

Plugins Python API

All SocExplorer plugins expose some common functions and their own functions to the embedded Python terminal.

Common Functions

All these functions are either implemented in the base class **socexplorerplugin** or in the plugin itself. They are described with their C++ interface since they are dynamically wrapped in the Python context. To have a better understanding of how the arguments are converted between Python and C++ you can have a look [here](#). You can call any of the following methods from any plugin instance; **MyPluginInstance.Method(...)**.

Function list:

- [QVariantListRead](#) (unsigned int address, unsigned int count);
- [voidWrite](#) (unsigned int address, QList<QVariant> dataList);
- [voidcloseMe](#) ();
- [voidactivate](#) (bool flag);
- [voidsetInstanceName](#) (const QString & newName);
- [booldumpMemory](#) (unsigned int address, unsigned int count, QString file);
- [booldumpMemory](#) (unsigned int address, unsigned int count, QString file, const QString & format);
- [boolmemSet](#) (unsigned int address, int value, unsigned int count);
- [boolloadbin](#) (unsigned int address, QString file);
- [boolloadfile](#) (abstractBinFile * file);

[QVariantListRead](#) (unsigned int address, unsigned int count)

Reads target memory at given address and return its content. On any root plugin it will read system memory and by default on child plugin it will forward request to parent plugin until it reaches root plugin and read system bus.

Note that this function could be re-implemented on a child plugin and have a different behavior.

The returned list is a **Word** list which means that the smallest data you can read is a word(32 bits) and **count** is the number of words to read. The function respects host endianness so it will convert data depending on target endianness.

See also [voidWrite](#) (unsigned int address, QList<QVariant> dataList);

[voidWrite](#) (unsigned int address, QList<QVariant> dataList)

Writes given **datalist** at given **address** in target system bus. On any root plugin it will write system memory and by default on child plugin it will forward request to parent plugin until it reaches root plugin and writes system bus.

Note that this function could be re-implemented on a child plugin and have a different behavior.

The given list is a Word list which means that the smallest data you can write is a word(32 bits). The function respects host endianness so it will convert data depending on target endianness.

Example:

Let's consider we have a Leon3 with some RAM at 0x40000000 and we want to write 1 2 3 and 0xffffffff. We are connected to the target through **rootplugin** which can be replaced by the plugin we are using.

```
rootplugin.Write(0x40000000, [1, 2, 3, 0xffffffff])
```

See also [QVariantListRead](#) (unsigned int address, unsigned int count);

[voidcloseMe](#) ()

Closes the plugin.

voidactivate (bool flag)

Activates the plugin GUI, this function is called by **SocExplorer** you may not call this function unless you know exactly what you do. By default if the plugin GUI is disabled, it means that the root plugin isn't connected to the target and that any operation on child plugin are forbidden.

voidsetInstanceName (const QString & newName)

This method will be removed from python context since it may not be used from python terminal.

booldumpMemory (unsigned int address, unsigned int count, QString file)

Dumps the memory content at given **address** in given **file**. As for [Read](#) method it will read memory word by word and it will handle host and target endianness. The data will be written in ascii with the following format:

```
0x40000000: 0xad4c0136
0x40000004: 0x665b89c0
0x40000008: 0xabc04748
0x4000000c: 0x8e110724
```

Where left column is address and right column is the corresponding data word.

Example:

One interesting usage of this method is to dump memory space while your system is running to see if some memory space got modified. In this example we will load an elf file containing an executable to a Leon3 target through the AHBUARTplugin and dump the first 16 bytes before and after execution and compare them. We also assume that the device is attached to **ttyUSB0**.

```
proxy.loadSysDriver("AHBUARTplugin", "AHBUARTplugin0")
proxy.loadSysDriverToParent("dsu3plugin", "dsu3plugin0", AHBUARTplugin0)
AHBUARTplugin0.open("/dev/ttyUSB0", 3000000)
dsu3plugin0.openFile("/somePath/someElfFile")
dsu3plugin0.flashTarget()
AHBUARTplugin0.dumpMemory(0x40000000, 16, "/somePath/First_dump.txt")
dsu3plugin0.run()
waitSomeTime() # You can wait or process something or do whatever you want
AHBUARTplugin0.dumpMemory(0x40000000, 16, "/somePath/Second_dump.txt")
```

Then you can for example diff your two files:

```
diff -u /somePath/First_dump.txt /somePath/Second_dump.txt
```

See also [booldumpMemory](#) (unsigned int address, unsigned int count, QString file, const QString & format);

booldumpMemory (unsigned int address, unsigned int count, QString file, const QString & format)

This function behaves like [booldumpMemory](#) (unsigned int address, unsigned int count, QString file) except that it allows you to set output file format. Possible format are "srec", "bin" and "hexa".

See also [booldumpMemory](#) (unsigned int address, unsigned int count, QString file, const QString & format);

boolmemSet (unsigned int address, int value, unsigned int count)

Sets memory space with given **value** at given **address**, as for [Write](#) method it will write memory word by word and it will handle host and target endianness.

Example:

In this example we will first clear memory from 0x40000000 to 0x4000000C and then we will write 0x1234 from 0x40000010 to

0x4000002C

```
proxy.loadSysDriver("AHBUARTplugin", "AHBUARTplugin0")  
AHBUARTplugin0.open("/dev/ttyUSB0", 3000000)  
AHBUARTplugin0.memSet(0x40000000, 0, 4)  
AHBUARTplugin0.memSet(0x40000010, 0x1234, 8)
```

boolloadbin (unsigned int address, QString file)

boolloadfile (abstractBinFile * file)
