Ocan 1.00

A device driver for the Intel 82527 CAN controller
February 2006

Alessandro Rubini
Rodolfo Giometti

The ocan Package

This manual documents the 1.00 release of the *ocan* device driver and associated user utilities. The driver currently works on a range of 82527 devices. I chose not to use the device driver by Arnaud Westenberg because I needed to run my device in a very short time, and his work is much more complex so it needed time to study. Besides, I needed to have *poll* and complete documentation.

Nowadays there are several CAN drivers, and this might now be the best choice for you.

1 Introduction

This driver allows user application to access registers and message objects in the 82527 CAN controller by Intel, as well controlling individual configuration items via an higher level abstraction than bit operations on individual registers. This package is the result of changing a 2.2-only driver based on '`/proc`' to a 2.2/2.4 driver based on *ioctl*.

Linux-2.2 was supported until 0.93 included, later versions added 2.6 support and removed 2.2 to keep the code simpler. The package reveals its age nonetheless, and if you read the code you'll find it's not a good 2.6 programming example.

Device special files managed by this driver are both general purpose devices and special purpose devices. The latter include device files bound to a specific message object and device files use to access I/O ports or error information.

In addition to *ioctl*, some driver parameters can be read and written by reading and writing files in */proc/sys/dev/ocan/*.

2 Contributed Code

As of version 0.90, the *ocan* package includes two major external contributions, in the '`contrib`' directory.

One is a patch by Philippe Gagnon, which adds support for a new hardware device and changes data management in a few ways. The other is a port to *RTAI* by Seb James called *rtcan*. The former is going to be integrated in the official *ocan* source code, but integration is not as immediate as it can seem, as documentation must be produced and some design details must be dealt with. On the other hand, the *rtcan* tarball is no more distributed with *ocan* and the *rtcan* '`README`' now points to the project's home page. This will allow users to get the latest version of the package.

Both contributions are worth looking at, though. More details are available in the '`README`' file associated to each contributed package.

3 Installing the Package

3.1 Compiling the Package

To compile the driver and associated utilities, issue `make`. If needed, issue `make install`. The driver is meant to work with both Linux-2.4 and Linux-2.6.

Please note that the installation part is the least tested corner of the package. Also, due to kernel-version issues, you might want to check the `Makefile` and edit a few variables.

If the kernel source for the version you compile against is not installed as '`/usr/src/linux/`', please specify its location in the command line of '`make`'. For example, this is how I compile for version 2.4 of the kernel:

```
make LINUX=/usr/src/linux-2.4
```

You can set `LINUX` in the environment, if you prefer that to the command line of '`make`'.

Installation of the package places it by default under '`/usr/local`', while the module is installed in '`/lib/modules/kernel-version/misc`'. Under 2.6, installation is performed by the kernel Makefile and the module will be '`/lib/modules/kernel-version/extra`'. You can specify `DESTDIR` and/or `prefix`. See the '`Makefile`' for details.

3.2 Loading the Kernel Module

The '`ocan_load`' script deals with loading the device driver. It looks for the module in the current directory and then in the usual places. The script prints the location where the module is found, so you can at least know what is going on. You are expected to edit the script to configure permission and owner/group of the device files being created (by default anyone can access the CAN controller).

I expect most users will simply run '`./ocan_load`' from the top-level directory of the package, although it can be installed to '`/usr/local`' along with other tools.

Any arguments that you pass to the ocan_load command line are passed to '`insmod`'. This allows load-time configuration of internal variables and, if you feel so inclined, to specify the `-f` option to force loading the module for a different kernel version (see '`insmod`' documentation

You can pass a few command-line parameters to the device driver. The following table lists all of them.

`major`

The integer parameter tells which major number should be used by this driver. By default the major number is dynamically allocated (and you'll most likely get 254 from such allocation).

`base`

The parameter lists the hardware addresses where CAN controllers are to be found. Each item in the array defaults to 0 and you must explicitly state the base addresses of your devices (either in I/O or in memory space). The array can host 8 base addresses, as the driver can handle up to 8 CAN controllers.

`irq`

Like *base*, this lists the IRQ numbers, and all of them defaults to 0. In order for one CAN controller to be used by the driver, both the base address and the IRQ number must be set to non-zero values.

`type`

The *type* is an array of integer values, used to select one of the supported hardware abstraction layers. Its use may be optional according to the hardware you run; the driver can tell I/O-mapped controllers from memory-mapped controllers but some base addresses can refer to more than one supported device. See Section 3.4 [Device Types], page 3.

`remap_isaio`

This parameter is needed to force use of memory-mapped access for ISA (PC104) I/O based devices. This is not usually needed, but some port may not properly define *readb*/*writeb* to be compatible with ISA devices plugged in the platform.

You don't usually need to set this argument, but it is mandatory, for example, on the TS-7200 embedded platform (EP9301 ARM CPU), where you need to specify `remap_isaio=0x11e00000` to access any PC104 ocan device.

`use_bottom_half`
`use_shared_irq`
`use_exclusive_irq`
`use_irq_stamps`
`use_complete_log`
`cpu_khz`

These global parameters can be set both at load time and at run time though *sysctl*. Such items are described in Section 8.1 [Global sysctl Parameters], page 10.

For example, this commands activates four CAN controllers: two of them on a PC-ECAN ISA device (I/O mapped) and two of them on the Eurotech memory-mapped device:

```
./ocan_load type=3,3 base=0x380,0x382,0xd8000,0xd8100 irq=5,10,6,9
```

The following command, instead, loads the driver for a single MSMCAN device set up with its factory defaults:

```
./ocan_load type=4 base=0x340 irq=9
```

The following one enforces factory defaults for the Kontron device:

```
./ocan_load type=5 base=0x2000 irq=5
```

3.3 Unloading the Module

The *ocan_unload* script takes care of unloading the driver and cleaning up the `/dev/` directory.

3.4 Device Types

The driver uses a simple hardware abstraction layer in order to support different device types. Each type is identified by a number; a type of 0 (the default) specifies automatic detection of the device type. Automatic detection works by asking to each supported type whether or not the base address specified can be its own. If more than one device type replies affirmatively, the driver refuses to load and prints an error message. Thus, autodetection can' work with ISA I/O-based devices, since the package supports several such device types.

Currently, the following device types are supported:

1: isamem

This is a memory-mapped mounting of the 527 chip, in the ISA I/O memory space. The base address for this device type must be a multiple of 256 and must be in the range from 0xa0000 to 0xef000. The type number, 1, has been chosen because it resembles the I of "isamem". These devices include Eurotech motherboards, Eurotech CAN cards and their COM1270 (one Ethernet, 8 serial ports, two 527 controllers). I thank them for donating hardware samples.

2: isa-io

This is a generic driver for I/O mapped devices in the ISA address space. If your device uses an address register at the *base* I/O port and a data port at port *base+1*, then you can use this driver. I currently have no such device to test.

3: pcecan

The PC-ECAN device supports up to two 527 controllers mapped in I/O memory, using ISA. Each pair uses a range of 8 I/O ports and uses an address-register/data-register pair to access I527 registers and a separate port to reset one or both controllers. This driver is like *isa-io* above, with additional support for the reset bits. The type number, 3, resembles a flipped E (as in ECAN).

4: msmcan

The MSMCAN device is manufactured by Digital Logic AG. It's an I/O mapped device with the address register at port *base+1* and the data register at port *base*. My specimen of the device has been contributed by Clay Barclay of Harris Corporation.

5: isa-dio

This back-end supports devices whose registers are mapped to a continuous range of 256 I/O ports (the name stands for ISA direct I/O). This applies for example to the MOPS devices built by Kontron, who sent me a complimentary device.

6: gea

The PC/104 board developed by GEA-Automotive is similar to PC-ECAN, but uses 16 I/O ports. Once again, the number, 6, is chosen because is resembles G as in GEA. This device, though, has extra functionalities besides carrying one or two 527 chips. See Section 8.2 [GEA sysctl Files], page 11.

8: TQM8xxL

TQ manufactures embedded PowerPC modules with 82527 devices on-board. This device type supports such modules (code contributed by Wolfgang Grandegger). This device type is only compiled in if you build for the PowerPC platform (while the others are currently only compiled in if you are not compiling for PowerPC).

4 Device Special Files

The script '`ocan_load`' creates *ocan* special files within the directory '`/dev/ocan`'. The various 82527 controllers are identified by a lowercase letters, starting from `a`. The following device files are created for the first controller being used by the driver:

`/dev/ocan/a`
: The general entry point to the controller. While *read* and *write* can't be used on this device, it implements all the *ioctl* commands and *poll* (*select*). The *ioctl* commands use to read and write CAN messages behave like the *read* and *write* system calls, in that they respect `O_NONBLOCK` and `O_SYNC`.

`/dev/ocan/a1 .. /dev/ocan/a15`
: Specific entry points to access one of the 82527 message buffers. These files support *read* and *write*, but the result of such operations is undefined unless the message object has first been configured. About the exact semantics of *read* and *write* see below, in Chapter 5 [Device Methods], page 4. File '`a15`' is read-only. **Warning:** not yet implemented.

`/dev/ocan/a-error`
: The error file is used to report errors to user space. The first *read* after opening the file returns the current status byte. Successive reads will block until a status interrupt notifies the driver about a status-change event. Each *read* returns one byte, the current status byte. The application can use *select*/*poll* to know when *read* would or would not block. See Section 10.2 [Demonstrating Error Management], page 17.

`/dev/ocan/a-io1` `/dev/ocan/a-io2`
: The devices access the two 8-bit I/O ports. The ports are called 1 and 2 (instead of 0 and 1) to be consistent with Intel documentation about their CAN controller. Each *read*() or *write*() system call will transfer a single byte of information from/to the hardware device. Configuration for the ports (i.e., whether each bit is set up as input or output) is performed via *ioctl*(). The implementation of *ioctl* is shared by all devices, and offers commands for direct input/output too (see Chapter 6 [Ioctl], page 5); these devices are therefore mainly meant for shell scripting. The hardware implementation for the specific device may or may not force a default at hardware initialization and reset; this currently only happens for the GEA device.

5 Device Methods

A device driver is plugged in the system by means of a table of "device operations" (or methods) that it takes care of. The implementation (or lack of) used in the *ocan* grabber is described below.

open
: When *open* is first called for a device, its interrupt line is enabled. The method also makes a few consistency checks (for example, '`/dev/ocan/a15`' cannot be opened for writing as the message object is receive-only at hardware level). Single-open behavior will be implemented for message-object device files so different processes can't make a mess of their data.

  Note that nothing is initialized at open time. This specific behavior has been chosen to allow configuring the controller via '`ocan_control`' (see Section 9.1 [ocan_control], page 12) and then reading and writing from a different process.

release
: When a file descriptor is last closed, any message object owned by that file descriptor is freed, even if it is currently transmitting. This means that you should not close a file before your transmission is over; the behavior allows recovery of message objects when transmission can't be performed due to bus errors and no IRQ is reported for the transmitting messages. Also, the release method disables interrupt reporting when a CAN controller is closed for the last time. No reset is performed, for the same reason why *open* doesn't initialize the device.

read

The *read* system call should be implemented for special files that refer to a specific message object, but it currently is not.

The system call is available for error management and digital I/O, though.

write

The *write* system call should be implemented for special files that refer to a specific message object but it currently is not.

The system call is implemented for digital I/O, though.

poll

The *poll* method is the back-end of both *poll* and *select*. It is implemented and it can be associated with either *read*/*write* or *ioctl*. When a file calls *poll*, the list of message objects is scanned and if the file owns more than one message object, the result will be inclusive of all devices. The application developer is strongly invited to only access one message object from each open file. A future enhancement will restrict special files that refer to a specific message object to only work with that message object (even if *ioctl* commands allow to specify which message object to use), in order to get better performance out of *poll*.

ioctl

Most of the operations that can be performed on the device are available via *ioctl*. The list of *ioctl* commands is available in Chapter 6 [Ioctl], page 5.

fasync

The method is not currently implemented but I plan to add it.

llseek

Returns `ESPIPE`, since seeking a CAN device is not possible.

flush
fsync

The *flush* method is called when a file is closed (even if not the last close, while *release* is only called on last close). *fsync* is the back-end of the *fsync* system call. Both will be implemented (and will do the same) but currently aren't.

mmap

Not implemented.

6 Ioctl

The *ioctl* method is used to act on the device, both at low level (i.e., reading and writing registers) and at higher level (i.e., reading and writing messages and configuration parameters ignoring the bit position on the physical device).

I implemented no kind of protection on the device: you must protect it using the normal Unix permission/owner techniques (however, by default the devices are open to everyone, feel free to change *ocan_load* if needed). It might make sense to implement some access restriction in the device, but I'm not sure about it.

The following list describes all the commands currently implemented in the driver and the ones I plan to implement. The type of the third argument (if any) is specified in parenthesis. All of the commands can also be issued by means of the *ocan_control* application (see Section 9.1 [ocan_control], page 12).

6.1 Low-level Commands

`OCAN_IOCRESET (no third argument)`
: The command can be used to reset the CAN controller. It uses the *h_reset* method of the hardware abstraction layer, and fails if that method fails. Thus, if no hardware reset is available `EOPNOTSUPP` is returned. Hardware reset does not imply software reset (`IOCSOFTRESET`).

`OCAN_IOCSOFTRESET (no third argument)`
: The command re-registers the interrupt handler and asks the driver to do software reset. Unregistering and re-registering the interrupt handler can help with some very rare hardware problem, although the same effect can be achived by closing and reopening the device, that is inpractical when several processes are using the device at the same time. Then, the initialization sequence for the CAN controller is called and all flags and buffers associated to device objects are reset. If any message object is marked as busy, it is released. An object can be stuck busy if an application requested transmission but a fatal error happened and no interrupt has been reported for the successful transmission (still, you can know about errors by reading the error device). Software reset does not imply hardware reset (`IOCRESET`).

`OCAN_IOCREADREG (struct ocan_reg * argument)`
`OCAN_IOCWRITEREG (struct ocan_reg * argument)`
: The commands can be used to read and write individual device registers. The structure `struct ocan_reg` has two 8-bit fields: `reg` and `val`. The user must fill one or both of the fields; writing has no effect on the data structure, while reading sets `val`.

`OCAN_IOCREADMULTI (struct ocan_multireg * argument)`
`OCAN_IOCWRITEMULTI (struct ocan_multireg * argument)`
: To avoid excessive system-call overhead, these commands can be used to read and write multiple consecutive registers up to a maximum of `OCAN_MULTIREG_MAX`, currently 16. Please note that you can't set `OCAN_MULTIREG_MAX` to arbitrary values and recompile; you'll also need to check `IOC_BUFSIZE` in the implementation of *ioctl*.

`OCAN_IOCREADALL (struct ocan_allregs * argument)`
: This commands reads all 256 registers. The third argument can be a simple pointer to an area of 256 bytes, even though I had to define a data structure in order to define the command.

6.2 Higher-level Commands

`OCAN_IOCGETMASKS (struct ocan_masks * argument)`
`OCAN_IOCSETMASKS (struct ocan_masks * argument)`
: The commands are used to read and write the two global masks and the message-15 mask of the CAN controller. The masks are 16 bits or 32 bits long, and the least significant bits are ignored (refer to the 82527 data sheet for details about mask layout).

`OCAN_IOCGETTIMES (struct ocan_times * argument)`
`OCAN_IOCSETTIMES (struct ocan_times * argument)`
: The commands are used to read and write the current timing configuration of the CAN controller. The fields of the data structure include the various clock pre-scalers, clock output control and bit length control. Arguments in the data structure are as wide as the bit fields in 82527 registers, but shifted to the least significant bits. If any higher bit is set when changing device configuration the result is undefined.

`OCAN_IOCGETBUSCONF (struct ocan_reg * argument)`
`OCAN_IOCSETBUSCONF (struct ocan_reg * argument)`
: The commands read and write the bus configuration register (0x2f) of the CAN controller. Strictly speaking, a `IOCREADREG` call could simply be used in place of `IOCGETBUSCONF`; however to change register 0x2f you first need to set the "change configuration enable" bit, so both commands are implemented for symmetry. Only the *val* field of `struct ocan_reg` is used (the *reg* field is ignored and is not changed).

`OCAN_IOCWRITEMSG (struct ocan_msg * argument)`
`OCAN_IOCSETUPMSG (struct ocan_msg * argument)`
`OCAN_IOCTXMSG (unsigned long argument)`

The `IOCWRITEMSG` command configures a message and transmits it, while `IOCSETUPMSG` and `IOCTXMSG` allow the task to be slit in two steps, since setting up a message object is a much longer task than actual transmission. Moreover, a message could be set up once and transmitted several times. All configuration information is hosted in `struct ocan_msg`, but you only need to pass the message-object number in order to transmit a configured message.

Both configuring a message and transmitting it are potentially blocking calls, to prevent any interference with an already ongoing transmission. Although the delay will be small, as no more than one hardware transmission can be pending, you can use a non-blocking file descriptor and rely on `EAGAIN` to be returned. The message object can be busy only due to a previous `TXMSG` or `WRITEMSG` issued through the same file descriptor.

For details about the transmission mechanism, see Chapter 7 [Sending and Receiving Packets], page 8.

`OCAN_IOCRXMSG (struct ocan_msg * argument)`

The command receives a message from the specified message objects. The call is blocking if no message has already been received, unless `O_NONBLOCK` is set for the current file. When the command blocks if behaves like a blocking *read*.

For details about the transmission mechanism, see Chapter 7 [Sending and Receiving Packets], page 8.

`OCAN_IOCPEEKMSG (struct ocan_msg * argument)`

The command peeks in the queue of received messages and extracts one with the specified identifiers. It can only be used on a message object opened for reading and does never block (since it isn't easy to match this with *select*). The caller must fill the *id* field of the data structure and the `I527_XTD` bit in the *config* field; the driver will return a packet matching *id* and `XTD`, or -1 if no matching packet is there (with `errno` set to `EAGAIN`). The packet is removed from the queue, unless `OCAN_MSGFLAG_PEEKONLY` is set.

`OCAN_IOCRELEASEMSG (unsigned long argument)`

The command is used to declare the current process (actually, the current file) is not interested in the message object any more. No more packets will be received by this message after it is released and before another file declares interest in it.

For details about message ownership, see Section 7.1 [Message Ownership], page 8.

`OCAN_IOCWRITEQ (struct ocan_msg * argument)`

The command enqueues messages (instead of writing one of them to hardware registers like `IOCWRITEMSG` does). Whenever the queue is full, the process is put asleep (unless it is non-blocking, in that case `EAGAIN` is returned). Sleeping processes are awaken when at least half of the queue is empty. Technical details about how locking and sleeping is performed are available in the text file '`README.locks`' within the source code.

`OCAN_IOCGIRQCOUNT (unsigned long *argument)`

The command returns to user space the number of interrupts received from this driver after it has been loaded. If the interrupt handler is shared, only valid interrupts for this device are reported. This is therefore different from reading '`/proc/interrupts`', and sometimes useful information.

`OCAN_IOCREQSIGNAL (unsigned long * argument)`

The command requests signal notification on error interrupts. The argument points to the signal that the current process wants to receive on error. If the signal is 0, then the current process is removed from the list of processes being notified. If the signal specified is too high, `EINVAL` is returned; if the table of processes requesting a signal is full, `EBUSY` is returned. The default length of the table is 4 processes.

When the file is last closed, signal notification is removed. Please note that if signal notification is activated and then the file is passed to a child via *fork*, the signal will

still be delivered to the parent process, as the *pid* is recorded when *ioctl* is invoked. This might be a security issue, but it is not because *ioctl* can only be invoked by the superuser.

`OCAN_IOCINPUT (struct ocan_reg * argument)`
`OCAN_IOCOUTPUT (struct ocan_reg * argument)`
`OCAN_IOCIOCFG (struct ocan_reg * argument)`

The three commands are used to read or write an 8-bit register in the 82527 controller, and are functionally equivalent to `IOCREADREG` and `IOCWRITEREG`. The commands, therefore, only exist to ease the user (who can avoid using register numbers to act on I/O ports). The *reg* field of the structure must be either 1 or 2. `IOCINPUT` fills the *val* field, the other two commands copy the *val* field to hardware registers.

Configuring an I/O port means setting what bits are used as input and what bits are used as output. Bits set to 1 configure the pin as output, bits set to 0 configure it as input.

7 Sending and Receiving Packets

Transmission and reception of packets is performed via either *ioctl* or *read*/*write* (although the latter method is not implemented in early versions of the driver).

Independently of the interface chosen by the application, internally everything is implemented by assigning ownership of message objects to the file using them and by transferring information using the `ocan_msg` data structure.

7.1 Message Ownership

In order for a message object to be used in transferring data packets, it must be owned by a file. This choice allows control of whether a message object is configured or not, and some form of access control for message objects. Since message ownership is associated to the file and not to the process that opened it, two clones of the same file share their ownership (this happens when *dup*(2) or *fork*(2) are used). Similarly, ownership is preserved across *fork*(2)/*exec*(2).

A file becomes the owner of a message object if the device special file being opened is specific to a message object (for example, '`/dev/ocan/a4`'), or when one of `IOCSETUPMSG` or `IOCWRITEMSG` is issued via *ioctl* (in this case, independently of the device file being opened). A file trying to access a message object owned by another file receives an error of `EBUSY` (either on *open*(2) or on *ioctl*(2)).

When a file is the owner of a message object it should configure it before using it for message transmission (i.e., it should set a CAN identifier, choose whether to use standard or extended id's, select whether remote frames must be used or not). However, configuring the file is not mandatory.

Only the owner of a message object can send or receive files through that object. Transmission and reception can both be performed via *ioctl*, while a file that opened a message-specific device ('`/dev/ocan/b12`' or similar can only read or write until the file is closed. To enforce that, `open(O_RDWR)` is not allowed on such device files.

If a file tries to issue `IOCTXMSG` or `IOCRXMSG` without being the owner of the message object, `EPERM` ("Operation not permitted") is returned. Neither command checks message flags

Ownership is released either by closing the file or by issuing `IOCRELEASEMSG`. Please note that in either case the message object is released even if it is currently transmitting, so you should use `IOCRELEASEMSG` (and *close*) with care. A message is released even if it is transmitting in order to recover it from the "hardware busy" status in case errors happen (i.e., when transmission was requested but no interrupt reported it as successfully completed).

The rationale behind this design is in allowing use by either compiled applications or shell scripts while preventing concurrent access. Use by a shell script means that configuration of the message object and packet transfer must be performed by different processes. Thus, message configuration (id, extended flag, remote flag) survives a change in message ownership.

7.2 Use of ocan_msg

This section describes how the fields of `struct ocan_msg` are used in the device driver and *ioctl* commands. The structure is defined in '`ocan.h`'.

In the following description, `IOCWRITEMSG` does not appear because it behaves exactly like `IOCSETUPMSG`.

`__u32 id`
: The CAN identifier for this message object. The user must set this field in `IOCSETUPMSG`, the driver returns it in `IOCRXMSG`. The field is laid out like 82527 registers: for standard messages only the top 11 bits are meaningful, for extended messages only the top 29 bits are meaningful.

`__u8 msgobj`
: The number of the message object. The field must be set by the application both in `IOCSETUPMSG` and `IOCRXMSG`.

`__u8 flags`
: Either `OCAN_MSGFLAG_READ` or `OCAN_MSGFLAG_WRITE`. It must be `OCAN_MSGFLAG_WRITE` when calling `IOCWRITEMSG`. The driver sets this field to `OCAN_MSGFLAG_READ` when a packet is received (and returned to user space).

  The flag `OCAN_MSGFLAG_PEEKONLY` is used by `IOCPEEKMSG` to peek in the queue of received messages without removing data from the queue itself.

`__u16 control`
: Currently unused. Still, it's useful as alignment.

`__u8 data[8]`
: Data bytes. They must be filled before calling `IOCSETUPMSG` if the `OCAN_MSGFLAG_WRITE` is set. The driver fills it on `IOCRXMSG`.

`__u8 config`
: This field is laid out like the *config* hardware register associated to the message object. When `IOCSETUPMSG` is called, the driver only uses the *extended* bit (since data length is taken from the *dlc* field and the direction bit is derived from the *flags* field. After `IOCRXMSG` all bits of the field are valid.

`__u8 dlc`
: Data length counter. The field must be set by the application before calling `IOCSETUPMSG` with `OCAN_MSGFLAG_WRITE` and is ignored when setting up a message for reading. When receiving a message the driver sets it to the number of data bytes received (also available from the high nibble of *config*.

`__u16 error`
: A bit-mask of error flags. The flag `OCAN_ERROR_MSGLST` is set for a message when the "message lost" flag is set in hardware; this means a message has been lost before this one. The flag `OCAN_ERROR_OVRFLW` is set in a message when the following message has been discarded by software because the internal buffer overflowed; the number of pending messages is defined in '`ocan.h`' as `OCAN_BUFSIZE`, and it's currently not configurable after compilation.

7.3 Use of O_NONBLOCK and O_SYNC

The driver honors the `O_NONBLOCK` file flag when reading a message.

If no message is available, the *ioctl* or *read* system calls will either block or return `EAGAIN` according to whether or not `O_NONBLOCK` is set in the file flags.

When no message is available, the process can use the *select* or *poll* system calls to wait for a message (the calls work whether or not the file is non-blocking, just like they work with *read*).

If the file that calls *select* or *poll* owns more than one message object, the file descriptor will be reported as readable when at least one of the message objects has new data. Thus, if you use *select* or *poll* while owning more than one buffer object you'll need to set `O_NONBLOCK` and

try to receive from the various objects you own. The suggested approach to read from several message objects is using a different file descriptor for each of them.

`O_SYNC` is not currently supported but will be.

8 /proc/sys/dev/ocan

The *sysctl* interface is used for setting configuration variables for run-time behaviour and for device-specific extended features. The latter is currently only used for GEA devices.

Later versions will allow reading and writing message identifiers and masks vie '`/proc/sys`', but this is currently not supported.

All files in '`/proc/sys`' can be read and written from user space, but you need superuser privileges to change configuration variables.

8.1 Global Configuration

The following global parameters can be read and written via either '`/proc`' or *sysctl*. For the latter tool, magic numbers appear in '`ocan.h`'.

All values are boolean, and can't be set to anything but 0 or 1. When the values are modified, they have immediate effect unless otherwise noted.

Default values are compiled-in, and match the behaviour of previous releases of *ocan*, but new command line parameters for *insmod* have been introduced so you can change the default values without passing through *sysctl* or '`/proc`'.

In the following table, the '`/proc`' name appears together with the command line parameter in parenthesis.

`bh (use_bottom_half)`
: Enable split interrupt handling (i.e., use of bottom halves, whence the name. This is set by default for the i386 platform, unset for other platforms. If bottom halves are disabled, all interrupt processing occurs in interrupt context, without giving control back to Linux after the interrupt source is acknowledged. You might want to disable bottom halves if you need to receive several back-to-back packets, as bottom-half processing can be slightly delayed from hardware interrupt management if other asynchronous events are pending or other interrupts are received.

  Since most supported boards are ISA (PC104) devices, the bottom-half code was designed with them in mind and proved not to work with level-triggered interrupts, that's why it's disabled for non-x86 platforms. This misbehaviour is a known bug and will be fixed as time permits.

`shirq (use_shared_irq)`
: Register a shared interrupt handler. This is set by default and is used when the device is first opened, as the interrupt line is released back to Linux when the device is closed by all processes that were using it. Usually registering a shared handler is the right thing to do, because a shared interrupt is better than no interrupt at all. Sometimes, though, you might want to prevent a shared handler from being installed.

`exclirq (use_exclusive_irq)`
: Register an exclusive interrupt handler. This is clear by default and is used when the device is first opened, as the interrupt line is released back to Linux when the device is closed by all processes that were using it. An "exclusive" interrupt handler is one that keeps all other interrupts disabled while running (i.e., it uses the `SA_INTERRUPT` flag when calling *request_irq*). You might want to keep other interrupts disabled if you are very concerned about speed in CAN processing.

`irqstamp (use_irq_stamps)`
: Request to print time stamps for hardware interrupts and bottom half processing (not set by default). When stamping is activated, the driver will print the delay, in microseconds, of bottom-half processing from the hardware interrupt as well as

the delay of each hardware interrupt from the previous one. This feature uses the "time stamp counter" (TSC) found on some CPU cores and the CPU frequency set forth by the user (see below) to convert the delays into microseconds. If you run under 2.2, no *cpu_khz* exists and the driver performs no conversion at all, so delays are printed as raw TSC counts. If your CPU has not time stamp counter all delays will appear as 0. If the counter of your CPU runs at a different pace than the CPU core, you'll need to look for `cpu_khz` in *ocan* code and fix the calculation (this only happens with some non-x86 architectures).

`buslog (use_complete_log)`
: This parameter should enable complete logging of transmitted/received packets, but is not currently implemented.

`cpu_kHz (cpu_kHz)`
: The frequency of the CPU, used in making time calculations using the time stamp counter. While the kernel has a global variable called `cpu_khz` (lowercase), that value is not exported to modules, so *ocan* need to know otherwise. The user can specify the CPU frequency at module load time or by writing to '`/proc/sys/dev/ocan/cpu_kHz`'. Please note that you can write any number in there, as long as it's greater than 10000. Time measures, enabled by `irqstamp` as described above, will change according to the assumed CPU frequency. The default value for core frequency, if not specified, is 100MHz.

8.2 GEA sysctl Files

The device manufactured by *GEA Automotive* supports an internal 16-bit counter (using leading edges of one of the digital inputs) and a timer interrupt, that fires with a configurable period, multiple of a millisecond.

If you install more than one card (for example to have more than two CAN busses), you'll still only access the timer and counter on the first card. Similarly, there's not support to actually use the timer as a timing source for user-space processes, something that might be very useful (for example implementing something on the lines of '`/dev/rtc`', but exploiting the multiple-of-a-millisecond time interval offered here.

Both problems will be solved as soon as I implement device-specific minor numbers (so you'll be able to use *poll* and *ioctl* with specific commands, blocking and non-blocking I/O, and so on).

Currently, the following three files implement the timer and the counter functionality, all of three live in '`/proc/sys/dev/ocan`' and can be accessed either via text I/O and via the *sysctl* system call. I'll add a demonstration program to use the *sysctl* binary interface.

counter
: The counter is a 32-bit number. The hardware counter is a read-only 16-bit value, but it's *overflow* interrupt is used to extend the width of the count. The content of this file is reset to 0 at module load time.

  The counter is not reset when '`/dev/ocan/a`' is opened for the first time, since its overflow interrupt will be handled even when the device is not in use. Applications will therefore need to keep track of the initial value of '`counter`'.

  The '`counter`' file is read-only

timerstep
: The delay between timer interrupts. The value represents the number of milliseconds between successive timer interrupts. A value of 0 disables the interrupt altogether; any value in the range 1-255 activates the interrupt. The file is read-write.

timer
: The number of timer interrupts since the module has been loaded. The values is not reset at device open, since its interrupt are handled even when the CAN device is not in use. Thus, applications using the timer will need to keep track of the initial value of the timer. Please note also that no locking or protection is implemented on the value of '`timerstep`': any *root* process can change the value and no application gets notified.

9 User Space Tools

The package includes a few user-space programs to act on the device.

9.1 ocan_control

'`ocan_control`' is a front-end to the *ioctl* system call. In general, all implemented *ioctl* commands are also available from '`ocan_control`'. The program reads commands from either the command line or standard input.

By default the program acts on '`/dev/ocan/a`', but you can specify a device pathname as either the first or last command line argument. Alternatively, you can set `OCANDEVICE` in the environment to select a default device name. An additional argument of `-t` requires terse (i.e., non-verbose) operation; due to the simple-minded implementation, the option must follow the device name if you chose to specify it as the first (rather than last) argument, passing arguments with the option before the device name won't work as expected. If the only remaining argument is `-`, then the commands to issue are read from standard input, otherwise the commands are read from the command line.

If you call the program without arguments, it prints the list of supported command. Each command is invoked followed by a number of integer arguments (read as `"%i"`, so hex numbers can be passed using `0x` as prefix).

```
fino% ./ocan_control
Use: ./ocan_control [command [arg] ...]
The device used is /dev/ocan/a, or $OCANDEVICE
Commands are:
  reset           (OCAN_IOCRESET       ,  0 numeric args)
  softreset       (OCAN_IOCSOFTRESET   ,  0 numeric args)
  readreg         (OCAN_IOCREADREG     ,  1 numeric arg)
  writereg        (OCAN_IOCWRITEREG    ,  2 numeric args)
  readmulti       (OCAN_IOCREADMULTI   ,  2 numeric args)
  writemulti      (OCAN_IOCWRITEMULTI  ,  3 to 18 numeric args)
  readall         (OCAN_IOCREADALL     ,  0 numeric args)
  getmasks        (OCAN_IOCGETMASKS    ,  0 numeric args)
  setmasks        (OCAN_IOCSETMASKS    ,  3 numeric args)
  gettimes        (OCAN_IOCGETTIMES    ,  0 numeric args)
  settimes        (OCAN_IOCSETTIMES    ,  9 numeric args)
  set3times       (OCAN_IOCSETTIMES    ,  3 numeric args)
  getbusc         (OCAN_IOCGETBUSCONF  ,  0 numeric args)
  setbusc         (OCAN_IOCSETBUSCONF  ,  1 numeric arg)
  setupmsg        (OCAN_IOCSETUPMSG    ,  2 to 10 numeric args)
  writemsg        (OCAN_IOCWRITEMSG    ,  3 to 10 numeric args)
  writeq          (OCAN_IOCWRITEQ      ,  3 to 10 numeric args)
  txmsg           (OCAN_IOCTXMSG       ,  1 numeric arg)
  rxmsg           (OCAN_IOCRXMSG       ,  1 numeric arg)
  peekmsg         (OCAN_IOCPEEKMSG     ,  2 to  3 numeric args)
  input           (OCAN_IOCINPUT       ,  1 numeric arg)
  output          (OCAN_IOCOUTPUT      ,  2 numeric args)
  iocfg           (OCAN_IOCIOCFG       ,  2 numeric args)
  releasemsg      (OCAN_IOCRXMSG       ,  1 numeric arg)
```

Most of the textual commands map directly to the *ioctl* commands, but not all of them. Moreover, the *help* command is available: it prints the same message as show above unless followed by a command name, in that case it prints more detailed information about the specific command.

The *getmasks* and *setmasks* use a different field order than the fields in the data structure (since the data structure is ordered to maximize alignment, and the textual command is in logical order). The three masks are, in order, standard global mask, extended global mask, message-15 mask.

The *gettimes* and *settimes* use their numeric arguments in the same order; to know their order the user is invited to use *gettimes* first. The simplified *set3times* commands only sets the baud rate pre-scaler, the TSEG1 and the TSEG2 values, in this order, preserving the other timing values. This allows to set the bit rate without affecting clock-out timings and internal clocks.

A sample session with the command looks like this:

```
fino.root# ./ocan_control readreg 0 readmulti 0 3 readall
ioctl("/dev/ocan/a", OCAN_IOCREADREG, ...) = 0x00 = 0x0a (ret 0)
ioctl("/dev/ocan/a", OCAN_IOCREADMULTI, ...) = 0x00-0x03 = 0a 00 01 01 (ret 0)
ioctl("/dev/ocan/a", OCAN_IOCREADALL, ...) = all registers:
0a 00 01 01  00 00 ff ff  ff ff ff f8  ff ff ff f8
55 55 01 00  00 00 00 00  00 00 00 00  00 00 00 30
55 55 02 00  00 00 00 00  80 00 00 00  00 00 00 00
55 55 03 00  00 00 00 00  00 00 00 00  00 00 00 00

55 55 04 00  00 00 00 00  00 00 00 00  00 00 00 c6
55 55 05 00  00 00 00 00  00 00 00 00  00 00 20 00
55 55 06 00  00 00 00 00  00 00 00 00  00 00 00 ff
55 55 07 00  00 00 00 00  00 00 00 00  00 00 00 ff

55 55 08 00  00 00 00 00  00 00 00 00  00 00 00 ff
55 55 09 00  00 00 00 00  00 00 00 00  00 00 00 00
55 55 0a 00  00 00 00 00  00 00 00 00  00 00 00 03
55 55 0b 00  00 00 00 00  00 00 00 00  00 00 00 00

55 55 0c 00  00 00 00 00  00 00 00 00  00 00 00 01
55 55 0d 00  00 00 00 40  00 00 00 00  00 00 00 00
55 55 0e 00  00 00 00 00  00 00 00 00  00 00 00 01
55 55 0f 00  00 00 00 00  00 00 00 00  00 00 00 ff
(ret 0)
```

The following example shows use of the command from standard input:

```
fino.root# ./ocan_control -
readreg 0
ioctl("/dev/ocan/a", OCAN_IOCREADREG, ...) = 0x00 = 0x0a (ret 0)
getmasks
gettimes
ioctl("/dev/ocan/a", OCAN_IOCGETTIMES, ...) = DSC, DMC = 0 0
CLKOUTDIV, CLKOUTSL = 0 3
SPL, SJW = 1 0
BRP, TSEG1, TSEG2 = 0 6 4
 (ret 0)
set3times 7 3 4
ioctl("/dev/ocan/a", OCAN_IOCSETTIMES, ...) =  (ret 0)
help setupmsg
Command "setupmsg":
    2 to 10 numeric arguments
    Use: "setupmsg <msgnum> <id> [<databyte> ...]"
setupmsg 1 0x3321 0x10 0x3f 0x48
ioctl("/dev/ocan/a", OCAN_IOCSETUPMSG, ...) =  (ret 0)
txmsg 1
ioctl("/dev/ocan/a", OCAN_IOCTXMSG, ...) =  (ret 0)
```

Note that both *setupmsg* and *txmsg* are able to configure both standard and extended identifiers. The identifier is considered a 32-bit value; if all the top 16 bits are zero, then it is considered a standard identifier, and only the top 11 bits are meaningful; if at least one of the top 16 bits is set, then it is considered an extended identifier and the top 29 bits are meaningful. This is implemented by properly setting the `I527_XTD` flag in the control register

and by shifting left any identifier whose top 16 bits are zero, to match the behavior of the *id* field of *struct ocan_msg* only uses the most significant bits. See Section 7.2 [Use of ocan_msg], page 9.

The behavior of *rxmsg* matches that of *txmsg*: extended identifiers are reported as 32-bit numbers (whose top 29 bits are meaningful) and standard identifiers are reported as 16-bit numbers (whose top 11 bits are meaningful). The contents of the *error* bit mask are reported in square brackets, if any.

```
rxmsg 1
from <3e00>: e0 ee e2 e3 e4 e5 e6 e7
rxmsg 1
from <3e00>: e0 e1 e2 e3 [error: OVERFLOW]
```

The *peekmsg* command receives as arguments the message object and the identifier to look for. If the third argument is there and it's not zero, then the `PEEKONLY` flag will be set, so returned data won't be removed from the queue of received packets.

10 Demonstration Programs

The directory '`demo`' includes a few demonstration programs, whose code is placed in the public domain (as far as law permits).

Warning: Contrary to previous versions, the demonstration programs do not reconfigure the bus according to compile-time configuration. Such a feature was handy in early versions, but now that devices declare their preferred configuration, it isn't needed any more.

10.1 Demonstrating Communication

This release includes a simple programs to demonstrate communication; some of them are stand-alone programs and some are pairs aimed to be run on different nodes of the bus.

10.1.1 Stand-alone Demo Programs

The stand-alone demonstration programs are very simple. They are placed in the public domain, since their building blocks are just basic use of the data structures and *ioctl* commands.

The following programs are included in the distribution:

demo-rxmsg
: The program takes two arguments, the message-object number to use and the identifier used in receiving messages. It will set up bus configuration and timings according to the default values and will loop forever waiting for CAN messages. Every message received is dumped to stdout. If the identifier specified is less than 16-bit wide, it is considered a standard identifier (whose top 11 bits are meaningful), if any of the top 16 bits is set then it is considered an extended identifier, and the top 29 bits are meaningful. The same convention is used to dump identifiers of received data, and is consistent with commands in *ocan_control*. Similarly, error flags are reported like *ocan_control* does in *rxmsg*. See Section 9.1 [ocan_control], page 12.

demo-select
: The program behaves exactly like *demo-rxmsg*, but it uses select to wait for a message instead of blocking on *ioctl*. The same conventions as for *demo-rxmsg* above apply to the *id* argument and to the output format.

demo-dump
: The program configures message 15 to receive every message that travels on the bus and then prints to *stdout* all received messages, using the same textual representation as *demo-rxmsg* uses. This can be useful to diagnose ongoing communication on the bus. The "`-e`" option tells it to listen to extended messages instead of standard ones; the "`-d`" option asks to report time differences across packets instead of the absolute time; the "`-i` *interface*" option asks to dump packets that are received in

that network interface too. This last option is useful if you are interested in inter-dependencies between CAN events and Ethernet events. No filtering of Ethernet packets is implemented; for the representation used see the source code.

Time stamps are reported using either *get_cycles* (if the underlying CPU supports it) or *gettimeofday*. If you want to use *gettimeofday* even on a CPU that offers a TSC, you can set `FORCETV` in the environment. No conversion from TSC times to absolute time is performed by the sample tool.

All three programs accept as extra argument the device name to use, as either first of last command-line argument. Additionally, if the `-n` option is specified before the message number, it is ignored (in previous versions it prevented reconfiguration of the bus, but this is now the default).

The programs do not use remote frames. Such feature will be added as soon as the driver supports it.

The following screen dump shows use of *demo-select*.

```
fino.root# demo/demo-select 1 0x1100
configuring msg 1 for standard id 0x1100
waiting for data
from <1100>: 01 02 03 04
from <1220>: 04 05 06
```

Before running the program, the global mask for standard messages was set to `0xf0000`, thus allowing the program to receive messages from different sources. The messages shown above were sent by issuing the following commands to *ocan_control*:

```
setupmsg 1 0x1100 1 2 3 4
setupmsg 2 0x1200 4 5 6
txmsg 1
txmsg 2
```

Note that the identifiers are only 11 or 29 bits wide, so low bits are discarded; this may lead to differences between what you believe to send and what you receive (for example, and id of `0x1111` is received as `0x1100`).

The following example shows exchange of an extended message. The sender is issuing commands to *ocan_control* and the receiver is started before the message is actually sent.

```
setupmsg 1 0x30001230 2 3 4 5
= 0 (0x0)
txmsg 1
= 0 (0x0)

fino.root# demo/demo-select 14 0x30000000
configuring msg 14 for extended id 0x30000000
waiting for data
from <30001230>: 02 03 04 05
```

10.1.2 Demo Program Pairs

Two program pairs are provided in the distribution. Unlike what happens for stand-alone programs, the pairs are released according to the GPL.

This section of the manual is about the following programs:

system-status-server
: The program configures one message object for writing and one for reading. It then transmits a packet through the former message object every time it receives a packet from the latter message object. The transmitted packet carries 8 bytes of status information: the uptime expressed in seconds, the current load average and the amount of free memory expressed in KiBytes. Such information is collected from files in '`/proc`'. By default the program uses message objects 1 and 2, receiving packets addressed to ID 0x3500 and sending replies to ID 0x3400.

system-status-client
: The program configures one message object for writing and one for reading. It then transmits empty packets and expects to receive a reply packet for each packet sent. The reply packet is dumped to stdout assuming it uses the format written by *system-status-server*. By default the program uses message objects 1 and 2, receiving packets addressed to ID 0x3400 and sending replies to ID 0x3500. It delays a configurable time lapse after each packet exchange, the default delay is 100ms.

demo-source
: The program configures a message object for reading. It then prints to *stdout* the data bytes received on that message object. By default the program uses message object 3, receiving packets addressed to ID 0x4400. Error flags found in received packets are reported to *stderr*.

demo-sink
: The program configures a message object for writing. It then prints transmits using that message object every data received from *stdin*, either 8 bytes at a time or less (if *read* returns less bytes. The program exits when it finds end-of-file in its standard input. By default the program uses message object 3, sending packets to ID 0x4400.

The source code for the four programs is very similar, and all of them take a similar set of command-line options. All options get a default value at compile time, but such defaults can be overridden by predefining a C preprocessor macro. For example the *delay* option for *system-status-client* defaults to `OCAN_SSC_DELAY`. Please check the source code for the list of options.

The programs pairs with the default configuration can communicate if run on two different host computers; if you connect two interfaces controlled by the same computer you'll need to use command line options. For example, the following pair of commands establish local communication:

```
demo/system-status-client
demo/system-status-server -f /dev/ocan/b -m 5,6
```

All programs spit a short help message if you pass them any unrecognized option (like `-h` or `--help`). The following list details the meaning of each option:

`-d` *delay*
: A delay, expressed in milliseconds, added after sending each data packet (*demo-sink*, default 10ms) or across transactions (*system-status-client*, default 100ms).

`-f` *devicefile*
: The device being used. It defaults to '`/dev/ocan/a`'.

`-i` *idr*`,`*idw*
`-i` *idr*
`-i` *idw*
: The message ID or ID pair. These numbers can represent either standard or extended identifiers, with the same syntax used by *ocan_control* and *demo-rxmsg*. See Section 10.1.1 [Stand-alone Demo Programs], page 14. See Section 9.1 [ocan_control], page 12.

`-m` *msgr*`,`*msgw*
`-m` *msgr*
`-m` *msgw*
: The message object or message object pair used in communication.

`-p`
: "Prefill" mode for *system-status-server*. Pre-filling the data packets can increase throughput as seen by *system-status-client*, especially if you use a delay of zero.

`-s`
: "Synchronous" more for *demo-source*. If set The program will disable buffering on *stdout*.

`-v`

"Verbose" mode for all programs. The flag may or may not have any effect. Neither the current implementation of verbose mode nor future changes will be documented here.

10.2 Demonstrating Error Management

Errors are notified to the driver via the status-register interrupt. Such interrupt is fired when the *warning* or *error* bits get set in the status register. When such an interrupt happens, any process that is waiting for error notification is awakened and can read the current value of the status register, either via *ioctl* or via *read* from '`/dev/ocan/a-error`' or equivalent file. If the cable is not connected the *warning* and the *error* bits will be quickly set one after another; it's likely a process will only read the second change, since only the current status register can be returned, and not the history of changes.

A process can be notified of an error in several ways; all of them are shown in the following demonstration programs:

demo-error-signal

The program shows how to receive a signal when the error file becomes readable. An argument on the command line is used to open a file different from the default '`/dev/ocan/a`'. Note that there's no need to open '`/dev/ocan/a-error`' to receive error notification via a signal.

demo-error-read

A program using the *read* system call to wait for error notification. An argument on the command line is used to open a file different from the default '`/dev/ocan/a-error`'. The file specified must be an error file. The program is a shell script, since there's no need to write it in C.

demo-error-select

Like *demo-error-read*, but using *select* to wait for an error notification. **Not Implemented, yet**.

11 Known Bugs and Issues

Read and write are still not implemented

There is no control about change of direction for a message object; the application is required to always use the object consistently; any change of direction must be performed during inactivity of the message object.

12 Mailing List

There is a mailing list for discussons about Ocan. You can post suggestions, requests and bugs you encounter while using this package.

To subscribe to the mailing list, send an empty message to *ocan-request@ar.linux.it* with an empty body and *subscribe* in the subject.

There is also a read-only mailing list for CVS commits. Subscribe to this mailing list if you want to be notified by e-mail of CVS changes.

To subscribe to the CVS mailing list, send an empty message to: *ocan-commit-request@ar.linux.it* with an empty body and *subscribe* in the subject.

Table of Contents