com-1274

February 2011

How to run Eurotech/Parvus com-1274 and com-1270 under Linux-2.6

Alessandro Rubini (rubini@gnudd.com)

Introduction

"Com-1270" by Eurotech is a PC/104 board with 8 serial ports, 2 CAN interfaces and a 10Mbit RTL8019 Ethernet port. It has been replaced by "Com-1274", distributed by both Eurotech and Parvus (`parvus.com`) which has 8 serial ports and two CAN interfaces, without Ethernet (which is now on most mother boards, at 100Mbit).

This document explains how to run com-1274 and com-1270, the latter only as far as the multi-serial port and CAN interfaces are concerned.

The information in this document has been gathered on a "CompuLab CM-iGLX" motherboard, contributed by Evan V. Hansen of `3d-p.com`, who also donated the 1274 board and sponsored the associated work, allowing the public release of this documentation.

Any opinions expressed here, however, are only my own, and don't necessarily reflect the ones of any person of company referenced here.

The kernel versions used here are 2.6.24 and 2.6.36, in their vanilla flavors.

1 Configuring the board

The 1274, like the 1270, features configurable addresses and interrupt lines, but configuration must be performed in DoS (which used to stand for "disk operating system" but this millennium is definitely "denial of service").

We'd love the manufacturer to release information about the configuration protocol to do that directly in GNU/Linux, but this hasn't happened, yet – well, on request they release their ASM source files, but turning them to useful code in a sane environment would really take too much.

To configure under DOS, you could run *dosemu* with the released *freedos* image. What I've done is changing the configuration file to open access to a number of I/O ports with this line (it's a single line in `/etc/dosemu/dosemu.conf`):

```
$_ports = "range 0x110,0x1a0 range 0x280,0x2c0 0x80 0x1e0
                  0x300 0x30a 0x213 0xa79 range 0x99,0x9c"
```

With this in place, run `dosemu -s` and run their `setup.exe`. (The ports listed above have been identified by looking at the log file of *dosemu* after activating debugging for I/O accesses: `-D+i`).

Whether or not the address ranges and interrupts work with your own motherboard is a roll of dice as usual. This document assumes you have found the working parameters using some black magic, possibly sending arcane commands to your PCI-to-ISA bridge in order to open access to the relevant areas. Or you may need to configure other chips like the "South Bridge" or "Companion Chip". This document is not concerned about those steps, even though they are usually the most time-consuming. You may want to use `msr.ko` (see Chapter 6 [Other Tools], page 7) to this aim.

2 Booting 2.6.24

To boot 2.6.24 on the embedded system I used for x86 development I used the configuration found as `configs/2.6.24-iGLX.config` (this configuration includes the CAN modules as well). The following points proved important in my specific case:

`CONFIG_SERIAL_8250_PNP`
: The variable must be unset, or this specific motherboard freezes at boot time. To be able to deactivate the configuration item I had to enable `CONFIG_EMBEDDED` and then disable `CONFIG_PNP`.

`SERIAL_8250_NR_UARTS`
: You must set this value high enough, to fit all the serial ports of your system. You should count 4 for the motherboard (but 2 may suffice, according to the motherboard) and 8 for each 8-port board you plug. Setting it to 12 or 20 won't hurt.

`CONFIG_SERIAL_8250_RUNTIME_UARTS`
: This should be the same value as above. I didn't dig about why two options are used. I use 12 as above.

`SERIAL_8250_EXTENDED`
: This item must be set in order to enable shared interrupts for the serial ports.

`SERIAL_8250_SHARE_IRQ`
: This option must be set, as we have 8 ports on a single interrupt line.

`SERIAL_8250_EXAR_ST16C554`
: This option must remain unset, even if you noticed the devices on the 1270/1274 are exactly this chip. Actually, the option selects a driver similar to ours, but which features hardwired I/O addresses and interrupt numbers. The question is only asked if you have `CONFIG_ISA` and `CONFIG_SERIAL_8250_MANY_PORTS` (the former is mandatory, the latter is not).

`SERIAL_8250_DETECT_IRQ`
: Say no (as suggested) as this is unsafe and unneeded. If you run the 1270/1274 you already know everything about its settings, unfortunately, so it's better not to probe for IRQ.

Moreover, please note that I haven't been able to compile with *gcc-4.3*, as the system was reporting a kernel panic at boot time, for a division by zero within `update_wall_time`.

Being unable to find the bug, I finally installed *gcc-4.1* (there is a Debian package in *etch*) and compiled with the following command:

```
make CC=gcc-4.1
```

3 Booting 2.6.36

To boot 2.6.36 I used the configuration found in this package as `configs/2.6.36-iGLX.config` which includes the can modules, like the 2.6.24 configuration described earlier. The following points proved important in my specific case:

`CONFIG_SERIAL_8250_PNP`
: Like for 2.6.24, the variable must be unset for my motherboard freezes at boot time. To unset it you need to enable `CONFIG_EMBEDDED` but you don't need to add `CONFIG_ISA`.

`SERIAL_8250_NR_UARTS`
: You must set this value high enough, but the default is already 32.

`CONFIG_SERIAL_8250_RUNTIME_UARTS`
: This should be high enough. I used 12 (4 uarts on the motherboard and 8 on the 1274. Setting it higher won't raise any problem.

`SERIAL_8250_EXTENDED`
: This item must be set, but in 2.6.36 it is set by default.

`SERIAL_8250_EXAR_ST16C554`
: This option must remain unset. The question is only asked if you have `CONFIG_ISA` and `CONFIG_SERIAL_8250_MANY_PORTS` set.

`SERIAL_8250_SHARE_IRQ`
: This option must be set, but again it is active by default.

`SERIAL_8250_DETECT_IRQ`
: Say no (as suggested) as this is unsafe and unneeded. If you run the 1270/1274 you already know everything about its settings, unfortunately, so it's better not to probe for IRQ.

Unlike with 2.6.24, no specific compiler choice is needed at this time, as far as I know.

4 The Serial Ports

The ports are standard 16550 UART devices mapped in the I/O space. The only peculiar feature is that the interrupt line is shared among them. There is a special I/O port that acts as a multiplexer, so you can look there in order to avoid polling all ports for an interrupt condition.

What is described here applies to both 2.6.24, 2.6.36 amd 2/6/38-rc2. Here we'll assume the following configuration for I/O addresses:

`0x280`
`0x288`
`0x290`
`0x298`
`0x2a0`
`0x2a8`
`0x2b0`
`0x2b8` Uart registers for each of the ports.

`0x2c0` Status port (interrupt multiplexer)

You can configure all the ports as `ttyS` ports using the standard 8250/16550 device driver. To do that you need to load the a module that registers the ports, which is included in this package, in the directory *modules*. The module receives the following parameter on the command line:

`addr` The base I/O address for the first port. Default: 0x280.

`status_addr`
: The base I/O address for the first port. Default: *addr* + 0x40.

`irq` The shared interrupt for this port-set. Default: unset.

To compile the module, enter *./modules/* in this package, set the `LINUX` environment variable to point to your own Linux sources and run *make*. If you installed the header package to develop external headers, you can point `LINUX` to that directory, instead of the complete source tree.

After compilation, you'll have `uart-127x.ko`, which you need to load on your target system, with a command similar to:

```
insmod uart-127x.ko irq=5
```

If you forget to pass the `irq=` argument, you'll get `EINVAL`.

What I get back in *insmod* is this:

```
serial8250.6: ttyS4 at I/O 0x280 (irq = 5) is a 16550A
serial8250.6: ttyS5 at I/O 0x288 (irq = 5) is a 16550A
serial8250.6: ttyS6 at I/O 0x290 (irq = 5) is a 16550A
serial8250.6: ttyS7 at I/O 0x298 (irq = 5) is a 16550A
serial8250.6: ttyS8 at I/O 0x2a0 (irq = 5) is a 16550A
serial8250.6: ttyS9 at I/O 0x2a8 (irq = 5) is a 16550A
serial8250.6: ttyS10 at I/O 0x2b0 (irq = 5) is a 16550A
```

```
serial8250.6: ttyS11 at I/O 0x2b8 (irq = 5) is a 16550A
```

and the ports are all properly working. The fast way to test them is using *stty*, *cat* and *echo* with I/O redirection after looping back one UART to another. However, you must remember to remove echoing or you'll get in trouble (`stty -echo -F /dev/ttySx`).

Please note that removing the module is not fully supported; the problem is not in the module itself but in the 8250 core driver. On unload you will get this message on the console:

```
Device 'serial8250.6' does not have a release() function,
                it is broken and must be fixed.
WARNING: at drivers/base/core.c:107 device_release()
```

5 The CAN Interfaces

5.1 Running Socketcan on 2.6.24

In order to run the *socketcan* device drivers on 2.6.24, I had to back-port the CAN patches from more recent kernels, as well as add the *old* drivers from an *svn* checkout of the *socketcan* project.

I cherry-picked or generated these patches in my *git* tree, and then saved with "*git format-patch*" into *patches/2.6.24* in this package. The patch list is the following:

```
0001-Makefile-added-libgcc.patch
0002--CAN-Allocate-protocol-numbers-for-PF_CAN.patch
0003--CAN-Add-PF_CAN-core-module.patch
0004--CAN-Add-raw-protocol.patch
0005--CAN-Add-broadcast-manager-bcm-protocol.patch
0006--CAN-Add-virtual-CAN-netdevice-driver.patch
0007--CAN-Fix-plain-integer-definitions-in-userspace-he.patch
0008--CAN-Add-missing-Kbuild-entries.patch
0009--CAN-Clean-up-module-auto-loading.patch
0010--CAN-Move-proto_-un-register-out-of-spin-locked.patch
0011--CAN-Minor-clean-ups.patch
0012--CAN-Add-documentation.patch
0013--CAN-Update-documentation-of-struct-sockaddr_can.patch
0014-CAN-use-hrtimers-in-can-bcm-protocol.patch
0015-can-Fix-copy_from_user-results-interpretation.patch
0016-can-Fix-can_send-handling-on-dev_queue_xmit-fai.patch
0017-can-add-sanity-checks.patch
0018-can-Add-documentation-for-virtual-CAN-driver-usage.patch
0019-can-Fix-CAN_-EFF-RTR-_FLAG-handling-in-can_filter.patch
0020-can-omit-received-RTR-frames-for-single-ID-filter-l.patch
0021-socketcan-copied-drivers-net-can-old-and-include-so.patch
0022-socketcan-added-i82527-kconfig-and-makefile.patch
0023-socketcan-add-sja1000-directory-to-obj-m.patch
```

The role of the patches is as follows:

`0001-Makefile-added-libgcc.patch`
: This patch adds `libgcc.a` to the link, or version 2.6.24 won't compile with current compilers (undefined symbols `__umoddi3` and `__udivdi3`). The problem was fixed after 2.6.24.4, but you can apply this set to 2.6.24.6 or later anyways, as the correct fix is orthogonal to this hack.

`0002 .. 0020`
: These are just rebased patches from more recent kernels. The core went in before `2.6.25-rc1`, other patches come from later versions, up to `2.6.28`.

`0021-socketcan-copied-drivers-net-can`
: This patch creates new files copying from *svn* version 1060 of the svn repository, checked out as described later.

`0022 .. 0023`
: These add actual compilation of the old modules. We only need the 82527 for the `com1274` board, but I have a motherboard with an SJA1000, so I added both.

You can apply with either `patch -p1` or `git am`, as you prefer. With these in places, you can activate the CAN modules (they are already activated in the example configuration provided).

If you already booted what you compiled in Chapter 2 [Booting 2.6.24], page 1, you don't need do reboot, as the patch set only adds new modules. You can just run `make oldconfig` and `make modules`, possibly setting `CC=gcc-4.1` to match how you compiled the kernel. The following configuration options should be set in the `oldconfig` step:

```
CONFIG_CAN=m
CONFIG_CAN_RAW=m
CONFIG_CAN_OLD_DRIVERS=m
CONFIG_CAN_SJA1000_OLD=m
CONFIG_CAN_I82527_OLD=m
```

You can safely reply no to other can-related questions.

On the target board, then, you'll need to load the modules for CAN support: `can.ko` and `can_raw.ko`.

5.2 Running Socketcan on 2.6.36

In 2.6.36 *socketcan* is already part of the official kernel but the "old" 82527 is not. Therefore, I rebased the last patches for 2.6.24 listed above and fixed them to compile in 2.6.36. The same patch set works unchanged in 2.6.38-rc2 (the last one I used on my target board, at time of writing).

The patch set, saved by "*git format-patch*" is available in *patches/2.6.36* in this package. The patch list is the following:

```
0001-socketcan-copied-drivers-net-can-old-and-include-so.patch
0002-can-removed-old-headers-from-previous-patch.patch
0003-socket-can-added-i82527-kconfig-and-makefile.patch
0004-socketcan-add-sja1000-directory-to-obj-m.patch
0005-fixes-for-old-i82527-can-driver.patch
0006-fixes-for-old-sja1000-can-driver.patch
```

The role of the patches is as follows:

`0001-socketcan-copied-drivers-net-can`
: This patch corresponds to patch 0021 for 2.6.24 (see above). It copies files from from *svn* version 1060 of the *socketcan* repository. This adds more stuff than needed (see patch 0002 below), but it a simple rebase of what I've been doing for 2.6.24.

`0002-can-removed-old-headers-from-pre`
: After *socketcan* was included in the mainline kernel, some headers were moved to a new position. Patch 0001 however comes from an older repository, so this removes the headers that have been wrongly added.

`0003 .. 0004`
: These two add compilation of the old modules to `Kconfig` and `Makefile`. As above I add SJA1000 as well as I use it for the motherboard CAN device.

`0005 .. 0006`
: The patches deal with changes in the internal CAN API between *svn* revision 1060 and this kernel version.

With these patches in place, you can compile the CAN driver for this Eurotech/Parvus board, as described in the next section.

5.3 The i82527 Driver

The driver for 82527 is not part of the upstream kernel, so I took the one from the *svn* repository of the *socketcan* project. The patches used on the previous sections add the driver to the kernel source tree, but you might prefer compile it externally, following the documentation that's part of the package. I used version 1060 of the *svn* repository (2009-09-18):

```
svn checkout svn://svn.berlios.de/socketcan/trunk -r 1060
```

While the version is somewhat old, it matches my need to run on 2.6.24.

After patching the kernel (see previous sections) and running `make modules`, you'll find the following modules that are relevant to CAN and our chip:

```
net/can/can.ko
net/can/can-raw.ko
drivers/net/can/old/i82527/i82527-pc7io.ko
drivers/net/can/old/i82527/i82527-esdio.ko
drivers/net/can/old/i82527/i82527-iomem.ko
```

The first two modules are the core CAN support, and must be loaded, while the only '527 module we need is `i82527-iomem.ko`, as it is the one that matches how the chip is mounted on the *com1274* board.

To load the driver, we need to specify the memory addresses and interrupt numbers. For the two-port version of the 1274, with my own settings for memory address and interrupt number, I use the following command:

```
insmod i82527-iomem.ko base=0xc8000,0xc8100 irq=11,12 bcr=0x4c,0x4c
```

The *bcr* argument is the *Bus Configuration Register*, whose correct value depends on how the hardware is mounted. The 1274 the correct value is `0x4c`.

```
i82527-iomem driver v0.0.4 (2007-08-03)
i82527-iomem - options [clk 16.000000 MHz] [restart_ms 100ms] [debug 0]
i82527-iomem - options [force_dmc 0] [irq_mode 1]
i82527-iomem: checking for i82527 on address 0xC8000 ...
i82527-iomem: base 0xC00C8000 / irq 11 / speed 500 / btr 0x0 / rx_probe 0 / mo15 sff
i82527-iomem: checking for i82527 on address 0xC8100 ...
i82527-iomem: base 0xC00C8100 / irq 12 / speed 500 / btr 0x0 / rx_probe 0 / mo15 sff
```

If you have problems in accessing I/O memory (for example for a misconfiguration in the PCI-ISA bridge) you'll get this instead:

```
i82527-iomem: checking for i82527 on address 0xC8000 ...
i82527-iomem: probing  0xC00C8000 failed (reset)
i82527-iomem: probably missing controller hardware
```

Later, when you run `ifconfig can1 up` (you have both *can0* and *can1*) you'll get the following messages:

```
can1: calculated best baudrate: 500000 / btr is 0x4067
can1: setting BTR0=40 BTR1=67
can1: i82527: using msg object 15 for SFF reception.
```

5.4 Using Socketcan

If you are new to socketcan, you may want to check my *samplecan* sample application, found within *tools/* in this package. I wrote it while reading `Documentation/networking/can.txt`, that you may find useful.

I always use the *samplecan* tool to check that my setup is working. The typical problems you may find are:

- You forgot to load `can.ko` or `can-raw.ko`. In this case the tool exits 1 as it receives "Address family not supported by protocol".

- You get "NETDEV WATCHDOG: can0: transmit timed out". This means there is no other host on the bus, or your interrupt setup is wrong (again, ISA bridge or other strange stuff in the motherboard.

If it works, the tool sends one packet per second at default speed, and reports any packet it receives from the bus. There are a few options, like listen-only, or changing the CAN interface. Please check the sources.

6 Other Tools

While getting mad in looking for motherboard issues (mainly interrupt routing and access to ISA memory), I wrote three kernel modules that are under *modules* in this package. Both modules make a specific action at load time and then return `EAGAIN`, so you can save unloading them.

I'm not proud of them, as it's all quick-and-dirty stuff, but you may find them useful.

The modules are

`allirq.ko`
: The module registers a shared handler for all system interrupts (from 0 to `NR_IRQS`), reporting any failure through *printk*. Each handler prints a message the first time it is invoked, and at unload time the total number of interrupts received by each handler is reported. I used this to check whether interrupts reached the motherboard or not. Sure, you can put voltage on the bus wires and look at what happens at software level.

`msr.ko`
: This module reads or write a *model-specific register*. You can use it to configure the Geode companion chip. It takes a few arguments on the *insmod* command line: `write=1` (default 0: read the register); `msr=` (number of register); `vall=` (the low 32 bits, used for writing); `valh=` (the high 32 bits, used for writing). The module reports through *printk* what it reads or writes. You need one *insmod* for each register access, but for configuring it's enough.

`remap512.ko`
: The module remaps 512 bytes of I/O memory and prints the contents, reading one byte at a time. The base address for remapping is the module argument `base=`. I used this to make sure that 82527 access worked fine, as `/dev/mem` gave unexpected results. For the record, this gave the same results.

Table of Contents