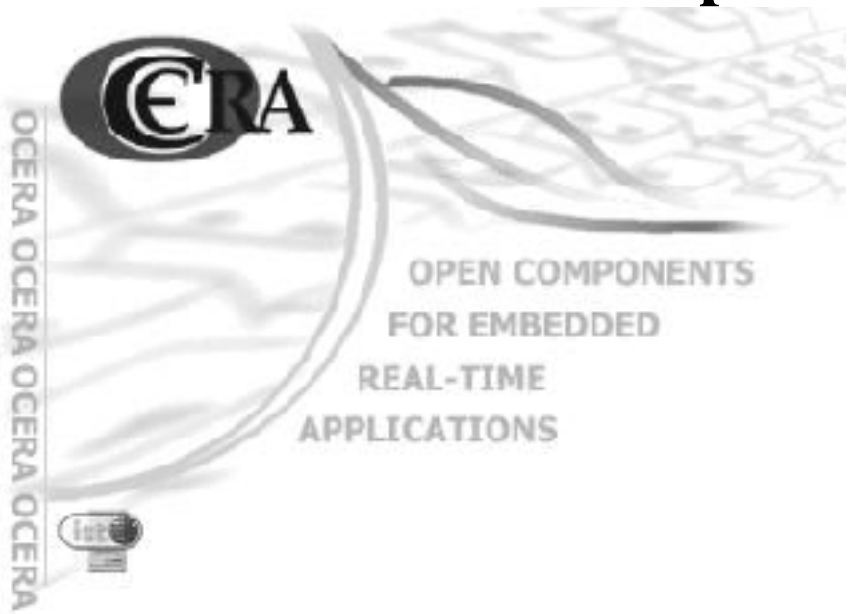
WP7 - Communication Components



Deliverable D7.3 - Design of new communication drivers

WP7 - Communication components : Deliverable 7.3 - Design of new communication drivers
by Pavel Pisa
by Frantisek Vacek

Published July 2003

Table of Contents

Chapter 1. Linux CAN Driver (LINCAN)

The LINCAN is an implementation of the Linux device driver supporting more CAN controller chips and many CAN interface boards. Its implementation has long history already. The OCERA version of the driver adds new features, continuous enhancements and reimplementation of structure of the driver. The usage of the driver is tightly coupled to the virtual CAN API interface component which hides driver low level interface to the application programmers.

1.1. Summary

Name of the component
: Linux CAN Driver (LINCAN)

Author
: Arnaud Westenberg
: Tomasz Motylewski
: Pavel Pisa

Reviewer
: The previous driver versions were tested by more users. The actual version was tested at CTU by more OCERA developers and by BFAD GmbH, which use pre-OCERA and current version of the driver in their products.

Layer
: High-level available

Version
: 0.7-pi3.4 alpha

Status
: Alpha

Dependencies
: The driver requires CAN interface hardware.
: Linux kernels from 2.2.x and 2.4.x series are fully supported.
: Support for latest 2.5.x and future 2.6.x. is in preparation
: The use of VCA API library is suggested for seamless application transitions between driver kinds and versions.

Release date
: N/A

1.2. Description

The LINCAN driver is the loadable module for the Linux kernel which implements CAN driver. The driver communicates and controls one or more CAN controllers chips. The each chip/CAN interface is represented to the applications as one or more CAN message objects through the character device interface. The application can open the character device and use `read`/`write` system calls for CAN messages transmission or reception through the connected message object. The parameters of the message object can be modified by the `IOCTL` system call. The closing of the character device releases resources allocated by the application.

The present version of the driver supports three most common CAN controllers:

- Intel i82527 chips
- Philips 82c200 chips

- Philips SJA1000 chips in standard and PeliCAN mode

The intelligent CAN/CANopen cards should be supported by future versions. One of such cards is P-CAN series of cards produced by Unicontrols. The driver contains support for more than ten CAN cards basic types with different combinations of the above mentioned chips. Not all card types are held by OCERA members, but CTU has and tested more SJA1000 type cards and will test some i82527 cards in near future.

1.3. API / Compatibility

1.3.1. Driver API Overview

Each driver is a subsystem which has no direct application level API. The operating system is responsible for user space calls transformation into driver functions calls or dispatch routines invocations. The CAN driver is implemented as a character device with the standard device node names `/dev/can0`, `/dev/can1`, etc. The application program communicates with the driver through the standard system low level input/output primitives (`open`, `close`, `read`, `write`, `select` and `ioctl`). The CAN driver convention of usage of these functions is described in the next subsection.

The `read` and `write` functions need to transfer one or more CAN messages. The structure `canmsg_t` is defined for this purpose and is defined in include file `can/can.h`. The `canmsg_t` structure has next fields:

```
struct canmsg_t {
    short flags;
    int cob;
    unsigned long id;
    unsigned long timestamp;
    unsigned int length;
    unsigned char
    data[CAN_MSG_LENGTH];
} PACKED;
```

flags
: The flags field holds information about message type. The bit `MSG_RTR` marks remote transmission request messages. Writing of such message into the CAN message object handle results in transmission of the RTR message. The RTR message can be received by the read call if no buffer with corresponding ID is pre-filled in the driver. The bit `MSG_EXT` indicates that the message with extended (29 bit) ID will be send or was received. The bit `MSG_OVR` is intended for fast indication of the reception message queue overfill.

cob
: The field reserved for a holding message communication object number. It could be used for serialization of received messages from more message object into one message queue in the future.

id
: CAN message ID.

timestamp
: The field intended for storing of the message reception time.

length
: The number of the data bytes send or received in the CAN message. The number of data load bytes is from 0 to 8.

data
: The byte array holding message data.

As was mentioned above, direct communication with the driver through system calls is not encouraged because this interface is partially system dependent and cannot be ported to all environments. The suggested alternative is to use OCERA provided VCA

library which defines the portable and clean interface to the CAN driver implementation.

The other issue is addition of the support for new CAN interface boards and CAN controller chips. The subsection Driver Board Support Interface describes template functions, which needs to be implemented for newly supported board. The template of board support can be found in the file `src/template.c`.

The other task for more brave souls is addition of the support for the unsupported chip type. The source supporting the SJA1000 chip in the PeliCAN mode can serve as an example. The full source of this chip support is stored in the file `src/sja1000p.c`. The subsection Driver Chip Support Interface describes basic functions necessary for the new chip support.

1.3.2. CAN Driver File Operations

open

Name

`open` — message communication object open system call

Synopsis

```
int open (const char * pathname, int flags);
```

Arguments

`pathname`

The path to driver device node is specified there. The conventional device names for Linux CAN driver are `/dev/can0`, `/dev/can1`, etc.

`flags`

flags modifying style of open call. The standard `O_RDWR` mode should be used for CAN device. The mode `O_NOBLOCK` can be used with driver as well. This mode results in immediate return of `read` and `write`.

Description

Returns negative number in the case of error. Returns the file descriptor for named CAN message object in other cases.

close

Name

`close` — message communication object close system call

Synopsis

```
int close (int fd);
```

Arguments

fd
: file descriptor to opened can message communication object

Description

Returns negative number in the case of error.

read

Name

`read` — reads received CAN messages from message object

Synopsis

```
ssize_t read(int fd, void * buf, size_t count);
```

Arguments

fd
: file descriptor to opened can message communication object

buf
: pointer to array of `canmsg_t` structures.

count
: size of message array buffer in number of bytes

Description

Returns negative value in the case of error else returns number of read bytes which is multiple of `canmsg_t` structure size.

write

Name

`write` — writes CAN messages to message object for transmission

Synopsis

```
ssize_t write(int fd, const void * buf, size_t count);
```

Arguments

fd
: file descriptor to opened can message communication object

buf
: pointer to array of `canmsg_t` structures.

count
: size of message array buffer in number of bytes. The parameter informs driver about number of messages prepared for transmission and should be multiple of `canmsg_t` structure size.

Description

Returns negative value in the case of error else returns number of bytes successfully stored into message object transmission queue. The positive returned number is multiple of `canmsg_t` structure size.

1.3.3. Driver Board Support Interface

template_request_io

Name

`template_request_io` — reserve io memory

Synopsis

```
int template_request_io (unsigned long io_addr);
```

Arguments

`io_addr`

The reserved memory starts at *`io_addr`*, which is the module parameter *`io`*.

Description

The function `template_request_io` is used to reserve the io-memory. If your hardware uses a dedicated memory range as hardware control registers you will have to add the code to reserve this memory as well. `IO_RANGE` is the io-memory range that gets reserved, please adjust according your hardware. Example: #define IO_RANGE 0x100 for i82527 chips or #define IO_RANGE 0x20 for sja1000 chips in basic CAN mode.

Return Value

The function returns zero on success or `-ENODEV` on failure

File

src/template.c

template_release_io

Name

`template_release_io` — free reserved io-memory

Synopsis

```
int template_release_io (unsigned long io_addr);
```

Arguments

`io_addr`

Start of the memory range to be released.

Description

The function `template_release_io` is used to free reserved io-memory. In case you have reserved more io memory, don't forget to free it here. IO_RANGE is the io-memory range that gets released, please adjust according your hardware. Example: #define IO_RANGE 0x100 for i82527 chips or #define IO_RANGE 0x20 for sja1000 chips in basic CAN mode.

Return Value

The function always returns zero

File

src/template.c

template_reset

Name

`template_reset` — hardware reset routine

Synopsis

```
int template_reset (int card);
```

Arguments

card

Number of the hardware card.

Description

The function `template_reset` is used to give a hardware reset. This is rather hardware specific so I haven't included example code. Don't forget to check the reset status of the chip before returning.

Return Value

The function returns zero on success or `-ENODEV` on failure

File

src/template.c

template_init_hw_data

Name

`template_init_hw_data` — Initialize hardware cards

Synopsis

```
int template_init_hw_data (int card);
```

Arguments

card
Number of the hardware card.

Description

The function template_init_hw_data is used to initialize the hardware structure containing information about the installed CAN-board. RESET_ADDR represents the io-address of the hardware reset register. NR_82527 represents the number of Intel 82527 chips on the board. NR_SJA1000 represents the number of Philips sja1000 chips on the board. The flags entry can currently only be PROGRAMMABLE_IRQ to indicate that the hardware uses programmable interrupts.

Return Value

The function always returns zero

File

src/template.c

template_init_chip_data

Name

template_init_chip_data — Initialize chips

Synopsis

```
int template_init_chip_data (int card, int chipnr);
```

Arguments

card
Number of the hardware card
chipnr
Number of the CAN chip on the hardware card

Description

The function template_init_chip_data is used to initialize the hardware structure containing information about the CAN chips. CHIP_TYPE represents the type of CAN chip. CHIP_TYPE can be “i82527” or “sja1000”. The *chip_base_addr* entry represents the start of the ’official’ memory map of the installed chip. It’s likely that this is the same as the *io_addr* argument supplied at module loading time. The *clock* entry holds the chip clock value in Hz. The entry *sja_cdr_reg* holds hardware specific options for the Clock Divider register. Options defined in the sja1000.h file: CDR_CLKOUT_MASK, CDR_CLK_OFF, CDR_RXINPEN, CDR_CBP, CDR_PELICAN The entry *sja_ocr_reg* holds hardware specific options for the Output Control register. Options defined in the sja1000.h file: OCR_MODE_BIPHASE, OCR_MODE_TEST, OCR_MODE_NORMAL, OCR_MODE_CLOCK, OCR_TX0_LH, OCR_TX1_ZZ. The entry *int_clk_reg* holds hardware specific options for the Clock Out register. Options defined in the i82527.h file: iCLK_CD0, iCLK_CD1, iCLK_CD2, iCLK_CD3, iCLK_SL0, iCLK_SL1. The entry *int_bus_reg* holds hardware specific options for the Bus Configuration register. Options defined in the i82527.h file: iBUS_DR0, iBUS_DR1, iBUS_DT1, iBUS_POL, iBUS_CBY. The entry *int_cpu_reg* holds hardware specific

options for the cpu interface register. Options defined in the i82527.h file: iCPU_CEN, iCPU_MUX, iCPU_SLP, iCPU_PWD, iCPU_DMC, iCPU_DSC, iCPU_RST.

Return Value

The function always returns zero

File

src/template.c

template_init_obj_data

Name

template_init_obj_data — Initialize message buffers

Synopsis

```
int template_init_obj_data (int chipnr, int objnr);
```

Arguments

chipnr
: Number of the CAN chip

objnr
: Number of the message buffer

Description

The function template_init_obj_data is used to initialize the hardware structure containing information about the different message objects on the CAN chip. In case of the sja1000 there's only one message object but on the i82527 chip there are 15. The code below is for a i82527 chip and initializes the object base addresses The entry *obj_base_addr* represents the first memory address of the message object. In case of the sja1000 *obj_base_addr* is taken the same as the chips base address. Unless the hardware uses a segmented memory map, flags can be set zero.

Return Value

The function always returns zero

File

src/template.c

template_program_irq

Name

template_program_irq — program interrupts

Synopsis

```
int template_program_irq (int card);
```

Arguments

card
Number of the hardware card.

Description

The function `template_program_irq` is used for hardware that uses programmable interrupts. If your hardware doesn't use programmable interrupts you should not set the *candevices_t*->flags entry to `PROGRAMMABLE_IRQ` and leave this function unedited. Again this function is hardware specific so there's no example code.

Return value

The function returns zero on success or `-ENODEV` on failure

File

src/template.c

template_write_register

Name

`template_write_register` — Low level write register routine

Synopsis

```
void template_write_register (unsigned char data, unsigned long address);
```

Arguments

data
data to be written
address
memory address to write to

Description

The function `template_write_register` is used to write to hardware registers on the CAN chip. You should only have to edit this function if your hardware uses some specific write process.

Return Value

The function does not return a value

File

src/template.c

template_read_register

Name

`template_read_register` — Low level read register routine

Synopsis

```
unsigned template_read_register (unsigned long address);
```

Arguments

address
memory address to read from

Description

The function `template_read_register` is used to read from hardware registers on the CAN chip. You should only have to edit this function if your hardware uses some specific read process.

Return Value

The function returns the value stored in *address*

File

src/template.c

1.3.4. Driver Chip Support Interface

sja1000p_chip_config

Name

`sja1000p_chip_config` — can chip configuration

Synopsis

```
int sja1000p_chip_config (struct chip_t * chip);
```

Arguments

chip
pointer to chip state structure

Description

This function configures chip and prepares it for message transmission and reception. The function resets chip, resets mask for acceptance of all messages by call to `sja1000p_extended_mask` function and then computes and sets baudrate with use of function `sja1000p_baud_rate`.

Return Value

negative value reports error.

File

src/sja1000p.c

sja1000p_extended_mask

Name

sja1000p_extended_mask — setup of extended mask for message filtering

Synopsis

```
int sja1000p_extended_mask (struct chip_t * chip, unsigned long code, unsigned long mask);
```

Arguments

chip
: pointer to chip state structure

code
: can message acceptance code

mask
: can message acceptance mask

Return Value

negative value reports error.

File

src/sja1000p.c

sja1000p_baud_rate

Name

sja1000p_baud_rate — set communication parameters.

Synopsis

```
int sja1000p_baud_rate (struct chip_t * chip, int rate, int clock, int sjw, int sampl_pt, int
flags);
```

Arguments

chip
: pointer to chip state structure

rate
: baud rate in Hz

clock
: frequency of sja1000 clock in Hz (ISA osc is 14318000)

sjw
: synchronization jump width (0-3) prescaled clock cycles

sampl_pt
: sample point in % (0-100) sets (TSEG1+1)/(TSEG1+TSEG2+2) ratio

flags
: fields BTR1_SAM, OCMODE, OCPOL, OCTP, OCTN, CLK_OFF, CBP

Return Value
negative value reports error.

File
src/sja1000p.c

sja1000p_read

Name
`sja1000p_read` — reads and distributes one or more received messages

Synopsis
```
void sja1000p_read (struct chip_t * chip, struct canfifo_t * fifo);
```

Arguments
chip
: pointer to chip state structure

fifo
: pinter to CAN message queue information

File
src/sja1000p.c

sja1000p_pre_read_config

Name
`sja1000p_pre_read_config` — prepares message object for message reception

Synopsis
```
int sja1000p_pre_read_config (struct chip_t * chip, struct msgobj_t * obj);
```

Arguments
chip
: pointer to chip state structure

obj
: pointer to message object state structure

Return Value
negative value reports error. Positive value indicates immediate reception of message.

File
src/sja1000p.c

sja1000p_pre_write_config

Name

sja1000p_pre_write_config — prepares message object for message transmission

Synopsis

```
int sja1000p_pre_write_config (struct chip_t * chip, struct msgobj_t * obj, struct canmsg_t *
msg);
```

Arguments

chip
: pointer to chip state structure

obj
: pointer to message object state structure

msg
: pointer to CAN message

Description

This function prepares selected message object for future initiation of message transmission by sja1000p_send_msg function. The CAN message data and message ID are transfered from *msg* slot into chip buffer in this function.

Return Value

negative value reports error.

File

src/sja1000p.c

sja1000p_send_msg

Name

sja1000p_send_msg — initiate message transmission

Synopsis

```
iint sja1000p_send_msg (struct chip_t * chip, struct msgobj_t * obj, struct canmsg_t * msg);
```

Arguments

chip
: pointer to chip state structure

obj
: pointer to message object state structure

msg
: pointer to CAN message

Description

This function is called after `sja1000p_pre_write_config` function, which prepares data in chip buffer.

Return Value

negative value reports error.

File

src/sja1000p.c

sja1000p_check_tx_stat

Name

`sja1000p_check_tx_stat` — checks state of transmission engine

Synopsis

```
int sja1000p_check_tx_stat (struct chip_t * chip);
```

Arguments

chip
: pointer to chip state structure

Return Value

negative value reports error. Positive return value indicates transmission under way status. Zero value indicates finishing of all issued transmission requests.

File

src/sja1000p.c

sja1000p_set_btregs

Name

`sja1000p_set_btregs` — configures bitrate registers

Synopsis

```
int sja1000p_set_btregs (struct chip_t * chip, unsigned short btr0, unsigned short btr1);
```

Arguments

chip
: pointer to chip state structure

btr0
: bitrate register 0

btr1
: bitrate register 1

Return Value
negative value reports error.

File
src/sja1000p.c

sja1000p_start_chip

Name
`sja1000p_start_chip` — starts chip message processing

Synopsis
```
int sja1000p_start_chip (struct chip_t * chip);
```

Arguments
chip
pointer to chip state structure

Return Value
negative value reports error.

File
src/sja1000p.c

sja1000p_stop_chip

Name
`sja1000p_stop_chip` — stops chip message processing

Synopsis
```
int sja1000p_stop_chip (struct chip_t * chip);
```

Arguments
chip
pointer to chip state structure

Return Value
negative value reports error.

File
src/sja1000p.c

sja1000p_remote_request

Name

sja1000p_remote_request — configures message object and asks for RTR message

Synopsis

```
int sja1000p_remote_request (struct chip_t * chip, struct msgobj_t * obj);
```

Arguments

chip
pointer to chip state structure

obj
pointer to message object structure

Return Value

negative value reports error.

File

src/sja1000p.c

sja1000p_standard_mask

Name

sja1000p_standard_mask — setup of mask for message filtering

Synopsis

```
int sja1000p_standard_mask (struct chip_t * chip, unsigned short code, unsigned short mask);
```

Arguments

chip
pointer to chip state structure

code
can message acceptance code

mask
can message acceptance mask

Return Value

negative value reports error.

File

src/sja1000p.c

sja1000p_clear_objects

Name

sja1000p_clear_objects — clears state of all message object residing in chip

Synopsis

```
int sja1000p_clear_objects (struct chip_t * chip);
```

Arguments

chip
pointer to chip state structure

Return Value

negative value reports error.

File

src/sja1000p.c

sja1000p_config_irqs

Name

sja1000p_config_irqs — tunes chip hardware interrupt delivery

Synopsis

```
int sja1000p_config_irqs (struct chip_t * chip, short irqs);
```

Arguments

chip
pointer to chip state structure
irqs
requested chip IRQ configuration

Return Value

negative value reports error.

File

src/sja1000p.c

sja1000p_irq_write_handler

Name

sja1000p_irq_write_handler — part of ISR code responsible for transmit events

Synopsis

```
void sja1000p_irq_write_handler (struct chip_t * chip, struct canfifo_t * fifo);
```

Arguments

chip
pointer to chip state structure

fifo
pointer to attached queue description

Description

The main purpose of this function is to read message from attached queues and transfer message contents into CAN controller chip. This subroutine is called by `sja1000p_irq_write_handler` for transmit events.

File

src/sja1000p.c

sja1000p_irq_handler

Name

`sja1000p_irq_handler` — interrupt service routine

Synopsis

```
void sja1000p_irq_handler (int irq, void * dev_id, struct pt_regs * regs);
```

Arguments

irq
interrupt vector number, this value is system specific

dev_id
driver private pointer registered at time of `request_irq` call. The CAN driver uses this pointer to store relationship of interrupt to chip state structure - *struct* chip_t

regs
system dependent value pointing to registers stored in exception frame

Description

Interrupt handler is activated when state of CAN controller chip changes, there is message to be read or there is more space for new messages or error occurs. The receive events results in reading of the message from CAN controller chip and distribution of message through attached message queues.

File

src/sja1000p.c

1.4. Implementation issues

The development of the CAN drivers for Linux has long history. We have been faced before two basic alternatives, start new project from scratch or use some other project as basis of our development. The first approach would probably lead faster to more simple and clean internal architecture but it would mean to introduce new driver with probably incompatible interface unusable for already existing applications. The support of many types of cards is thing which takes long time as well. More existing projects aimed to development of a Linux CAN driver has been analyzed:

Original LDDK CAN driver project

: The driver project aborted on the kernel evolution and LDDK concept. The LDDK tried to prepare infrastructure for development of the kernel version independent character device drivers written in meta code. The goal was top-ranking, but it proves, that well written "C" language driver can be more portable than the LDDK complex infrastructure.

can4linux-0.9 by PORT GmbH

: This is version of the above LDDK driver maintained by Port GmbH. The card type is hard compiled into the driver by selected defines and only Philips 82c200 chips are supported.

CanFestival

: The big advantage of this driver is an integrated support for the RT-Linux, but driver implementation is highly coupled to one card. Some concepts of the driver are interesting but the driver has the hardcoded number of message queues.

can-0.7.1 by Arnaud Westenberg

: This driver has its roots in the LDDK project as well. The original LDDK concept has been eliminated in the driver source and necessary adaptation of the driver for the different Linux kernel versions is achieved by the controllable number of defines and conditional compilation. There is more independent contributors. The main advantages of the driver are support of many cards working in parallel, IO and memory space chip connection support and more cards of different types can be selected at module load time. There exist more users and applications compatible with the driver interface. Disadvantages of the original version of this driver are non-optimal infrastructure, non-portable make system and lack of the select support.

The responsible OCERA developers selected the can-0.7.1 driver as a base of their development for next reasons:

- Best support for more cards in system
- Supports for many types of cards
- The internal abstraction of the peripheral access method and the chip support

The most important features added in the first stage of OCERA development are:

- Added the select system call support
- The support for our memory mapped ISA card added, which proved simplicity of addition of the support for new type of CAN cards
- Revised and bug-fixed the IRQ support
- Rebuilt the make system to compile options fully follow the running kernel options, cross-compilation still possible when the kernel location and compiler is specified. The driver checked with more 2.2.x and 2.4.x kernels and hardware configurations. (compiles even with latest 2.5.x kernels for UP, but needs more work to be ready for 2.6.x kernels)
- Added devfs support

We are planning next changes in the driver in the next stage of the development

- Full support for 2.6.x kernels

- The second deeper rebuilt of the driver infrastructure to enable porting to more systems (most important RT-Linux). The big advantage of continuous development should be ability to keep compatibility with many cards and applications
- Support for multiple opening of the single minor device
- Support for intelligent CAN/CANopen cards
- PCI and USB hardware hot-swapping and PnP recognition

The

1.5. Tests

No heavy tests have been run yet. The driver has been used with more CAN devices (commercial and CTU made) on more Linux system setups (different kernels 2.4.18, 2.4.19, 2.2.19, 2.2.22) with more compilers (GCC 2.95.3 and 3.2.x). The test application from the driver sources and VCA API sources has been tested. The driver is used for the CanMonitor development and other CTU CAN related projects. The success has been reported from the BFAD company as well.

1.6. Examples

The simple test utilities can be found in the `utils` subdirectory of the LINCAN driver source subtree. These utilities can be used as base for user programs directly communicating with the LINCAN driver. We do not suggest to build applications directly dependent on the driver operating system specific interface. We suggest to use the VCA API library for communication with the driver which brings higher level of system interface abstraction and ensures compatibility with the future versions of LINCAN driver and RT-Linux driver clone versions.

The basic utilities provided with LINCAN driver are:

rxtx
: the simple utility to receive or send message which guides user through operation, the message type, the message ID and the message contents by simple prompts

send
: even more simplistic message sending program

readburst
: the utility for continuous messages reception and printing of the message contents. This utility can be used as an example of the `select` system call usage.

sendburst
: the periodic message generator. Each message is filled by the constant pattern and the message sequence number. This utility can be used for throughput and message drops tests.

can-proxy
: the simple TCP/IP to CAN proxy. The proxy receives simple commands from IP datagrams and processes command sending and state manipulations. Received messages are packed into IP datagrams and send back to the client.

1.7. Installation Instructions

1.7.1. Installation Prerequisites

The next basic conditions are necessary for the LINCAN driver usage

- some of supported types of CAN interface boards (high or low speed)
- cables and at least one device compatible with the board or the second computer with an another CAN interface board

- working Linux system with any recent 2.4.x or 2.2.x kernel (successfully tested on 2.4.18, 2.4.19, 2.2.19, 2.2.20, 2.2.22 kernels) or working setup for kernel cross-compilation
- installed native and or target specific development tools (GCC and binutils) and pre-configured kernel sources corresponding to the running kernel or intended target for cross-compilation

Every non-archaic Linux distribution should provide good starting point for the LINCAN driver installation.

1.7.2. Quick Installation Instructions

Change current directory into the LINCAN driver source root directory

```
cd lincan-dir
```

invoke make utility. Just type '**make**' at the command line and driver should compile without errors

```
make
```

If there is problem with compilation, look at first lines produced by 'make' command or store make output in file. More about possible problems and more complex compilation examples is in the next subsection.

Install built LINCAN driver object file (`can.o`) into Linux kernel loadable module directory (`/lib/modules/2.x.y/kernel/drivers/char`). This and next commands needs root privileges to proceed successfully.

```
make install
```

If device filesystem (devfs) is not used on the computer, device nodes have to be created manually.

```
mknod -m666 /dev/can0 c 91 0
mknod -m666 /dev/can1 c 91 1
...
mknod -m666 /dev/can7 c 97 7
```

The parameters, IO address and interrupt line of inserted CAN interface card need to be determined and configured. The manual driver load can be invoked from the command line with parameters similar to example below

```
insmod can.o hw=pip5 irq=4 io=0x8000
```

This commands loads module with selected one card support for PIP5 board type with IO port base address `0x8000` and interrupt line `4`. The full description of module parameters is in the next subsection. If module starts correctly utilities from `utils` subdirectory can be used to test CAN message interchange with device or another computer. The parameters should be written into file `/etc/modules.conf` for subsequent module startup by modprobe command.

Line added to file `/etc/modules.conf` follows

```
options can hw=pip5 irq=4 io=0x8000
```

The module dependencies should be updated by command

```
depmod -a
```

The driver can be now stopped and started by simple **modprobe** command

```
modprobe -r can modprobe can
```

1.7.3. More Installation Instructions

The LINCAN make solutions tries to fully automate native kernel out of tree module compilation. Make system recurses through kernel `Makefile` to achieve selection of right preprocessor, compiler and linker directives. The description of make targets after make invocation in driver top directory follows

lincan-drv/Makefile (all)

LINCAN driver top makefile

lincan-drv/src/Makefile (default or all -> make_this_module)

Needs to resolve target system kernel sources location. This can be selected manually by uncommenting the `Makefile` definition **KERNEL_LOCATION=/usr/src/linux-2.2.22**. The default behavior is to find the running kernel version and look for path to sources of found kernel version in `/lib/modules/2.x.y/build` directory. If no such directory exists, older version of kernel is assumed and makefile tries the `/usr/src/linux` directory.

lib/modules/2.*x.y*/build/Makefile SUBDIRS=.../lincan-drv/src (modules)

The kernel supplied `Makefile` is responsible for defining of right defines for preprocessor, compiler and linker. If the Linux kernel is cross-compiled, Linux kernel sources root `Makefile` needs be edited before Linux kernel compilation. The variable CROSS_COMPILE should contain development toolchain prefix, for example **arm-linux-**. The Linux kernel make process recurses back into LINCAN driver `src/Makefile`.

lincan-drv/src/Makefile (modules)

This pass starts real LINCAN driver build actions.

If there is problem with automatic build process, the next commands can help to diagnose the problem.

```
make clean make >make.out 2>&1
```

The first lines of file `make.out` indicates autodetected values and can help with resolving of possible problems.

```
make -C src default ;
make -C utils default ;
make[1]: /scripts/pathdown.sh: Command not found
make[1]: Entering directory `/usr/src/can-0.7.1-pi3.4/src'
echo >.supported_cards.h echo \#define ENABLE_CARD_pip 1 >>.supported_cards.h ; ...
Linux kernel version 2.4.19
echo Linux kernel sources /lib/modules/2.4.19/build
Linux kernel sources /lib/modules/2.4.19/build
echo Module target can.o
Module target can.o
echo Module objects proc.o pip.o pccan.o smartcan.o nsi.o ...
make[2]: Entering directory `/usr/src/linux-2.4.19'
```

The driver size can be decreased by restricting of number of supported types of boards. This can be done by editing of definition for SUPPORTED_CARDS variable.

There is complete description of driver supported parameters.

```
insmod can.o hw='your hardware' irq='irq number' io='io address' <more options>
```

The more values can be specified for *hw*, *irq* and *io* parameters if more cards is used. Values are separated by commas in such case. The *hw* argument can be one of:

- `pip5`, for the pip5 computer by MPL
- `pip6`, for the pip6 computer by MPL
- `pccan-q`, for the PCcan-Q ISA card by KVASER
- `pccan-f`, for the PCcan-F ISA card by KVASER
- `pccan-s`, for the PCcan-S ISA card by KVASER
- `pccan-d`, for the PCcan-D ISA card by KVASER

- `nsican`, for the CAN104 PC/104 card by NSI
- `cc104`, for the CAN104 PC/104 card by Contemporary Controls
- `aim104`, for the AIM104CAN PC/104 card by Arcom Control Systems
- `pc-i03`, for the PC-I03 ISA card by IXXAT
- `pcm3680`, for the PCM-3680 PC/104 card by Advantech
- `m437`, for the M436 PC/104 card by SECO
- `bfadcan` for sja1000 CAN embedded card made by BFAD GmbH
- `pikronisa` for ISA memory mapped sja1000 CAN card made by PiKRON Ltd.
- `template`, for yet unsupported hardware (you need to edit `src/template.c`)

The *<more options>* can be one or more of:

- *major*=*<nr>*, major specifies the major number of the driver.
- *minor*=*<nr>*, you can specify which minor numbers the driver should use for your hardware
- *extended*=*[*1/0*]*, configures the driver to use extended message format
- *pelican*=*[*1/0*]*, configures the driver to set the CAN chips into pelican mode.
- *baudrate*=*<nr>*, sets the baudrate of the device(s)
- *clock_freq*=*<nr>*, the frequency of the CAN quartz
- *stdmask*=*<nr>*, sets the standard mask of the device
- *mo15mask*=*<nr>*, sets the mask for message object 15 (i82527 only)