|  |  |  |
| --- | --- | --- |
| Bochs Developers Guide | | |
| [Prev](http://docs.google.com/directory-structure.html) | Chapter 2. About the code | [Next](http://docs.google.com/configure-scripting.html) |

2.3. Emulator Objects

2.3.1. Weird macros and other mysteries

Bochs has many macros with inscrutable names. One might even go as far as to say that Bochs is macro infested. Some of them are gross speed hacks, to cover up the slow speed that C++ causes. Others paper over differences between the simulated PC configurations. Many of the macros exhibit the same problem as C++ does: too much stuff happens behind the programmer's back. More explicitness would be a big win.

2.3.2. Static methods hack

C++ methods have an invisible parameter called the this pointer - otherwise the method wouldn't know which object to operate on. In many cases in Bochs, there will only ever be one object - so this flexibility is unnecessary. There is a hack that can be enabled by #defining BX\_USE\_CPU\_SMF to 1 in config.h that makes most methods static, which means they have a "special relationship" with the class they are declared in but apart from that are normal C functions with no hidden parameters. Of course they still need access to the internals of an object, so the single object of their class has a globally visible name that these functions use. It is all hidden with macros.

Declaration of a class, from iodev/pic.h:

...  
#if BX\_USE\_PIC\_SMF  
# define BX\_PIC\_SMF static  
# define BX\_PIC\_THIS thePic->  
#else  
# define BX\_PIC\_SMF  
# define BX\_PIC\_THIS this->  
#endif  
...  
class bx\_pic\_c : public bx\_pic\_stub\_c {  
  
public:  
 bx\_pic\_c();  
 ~bx\_pic\_c();  
...  
 BX\_PIC\_SMF void service\_master\_pic(void);  
 BX\_PIC\_SMF void service\_slave\_pic(void);  
 BX\_PIC\_SMF void clear\_highest\_interrupt(bx\_pic\_t \*pic);  
};

And iodev/pic.cc:

...  
#define LOG\_THIS thePic->  
  
bx\_pic\_c \*thePic = NULL;  
...  
void bx\_pic\_c::service\_master\_pic(void)  
{  
 Bit8u unmasked\_requests;  
 int irq;  
 Bit8u isr, max\_irq;  
 Bit8u highest\_priority = BX\_PIC\_THIS s.master\_pic.lowest\_priority + 1;  
 if(highest\_priority > 7)  
 highest\_priority = 0;  
  
 if (BX\_PIC\_THIS s.master\_pic.INT) { /\* last interrupt still not acknowleged \*/  
 return;  
 }  
  
 isr = BX\_PIC\_THIS s.master\_pic.isr;  
 if (BX\_PIC\_THIS s.master\_pic.special\_mask) {  
 /\* all priorities may be enabled. check all IRR bits except ones  
 \* which have corresponding ISR bits set  
 \*/  
 max\_irq = highest\_priority;  
 }  
 else { /\* normal mode \*/  
 /\* Find the highest priority IRQ that is enabled due to current ISR \*/  
 max\_irq = highest\_priority;  
...  
}  
...

Ugly, isn't it? If we use static methods, methods prefixed with BX\_PIC\_SMF are declared static and references to fields inside the object, which are prefixed with BX\_PIC\_THIS, will use the globally visible object, thePic->. If we don't use static methods, BX\_PIC\_SMF evaluates to nothing and BX\_PIC\_THIS becomes this->. Making it evaluate to nothing would be a lot cleaner, but then the scoping rules would change slightly between the two Bochs configurations, which would be a load of bugs just waiting to happen. Some classes use BX\_SMF, others have their own version of the macro, like BX\_PIC\_SMF above.

2.3.3. CPU und memory objects in UP/SMP configurations

The CPU class is a special case of the above: if Bochs is simulating a uni- processor machine then there is obviously only one bx\_cpu\_c object and the static methods trick can be used. If, on the other hand, Bochs is simulating an smp machine then we can't use the trick. The same seems to be true for memory: for some reason, we have a memory object for each CPU object. This might become relevant for NUMA machines, but they are not all that common -- and even the existing IA-32 NUMA machines bend over backwards to hide that fact: it should only be visible in slightly worse timing for non-local memory and non-local peripherals. Other than that, the memory map and device map presented to each CPU will be identical.

In a UP configuration, the CPU object is declared as bx\_cpu. In an SMP configuration it will be an array of pointers to CPU objects (bx\_cpu\_array[]). For memory that would be bx\_mem and bx\_mem\_array[], respectively. Each CPU object contains a pointer to its associated memory object. Access of a CPU object often goes through the BX\_CPU(x) macro, which either ignores the parameter and evaluates to &bx\_cpu, or evaluates to bx\_cpu\_array [n], so the result will always be a pointer. The same goes for BX\_MEM(x). If static methods are used then BX\_CPU\_THIS\_PTR evaluates to BX\_CPU(0)->. Ugly, isn't it?

2.3.4. The configuration parameter tree

Starting with version 1.3, the Bochs configuration parameters are stored in parameter objects. These objects have get/set methods with min/max checks and it is possible to define parameter handlers to perform side effects and to override settings. Each parameter type has it's own object type with specific features (numeric, boolean, enum, string and file name). A special object type containing a list of parameters is designed for building and managing configuration menus or dialogs automatically. In the original implementation the parameters could be accessed only with their unique id from a static list or a special structure containing pointers to all parameters.

Starting with version 2.3, the Bochs parameter object handling has been rewritten to a parameter tree. There is now a root list containing child lists, and these lists can contain lists or parameters and so on. The parameters are now accessed by a name build from all the list names in the path and finally the parameter name separated by periods.

Bit32u megs = SIM->get\_param\_num("memory.standard.ram.size")->get();

The example above shows how to get the memory size in megabytes from the simulator interface. In the root list (".") there is child list named "memory" containing a child list "standard". It's child list "ram" contains the numeric parameter type "size". The SIM->get\_param\_num() methods returns the object pointer and the get() method returns the parameter value.

The table below shows all parameter types used by the Bochs configuration interface.

**Table 2-2. Parameter types**

|  |  |
| --- | --- |
| Type | Description |
| bx\_object\_c | Base class for all the other parameter types. It contains the unique parameter id and the object type value. |
| bx\_param\_c | Generic parameter class. It contains the name, label, description and the input/output formats. |
| bx\_param\_num\_c | Numerical (decimal/hex) config settings are stored in this parameter type. |
| bx\_param\_bool\_c | This parameter type is based on bx\_param\_num\_c, but it is designed for boolean values. A dependency list can be defined to enable/disable other parameters depending on the value change. |
| bx\_param\_enum\_c | Based on bx\_param\_num\_c this parameter type contains a list of valid values. |
| bx\_param\_string\_c | Configuration strings are stored in this type of parameter. |
| bx\_param\_filename\_c | Based on bx\_param\_string\_c this parameter type is used for file names. |
| bx\_list\_c | Contains a list of pointers to parameters (bx\_param\_\*\_c and bx\_list\_c). In the config interface it is used for menus/dialogs. |

2.3.5. The save/restore feature

The save/restore feature is based on an extension to the parameter tree concept. A subtree (list) called "bochs" appears in the root of the parameter tree and some new "shadow" parameter types store pointers to values instead of the values itself. All the hardware objects have register\_state() methods to register pointers to the device registers and switches that need to be saved. The simulator interface saves the registered data in text format to the specified folder (usually one file per item in the save/restore list). Large binary arrays are registered with a special parameter type, so they are saved as separate files. The filename is then created from the full parameter path without the prefix "bochs.".

The table below shows the additional parameter types for save/restore.

**Table 2-3. Save/restore parameter types**

|  |  |
| --- | --- |
| Type | Description |
| bx\_shadow\_num\_c | Based on bx\_param\_num\_c this type stores a pointer to a numerical variable. |
| bx\_shadow\_bool\_c | This parameter type stores a pointer to a boolean variable (bit #0 only) or a numerical one (only one selected bit). |
| bx\_shadow\_data\_c | This special parameter type stores pointer size of a binary array. The data is saved in a separate file and the text file uses the file name as the value. |
| bx\_shadow\_filedata\_c | This special parameter type stores the descriptor of an open file (added in Bochs 2.5). |

It is also possible to use the bx\_param\_num\_c object with parameter save/restore handlers. With this special way several device settings can be saved to and restored from one single parameter. The disk image state is also handled this way (see below).

All devices can uses these two save/restore specific methods:

* register\_state() is called after the device init() to register the device members for save/restore
* after\_restore\_state() is an optional method to do things directly after restore (e.g. vga: force a display update)

To implement save/restore for hard drive images, the new method register\_state() has been added to the base class of the disk image objects. It creates a bx\_param\_bool\_c object called "image" and installs static save and restore handlers. The save operation finally sets the parameter's value (1 = success) and the save/restore handlers are doing the main job. The static handlers call the class-specific code. Depending on the image "mode" they copy either the whole image file or the file containing changes (journal). The files are saved similar to binary arrays with the same naming convention. The restore methods are doing some format or coherency checks, close the open image, copy the file(s) and finally re-open the image.

|  |  |  |
| --- | --- | --- |
| [Prev](http://docs.google.com/directory-structure.html) | [Home](http://docs.google.com/index.html) | [Next](http://docs.google.com/configure-scripting.html) |
| Directory Structure | [Up](http://docs.google.com/about-the-code.html) | Configure Scripting |