Systems/DBG Debugger
Version 2.30

Systems/DBG Debugger
Version 2.30

Contents

How To Use This Book

This book describes the Systems/DBG debugger, **DDBG**. **DDBG** is used to debug assembly, C and C++ programms running on z/OS. This book describes how to run **DDBG**, and use it to debug z/OS programs.

To learn more about the Systems/ASM assembler, refer to the *Systems/ASM* manual.

To learn more about the Systems/C compiler, refer to the *Systems/C Compiler*, *Systems/C Utilities* and *Systems/C Library* manuals.

To learn more about the Systems/C++ compiler, refer to the *Systems/C++ Compiler* and *Systems/C++ Library* manual.

For further information, contact Dignus, LLC at (919) 676-0847, or visit http://www.dignus.com.

The Systems/DBG Debugger
DDBG

Systems/DBG Overview

The Dignus Systems/DBG Debugger, **DDBG**, is a debugger for the 390 and zSeries architectures executing the z/OS operating system. It supports debugging Dignus Systems/C and Systems/C++ programs as well as basic assembly language programs.

DDBG is architected in two pieces; the debugger "kernel" running remotely on z/OS (**DDBGKERN**), and a debugger user interface (**DDBG**) running locally.

The "kernel" uses TCP/IP to communicate with the user interface.

To begin a debugging session, the user initiates the **DDBG** command on the workstation. **DDBG** then displays the TCP/IP name and the selected port number the kernel will use to connect.

The kernel **DDBGKERN** is then started on the mainframe. This can be done in TSO or BATCH JCL, or under OpenEdition. The options on the **DDBGKERN** command specify the host name and port number for connecting back to the **DDBG** program.

The programmer then interacts with the **DDBG** command line interface to initiate the target program (the program to be debugged), examine memory, execute, single step, etc.

DDBG User Interface

Initiating DDBG

To start the **DDBG** user interface, simply issue the `ddbg` command in a terminal or command window.

DDBG will then display a banner and the TCP/IP connection information and wait for a connection from the **DDBGKERN** program on the mainframe. The TCP/IP connection information includes the host name and port number for the **DDBGKERN** program to use to connect to the **DDBG** program.

You can then start **DDBGKERN** via either TSO, BATCH or OpenEdition on the mainframe, specifying the TCP/IP connection information.

DDBG options

–n*portnumber* Specify a particular port number for a waiting connection from **DDBGKERN**. If –n is not specified, then **DDBG** requests an available port number from the operating system.

–k*script* On UNIX-style systems, *script* is a shell script that is executed to remotely initiate the debugging kernel **DDBGKERN** on z/OS. For example, the script could remotely execute the **DDBG** command and pass the TCP/IP connection information.

DDBG commands

The **DDBG** debugger has commands that allow you to examine memory, examine the values of registers, set breakpoints, initiate a program to debug, pass arguments to a program, etc.. It also includes an expression parser and evaluation engine to allow for flexibility in specifying arguments to the various commands.

backtrace - display runtime stack information

The `backtrace` command examines the program and displays the current stack frame information. The stack frame information is the stack of invoked function frames, as each function was entered.

Currently **DDBG** examines the instructions at the start of the current function to look for recognized function start sequences. It then examines the register save areas to display saved register values, etc...

The `backtrace` command has an optional sub-option that is an integer indicating how many stack frames to display.

The `backtrace` command may be abbreviated as `bt`.

The `where` command is a synonym for `backtrace`.

break - set a break point

Specify a break point at an address. The `break` command accepts an argument that indicates the address to use.

If the argument begins with `*`, then the following expression is evaluated to determine the address of the breakpoint.

If the argument begins with a digit, then it is taken to be the line number in the source program of where to set the break point.

Otherwise, the argument is taken to be a symbol which should be the start of a function for the break point.

For example:

```
break *$r12+240
```

sets a breakpoint at the address specified as 240(0,12).

```
break main
```

sets a breakpoint at the address specified by the symbol "`main`".

Breakpoints are added to the current breakpoint list. The breakpoint list can be examined with the `info breakpoints` command, disabled with the `disable breakpoint` command, enabled with the `enable breakpoint` command and removed with the `delete breakpoint` command.

Multiple breakpoints can be added to a given location, which can be useful when breakpoint conditions are also employed.

The `break` command may be abbreviated as `b`, `br` or `brea`.

condition - add a conditional test to a break point

The `condition` command adds an test expression to an existing breakpoint.

The `condition` command accepts up to two suboptions, the breakpoint number and an optional debugger expression to evaluate. If no expression is provided, the breakpoint is set to "unconditional", effectively removing any previous condition.

Normally, execution stops when the program reaches the address of a break point. Such a breakpoint is called "unconditional".

If a conditional test is added to the breakpoint, when program execution reaches the breakpoint, the expression is evaluated. If the expression is non-zero, the debugger halts program execution; otherwise, the debugger continues program execution as if the break point had not been present.

Note that multiple breakpoints can reside at the same location, each will be evaluated in turn to determine if any conditions evaluate to a non-zero value.

continue - continue program execution

The `continue` command continues program execution after a break point has been hit. `continue` takes an optional numeric argument specifying how many times to pass over the current breakpoint without stopping.

The `continue` command may be abbreviateed as `c`.

delete - delete an existing breakpoint or display items

The `delete` command will remove an existing breakpoint or display from the current breakpoint or display list.

The `delete` command accepts a list of integer values indicating which breakpoint or display to remove.

Before the list of values, the kind of item to be removed can be specified, with either a `"breakpoints"` or `"display"` sub-option preceeding the list. `"breakpoints"` or `"display"` may be abbreviated. If no sub-option is specified, `"breakpoints"` is assumed.

If no list is specified, the entire list of current breakpoints or display items will be deleted.

The `delete` command may be abbreviated as `d`.

The `unset` command is recognized as a synonym for `delete`.

dir - add a source search path

The `dir` command will add a directory to the source search path list. The source search paths are used when **DDBG** needs to locate a source file (for the `list` command) or a debug information side file (`.dbg` file).

The search begins with the source path as it was originally specified on the compiler commandline. If that path is a Unix-style absolute path (beginning with a directory separator), it is used as-is. If it is a Windows-style absolute path (beginning with a drive specifier), it is tried as-is and then the search may proceed as if the drive specifier had been stripped off of the path. Then a path is tried that is a combination of the directory the compiler was run in with the source filename. Finally, the entries in the source search path list are combined with the source filename.

If `dir` is not given any arguments, then the source search paths are displayed. The `show directories` command can also be used to view the source search paths.

disable - disable a list of breakpoints or display items

The `disable` command will mark a list of breakpoints, or display items as inactive. The breakpoint or display remains in the list, but isn't operating.

The `disable` command accepts a list of integer values indicating which breakpoint or display to remove.

Before the list of values, the kind of item to be disabled can be specified, with either a `"breakpoints"` or `"display"` sub-option preceeding the list. `"breakpoints"` or `"display"` may be abbreviated. If no sub-option is specified, `"breakpoints"` is assumed.

If no list is specified, the entire list of current breakpoints or display items will be disabled.

The `disable` command may be abbreviated as `disa` or `dis`.

disassemble - disassemble program memory

The `disassemble` command has up to 2 expression arguments.

If no arguments are provided, the disassembly address is the current value of `$pc`.

If one argument is provided, it is the starting address for the disassembly.

If two arguments are provided, the first is the starting address and the second is the ending address.

If no ending address is given, then the end is taken as the start plus 24 bytes.

Instructions are disassembled begining at the start address until the end address is reached.

display - auto-display expressions

The `display` command automatically evaluates and displays an expression each time the target program being debugged stops.

`display` accepts a single argument, the expression to display.

The `display` command can be decorated with a format indicator to override the type of the value to print. The `/FMT` suffix on the `display` command is used to specify the format. The value for `FMT` can be any of the types described in the `x` (examine memory) command.

Items to display are added to the current display item list. The display item list can be examined with the `info display` command, disabled with the `disable display` command, enabled with the `enable display` command and removed with the `delete display` command.

down - move earlier in the stack frame execution

The `down` command moves "down" in the stack frame, toward frames that were executed more recently than the current frame.

The `down` command accepts one sub option which is an integer indicating the number of frames to move. If the option isn't given, the default is 1 frame.

The `down` command may be abbreviated as `do`.

enable - enable a list of breakpoints or display items

The `enable` command will mark a list of breakpoints, or display items as active. A previously inactive breakpoint or display item would then be operating.

The `enable` command accepts a list of integer values indicating which breakpoint or display to remove.

Before the list of values, the kind of item to be enabled can be specified, with either a `"breakpoints"` or `"display"` sub-option preceeding the list. `"breakpoints"` or `"display"` may be abbreviated. If no sub-option is specified, `"breakpoints"` is assumed.

If no list is specified, the entire list of current breakpoints or display items will be enabled.

The `enable` command may be abbreviated as `en`.

file - name a program to debug

The `file` command names the program to debug. The debugger kernel **DDBGKERN** can also provide the name of the program to debug on the **DDBGKERN** command line.

The `file` command causes the remote kernel to load the desired program and prepare it for execution in a debugging environment.

help - print out help on DDBG commands

The `help` command displays information about the **DDBG** commands. It can be abbeviated as `h`.

info - provide machine and debugger state informtion

The `info` command displays information about the machine state of the program debugged, or information about the debugger's state. It accepts an argument that indicates what information to display.

address	Display the type and address of a symbol. If debugging information has been loaded, this includes program symbols, otherwise it would only include the `LD` and `SD` symbols from the loaded program. The `address` sub-option may be abbreviated as `addr`.
all-registers	Display the general registers, the floating-pt registers and `$pswa`, `$pswm` and `$pc`.
breakpoints	Display information about the currently set breakpoints.
display	Display information about the currently set display items.
registers	Display the general registers and the value of `$pc`.
stack	Synonym for the `backtrace` command.

The `info` command may be abbreviated as `i` or `inf`.

list - list program source

The `list` command lists the source of the program being debugged.

If no options are given, an no previous `list` command was given, the lists the source from the current program location in the current stack frame. If the previous command was a `list` command, and no options are given, the list continues from where it left off.

If an option is given, the debugger first tries to interpret it as a function name. If the name matches a symbol from the debugging information, the source associated with that function is listed.

If the option does not appear to be a symbol in the debugging information, the debugger tries to interpret it in the format of "*filename*:*linenum*" and looks for the line number from the given source file name.

Otherwise, the debugger tries to interpret the option as a line number within the last visited source file.

The debugger will list 10 lines at a time.

The `list` command may be abbreviated as `l`.

next - step one source line, step over a function call

The `next` command single steps one source line; if the source has a function call, the `next` command doesn't enter the function. `next` takes an optional numeric argument specifying how many lines to skip over.

nexti - step one machine instruction, step over a call

The `nexti` command single steps one machine instruction, unless that instruction is a function call. In that case `nexti` will skip over the call. `nexti` takes an optional numeric argument specifying how many instructions to skip over.

print - display expression once

The `print` command evaluates an expression and displays its result. The result is also stored in a numbered variable (i.e., `$1`) which can be used in subsequent expressions.

By default, `print` command uses the type of the expression to format its result. If the result of the expression is an array or a structure, `print` will print all the members of the array or structure.

The `print` command can be decorated with a format indicator to override the type of the value to print. The `/FMT` suffix on the `print` command is used to specify the format. The value for `FMT` can be any of the types described in the `x` (examine memory) command.

quit - exit the debugger

The `quit` command is used to end the debugging session. If connected to the remote kernel **DDBGKERN**, `quit` will end the remote kernel and the debugger session.

The `quit` command may be abbreviated as `q`.

run - run the program

The `run` command starts execution of the program.

Any characters after the `run` command are taken as parameters to pass to the program at program start-up.

The `run` command may be abbreviated as `r`.

set - set a target to a value

The `set` command is used to set a value in the memory of the target program being debugged, the value of a floating point or general register, or the value of the *pswaor*pc.

The `set` command is followed by an assignment statement of the format *target*=*source*, where the *target* and *source* are debugger expression. The *target* expression must be a value "l-value" in terms of C syntax. The *source* is interpreted and its value is retrieved.

For example, to set general registers #1 to the value 500, this command could be used:

```
set $r1=500
```

The general C expression syntax is allowed in either *target* or *source*. For example, to set the 4 byte integer at the offset addressed by general register #12 plus an offset of 26 to the value 10, you can use:

```
set *((int *)($r12+26))=10
```

Similarly, if the target debug program had the integer variables `i` and `j` in scope at the current location, then this command would set the variable `i` to the value of `j` plus 13:

```
set i=j+13
```

show - display debugger information

The `show` command will display information about the debugger itself. Two sub-commands are supported. `show version` will display **DDBG**'s version number. `show directories` displays the current source search paths (specified with the `dir` command).

shell - execute shell command

The `shell` command invokes the system's standard interactive shell to execute the command specified in the options.

On UNIX-style systems, if no options are specified a default interactive shell is initiated.

The `shell` command may be abbreviated as `sh`.

step - step one source line

The `step` command single steps program execution one source line. `step` takes an optional numeric argument specifying how many lines to skip over.

stepi - step one machine instruction

The `stepi` command single steps one machine instruction. `stepi` takes an optional numeric argument specifying how many instructions to skip over.

The `stepi` command may be abbreviated as `si`.

up - move later in the stack frame execution

The `up` command moves "up" in the stack frame, toward frames that were executed more after than the current frame.

The `up` command accepts one sub option which is an integer indicating the number of frames to move. If the option isn't given, the default is 1 frame.

x - examine memory

The `x` command examines memory. The produced output can be formatted according to several options.

Formatting options on the `x` command are indicated by a slash `/` and a format specification. The format specification is of the form *nnnfes* where *nnn* is an optional count value (default 1), *f* is an optional format indicator (default "`x`"), *e* is an optional encoding, and *s* is the size indicator (default "`w`"). The format indicators are:

o	octal
x	hex
d	decimal
u	unsigned decimal
t	binary
f	float
a	address
i	instruction
c	char
s	string

The size indicators are:

b	byte
h	halfword (2 bytes)
w	word (4 bytes)
g	giant (8 bytes)
v	very large (16 bytes)

The meaning of the encoding depends on which format is specified. For "`f`" (float) format, the encoding can be either "`h`" (Hex Floating Point), "`b`" (Binary Floating Point), or "`d`" (Decimal Floating Point). For "`c`" (char) or "`s`" (string) formats, the encoding can be either "`e`" (EBCDIC, the default) or "`a`" (ASCII). For other formats, there are no encoding choices.

The optional argument after the format specification is an expression indicating the first address of memory to examine. If no argument is given, then the address following a previous `x` command is used.

For example, to disassemble 10 instructions starting at the symbol "`main`" this command could be used:

```
x/10i main
```

whatis - display the type of an expression

The `whatis` command displays the type of the given debugger expression.

DDBG expressions

DDBG includes a C-like expression parser; expressions can be used as arguments to the DDBG commands.

There are several pre-defined symbols that provide access to register values and control over debugger settings.

Pre-defined names

The debugger includes these predefined names for the various registers:

$r0-$r15 General registers `R0` thru `R15`.

$f0-$f15 Floating point registers `F0` thru `F15`.

$pswa The PSW address value

$pswm The PSW mask values

$pc The PSW address expressed in the current execution AMODE.

Pre-defined aliases for the FP registers are also available with specific types. The prefix can be `h` (IBM Hexadecimal Floating Point), `b` (IEEEE Binary Floating Point), or `d` (Decimal Floating Point). An optional suffix species the size. No suffix indicates 8 bytes, while a suffix of `s` (short) indicates 4 bytes, and `l` (long) indicates 16 bytes (referencing both registers of the register pair). For example, `$bf4l` would reference the register pair of `$f4` and `$f6` as a 128-bit BFP value.

Arithmetic Constants

Integer constants are supported via C syntax; a prefix of `0x` indicates a hex value; a prefix of `0` indicates an octal value, otherwise the constant is taken to be an integer constant.

Floating point constants are evaluated to the default floating point type values, the size depending on the various constant suffixes. Currently, the default floating point type is HFP (Hexadecimal Floating Point.)

Expressions

The debugger includes a C-like expression parser and evaluation engine. For example, the expression:

```
$r13+128
```

evaluates to the current value in R13 plus 128.

The usual C expression operators, type casting, etc... can be used.

For example, to display memory at the address 256(10,12), this command could be used:

```
x (256+$r10+$r12)
```

Similarly, to print the value of field "myfield" in the structure "mystruct", simply:

```
print mystruct.myfield
```

Symbols from the target program can be directly used, just as they would be in original program.

When searching for a symbol from the execution environment, the normal scoping rules are employed. The debugger begins with the current stack frame and moves "up" to previous stack frames looking for the symbol.

Symbol values are similarly retrieved based on the current stack frame in the target program being debugged. The value of the general registers (`$r0-$r15`, `$pswa` and `$pc`) is also stack frame specific.

The "current" frame can be adjusted with the `up` and `down` commands.

Floating point operations are evaluated in 128-bit IEEE floating point values, regardless of the type of the operands or the host platform. Operands in floating point operations are converted to 128-bit IEEE floating point values, the operation is performed, and if needed the value is converted to the target floating point format. Thus, floating point operations in a **DDBG** may not produce exactly the result as the program being debugged.

DDBG kernel

Initiating the kernel on z/OS

The **DDBGKERN** can be started either via TSO or BATCH by executing the D̂DBGKERN program. D̂DBGKERN requires two options, the –h*hostname* and –n*portnumber* where *hostname* and *portnumber* are provided by the **DDBG** user interface.

DDBGKERN can also accept the name of the program to debug, and any parms to pass to the program, for facilitating JCL.

DDBGKERN is a 64-bit program, and keeps much of its data above-the-bar to make room more room for debugging 32-bit applications. There is also a 31-bit version **DDBGK31** which can be used if desired.

The debugger will need to write to your program code to set break points, etc... because of this, the target program cannot be loaded into read-only virtual storage, or the debugger will be unable to set breakpoints. Programs loaded from APF authorized libraries with the RENT setting can be loaded into read-only virtual storage. Programs running under OpenEdition are also loaded into read-only storage unless the `_BPX_PTRACE_ATTACH` environment variable is set to `"yes"`.

Initiating the kernel under TSO

To initiate the kernel under TSO, simply invoke the program. Note that TSO will make the parms upper-case by default, so the ÂSIS option is required. For example:

```
READY
call dignus.load(DDBGKERN) '-hmypc:2011' asis
```

This will start the kernel, indicating it should connect to the host "`mypc`" at port number `2011`. Both of these values are provided by the **DDBG** user interface when it starts.

This example does not name a program to debug, so the **DDBG** `file` command would be used to cause a program to be loaded for debugging.

Initiating the kernel under JCL

To alter JCL to debug in a batch session, simply replace the `PGM=` specification on an `EXEC` statement to be `DDBGKERN` instead of the original program name. Then, you can provide the program name and any options to pass to the program with the `PARM` value on the `DDBGKERN` program.

For example, if your original JCL was to run the program `MYPROG` with the parameter `parm1`:

```
//RUN EXEC PGM=MYPROG,PARM=('parm1')
```

then that would become:

```
//RUN EXEC PGM=DDBGKERN,PARM='-hHHHH:####,-pparm1,MYPROG'
```

This invocation of DDBGKERN specifies the host option (*HHHH*) and the port number (*####*) provided by the **DDBG** interface program. It then uses the –p option to indicate the parameter "`parm1`" is to be passed to the program to debug, and then it names the `MYPROG` program as the one to debug.

When the **DDBGKERN** kernel connects to the the **DDBG** interface program; it will pass the parameters and program name to **DDBG** which will then issue a `file` command to set up the program to debug.

Initiating the kernel under OpenEdition

In the OpenEdition environment, to start the **DDBGKERN** kernel simply execute they `ddbgkern` program with the appropriate arguments. For example:

```
ddbgkern -hmypc:2011
```

Note that when running under OpenEdition, the `_BPX_PTRACE_ATTACH` environment variable should be set to "yes" to load the program into writable memory so it can be debugged:

```
_BPX_PTRACE_ATTACH=yes
export _BPX_PTRACE_ATTACH
```

TCP/IP connection

The **DDBG** user interface begins the debugging session. When it starts, it displays the host name on which it is running, and a port number to use for connection. **DDBG** then waits on a TCP/IP connection from the debugging kernel program **DDBGKERN** on the remote system.

DDBGKERN must be passed the host name and port number to use for the connection.

If you would like a dedicated port number; then the –n option can be used on the **DDBG** command to specify a particular port number, otherwise **DDBG** will request an availabe port from the operating system.

This connection between **DDBG** and **DDBGKERN** requires a TCP/IP connection between the user workstation and the mainframe.

Kernel options

The **DDBGKERN** program can accept several options, optionally followed by the name of the program to debug.

If a program name is provided, that name is communicated back to the **DDBG** interface to indicate the name of the target program to debug. If a name is not provided in the **DDBGKERN** arguments, the name can be provided by the `file` command under the **DDBG** interface.

–h*name*	Specify the host name, and optionally port number, of the **DDBG** interface. The format of *name* is either *hostname* or *hostname*:*portnumber*. That is, `-hmypc:55555` is the same as `-hmypc -n55555`.
–n*portnumber*	Specify the port number for the connection to the bf DDBG interface if not provided in the –h option.
–p*parms*	Provide a parameter string to use when the target programs begins. This string will be communicated back to the bf DDBG interface program and used at program start up.

ASCII/EBCDIC Translation Table

The debugger, compiler and utilities use the following tables to translate characters between ASCII and EBCDIC. These tables represent the mapping of the IBM Code Page 1047 to ISO LATIN-1.

However, this is not the official IBM1047 mapping. The official mapping maps EBCDIC X'15' to LINEFEED X'85' and maps EBCDIC X'25' to NEWLINE X'0A'. This is reversed from their traditional mappings. Some vendors use the traditional mapping and some use the official mapping.

The debugger, compilers and utilities use the tradtitional mappings.

ASCII to EBCDIC

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00	01	02	03	37	2D	2E	2F	16	05	15	0B	0C	0D	0E	0F
1	10	11	12	13	3C	3D	32	26	18	19	3F	27	1C	1D	1E	1F
2	40	5A	7F	7B	5B	6C	50	7D	4D	5D	5C	4E	6B	60	4B	61
3	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	7A	5E	4C	7E	6E	6F
4	7C	C1	C2	C3	C4	C5	C6	C7	C8	C9	D1	D2	D3	D4	D5	D6
5	D7	D8	D9	E2	E3	E4	E5	E6	E7	E8	E9	AD	E0	BD	5F	6D
6	79	81	82	83	84	85	86	87	88	89	91	92	93	94	95	96
7	97	98	99	A2	A3	A4	A5	A6	A7	A8	A9	C0	4F	D0	A1	07
8	20	21	22	23	24	25	06	17	28	29	2A	2B	2C	09	0A	1B
9	30	31	1A	33	34	35	36	08	38	39	3A	3B	04	14	3E	FF
A	41	AA	4A	B1	9F	B2	6A	B5	BB	B4	9A	8A	B0	CA	AF	BC
B	90	8F	EA	FA	BE	A0	B6	B3	9D	DA	9B	8B	B7	B8	B9	AB
C	64	65	62	66	63	67	9E	68	74	71	72	73	78	75	76	77
D	AC	69	ED	EE	EB	EF	EC	BF	80	FD	FE	FB	FC	BA	AE	59
E	44	45	42	46	43	47	9C	48	54	51	52	53	58	55	56	57
F	8C	49	CD	CE	CB	CF	CC	E1	70	DD	DE	DB	DC	8D	8E	DF

EBCDIC to ASCII

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00	01	02	03	9C	09	86	7F	97	8D	8E	0B	0C	0D	0E	0F
1	10	11	12	13	9D	0A	08	87	18	19	92	8F	1C	1D	1E	1F
2	80	81	82	83	84	85	17	1B	88	89	8A	8B	8C	05	06	07
3	90	91	16	93	94	95	96	04	98	99	9A	9B	14	15	9E	1A
4	20	A0	E2	E4	E0	E1	E3	E5	E7	F1	A2	2E	3C	28	2B	7C
5	26	E9	EA	EB	E8	ED	EE	EF	EC	DF	21	24	2A	29	3B	5E
6	2D	2F	C2	C4	C0	C1	C3	C5	C7	D1	A6	2C	25	5F	3E	3F
7	F8	C9	CA	CB	C8	CD	CE	CF	CC	60	3A	23	40	27	3D	22
8	D8	61	62	63	64	65	66	67	68	69	AB	BB	F0	FD	FE	B1
9	B0	6A	6B	6C	6D	6E	6F	70	71	72	AA	BA	E6	B8	C6	A4
A	B5	7E	73	74	75	76	77	78	79	7A	A1	BF	D0	5B	DE	AE
B	AC	A3	A5	B7	A9	A7	B6	BC	BD	BE	DD	A8	AF	5D	B4	D7
C	7B	41	42	43	44	45	46	47	48	49	AD	F4	F6	F2	F3	F5
D	7D	4A	4B	4C	4D	4E	4F	50	51	52	B9	FB	FC	F9	FA	FF
E	5C	F7	53	54	55	56	57	58	59	5A	B2	D4	D6	D2	D3	D5
F	30	31	32	33	34	35	36	37	38	39	B3	DB	DC	D9	DA	9F