

MaRTE OS Misc utilities

Daniel Sangorrin
daniel.sangorrin@{unican.es, gmail.com}

rev 0.1: 2008-5-12

1. Circular Memory Buffer

This is a generic software component that allows the user to write some data in a buffer. This data is supposed to be extracted periodically. Otherwise, in case of overflow the old data will be overwritten. Mutual exclusion is provided through the use of a Mutex. The component has been developed to serve as a basis for a logging user application. The following files compose this component:

circular_memory_buffer.[h, c]: this is the module that implements the functionality of circular memory buffers. Its interface is very simple and consists on the following functions:

```
int membuffer_init(membuffer_t *mbuff, int capacity, int ceiling);
int membuffer_write(membuffer_t *mbuff, const void *data, int len);
int membuffer_read(membuffer_t *mbuff, void *data, int len);
int membuffer_destroy(membuffer_t *mbuff);
```

membuffer_driver.[h, c]: this is the implementation of the driver interface required by the operating system (MaRTE OS). Internally, the driver just makes the appropriate calls to the circular_memory_buffer interface. In the header file, a constant for the capacity of the memory buffer and the ceiling of the internal mutex can be set by the user of the driver.

```
#define MBUFFER_SIZE      100000
#define MBUFFER_CEILING  98

int membuffer_driver_create();
ssize_t membuffer_driver_read
    (int file_descriptor, void *buffer, size_t bytes);
ssize_t membuffer_driver_write
    (int file_descriptor, void *buffer, size_t bytes);
int membuffer_driver_remove();
```

membuffer_driver_import.ads: this is just the Ada spec file that imports the previous functions to Ada, **pragma** Import (C, Create, "membuffer_driver_xxx"), so it can be inserted in the MaRTE OS drivers table.

test_circular_memory_buffer.c: a test for the circular memory buffer. This test uses directly the interface of the module not the driver. For example:

```
err = membuffer_init(&mbuff, BUFFER_CAPACITY, BUFFER_CEILING);
```

test_membuffer_driver.c: a test for the memory buffer driver. This file tests the driver so it goes through the MaRTE OS filesystem. Therefore it uses a POSIX interface (Note, that the driver can also be used from Ada or C++ programs as the bindings are provided by MaRTE OS), like this:

```
fd = open("/dev/membuffer", O_RDWR);
count = write(fd, buff, nbytes);
count = read(fd, buff, BUFFER_LENGTH);
```

In the figures X and X we can see a graph showing the dependences of the module. The most important is pthread.h which is used for the internal mutex.

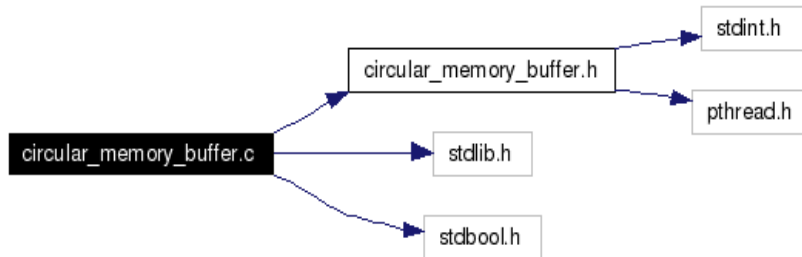


fig X.- circular_memory_buffer module include graph

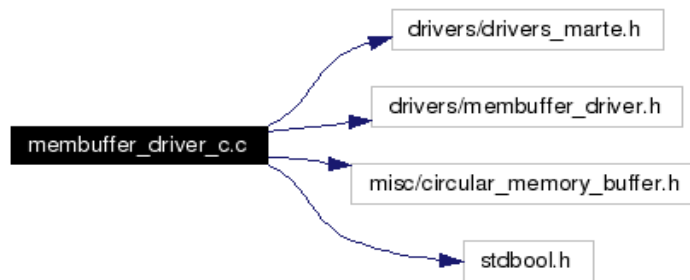


fig X.- membuffer_driver include graph

2. Logger

This is a software component that allows its users to retrieve data from the membuffer driver previously described and log it in a certain device, being that device a file in a disk, an ethernet card, a serial port, a wireless card (useful to retrieve data from equipment that is difficult to access) or the console for example. The component is composed of the following files:

logger.[h, c]: so far the logger has a simple interface that allows the user to either create a periodic thread that logs data periodically or log manually the data when the user wants. The constant `MAX_BYTES_TO_READ` defines how many bytes are read from the buffer (at the maximum) in each period or in each call to a manual log.

```
#define MAX_BYTES_TO_READ 1000

enum log_device_id_t {
    LOG_CONSOLE = 0,
    LOG_ETHERNET = 1
    ... other devices
};
```

```

int logger_init(enum log_device_id_t dev);
int logger_thread_create(struct timespec *period);
int logger_manual_call(void);

```

In the figure X we see the dependences of this module. Most of them are the typical POSIX header files. We use `timespec_operations.h` to operate on `timespec` values in the periodic thread and the headers to use the ethernet driver. In the future, support for other devices can be added easily by fulfilling a simple internal interface of two functions:

```

int (*init)(void);
int (*log_data)(int nbytes);

```

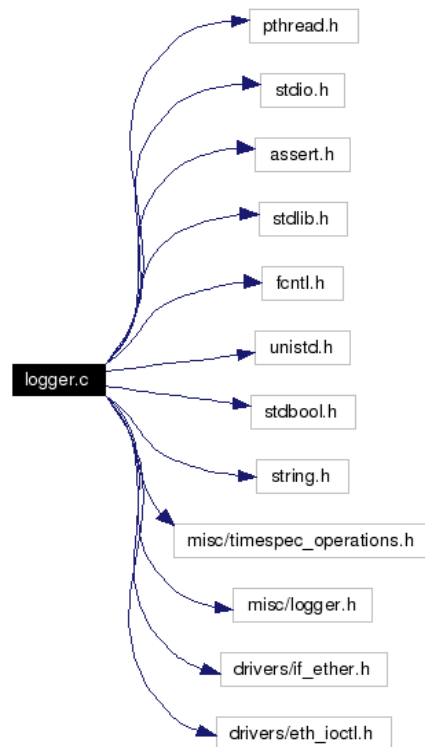


fig X.- logger include graph

test_logger.c: test for the logger that creates a logger thread and then writes some data to the memory buffer with a certain period. This data should be read by the logger and sent to the appropriate device (ethernet, disk, ...)

test_logger_manual.c: like the previous test but instead of creating a logging thread here we will use a manual trigger (a keyboard press event) for the logging.

linux_eth_receive_log.[h, c]: when we use LOG_ETHERNET as the logging device, the logger sends a set of ethernet frames with a broadcast address and a certain protocol number not allocated for other protocols. This module is a Linux program that uses raw sockets to receive those frames and write them to a log file. This log file can be then readed and treated with powerful Linux applications.

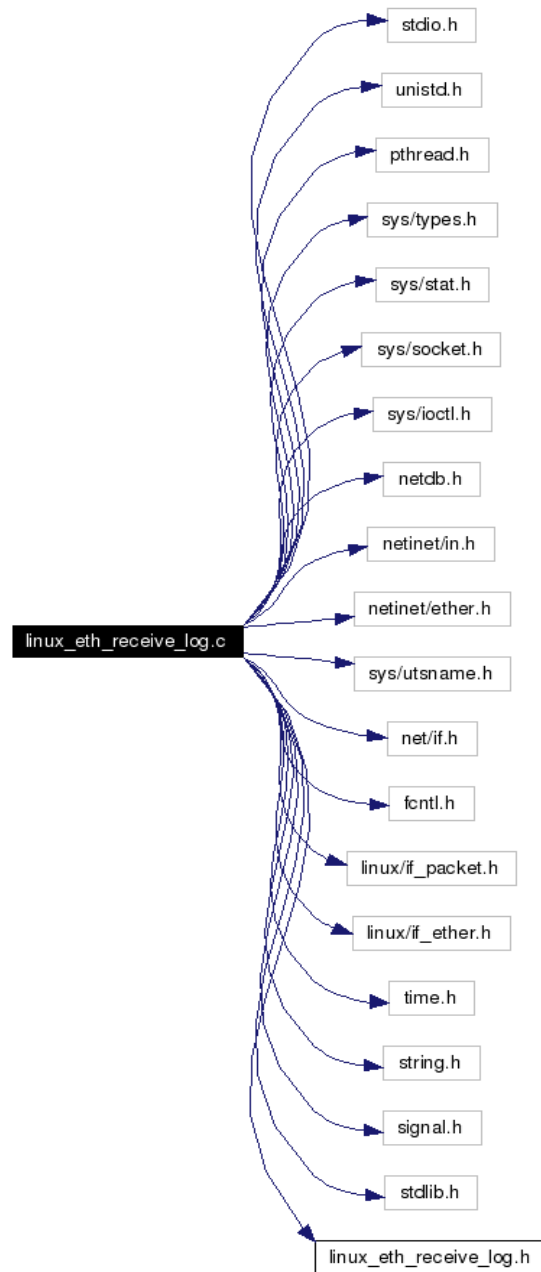


fig X.- Linux ethernet logger include graph

In the figure X we see the dependences of the Linux ethernet logger. The most important is the use of raw ethernet sockets.

3. Console switcher

So far we have two interesting components, one to put data bytes in a buffer and the other one to log those bytes into a specific device at a certain time (which can be periodic). Now, an interesting feature from the user point of view would be to send character strings (messages) to be logged. This is useful for debugging purposes for example. In order to give this functionality to the framework we have created a console_switcher driver that allows the user to change the console device dynamically so the messages printed to the console (i.e: with a **printf**) are redirected to the current device (the default device would be the normal console). The component is composed of the following files:

console_switcher.[h, c]: the implementation of the console_switcher driver. It provides the functions required by the MaRTE OS filesystem to be inserted and then used from the POSIX interface (**printf**). In order to change to a different console a non-standard ioctl option is used. We provide some simple macros to do that for us:

```
void SERIAL_CONSOLE_INIT(void);
void STANDARD_CONSOLE_INIT(void);
void MEMBUFFER_CONSOLE_INIT(void);
```

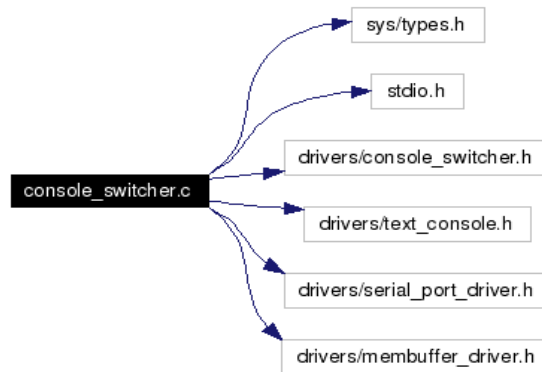


fig X.- console switcher include graph

In the figure X we see the dependences of the console switcher module. The most important ones are each of the console devices currently supported (text_console, serial port and membuffer).

test_console_switcher_membuffer.c: test for switching between the console and the memory buffer. It also uses the logger to retrieve the written messages and log them to a given device.

console_switcher_ioctl.ads: ioctl constants to use it from the Ada language

console_switcher_import.ads: import the driver functions to Ada to insert them in MaRTE OS's drivers table.

4. Time measurement library

So far, the framework we are describing is capable of redirecting printf messages to the memory buffer where they will wait to be logged at a “safer” time to a specific device (typically by sending them to a Linux host). Now we are going to go one step further in order to provide the feature of timestamping points of code that we will call trace points. This timestamps will be stored internally as efficiently as possible and at a “safe time” they will be flushed to the membuffer driver (with any additional information) so they can be logged (ie send them to a Linux host) and analyzed.

We have created two time measurement libraries that have different features and might be helpful in different situations:

- **POSIX time measurement library:**

This library is based on the POSIX function **clock_gettime()**. Its main good points are that, as a POSIX based library, it is portable (among operating systems and hardware platforms) and it allows the user to make measures on different kinds of clocks (for example, it is useful to know how much time has elapsed on a thread CPU time clock). On the other hand, it has a slight overhead that must be taken into account because it has to go through the POSIX interface and uses timespecs. The interface is quite simple so far, and it is intended to do begin-end time measurements:

```
int time_measure_posix_create(const char *name,
                             clockid_t clock_id,
                             time_measure_id_t *id);

int time_measure_posix_begin(time_measure_id_t id);

int time_measure_posix_end(time_measure_id_t id);
```

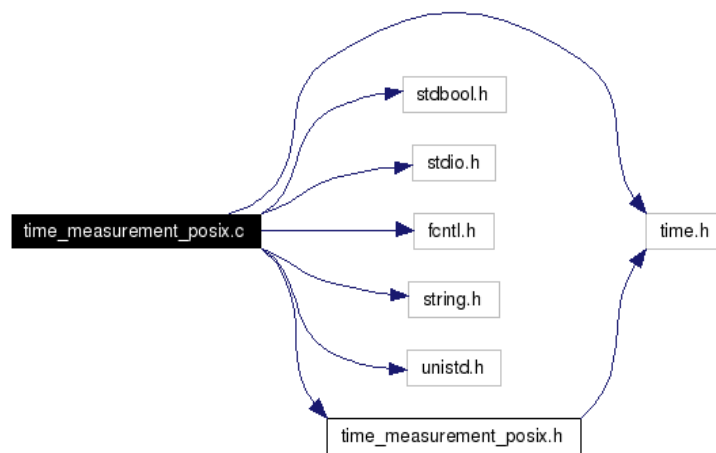


fig X.- console switcher include graph

In the figure X we see the dependences of the module. The most important are time.h, where the function `gettime` is declared, and `fcntl.h` and `unistd.h` that allows the module to open the memory buffer driver to store the measures whenever `time_measure_posix_end` is called. The component is composed of the following files.

time_measurement_posix.[h, c]: the time measurement implementation based on POSIX

test_time_measurement_posix_load.c: test for the `time_measurement_posix` module that measures the time taken to eat some time.

test_time_measurement_posix_ethernet.c: test for the `time_measurement_posix` module and ethernet. We get N measures of a packet go-return trip and then we log them manually.

– **HWTIME time measurement library:**

This library tries to avoid the overheads of using timespecs and the POSIX interface and reads and stores hardware timestamps (i.e for intel pentiums the `rdtsc` instruction gives the processor cycles). Its interface is a bit different to the previous one. We can't choose a CPU clock timer, only absolute timestamps can be taken. Also, it is not oriented to begin-end time measurements but to take measures of certain source code points (tracepoints). This is especially useful when we have a

execution path and we want to know where is the bottleneck because we just have to put some tracepoints and then performe substractions between the timestamps. The downside of this library is that it is not directly portable (although minor changes should be necessary, just the way to get the hardware time and the frequency typically). In the current implementation there are two main dependencies:

```
#define rdtsc11(val) \
    __asm__ __volatile__ ("rdtsc" : "=A" (val))

extern hwtime_t hardware_interface__get_hwclock_frequency ();
```

The component is composed of the following files:

time_measurement_hwtime.[h, c]: the library implementation. Its interface is as follows:

```
#define MX_TRACE_POINTS                20
#define MX_TRACE_POINT_CHARS          50
#define MX_TIMESTAMPS_PER_ID          1000

typedef unsigned int trace_point_id_t;    // 0 .. MX_TRACE_POINTS - 1

int time_measure_hwtime_init(trace_point_id_t id, const char *name);
void inline time_measure_hwtime_set_timestamp(trace_point_id_t id);
int time_measure_hwtime_write_membuffer(trace_point_id_t id);
```

test_time_measurement_hwtime_load.c: test for the time_measurement_hwtime module that measures the time taken to eat some time.

test_time_measurement_hwtime_ethernet.c: test for the time_measurement_hwtime module and ethernet. We get N measures of a packet go-return trip and then we log them manually.

So far we have our own way to format the log messages, but it may be desirable in the future to adapt them to support several analysis tools. We plan to study the difficulty of adapting them to tools like QNX TimeDoctor¹, Rapita's tools², KIWI³, a task inspector tool that exists for MaRTE OS⁴. For the moment, to make the analysis of the measures we are just using simple customized scripts written in Perl to parse the log messages and then copy the measures in an OpenOffice spreadsheet to extract a histogram ("FREQUENCY(Data;Bin)" + Ctrl-Shift-Enter), max, min and average values, variance, etc.

5. RT-EP Wireshark plug-in

This is a plug-in for the Wireshark sniffer in order to monitor the RT-EP protocol. The sniffer will be helpful to explain the number of messages transmitted in each transaction and therefore to check that the sum of the lower level message timings explain the timing measures of the high level measures (i.e the time to negotiate a transaction). The plug-in is in a directory inside the RT-EP protocol directory (x86_arch/drivers/rtep) and it contains the installation instructions.

¹ <http://sourceforge.net/projects/timedoctor>

² <http://www.rapitasystems.com/>

³ <http://rtportal.upv.es/apps/kiwi/>

⁴ <http://marte.unican.es>

6. Linux linked lists

The file '`x86_arch/include/misc/linux_list.h`' contains code extracted from the Linux kernel that implements double linked lists.

See <http://kernelnewbies.org/FAQ/LinkedLists>

This linked lists are very useful to link the same elements in different lists.

7. Freelist

The file '`x86_arch/include/misc/freelist.h`' contains the interface to the freelists which is an interesting data structure that implements a singly linked list that is used to find free cells in some external table. Parallel to the external table, there is a table of indexes organized as a singly linked list; it is the free list. A separate index, `free_cell`, indicates which is the first element of the singly linked list. The list is terminated with an index of -1.

A typical example of use is to have a pool of preallocated structures (in an array) and provide the user with an identifier (the position in that pool) which can be allocated and deallocated in O(1) time by using the freelist linked list.

```
int freelist_init(freelist_t *list, int size);
int freelist_alloc(freelist_t *list);
int freelist_free(freelist_t *list, int cell);
```

Check the file `test_freelist.c` to see an example.