item value="0x0A">Stage 2</item>
    <item value="0x16">Stage 3</item>
    <item value="0x22">Stage 4</item>
    <item value="0x2E">Stage 5</item>
    <item value="0x30">Stage 6</item>
    <item value="0x3E">Stage 7</item>
  </parameter>
  <script state="run">
    <action condition="(maincpu.pw@A38ED == DF40)">
      maincpu.pw@A38ED=DF40+param</action>
    <action condition="(maincpu.pw@A38EF == 0000)">
      maincpu.pw@A38EF=00FF</action>
  </script>
</cheat>
<cheat desc="Select Starting Stage PL2">
  <parameter>
    <item value="0x00">Stage 1</item>
    <item value="0x0A">Stage 2</item>
    <item value="0x16">Stage 3</item>
    <item value="0x22">Stage 4</item>
    <item value="0x2E">Stage 5</item>
    <item value="0x30">Stage 6</item>
    <item value="0x3E">Stage 7</item>
  </parameter>
  <script state="run">
    <action condition="(maincpu.pw@A38F3 == DF40)">
      maincpu.pw@A38F3=DF40+param</action>
    <action condition="(maincpu.pw@A38EF == 00FF)">
      maincpu.pw@A38EF=FFFF</action>
  </script>
</cheat>
<cheat desc="Finish Current Stage Now! PL1">
  <script state="on">
    <action>temp0=maincpu.pb@A38ED</action>
    <action condition="(temp0 LT 4A)">temp1=48</action>
    <action condition="(temp0 GE 4A) AND (temp0 LT 56)">temp1=54</action>
    <action condition="(temp0 GE 56) AND (temp0 LT 62)">temp1=60</action>
    <action condition="(temp0 GE 62) AND (temp0 LT 6E)">temp1=6C</action>
    <action condition="(temp0 GE 6E) AND (temp0 LT 70)">temp1=6E</action>
    <action condition="(temp0 GE 70) AND (temp0 LT 7E)">temp1=7C</action>
    <action condition="(temp0 GE 7E)">temp1=84</action>
    <action condition="(maincpu.pw@A391F == 0001) AND (maincpu.pb@A38E0 != FF)">maincpu.pb@A38E6=FF</action>
    <action condition="(maincpu.pw@A391F == 0001) AND (maincpu.pb@A38E0 != FF)">maincpu.pb@A38ED=temp1</action>
  </script>
</cheat>
<cheat desc="Finish Current Stage Now! PL2">
  <script state="on">
    <action>temp0=maincpu.pb@A38F3</action>
    <action condition="(temp0 LT 4A)">temp1=48</action>
    <action condition="(temp0 GE 4A) AND (temp0 LT 56)">temp1=54</action>
    <action condition="(temp0 GE 56) AND (temp0 LT 62)">temp1=60</action>
    <action condition="(temp0 GE 62) AND (temp0 LT 6E)">temp1=6C</action>
    <action condition="(temp0 GE 6E) AND (temp0 LT 70)">temp1=6E</action>
    <action condition="(temp0 GE 70) AND (temp0 LT 7E)">temp1=7C</action>
    <action condition="(temp0 GE 7E)">temp1=84</action>
    <action condition="(maincpu.pw@A391F == 0101) AND (maincpu.pb@A38E0 != FF)">maincpu.pb@A38E6=FF</action>
    <action condition="(maincpu.pw@A391F == 0101) AND (maincpu.pb@A38E0 != FF)">maincpu.pb@A38F3=temp1</action>
  </script>
</cheat>

```

## [batman] Batman (C) 1991 ATARI

Select starting level cheat: [#ci&cn](#) [#non-linearlevel](#)

- Performed a linear cheat search, no luck.
- Non-linear cheat search, for level value, found at 104D50.
- 01 – 08 – 04 – 09 – 07 – 0D – 1C – 14 are values of each level, from 1 to 8.
- Found a condition check byte, at 104CB2. It's set to 0x1388 and start to increase after 3/7 lives choice appears.
- As this condition check only has that value in normal game, demo games are not affected.

End sequence cheat: [#endsequence](#)

- Checked value of 104D50 (level value) after completing last level, 8. It has 0x0C value during end sequence.
- Checked 104CB2 condition again, its value is 0x0000 before 3/7 lives appear. End sequence execution adjusted to happen before life selection.

```
<cheat desc="Select Starting Level">
  <parameter>
    <item value="0x01">Level 1</item>
    <item value="0x08">Level 2</item>
    <item value="0x04">Level 3</item>
    <item value="0x09">Level 4</item>
    <item value="0x07">Level 5</item>
    <item value="0x0D">Level 6</item>
    <item value="0x1C">Level 7</item>
    <item value="0x14">Level 8</item>
  </parameter>
  <script state="run">
    <action condition="(maincpu.pw@104CB2 == 1388) AND
      (maincpu.pb@104D50 == 01)">maincpu.pb@104D50=param</action>
  </script>
</cheat>
<cheat desc="See End Sequence">
  <script state="run">
    <action condition="(maincpu.pw@104CB2 == 0000) AND
      (maincpu.pb@104D50 == 01)">maincpu.pb@104D50=0C</action>
  </script>
</cheat>
```

## [ddonpach] DoDonPachi (C) 1997 ATLUS

Select level to follow level 1 cheat: [#ci&cn](#) [#linearlevel](#) [#hook&customcode](#)

- Linear cheat search performed, 101978 found (current level number – starts from 0).
- Level number increase and compare routines briefly examined. Second Loop and final level info gathered from history.dat file. 10197A holds value of Second Loop.
- As it seemed inserting new code into current ROM area is impossible, hook method applied, “add” instruction code changed with “jsr”. Found that free region at 98000, for custom code.

```
0013FC: 5279 0010 1978      addq.w #1, $101978.1    ;after each level ends, increases
                                ;level number.
0013FC: 4EB9 0009 8000      jsr    $98000.1        ;branch to custom code at 98000
```

10197A value	101978 value	Incoming Level	(param) value
00	01	Level 2	1
00	02	Level 3	2
00	03	Level 4	3
00	04	Level 5	4
00	05	Level 6	5
00	06	Second Loop	7
01	06	Level 7 (special)	6
01	07	End Sequence	8

98000	4A79	0010	197A	test 10197A	;check second loop byte
	6644			bne end	;if not zero, go end (original routine)
	4A79	0010	1978	test 101978	;check current level byte
	663C			bne end	;if not zero, go end (original routine)
	103C	0001		move #01,D0	;here, (param) is stored into D0
	B03C	0001		cmp #01,D0	;is it 1?
	6732			beq end	;if equal to 1,go end (original routine)
	B03C	0006		cmp #06,D0	;check with 0x06
	6D24			blt r1	;if < 6, branch r1
	B03C	0006		cmp #06,D0	;if = 6
	6718			beq r2	;branch r2
	B03C	0008		cmp #08,D0	;check with 0x08
	6606			bne r3	;if != 8, it means it's 7, branch r3
	903C	0001		sub #01,D0	;here, D0 is 8. Minus one and becomes 7.
	600C			bra r2	;branch r2
r3	4279	0010	197A	clr 10197A	;clear second loop byte
	903C	0001		sub #01,D0	;decrease value of D0
	6006			bra r1	;branch r1 to store into current level
r2	5279	0010	197A	add #01,10197A	;set second loop on with 0x01 byte
r1	13C0	0010	1979	move D0,101979	;poke D0 into current level
	4E75			rts	
end	5279	0010	1978	add #01,101978	;here is original code, increases level
	4E75			rts	;number by 1.

Kill boss with 1 hit cheat: `#ci&cn #boss`

- When boss energy gauges appeared, performed `ci ud & cn` - method to find gauge memory locations.
- After some tries, found them easily:
- Boss energies are stored in 10098, as double word (100158 for level 6 & level 7 bosses).
- Calculation performed for current level, and poked double word 0 into appropriate locations (10098 or 100158).

```

<cheat desc="Select Level to follow Level 1">
  <comment>Second Loop is new harder game starting from Level 1, Level 7 is final
  stage with two bosses after Second Loop.</comment>
  <parameter>
    <item value="0x01">Level 2</item>
    <item value="0x02">Level 3</item>
    <item value="0x03">Level 4</item>
    <item value="0x04">Level 5</item>
    <item value="0x05">Level 6</item>
    <item value="0x07">Second Loop</item>
    <item value="0x06">Level 7 (Special)</item>
    <item value="0x08">End Sequence</item>
  </parameter>
  <script state="on">
    <action>temp0 =maincpu.md@13FC</action>
    <action>temp1 =maincpu.mw@1400</action>
  </script>
  <script state="run">
    <action>maincpu.md@13FC=4EB90009</action>
    <action>maincpu.mw@1400=8000</action>
    <action>maincpu.mq@98000=4A790010197A6644</action>
    <action>maincpu.mq@98008=4A7900101978663C</action>
    <action>maincpu.mw@98010=103C</action>
    <action>maincpu.mb@98012=00</action>
    <action>maincpu.mb@98013=param</action>
    <action>maincpu.mq@98014=B03C00016732B03C</action>
    <action>maincpu.mq@9801C=00066D24B03C0006</action>
    <action>maincpu.mq@98024=6718B03C00086606</action>
    <action>maincpu.mq@9802C=903C0001600C4279</action>
    <action>maincpu.mq@98034=0010197A903C0001</action>
    <action>maincpu.mq@9803C=600652790010197A</action>
    <action>maincpu.mq@98044=13C0001019794E75</action>
    <action>maincpu.mq@9804C=5279001019784E75</action>
  </script>
  <script state="off">
    <action>maincpu.md@13FC=temp0 </action>
    <action>maincpu.mw@1400=temp1 </action>
    <action>maincpu.mq@98000=FFFFFFFFFFFFFFFF</action>
    <action>maincpu.mq@98008=FFFFFFFFFFFFFFFF</action>
    <action>maincpu.mq@98010=FFFFFFFFFFFFFFFF</action>
  </script>

```

```

        <action>maincpu.mq@98018=FFFFFFFFFFFFFFFF</action>
        <action>maincpu.mq@98020=FFFFFFFFFFFFFFFF</action>
        <action>maincpu.mq@98028=FFFFFFFFFFFFFFFF</action>
        <action>maincpu.mq@98030=FFFFFFFFFFFFFFFF</action>
        <action>maincpu.mq@98038=FFFFFFFFFFFFFFFF</action>
        <action>maincpu.mq@98040=FFFFFFFFFFFFFFFF</action>
        <action>maincpu.mq@98048=FFFFFFFFFFFFFFFF</action>
        <action>maincpu.mq@98050=FFFFFFFFFFFFFFFF</action>
    </script>
</cheat>
<cheat desc="Kill Boss with 1 hit Now!">
    <comment>Set this cheat after any boss completely appears on screen.</comment>
    <script state="on">
        <action>temp0 =maincpu.pb@101979</action>
        <action condition="(temp0 LE 04) OR (temp0 == 06)">
            maincpu.pd@100198=00000000</action>
        <action condition="(temp0 GE 05)">maincpu.pd@100158=00000000</action>
    </script>
</cheat>

```

## [le2] Lethal Enforcers II (C) 1994 KONAMI

Finish current section cheat: [#detectautomate](#)

- Memory location of current section was known, found before this cheat investigation (C00022).
- Added C00022 check as a condition that values 0x07 and 0x10 were not allowed (bonus stage values).
- Also, kill bosses with one hit cheat was found before this, and found that boss energy is stored in C013F8.
- End of a section examined, before it, trace started and after trace analysis, found that when C00002 becomes 0x0400, automated skip to next section happens.
- For the last condition, same memory region monitored and found that C00002 byte must be 0x03 for in-game (no demo – not inside menus).
- Also second cheat line used for clearing boss energy (C013F8).

End sequence cheat: [#endsequence](#)

- All above info used (condition checks) and additionally, stored 0x16 (last section value) into C00022.
- As above, stored 0x0400 into C00002 to start automated process.

```

<cheat desc="Finish Current Section Now!">
    <comment>Proceeds to next Section. Also works in Stage 3 Boss Section
    (Gunslingers).</comment>
    <script state="on">
        <action condition="(maincpu.pb@C00002 == 03) AND (maincpu.pb@C00022
        != 07) AND (maincpu.pb@C00022 != 10)">maincpu.pw@C00002=0400</action>
        <action>maincpu.pb@C013F8=00</action>
    </script>
</cheat>
<cheat desc="See End Sequence Now!">
    <comment>Enable the cheat during play.</comment>
    <script state="on">
        <action condition="(maincpu.pb@C00002 == 03) AND (maincpu.pb@C00022
        != 07) AND (maincpu.pb@C00022 != 10)">maincpu.pb@C00022=16</action>
        <action condition="(maincpu.pb@C00002 == 03) AND (maincpu.pb@C00022
        != 07) AND (maincpu.pb@C00022 != 10)">maincpu.pw@C00002=0400</action>
    </script>
</cheat>

```

## [shdancer] Shadow Dancer (C) 1989 SEGA

Finish current section cheat: [#detectautomate](#)

- End of a section examined, before it, trace started and after trace analysis, found that FFC498 becomes 0x1010 in that process.
- For condition check, FFC403 memory location found. Its value is only 0x01 in normal game. (No bonus stage, no demo game etc.)

- Performed tests with directly storing 0x1010 into FFC498, realized that number of bombs remaining should be zero and thrown stars shouldn't be zero (if so, no star used big bonus points awarded).
- Found those two with **ci** & **cn +,1** (and **cn -,1**) method that FFC706 has number of bombs remaining and FFC40F has number of stars thrown in current section.
- Cheat prepared with FFC403 check, and poked appropriate values into FFC498, FFC706 and FFC40F.

Successful bonus stage cheat: [#detectautomate](#)

- Used above condition, FFC403; but its value should be 0x02 in bonus game.
- End of a bonus stage examined, just before automation, trace started and after trace analysis, found that FFC452 becomes 0x01 at the end of any bonus stage (successful or not), and automatic process starts.
- Tested directly poking that value, but enemy movements went on, so another method tested, applied **ci** & **cn -,1** for each killed enemy. Found at FFC60E, number of enemies remaining for bonus screen.
- Also analyzed a successful bonus stage ending and found that FFC450 becomes 0x01 for frozen enemies (no more movement).
- Cheat prepared as above, poked appropriate values after FFC403 check.

```
<cheat desc="Finish Current Section Now!">
  <comment>Proceeds to next Section. Works in both 1UP and 2UP game.</comment>
  <script state="on">
    <action condition="(maincpu.pb@FFC403 == 01)">
      maincpu.pb@FFC706=00</action>
    <action condition="(maincpu.pb@FFC403 == 01)">
      maincpu.pb@FFC40F=01</action>
    <action condition="(maincpu.pb@FFC403 == 01)">
      maincpu.pw@FFC498=1010</action>
  </script>
</cheat>
<cheat desc="Finish Bonus Stage Successfully Now!">
  <comment>Works in both 1UP and 2UP game.</comment>
  <script state="on">
    <action condition="(maincpu.pb@FFC403 == 02)">
      maincpu.pb@FFC60E=00</action>
    <action condition="(maincpu.pb@FFC403 == 02)">
      maincpu.pb@FFC452=01</action>
    <action condition="(maincpu.pb@FFC403 == 02)">
      maincpu.pb@FFC450=01</action>
  </script>
</cheat>
```

## [fffight] Final Fight (C) 1990 CAPCOM

Knife stab ability cheat: [#ci&cn](#)

- As known, Cody has special knife stab ability, can stab enemies without dropping knife, in short distances. This can be applied to Guy and Haggar?
- On player select screen, **ci** and **cn +** (also **cn -**) applied after cursor is over next player, found that FF1525 becomes 0x02 for Cody, 0x04 for Haggar and by default 0x00 for Guy.
- Afterwards, monitored read operations from there, and found that when selection is done and fire button pressed, its value is divided by two and stored in FF85E9 (0x00 Guy, 0x01 Cody and 0x02 Haggar).
- Another watchpoint created, to read from FF85E9. It's read and written to FF857C, and this address holds current PL1 character. Another watchpoint created to monitor reading from FF857C. Two routines triggered, while holding knife, original code:

00BC6E: 0C2E 0001 0014	cmpi.b	#\$1, (\$14,A6)	;A6+\$14 is FF857C. Is its value 1 (Cody)?
00BC74: 6606	bne	\$bc7c	;if not 1, branch BC7C
00BCA4: 0C2E 0001 0014	cmpi.b	#\$1, (\$14,A6)	;same as above, is it 1 (Cody)?
00BCAA: 6706	beq	\$bcb2	;if 1, branch BCB2

Cheat applied code:

00BC74: 4E71	nop	;added NOP and branch removed
00BCAA: 6006	bra	\$bcb2 ;always branch, as Cody behavior



```

<cheat desc="Knife stab ability for Guy and Haggar">
  <comment>Guy and Haggar can stab enemies with knife as Cody can.</comment>
  <script state="on">
    <action>temp0=maincpu.mw@0BC74</action>
    <action>temp1=maincpu.mb@0BCAA</action>
  </script>
  <script state="run">
    <action>maincpu.mw@0BC74=4E71</action>
    <action>maincpu.mb@0BCAA=60</action>
  </script>
  <script state="off">
    <action>maincpu.mw@0BC74=temp0</action>
    <action>maincpu.mb@0BCAA=temp1</action>
  </script>
</cheat>

```

Never drop weapons cheat: [#ci&cn](#)

- Predicted for holding weapons that, a memory location could be 0x00 when no weapons held, 0x01 for weapon held. So, after a few **ci**, **cn +,1** and **cn -,1** tries, found that FF85B2 is 0x01 when a weapon is held.
- Added watchpoint for reading from FF85B2, but limited that watchpoint to catch the moments that weapon should be dropped (due to different kinds of hits player can get).
- Found 3 routines for above, by testing all kinds of weapon drop cases. Noticed that they check FF85B2, if it's 0x00, branch and do not clear it again, but if it's not zero, clear it and other locations after those "beq" instruction codes. So, they were replaced with "bra". First one's branch range increased, to bypass "clr" instruction codes.

Original code:

00A6A2:	4A2E	004A	tst.b	(\$4a,A6)	;check FF85B2
00A6A6:	6712		beq	\$a6ba	;if zero, branch A6BA
00A6A8:	306E	004C	movea.w	(\$4c,A6), A0	;clear weapon data starts here
00A6AC:	116E	003E	move.b	(\$3e,A6), (\$3e,A0)	
00A6B2:	422E	004A	clr.b	(\$4a,A6)	;clear FF85B2
00A6B6:	422E	0096	clr.b	(\$96,A6)	
00A6BA:	422E	0098	clr.b	(\$98,A6)	
00A6BE:	426E	00A2	clr.w	(\$a2,A6)	
00A6C2:	422E	00A0	clr.b	(\$a0,A6)	
00A6C6:	422E	00A4	clr.b	(\$a4,A6)	
00A6CA:	4A2E	0040	tst.b	(\$40,A6)	
00A72E:	4A2E	004A	tst.b	(\$4a,A6)	;check FF85B2
00A732:	670E		beq	\$a742	;if zero, branch A742
00A734:	306E	004C	movea.w	(\$4c,A6), A0	
00A738:	116E	003E	move.b	(\$3e,A6), (\$3e,A0)	
00A73E:	422E	004A	clr.b	(\$4a,A6)	;clear FF85B2
00A742:	4A6E	0018	tst.w	(\$18,A6)	
00A878:	4A2E	004A	tst.b	(\$4a,A6)	;check FF85B2
00A87C:	6712		beq	\$a890	;if zero, branch A890
00A87E:	306E	004C	movea.w	(\$4c,A6), A0	
00A882:	116E	003E	move.b	(\$3e,A6), (\$3e,A0)	
00A888:	422E	004A	clr.b	(\$4a,A6)	
00A88C:	422E	0096	clr.b	(\$96,A6)	
00A890:	4A6E	0018	tst.w	(\$18,A6)	

Cheat applied code:

00A6A6:	6022	bra	\$a6ca	;forced further branch, to preserve ;weapon byte and avoid "clr" commands.
00A732:	600E	bra	\$a742	;branch A742, weapon byte preserved
00A87C:	6012	bra	\$a890	;branch A890, weapon byte preserved

```

<cheat desc="Never drop weapons">
  <comment>Weapons are never dropped unless player loses a life.</comment>
  <script state="on">
    <action>temp0=maincpu.mw@0A6A6</action>
    <action>temp1=maincpu.mb@0A732</action>
    <action>temp2=maincpu.mb@0A87C</action>
  </script>
  <script state="run">
    <action>maincpu.mw@0A6A6=6022</action>
    <action>maincpu.mb@0A732=60</action>
    <action>maincpu.mb@0A87C=60</action>
  </script>
  <script state="off">
    <action>maincpu.mw@0A6A6=temp0</action>
    <action>maincpu.mb@0A732=temp1</action>
    <action>maincpu.mb@0A87C=temp2</action>
  </script>
</cheat>

```

## [theglad] The Gladiator (C) 2003 IGS

One hit kills cheat: #ci&cn

- "Kill bosses with one hit" cheat was already found (with ci uw and cn - method).

```

<action>prot.pw@18015C7C=0001</action>           ;boss energy can be in one or
<action>prot.pw@18015EF4=0001</action>           ;all of these three memory
<action>prot.pw@1801616C=0001</action>           ;locations, depending on boss.

```

- When about to reach first boss, before the cutscene, took a snapshot. Added watchpoints for 3 memory locations above, to learn when initial energy of boss is stored.

- In that example, 0x07D0 (2000) was stored into 18015C7C. PC=802259C.

- Examined that routine, before 802259C:

08022592: 5A51	LDRH R1, [R2, R1]	;get value of (R1+R2) and store into R1
08022594: 3060	ADD R0, 60	;add 0x60 to R0
08022596: 2900	CMP R1, 00	;R1 is zero?
08022598: D100	BNE 0802259c (00)	;if not zero, branch 802259c
0802259A: 2101	MOV R1, 01	;store 0x01 into R1
0802259C: 8101	STRH R1, [R0, #008]	;store R1 into (R0+\$08)

- When boss energy was set, R1 was 0x07D0 and R0+\$08 was 18015C7C.

- Noticed that 802259A can store 0x01 for R1=health, but 8022598 branch needs to be disabled. (As it's known that this code is always executed for every new enemy appearing on screen, as patch is performed as above, all new enemies will have 0x01 health values.)

Here is cheat applied code (Byte sequence is reversed in this ARM7 CPU):

```

08022598: 2100          MOV R1, 00          ;changed to null mov, bne removed
                                           ;next line fixes R1 value to 0x01.

```

```

<cheat desc="One Hit Kills">
  <comment>In ARCADE MODE, for all enemies, including bosses.</comment>
  <script state="on">
    <action>temp0=prot.rb@08022599</action>
  </script>
  <script state="run">
    <action>prot.rb@08022599=21</action>
  </script>
  <script state="off">
    <action>prot.rb@08022599=temp0</action>
  </script>
</cheat>

```

## One button codes cheat:

#easteregg #movementcodes

- While additional game info research, found that at mode selection screen, pressing these buttons unlock hidden features and modes:

BCBCBC	Fight two bosses at once in Challenge Mode
BCCCCB	More bosses selectable in Challenge Mode
DDDBBB	Ranking Mode
DCBCBC	More skills in Ranking Mode

- At mode selection screen, trace started and applied one of above codes, when unlock happened, trace stopped.
- Examined that trace file briefly, unique results method, as unlocking happens only once inside the trace.
- Found that just before unlock done, button codes read from PC=8037166 and 80371682.
- After read memory was checked, found that 0809ADXX ROM area is read, button codes. And each progress ends with \$FF byte, means button combination OK.
- Tested all 4 combinations, added breakpoint for that routine, collected start address of all combinations. Noticed that 0x06 is code for B button, 0x07 is C and 0x08 is D.
- Applied one button codes for those 4; at their button code start location, poked 0X and FF bytes.
- Here they are, normally in reverse byte order, shown in normalized format, right side shows cheat applied:

0809ADD6:	060706070607FF	two boss at once	08FF	button D
0809ADCF:	060707070706FF	more bosses	07FF	button C
0809ADC8:	080808060606FF	ranking mode	06FF	button B
0809ADDD:	080708060706FF	more skills	06FF	button B

```
<cheat desc="One button codes for Mode Selection screen">
  <comment>In MODE SELECTION screen, press:
  Button B to unlock RANKING MODE, Button B again for more skills in RANKING
  MODE.
  Button C to unlock all bosses in CHALLENGE MODE.
  Button D to fight 2 bosses at once in CHALLENGE MODE.</comment>
  <script state="on">
    <action>temp0=prot.rw@0809ADC8</action>
    <action>temp1=prot.rw@0809ADDD</action>
    <action>temp2=prot.rw@0809ADCF</action>
    <action>temp3=prot.rw@0809ADD6</action>
  </script>
  <script state="run">
    <action>prot.rw@0809ADC8=FF06</action>
    <action>prot.rw@0809ADDD=FF06</action>
    <action>prot.rw@0809ADCF=FF07</action>
    <action>prot.rw@0809ADD6=FF08</action>
  </script>
  <script state="off">
    <action>prot.rw@0809ADC8=temp0</action>
    <action>prot.rw@0809ADDD=temp1</action>
    <action>prot.rw@0809ADCF=temp2</action>
    <action>prot.rw@0809ADD6=temp3</action>
  </script>
</cheat>
```

## Tips and tricks:

1. [Learn all essential instruction codes](#): Move methods, arithmetic methods, comparing, test operations, flags, which flags are affected after specific instruction codes:

68000 move examples:

```
303C 000F          move.w  #$f, D0          ;store 0x000F into D0
33ED 0072 0080 0166 move.w  ($72,A5), $800166.1 ;get word (A5+$72) and store into 800166
1B40 9101          move.b  D0, (-$6eff,A5)   ;move byte value of D0 into (A5-$6EFF)
```

Z80 arithmetic examples:

```
AF          xor  a          ;A XOR A operation, A will be 00. Flags are affected
ED 42       sbc  hl,bc      ;subtract BC and carry flag from HL
CB 19       rr  c          ;rotate right contents of C register
```

V30 compare & test examples:

```
81 FE F8 03      cmp     ix,3F8h          ;compare IX register with 0x03F8
80 3E 62 00 02    cmp     byte ptr [62h],2h  ;compare (DS+$62) byte with 0x02
F6 06 E0 38 FF    test    byte ptr [38E0h],0FFh ;is byte at (DS+$38E0) 0xFF?
```

2. [Learn branches, jumps and jump to subroutines](#):

TMS34010 branch & jump examples:

```
CEFC          JRN  FF804820h ;jump if N(negative) flag is 1, to FF804820
0D3F 05F9      CALLR FF80A810h ;call FF80A810 and return (short distances, and return back)
0167          JUMP A7        ;jump to (value of A7 register)
```

3. [To write a custom code, learn branch forward – backward and how it's calculated](#): (These calculations vary for different kinds of CPUs.)

68000 branch examples:

```
004A92: 672E          beq     $4ac2          ;short branch (total 2 bytes)
004A94: 102D 0065      move.b  ($65,A5), D0          ;4A94+$2E points 4AC2, will branch there
...
004B2E: 6700 FF5E          beq     $4a8e          ;long branch (total 4 bytes)
004B32: 3034 7000      move.w  (A4,D7.w), D0      ;4B32+2-(10000-FF5E) points 4A8E, branch
                                ;+2 as this branch is long one.
062586: 618A          bsr     $62512          ;short branch (total 2 bytes)
062588: 42AD 040E      clr.l  ($40e,A5)        ;62588-(100-8A) points 62512, branch
```

4. [Exercise above with MAME debugger](#): After game has started, open debugger and execute instruction codes one by one with F11 key (step into) and check left side of debugger, registers and flags are there. When yellow PC indicator is on a conditional branch instruction, try to guess if the branch will happen or not. Also check memory locations and value of them, with Ctrl + M (memory window). Memory addresses can be read or written with different kinds of addressing (immediate – direct – indirect – indexed – PC relative ...), so learn the basics.

5. [If you cannot find instruction code & operation code manual for a specific CPU](#): Use **dasm** command in MAME debugger to disassemble partial or full ROM area to the file you specify. You will notice all kinds of instruction codes and their operation codes there. Example for TMS34010:

```
dasm file.txt,ffa10000,10000 ;disassemble FFA10000 – FFA20000 region into file.txt
```

6. [To examine previous code when breakpoint/watchpoint you created is triggered and debugging stopped](#): Press Ctrl + D and on that disassembly window, replace highlighted “curpc” entry with “curpc-10” or something. Also, you might need to examine main routine (if current routine is a subroutine), then press Shift + F11 to step out to one upper level.

7. [Use extended wp command](#), to avoid watchpoint triggering very frequently:

```
wp 10c00,1,w,wpdata== 2 ;will only be triggered written value at 10C00 is 0x02.
wp ff85de,1,r,pc!=a33d && pc!=b258 ;will only be triggered if a read operation occurs from FF85DE, but PC
                                ;addresses are not equal to A33D and also B258.
```

8. Use extended bp command, to avoid breakpoint triggering every time:

`bp 770,1,d0!=0f` ;will only be triggered if D0 register is not equal to 0x0F when PC is at 770.

9. Use extended bp command, to poke value into RAM/ROM region and change registers:

`bp 634,1,{maincpu.mb@638=60;g}` ;when PC=634, poke 0x60 into 638 and go on execution.  
`bp fd4,1,{r0==ff;g}` ;when PC=FD4, store 0xFF into R0 register and go on execution.

Important note for the first example: If that breakpoint is executed, even only once, that poke operation will be permanent. Disabling/clearing breakpoint or even soft/hard reset won't restore its original value.

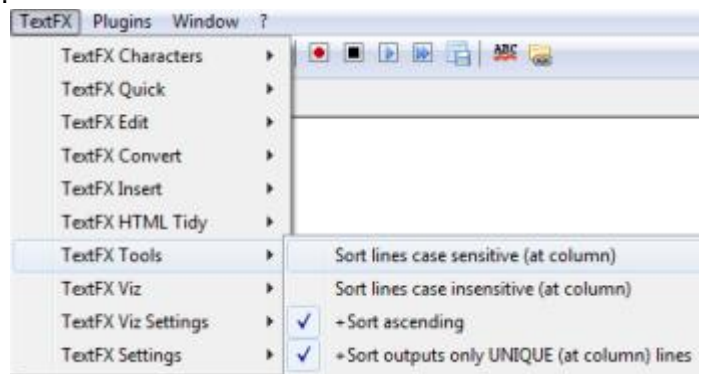
10. Use find command selectively, to find code location or memory region with one shot:

`find 0,100000,"INSERT"` ;search for "INSERT" string, address range is 0 to 100000. Watchpoint  
;(read) can be applied to found address(es), to find text displaying routine.  
`find 0,100000,w.6700,000a,0a41` ;search for 0x6700 followed by 0x000A and 0x0A41, same range.  
;chosen a low distance beq + eor operation codes, for less search results.

11. Trace file, remove duplicate lines:

To simplify the trace file, this is the first method. All duplicate lines will be removed and sort operation will be applied, so memory addresses executed will be sorted from low to high. It means if a specific code is executed several times in that trace, it will appear only once.

- Get Notepad++ from <http://notepad-plus-plus.org/download/> (Zip package recommended.)
- Start notepad++ and click: Plugins > Plugin Manager > Show Plugin Manager.
- On Plugin Manager window, in Available tab, scroll down, find "TextFX Characters", check it and click "Install".
- After restart of program, TextFX plugin will be available.
- Make sure "+sort ascending" and "+sort outputs only UNIQUE ..." are checked, as in screenshot.
- Open trace file in notepad++ and press Ctrl + A to select all.
- Click TextFX > TextFX Tools > Sort lines case sensitive (at column)
- After applying above, trace file will be sorted and all duplicate lines will be removed.



12. Trace file, getting unique instructions:

Unique instructions in a trace file are executed code lines those were executed ONLY ONCE. So, if a code in a memory location is executed more than once, all entries of that memory location should be cleared. After this operation, new trace file will be very simple and all lines will refer to code those were executed only once. It helps a lot for special analysis, i.e. after clearing a level/section in a game, increasing current level/section value process is executed only once, so that part of the code will appear in final trace file.

- In Windows OS, to be able to execute UN\*X commands, GNU utilities for Win32 is required.
- Download GNU utilities from <http://unxutils.sourceforge.net/>, named UnxUpdates.zip.
- Extract the zip file into a new folder.
- Copy original trace file to that folder, i.e. 1.txt.
- Open Command Prompt window, navigate to that folder where your trace file, uniq.exe and sort.exe files are in.
- From Command Prompt, give this command to create a new file, with unique instructions only:  
`sort 1.txt | uniq -u > 2.txt`
- After this command, 2.txt file will be generated. It has only unique instructions inside, and it's already sorted.



## World Wide Web links:

### MAME cheat database related:

<http://www.mamecheat.co.uk/>

<http://www.mamecheat.co.uk/forums/>

### MAME resources:

<http://www.mamedev.org/>

<http://www.arcade-history.com/>

<http://mameinfo.mameworld.info/>

### CPU instruction codes related:

<http://www.mamecheat.co.uk/guides.htm>

[http://emudocs.org/?page=All Chips](http://emudocs.org/?page=All+Chips)

<http://www.obelisk.demon.co.uk/6502/reference.html>

TMS34010 Instructions set

### Useful software:

<http://notepad-plus-plus.org/download/>

<http://unxutils.sourceforge.net/>

<http://mh-nexus.de/en/hxd/>

<https://code.google.com/p/dngrep/>

<http://www.scootersoftware.com/moreinfo.php>

<http://www.7-zip.org/>

Pugsy's Cheats (latest XML cheat collection for MAME)

MAME (and other systems) cheats forum

Official MAME Development Team website

history.dat for MAME can be downloaded from here

mameinfo.dat for MAME can be downloaded from here

several instruction codes for different CPUs

more CPU instruction sets

6502 (8-bit) instructions reference

TMS34010 (used in Midway games) instructions

Notepad++

GNU utilities win32 port

HxD Hex Editor

GREP for win32 with GUI

Beyond Compare software

7-Zip archiver