MaRTE OS

Minimal Real Time Operating System for Embedded Applications

Authors:	Mario Aldea Rivas	aldeam@unican.es
	Michael González Harbour	mgh@unican.es

User's Guide

(Last updated for MaRTE OS version 1.9_13Jan2009)

Contents of this Guide

1. Making Applications
2. Cross development environment (architecture x86)
3. Debugging Applications
4. Configuring a Stand-Alone Target
5. Compiling MaRTE Kernel and Libraries
6. Optimizing MaRTE Kernel for Speed and Size
7. Enabling Debug Checks and Messages in the Kernel
8. MaRTE OS device drivers
9. Using the 'tasks_inspector' Tool
10. Changing MaRTE OS Configuration Parameters
11. Miscellany of Utilities

1.- Making Applications

Applications are made using the scripts **`mgcc`** and **`mgnatmake.`** They are invoked in almost the same way that **`gcc`** and **`gnatmake`**. They link the user's application with the MaRTE OS libraries.

Examples of use:

```
$ mgcc -g -O2 extra_obj.o my_program.c

$ mgcc use_math.c -lm

$ mgnatmake -gnato my_program.adb
```

For Ada programs also **`mgnatbind`** is provided.

Including MaRTE OS source directories

In order to use MaRTE Ada packages in a user's Ada application it is necessary to include in the compilation command the directories where MaRTE sources are located. It can be done very easily by including a MaRTE specific flag for mgnatmake: `-Imarte_src_dirs`

```
$ mgnatmake -gnata -Imarte_src_dirs my_program.adb
```

(using `-Imarte_src_dirs` is equivalent to `-aI$MPATH/kernel -aI$MPATH/sll -aI$MPATH/arch/hwi -aI$MPATH/misc -aI$MPATH/posix5 -aIall_the_drivers_directories -aO$MPATH/lib`)

2.- Cross development environment (architecture x86)

A cross development environment uses two computers: the **host** and the **target**. The host computer is a PC with Linux, GNAT and MaRTE_OS, where we write and make the application. The target is the computer where the application will be executed. As target MaRTE OS only requires a PC with a 386 processor or above and a boot device: floppy, hard disk, Flash-RAM, PXE, ...

Usualy the application will be transferred from the host to the target across Ethernet. Probably the easiest way of doing it is using a floppy disk and "Etherboot" (for more information visit "MaRTE OS Boot process (x86 architecture)" it also describes other booting alternatives involving hard disk, Flash-RAM, PXE, etc.).

It can also be used and emulator to emulate a real target computer. More information about this topic in the documentation section of the MaRTE OS web page.

[Note] "Etherboot": http://www.etherboot.org/. An Etherboot image can be created at http://rom-o-matic.net/ as explained in the installation guide "INSTALL".

[Note] "MaRTE OS Boot process (x86 architecture)": MaRTE OS Boot process (x86 architecture) in the documentation section of the MaRTE OS web page.

[Note] "Hello MaRTE OS using an emulator": Hello MaRTE OS using an emulator in the documentation section of the MaRTE OS web page.

3.- Debugging Applications

Architecture x86

For the x86 architecture if you are using a target PC you can debug it using a serial line (RS-232) connecting host and target.

In order to debug an application it should be compiled with the -g flag. To synchronize the execution with the remote debugger running in the host is necessary to add these lines to the application code (possibly at the very beginning of your program):

Ada program:

```
...
with Debug_Marte; use Debug_Marte;
...
procedure My_Proc is
begin
  ...
  Debug_Marte.Init_Serial_Communication_With_Gdb (Serial_Port_1);
```

```
  Debug_Marte.Set_Break_Point_Here;

  Application's code;
end My_Proc;
```

C program:

```
...
#include <debug_marte.h>
...
int main()
{
  ...
  init_serial_communication_with_gdb (SERIAL_PORT_1);
  set_break_point_here;

  Application's code;
}
```

The application executes until the first **`Debug_Marte.Set_Break_Point_Here`** or **`set_break_point_here`** is reached. At this point the execution is stopped and the target is ready to synchronize with the debugger.

At the MaRTE installation path run **`gdb`**:

```
$ gdb mprogram
```

The connection with the target is performed by executing the following "gdb" command:

```
(gdb) target remote /dev/ttyS0
```

Where '/dev/ttyS0' is the device file for the serial port 1. After this you should get a message like this:

```
Remote debugging using /dev/ttyS0
main () at hello_world.c:25
25 printf("\nHello, I'm a C program running on the MaRTE OS.\n\n");
```

All these steps can be performed at once executing:

```
$ gdb -x /installation/path/marte/utils/marte_db
```

When executing gdb as above it is possible to reconnect with the target using the macro connect:

```
(gdb) connect
```

Architecture x86 using the QEMU emulator

For the x86 architecture if you are using the QEMU emulator as a target PC you can use its built-in debugging capabilities. (More in the tutorials section):

```
$ qemu -s -S disk.img &

$ ddd &

(gdb) target remote localhost:1234
```

```
(gdb) break main

(gdb) cont
```

Architectures Linux and Linux_Lib

If you have chosen Linux or Linux_lib as your architecture you can debug the program as any other Linux application:

```
$ cd $HOME/marte

$ mgnatmake -g hello_world.adb -o mprogram
```

or

```
$ mgcc -g hello_world_c.c -o mprogram

$ gdb mprogram
```

4.- Configuring a stand-alone target

The netbooting of user's applications is a very suitable method for the developing stage, but if you prefer it, the application can be executed in a target completely disconnected from any other computer. For this configuration neither the RS-232 nor the Ethernet are necessary for the target computer, only a boot device (floppy, hard disk, Flash-RAM) is required.

A very easy way of doing this is using the "grup-floppy image" provided in the download section of the MaRTE OS web page. In order to create a grup floppy, untar the file and copy the disk image to a floppy:

```
$ tar zxf grubfloppy.img.tgz

$ dd if=grubfloppy.img of=/dev/fd0
```

Then, every time you want to run a MaRTE OS application copy the `mprogram` to the floppy:

```
$ mcopy my_program a:mprogram
```

Finally insert the floppy in your target computer and reset it.

Remember the MaRTE application (mprogram) must fit in the disk (< 1.4 Mb).

Information about how to use other boot devices in: "MaRTE OS Boot process (x86 architecture)": MaRTE OS Boot process (x86 architecture) in the documentation section of the MaRTE OS web page.

5.- Compiling MaRTE kernel and libraries

The kernel and libraries are compiled using full optimization options during the installation stage, so if you are planning to use MaRTE OS "as is" to build your applications, just skip the rest of this chapter.

But there are some reasons why you can be interested on recompile MaRTE OS:

- In case you are making some modifications to MaRTE OS or adding some new functionality.
- If you want to compile the kernel with some compiler switches different than the default ones.
- If you want to enable some debug checks in the kernel (see chapter "Enabling Debug Checks and Messages in the Kernel").
- If you are installing a new device driver in the system (see chapter "MaRTE OS Devices").
- If you want to use the 'tasks_inspector' tool (see chapter "Using the 'tasks_inspector' Tool").
- In order to change the maximum number of resources allowed in a MaRTE application (see chapter "Changing MaRTE OS Configuration Parameters").

There are some scripts in the 'utils/' directory that allow you to recompile the kernel and/or the libraries:

msetcurrentarch:

When used without parameters, `msetcurrentarch` returns the current architecture. That is, the architecture for which `mgcc` and `mgnatmake` are currently configured to generate the executable.

This script can also be used to change the current architecture:

```
$ msetcurrentarch -march Linux
```

or

```
$ msetcurrentarch -march Linux_lib
```

or

```
$ msetcurrentarch -march x86 -mproc [i386 | pi | pii]
```

The flag `-mproc` allows to specify the processor to be used in the x86 architecture:

- `i386`: Intel x86 processor. PIT is used as timer and clock.
- `pi`: Pentium I. PIT is used as timer and TSC as clock.
- `pii`: Pentium II or above. LocalAPIC timer used as timer and TSC as clock.

mkmarte:

Compiles the kernel and drivers (calling 'mkdrivers'). It accepts gnat and/or gcc options, so if you want a kernel optimized for speed execute:

```
$ mkkernel -gnatn -O3 -gnatp
```

The "-f" flag can be added to force the recompilation of all the kernel, otherwise only modified and related packages will be compiled.

The `-compile_only_kernel` flag can be added to speed up compilation process when you don't need to compile the drivers nor the `libmc.a`.

mklibmc:

Generates the MaRTE OS version of the libc C standard library ('lib/libmc.a'). For full optimized library execute:

```
$ mklibmc -O3
```

mkdrivers:

Automatically run 'make' in each driver directory containing a 'GNUmakefile'. After that, all the objects files found in drivers directories are moved to the objects library `lib/`. (See also chapter "MaRTE OS devices").

mkrtsmarteuc:

Makes the GNAT run-time system for the current architecture.

```
$ mkrtsmarteuc -gnato -g
```

6.- Optimizing MaRTE kernel for speed and size

By default the kernel is optimized for speed (compiled with flags "-gnatn -gnatp -O3") and without any pragma "restrictions". If you desire to impose some restrictions to your application or use a simplified version of the run time with the use of "pragma Restricted_Run_Time" or "pragma Ravenscar" just put a 'gnat.adc' file with the desired pragmas in the `kernel/` directory and force recompilation of everything (flag "`-f`").

For example, for a minimum size kernel a '`gnat.adc`' file with the following pragmas can be used:

```
pragma Ravenscar;
pragma Restrictions (Max_Tasks => 2); -- Or the number of tasks of your application
```

And then recompile MaRTE kernel with the command:

```
$ mkmarte -gnatp -O3
```

The size of the final executable can be reduced by stripping all symbols that are not needed for relocation processing using the command:

```
 $ objcopy -O elf32-i386 --strip-unneeded mprogram
```

7.- Enabling Debug Checks and Messages in the Kernel

Some debugging checks are included in the kernel in the form of "Assert" pragmas. They can be used to detect internal errors inside the kernel before they cause any unexpected behaviour impossible to analyse. To enable these checks the kernel have to be recompiled with the '-gnata' flag:

```
$ mkmarte -gnata -a -f ("-f" force recompilations)
```

Enabling "Assert" pragmas can be useful if you are modifying MaRTE OS to check the consistency of your changes.

The kernel also can be configured to display some (lot) debugging messages on console. They can report all relevant events about context switches, mutexes, signals, timed events, etc. This functionality can be useful for kernel developers in some specific situations. In the general case the number of messages displayed is too huge to be analysed, and the use of the debugger will be preferred. To enable the debugging messages edit the file '`kernel/marte-debug_messages.ads`', set to true some (or all) the boolean constants below the label "General Messages" and recompile the kernel with assertions enabled:

```
$ mkmarte -gnata
```

8.- MaRTE OS Devices

MaRTE OS provides a standard method for installing and using device drivers. This method allow programmers to share their drivers with other people in a simple way.

The implemented model is similar what is used in most UNIX-like operating systems. Applications access devices through "device files" using standard file operations (open, close, write, read, ioctl).

The drivers installation in MaRTE OS is explained bellow. The best way of understanding that process is looking at drivers included in MaRTE OS distribution. The simpler examples are "Demo_Driver_C" and "Demo_Driver_Ada".

Driver code

A driver can be written using both Ada or C programming languages. The code file(s) must be included in a subdirectory of 'drivers/'.

An Ada driver can provide some the following functions (usually all the functions will be provided):

```
function My_Driver_Create return Int;

function My_Driver_Remove return Int;

function My_Driver_Open (Fd   : in File_Descriptor;
                         Mode : in File_Access_Mode)
                        return Int;

function My_Driver_Close (Fd : in File_Descriptor) return Int;

function My_Driver_Read (Fd            : in File_Descriptor;
                         Buffer_Ptr    : in Buffer_Ac;
                         Bytes_To_Read : in Unsigned_32)
                        return Int;

function My_Driver_Write (Fd             : in File_Descriptor;
                          Buffer_Ptr     : in Buffer_Ac;
                          Bytes_To_Write : in Unsigned_32)
                         return Int;

function My_Driver_Ioctl (Fd             : in File_Descriptor;
                          Request        : in Ioctl_Option_Value;
                          Ioctl_Data_Ptr : in Buffer_Ac)
                         return Int;
```

For a C driver the function prototypes are:

```
int my_driver_create (int arg);

int my_driver_remove ();

int my_driver_open (int file_descriptor, int file_access_mode);

int my_driver_close (int file_descriptor);

ssize_t my_driver_read (int file_descriptor, void *buffer, size_t bytes);
```

```
ssize_t my_driver_write (int file_descriptor, void *buffer, size_t bytes);

int my_driver_ioctl (int file_descriptor, int request, void* argp);
```

For C drivers a specification Ada file must be written to make accessible the C functions to MaRTE kernel. To write that file you can use **`drivers/demo_driver_c/demo_driver_c_import.ads`** as a template. Also a **`GNUmakefile`** file should be included in driver's directory since **`mkdrivers`** will automatically run **`make`** in each driver directory containing a **`GNUmakefile`**. After that, all the objects files found in drivers directories are copied to the drivers library **`lib/libdrivers.o`**.

Installing device drivers

Drivers and device files included in the system are registered in **`kernel/k-devices_table.ads`** file. In this file there are two tables:

- **`The_Driver_Table`**: device drivers included in the system. This table associates major numbers with device drivers.
- **`The_Device_Files_Table`**: device files defined in the RAM file system.

In order to include a new driver add and entry to **`The_Driver_Table`**, choosing an unused number. That number will be the major number that identifies your driver in the system. (You can take as example the commented entry for "Demo_Driver_Ada" or "Demo_Driver_C").

Now you must add unless one device file in **`The_Device_Files_Table`**. The name of your device file can be whatever you want ("/dev" prefix is recommended for analogy with UNIX systems).

As mayor number should be chosen the one previously associated with your driver. As minor number can be used whatever you want, this number is useful to distinguish between different files associated with the same major number.

In the field **`Ceiling`** you can put any value within the priorities range used in the system (usually between 0 and 31). To know more about this field read "Mutual exclusion among threads using the same file descriptor".

Installing device drivers for the standard input, output and error

A few special things is necessary to do in case you want to change one of the standard input, output or error devices.

Since we want to use these devices from the very beginning of an application execution, even before the file system has been properly setup, some direct hooks to basic functions of these devices have to be provided in file **`kernel/kernel_console.ads`**.

If you are changing the standard input device, you should modify the following "hook" in **`kernel/kernel_console.ads`**:

- **`Stdin_Direct_Read`**: basic function used to read a character from standard input. It is used to wait a key in order to reboot target computer after application finishes.

If you are changing the standard output device, you should modify the following "hooks" in **`kernel/kernel_console.ads`**:

- **`Stdout_Basic_Init`**: function called by MaRTE OS as application start execution. It can be used to perform a basic initialization of stdout device, just enough to allow **`Stdout_Direct_Write`** to

work (see below). This basic initialization should not do complex things (as installing interrupt handlers, use semaphores, etc.), that complex stuff should be done in the regular `Create` function included in `The_Driver_Table`.
- `Stdout_Direct_Write`: basic function used to write on standard output at the very beginning of execution, before the file system has been properly setup. If no function is provided MaRTE OS initialization messages will not be displayed. It is also used by `printc` to display messages directly to console without passing through the file system.
- `Stdout_End_Of_Kernel_Initialization`: to be called by MaRTE OS at the end of messages corresponding to kernel and devices initialization. It can be used by stdout device to filter out those messages or to display they in a different way (i.e. using a different color).

If you are changing the standard error device, you should modify the following "hooks" in `kernel/kernel_console.ads`:

- `Stderr_Basic_Init`: function called by MaRTE OS as applications start execution. It can be used to perform a basic initialization of stderr device, just enough to allow `Stderr_Direct_Write` to work (see below). This basic initialization should not do complex things (as installing interrupt handlers, use semaphores, etc.), that complex stuff should be done in the regular `Create` function included in `The_Driver_Table`.
- `Stderr_Direct_Write`: basic function used to write on standard error from the very beginning of execution, before the file system has been properly setup. This function is also used by MaRTE OS kernel and `printe` function to print error messages. Its use is more direct and less error prone than using the standard `write` function, so, it is more likely to work after fatal errors.

Restrictions for drivers and interrupt handlers code

There is an important restriction for interrupt handlers code: "they should not call any potentially blocking operation". This includes writing directly to `stderr` or `stdout` devices because a mutex is used to protect every driver in MaRTE OS. So, interrupt handlers are not allowed to use standard input/output functions as `printf` or the package `Ada.Text_IO`.

As alternative, C interrupt handlers must use `printc` (to write on console) and `printe` (to write on standard error). These functions write directly in the devices without passing through the file system. For Ada drivers package `Kernel_Console` must be used instead of `Ada.Text_IO`.

Apart from the explained before for interrupt handlers, there is no restrictions for code to be used inside drivers, so they could use any standard POSIX or Ada interfaces. Anyway, it is recommended for Ada drivers not to use the "POSIX.Ada" interface. There are two reasons for that:

- A driver that use this interface will introduce dependency to the interface even for C applications or Ada applications that do not uses POSIX by themselves. Then footprint is unnecessarily incremented in those cases.
- "POSIX.Ada" interface uses exceptions, so it is not possible to write a exceptions free kernel when using that driver (what could be desirable in some environments).

To avoid this dependency MaRTE OS provides two packages: `Marte_Hardware_Interrupts` and `Marte_Semaphores` to be used inside drivers instead of their POSIX equivalents (`POSIX_Hardware_Interrupts` and `POSIX_Semaphores`).

Compiling kernel and drivers

Each driver with files not written in Ada should include in its directory a 'GNUmakefile' with the rules to compile those files.

MaRTE utilities 'mkkernel' and 'mkdrivers' automatically run 'make' in each driver directory containing a 'GNUmakefile'. After that, all the objects files found in drivers directories are copied to the drivers library 'lib/libdrivers.a'.

'mkkernel', apart from calling 'mkdrivers', makes all MaRTE OS kernel. Then it must be executed after changing drivers and device files definition tables in 'kernel/marte-kernel-devices_table.ads'.

So, the general way of keeping everything updated after a change in device drivers code or definition tables is executing '`mkmarte`' (see also chapter "Compiling MaRTE kernel and libraries"). However, if you have only changed your driver code (without touching 'kernel/k-devices_table.ads') running '`mkdrivers`' would be enough.

It is important to notice that, when using Ada drivers from Ada applications, 'mgnatmake' will do all the work (it is not necessary to execute 'mkmarte' nor 'mkdrivers'). This is because all dependencies in an Ada application are known at compilation time (the Ada library records that information), and then, necessary compilations will be performed assuring the application consistency.

Mutual exclusion among threads using the same file descriptor

It is important to notice that, by default, there is no mutual exclusion among threads using the same device. This fact can be useful, for example, in devices that allows simultaneous write and read operations without any kind of synchronization between them.

Operations available for drivers

MaRTE OS provides some operations to be specifically used inside drivers:

- Get the major and minor numbers
- Get the Path associated to a file descriptor
- Set and reset the POSIX error for a driver operation
- Set and get specific data associated to a file descriptor
- Get file access mode for the file descriptor

These operations are provided by Ada package 'Drivers_Marte'. For more information about that functions read the source file `misc/divers_marte.ads`.

For drivers written in C exists the headers file `include/drivers/drivers_marte.h`.

9.- Using the 'tasks_inspector' Tool

This tool (in experimental phase) allows analysing the execution flow of an application. The relevant scheduling information can be stored or sent from the target to the host computer, to be analysed after the application execution finishes.

The scheduling information is parsed and transformed into a set of files that can serve as input for 'gnuplot', a plotting program with which the information is displayed graphically.

In order to use the 'tasks_inspector' tool first the boolean constant 'Tasks_Inspector_Messages' must be set to 'True' (file 'kernel/marte-debug_messages.ads') and then the kernel have to be recompiled:

```
$ mkmarte -gnata
```

In Linux_Lib architecture, the scheduling information always is stored in the file 'trace_marte.dat', but

in x86 architecture it can be chossen between differnt mechanisms to send the information from the target to the host:

- LOG_CONSOLE: trace data is written in the standard console
- LOG_ETHERNET: trace data is sent as a set of Ethernet frames with a broadcast address and a certain protocol number not allocated for other protocols. This data can be read with a Linux program that uses raw sockets to receive those frames and write them to a log file (`examples/logger/Linux_eth_receive_log.c`).
- LOG_DISK: trace data is written in a file. The name of the file is '/fat/log' in architecture x86 or 'trace_marte.dat' (created in the working directory) in the architecture Linux_Lib

In x86 architecture also the logger mechanism can be specified:

- DIRECT, data is directly written on the log device.
- USE_LOGGER_TASK data is stored in a memory buffer and is written in the log device by an auxiliary task.

Once the scheduling information is available in your host computer, the data can be displayed with 'gnuplot':

```
  $ .../tasks_inspector/tasks_inspector.pl trace_marte.dat
```

10.- Changing MaRTE OS Configuration Parameters

MaRTE OS (if using "Preallocated Resources") is a static system in which the maximum number of resources (i.e., threads, mutexes, thread stacks, number of priority levels, timers, etc.) is set at compilation time.

Also some specific services can be enabled or disabled to improve kernel performance or footprint. If you want to change the default values the kernel have to be recompiled with the new ones.

The configuration parameters are defined as constants in the file **`kernel/marte-configuration_parameters.ads`**. Below each parameter there is a short description of its meaning.

For example the maximum number of tasks (or threads) that is possible to create is set by constant **`Num_User_Tasks_Mx`** (20 by default in x86 architecture). If you want to create more tasks just change this value and recompile the kernel using the `mkmarte` script:

```
$ mkmarte -gnatn -gnatp -O3        (or whatever other compiler options you want)
```

11.- Miscellany of Utilities

In the directories 'misc/' and 'include/misc/' there are some extra utilities that although they do not belong to the kernel or the POSIX interface can be useful for some applications:

Console Management

Complete set of operations for the text console. They allow changing the text attributes, positioning the cursor and cleaning the screen. It also allows changing keyboard behaviour between to modes:

- "Cooked Mode": in this mode line editing is allowed, the textual unit of input is an entire "line" of text, where a "line" is a sequence of characters terminated by the line termination character CR. Thus, characters typed in this mode are not immediately made available to the calling program. They are first buffered to allow the user to perform line editing (erase characters) on them.
- "Raw Mode": every character is made available to the calling program as soon as it is typed, so no line editing is available in this mode.changing the console input configuration.

Files 'misc/console_management.ads', 'misc/console_management.adb' and 'include/misc/console_management.h'.

Execution Load

Allows applications to consume CPU for a given amount of time. Useful in examples that do nothing but show the scheduling of tasks and resources.

In Ada there are two versions of this package:

- Execution_Load_Loop: CPU time consumption is achieved by executing the appropriate number of loop iterations.
- Execution_Load: CPU time consumption is based on CPU time clocks.

Files 'misc/execution_load_loop.ads', 'misc/execution_load_loop.adb', 'misc/execution_load.ads', 'misc/execution_load.adb', 'misc/load.c' and 'include/misc/load.h'.

Console Switcher

Allows applications to change the output system console between the monitor and the serial line.

If you are using QEMU emulator you can redirect again the serial port to stdio to read the messages in the terminal (with scrolling!) with the followin flag: "-serial stdio"

```
$ qemu -hda disk.img -serial stdio
```

Files 'misc/console_switcher.ads' and 'include/drivers/console_switcher.h'.

Contact Address: aldeam@unican.es

MaRTE OS internet site: http://marte.unican.es

Department of Electrónica y Computadores
Group of Computadores y Tiempo Real
University of Cantabria
