You are not logged in.

Pages: **1**
Topic rating: **0**

05-06-2010 18:36 #1

Deji

Gender:
From: UK
Registered: 09-11-2008
Posts: 172

SITE

CLEO 4 Opcode Guide

Thought it might be suitable to write an opcode guide here in English. At first I was majorly confused by a lot of the opcodes but after translating as much of the readme as possible and learning from own experience, I've decided to post a detailed guide into using these wonderful features. Feel free to help here, since I don't know everything yet.

I'll leave out the audio opcodes and some of the more obvious things, since you probably know what they are already.

CLEO 4 has much more dynamic opcodes than ever seen before. It can automatically detect the type of parameter and change how it handles them, thus allowing one opcode to be used in a variety of ways.

Let's start with a simple opcode that has been changed since CLEO 3, but is still supported by CLEO 3 mods.

0A99

Code:

```
0A99: chdir "CLEO"
```

Previously 0A99 only took an integer 1 or 0, which would switch between two pre-defined locations. This time it takes a path, so we can customize the location ourselves.

However, 1 and 0 still work. CLEO 4 has functionality for the old method as well. So this is a simple example of how this works.

0A9A

Code:

```
0A9A: $hFILE = openfile "settings.ini" mode "rb" // IF and SET
```

CLEO 4 incorperates some of the style which many programming languages do. There used to be some slightly confusing hexadecimal characters for the mode parameter, now it's less confusing, because each letter means something in more of a human language.

Code:

```
r = read
w = write
a = append
b = binary
t = text
+ = create (if the file doesn't exist)
```

So by combining these (you can't just combine any random combination), you can open files in different ways.

Code:

```
"rb"  = read binary file
"wb"  = write binary file
"wb+" = write binary file (create it if it doesn't exist)
"ab"  = append binary file (write at the end)
"ab+" = append binary file (create it if it doesn't exist)
"rt"  = read text file
"wt"  = write text file
"wt+" = write text file (create it if it doesn't exist)
"at"   = append text file
"at+" = append text file (create it if it doesn't exist)
"rb+" = read/write binary file
"rt+" = read/write text file
```

You can't, however, do "atb+". Either readwrite, write, append or read etc. Either binary or text.

0AC6

Getting the pointer to threads is something that has been used in CLEO 3 a lot. It's very helpful for storing and retrieving information without the use of variables.

But last time we needed to a few more complex things in order to calculate the pointer. Now it's done for us.

Code:

```
0AC6: 0@ = label @label offset
```

Here's an example of how this works:

Code:

```
:Label1
wait 0
0AC6: 0@ = label @ThreadMemory offset
000A: 0@ += 1
0A8C: write_memory 0@ size 1 value 6 virtual_protect 0
0A93: end
```

Before

Code:

```
:ThreadMemory
hex
    00 00 00 00 00 00
end
```

After

Code:

```
:ThreadMemory
hex
    00 06 00 00 00 00
end
```

Here we wrote '6' to the second byte at :ThreadMemory by getting the pointer, bringing us to the first byte and then adding 1, bringing us to the second byte.

0AC7

This does similar to the last opcode. It gets the pointer, but this time it gets the pointer to a variable.

Code:

```
0AC7: 0@ = var 1@ offset
```

By doing this, we could use read_memory or write_memory to read/write the value in the variable, but this is particularly good for use with strings.

Let's say we wanted the second 2 characters of a 4-char string.

Code:

```
1@v = "ABCD"
0AC7: 0@ = var 1@ offset
0@ += 2  // skip the first 2 characters
0A8D: 1@ = read_memory 0@ size 2 virtual_protect 0  // read the next 2 characters
// 1@ now contains "CD"
```

CD is very special.

0AD0

The code above can be extended to display the "CD" text in a box...

Code:

```
1@v = "ABCD"
0AC7: 0@ = var 1@ offset
0@ += 2  // skip the first 2 characters
0A8D: 1@ = read_memory 0@ size 2 virtual_protect 0  // read the next 2 characters
0AD0: show_text_box 1@v
```

Notice how @v is only used with text related opcodes. In general, there is no difference between @ and @v, except how it is handled on the other end. In CLEO 4, you can use both in most cases, but only when the variable contains pointers to strings. More on that later.

I'll be updating this post with more info later.

0AC8 & 0AD3

Now is later...

Code:

```
0AD3: 0@v = format "%d + %d = %d" 2 2 4
```

Code:

```
2 + 2 = 4
```

Now we have many more ways to play with strings!

Not only can this opcode store text to a text variable, it can store it to a normal variable, which can be used as a pointer to the text.

Code:

```
0AD3: 0@ = format "We could never store this much text in one variable before!"
0ACA: show_text_box 0@
```

We can do this with any opcode that corresponds to the "string convention", as put in the readme. However we must also allocate enough memory to the variable first:

Code:

```
0AC8: 0@ = allocate_memory_size 59
```

59 corresponds with the amount of characters that we need to store "somewhere". 0@ is simply a pointer to those characters.

After we finish, it is good to release this memory.

Code:

```
0AC9: free_allocated_memory 0@
```

This is also good when used with 0AC7. More on that later, too.

0AD4

I got confused with the meaning of this at first. Thought it got the position of text within a string, but that can be done in another way... Again, more on that later.

Code:

```
0AD4: 4@ = scan_string 0@v format "%d + %d = %d" 5@ 6@ 7@ //IF and SET
```

This gets the amount of ____ in a string, stores each one in a variable and then returns the amount found. If used as a condition, it returns true if ALL of the ____'s are found. ____ refers to a number, string... In this case %d (an integer). I'll post the list of them later.

Useful for extracting bits of information from a string. Each bit can then be used as desired.

Code:

```
0AD3: 3@v = format "%d * %d" 4 10  // 4 * 10
if
    0AD4:    0@ = scan_string 3@v format "%d + %d" 1@ 2@  // IF and SET
then
    0ACA: show_text_box "Performing addition..."
    0A8E: 0@ = 1@ + 2@
    wait 2000
    0ACE: show_formatted_text_box "%d + %d = %d" 1@ 2@ 0@
end
if
    0AD4:    0@ = scan_string 3@v format "%d * %d" 1@ 2@  // IF and SET
then
    0ACA: show_text_box "Performing multiplication."
    0A90: 0@ = 1@ * 2@
    wait 2000
    0ACE: show_formatted_text_box "%d * %d = %d" 1@ 2@ 0@
end
```

This will print:

Code:

```
Performing multiplication.
[...]
4 * 10 = 40
```

The first if construct will not return true, because we didn't use the addition (+) character. If we'd have used it, we would have gotten an addition instead.

0AD4 takes the numbers we originally stored in 3@v (4 and 10) and multiplies them with opcode 0A90. Then prints the calculation and the result using 0ACE, a formatted text box. This is just like the other text box opcode but ultimately saves using the format opcode, THEN the show_text_box opcode.

0AD7

File operations offer the ability to do much more with CLEO. Previously, only reading files in binary was an option. Now we have the ability to read files as text and on a per-line basis.

By this I mean, we can read each line seperately or 1 by 1.

Code:

```
0AD7: read_string_from_file $hFile to 0@v size 15 // IF and SET
```

This reads the first 15 characters of a line from the file and stores it at 0@v. If the line is shorter than 15 characters, it will simply read whatever is on the line.

If the file was:

Code:

```
This is line #1
This is line #2
This is a really long line that won't be read fully!
```

0AD7 would return this on it's first use:

Code:

```
This is line #1
```

Then it would return this on it's second use:

Code:

```
This is line #2
```

Then it would return this on it's third use:

Code:

```
This is a reall
```

It would only return the first 15 characters of the really long line.

However, the next read would return:

Code:

```
y long line that
```

So you can keep reading until the end of the line is reached.

If this is used as a condition and if the end of the file had been reached and there is nothing left to read, this opcode will return **false**, otherwise it will return **true** to verify a successful read. Unfortunately, this opcode is not affected by the seek_file opcode and therefore a file must be closed, then opened again in order to read from the beginning.

0AD8

Simply put, this writes a string to a file.

Code:

```
0AD3: 3@v = format "The time is %d:%d" 1@ 2@
0AD8: write_string_to_file 0@ from 3@v //IF and SET
```

0AD9

The last code can be simplified with this opcode.

Code:

```
0AD9: write_formatted_text "The time is %d:%d" in_file 0@ 1@ 2@
```

Last edited by Deji (07-07-2010 16:36)

Opcode Database

GTAG v2 (all about modding) - Coming Soon.

07-06-2010 18:48 #2

Alien

Gender:
Registered: 12-10-2008
Posts: 531

Nice post. I will continue, if no objection.

Part 1.

[*Everything that is written below, is relevant to the version of the game US 1.0, CLEO 4.1.1.20 or higher.*]
Recently, in CLEO 4 appeared one unexplained, but useful feature. From CLEO 3 were opcodes 0AA5-0AA8, which are used to call the x86 native functions directly from the script. In the CLEO 4 it got an opportunity to pass as a parameter scm-line as ordinary C-line!
An example:

Code:

```
0AA5: call 0x588BE0 num_params 4 pop 4 0 permanent_flag 1 0 "permanent text box"
```

It displays a permanent text-box with an arbitrary string. Previously, this effect had to make a dirty ways, directly through the memory as follows:

Code:

```
0AC7: 0@ = var 1@v offset
1@v = "permanent text box"
0AA5: call 0x588BE0 num_params 4 pop 4 0 permanent_flag 1 0 0@
```

or follows:

Code:

```
0AC6: 0@ = label @damn_text_there offset
0AA5: call 0x588BE0 num_params 4 pop 4 0 permanent_flag 1 0 0@
...
:damn_text_there
hex
"permanent" 20 "text" 20 "box" 00
```

```
end
```

I think this innovation, many people will find useful...

Part 2.

[*Everything that is written below, is relevant to the version of the game US 1.0, CLEO 4.1.1.22 or higher.*]

Go to the next item. According to the standard of language C++ (which is written the game), when passed to the function an array (including array of characters, ie strings) is transmitted is not the whole array, but only a pointer to it. The following code passes a pointer to a local string variable 0@v.

Code:

```
0@v = "permanent text box"
0AA5: call 0x588BE0 num_params 4 pop 4 0 permanent_flag 1 0 0@v
```

However, no differences in the types of pointers, ie the types of what it points on. We can sneak in the called function pointer to a local or global scripting variable (not just a string), pre-defining it some value, or using it as a buffer for the function return value.
An example:

Code:

```
{$CLEO}
wait 0
while true
    wait 0
    if
        player.Defined(0)
    then
        0AA5: call 0x56E010 num_params 2 pop 2 -1 0@v
        0AD0: show_formatted_text_lowpriority "player pos: (%g, %g, %g)" time 2000 0@ 1@ 2@

    end
end
```

The function 0x56E010 retrieves the coordinates of player, writing them into the buffer passed by pointer. If you pass a pointer to the variable 0@, coordinates will be written in the variables 0@, 1@ 2@.
Original post on Russian here.

Jump to GTAModding Go