Srl API

This Chapter describe the use of the SRL API in DSX-VM (inspired from DSX SRL API).

The Srl API is an abstraction layer that provides the software programmer an easy acces to several software resources.

Thanks to the Srl (System Ressource Layer) API, the same code can be compiled and executed on several platforms such as

- a Linux/Posix workstation
- an MP-SoC architecture running the Mutek OS.

The code of the tasks is supposed to be written in C.

Mwmr Communication Channels

- `srl_mwmr_t channel = SRL_GET_MWMR(port_name)` defines a local variable associated to a MWMR channel acces port. The *port_name* argument corresponds to the port name defined in the task model defined in the DSX/L description.

- `srl_mwmr_read(channel, local_buffer, size)` reads *size* bytes from the MWMR channel to the local buffer. The *local_buffer* argument is a void*. The size argument must be a multiple of the channel width, and the the channel width must be a multiple of 4 bytes.

- `srl_mwmr_write(channel, local_buffer, size)` writes *size* bytes from the local buffer to the MWMR channel. The *local_buffer* argument is a void*. The *size* argument must be a multiple of the channel width, and the channel width must be a multiple of 4 bytes.

Locks

- `srl_lock_t lock = SRL_GET_LOCK(port_name)` defines a local variable associated to a lock. The *port_name* argument corresponds to the port name defined in the task model defined in the DSX/L description.

- `srl_lock_lock( lock )` takes a lock, waiting if necessary
- `srl_lock_unlock( lock )` releases the lock

Barriers

- `srl_barrier_t barrier = SRL_GET_BARRIER(port_name)` defines a local variable associated to a barrier. The *port_name* argument corresponds to the port name defined in the task model defined in the DSX/L description.

- `srl_barrier_wait( barrier )` waits for a barrier-global synchronization

Logging

Log API let you define several message levels. Levels allow you to keep the debug code in the source, and only compile it when needed.

In order, levels are:

- NONE
- TRACE
- DEBUG
- MAX

When writing your software, you decide what level the message is for. When compiling or running you software, you decide what minimal level your code must have to be printed.

- `srl_log(level, "message")` prints a message
- `srl_log_printf(level, "message_with_format", arguments...)` prints a printf-like message

Arguments in printf-like version may be not evaluated if level is not sufficient. Therefore you **MUST NOT** put expressions with side effects in the parameter list. ie do **not** do this:

```
srl_log_printf(DEBUG, "i=%d\n", i++);
```

Other services

- `srl_busy_cycles( N )` tells the simulation environment the simulation should run at least N cycles while in this call. This makes sense only for virtually synthetised tasks, otherwise, this call is a noop.
- `srl_assert( cond )` checks if *cond* is true, and fatally fails otherwise
- `srl_abort()` make the application (whatever the backend) abort now
- `srl_exit()` a way to correctly exit a task

Instrumentation services

- `srl_cycle_count()` returns the current cycle (most useful in a simulation context). On Posix, this returns the current millisecond since EPOCH (modulo $1^{32)}$.