ul_drv - uLan RS-485 Communication Driver

Pavel Pisa (pisa@cmp.felk.cvut.cz)

April 27, 2010

Contents

1 What is uLan

uLan provides 9-bit message oriented communication protocol, which is transferred over RS-485 link. Characters are transferred same way as for RS-232 asynchronous transfer except parity bit, which is used to distinguish between data characters and protocol control information. A physical layer consists of one twisted pair of leads and RS-485 transceivers.

Use of 9-bit character simplifies transfer of binary data and for intelligent controllers can lower the CPU load, because of the CPU need not to care about data characters send to other node. Producers of most microcontrollers for embedded applications know that and have implemented 9-bit extension in UARTs of most of today's MCUs. There is the list below to mention some of them :

- all Intel 8051 and 8096 based MCUs with UART
- members of Motorola 683xx family (68332, 68376, ...)
- Hitachi H8 microcontrollers

Intel has developed a multiprotocol UART i82510, which is very well suited for implementing 9-bit communication interface for PC computers. The second example of the chip, which is well suited for 9-bit communication, is OX16C954-PCI produced by Oxford Semiconductors.

One of the problems of 9-bit communications is missing standardization of message protocol. Drivers and formats of one possible implementation of uLan message protocol are described below.

2 uLan Message Protocol

2.1 Data Frame Format

The data frame is a basic communication unit of the uLan protocol. The frame has its destination (node address, general address or not addressed reply start), source node, frame type or command, end mark and integrity check xor_sum. The frame consists of sequence of 9-bit characters. Characters are transferred asynchronously, so every character has one start bit, nine data bits and one stop bit, see fig 1. Total transfer time of one character is equal to transfer of 11 bits. Control characters are transferred with bit D8 equal to one. These control characters appears only on begin and end of the data frame.

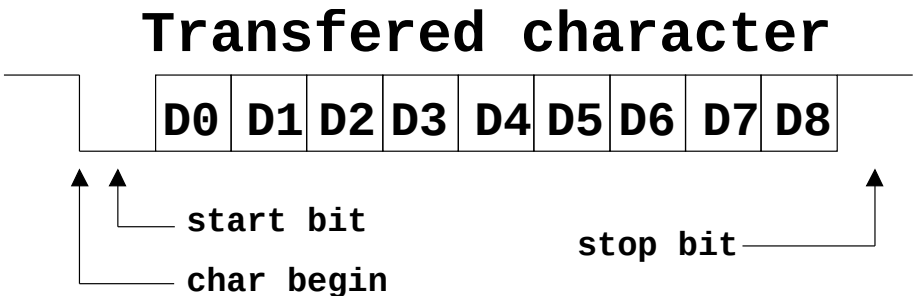

Figure 1: 9-bit Character Format

The data frame starts with a control character equal to the destination node address (DAdr) or an indication of the reply frame (uL_Beg). The receiving of this character wakes up all nodes and every node decides if rest of the frame will be received. The second character in the frame is an address of the source node (SAdr) and next character is command number or message type (Com). These characters and continuing frame data body are transferred with D8 equal to zero. No length information is sent with frame and frame data body can contain from none to maximum data characters defined by maximal admissible blocking time for other nodes. It is recommended, that data length should not exceed 2kB. The end of data body is marked by the frame end control character, which is followed by the integrity check code (xor_sum). The check code (xor_sum) is counted from all control and data characters of the frame as cumulative xor and addition of one.

The frame end control character selects, what should a node do after receive of the frame. There are four possible values (uL_End, uL_Arq, uL_Prq, uL_Aap).

uL_End frame should be transferred into the input queue

uL_Arq frame should be transferred into an input queue, but the sending node is waiting for the acknowledge about successful receiving of the frame. The frame in acknowledged by sending of the character (uL_ACK). In case of xor_sum error or other undefined problem negative acknowledge should be sent (uL_NAK). Delay equivalent to transfer time of more than three characters is considered by the sender as negative acknowledge or missing node with specified address too. The input queue full condition can be signaled to the sender by "wait please" negative acknowledge (uL_WAK). It solves the problem of overloading of the network by repeating attempts to deliver a message to the node, which has no room to receive messages at the moment. The sender node can wait or process other messages before next attempts. uL_Arq cannot be specified for frames with general destination address and reply frames, because of it can lead to parallel sending of uL_ACK from more nodes, which leads to line collision.

uL_Prq proceed request marks frame that needs immediate processing after reception. Next activity is fully defined by command of the frame (Com). Numerical values of these command should be greater than 80h. Processing of such frame can lead to sending or receiving of

next frame (for example memory read and write commands implied to use next frame for data).

uL_Aap same as uL_Prq, but acknowledge should be sent before a start of the frame processing. uL_NAK should be sent in case, that the command is unknown to the receiver node.

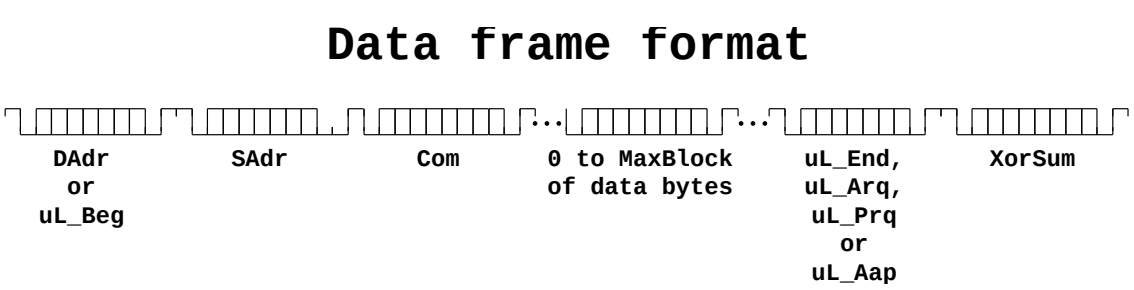

Figure 2: uLan Frame Format

2.2 Access Arbitration and Timing

The RS-485 line has no standard instruments to distinguish an access collision. The collision can be found only by receiving of corrupted message with bad xor_sum. Because the uLan protocol is relatively slow and it is designed to be used for real-time control, it would be bad to rely on such late collision detect. On the other side, usage of centralized arbiter or token passing behaves badly when some node dies and takes piece of bandwidth. That is why deterministic distributed arbitration scheme has been used.

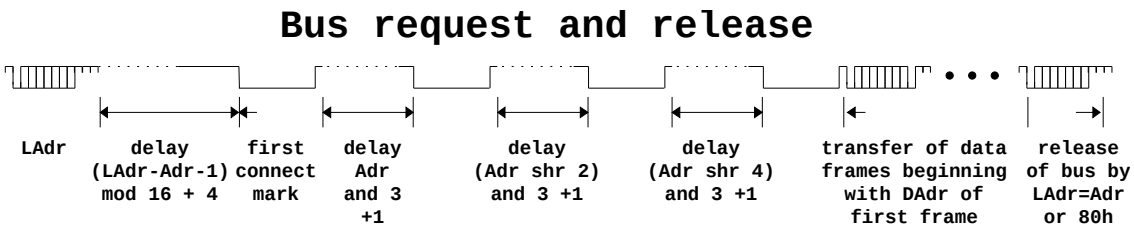

Figure 3: uLan Access Arbitration

The deterministic arbitration is achieved by timing rules for access sequence. The Sequence consist of sending of break characters (11 bits of zero) with enabled line driver and waiting for specified intervals with disabled line driver and listening for breaks from other nodes. All times are taken as multiples of time for sending one character, so no additional timer is needed. Before start of arbitration sequence the node must wait for 4 to 20 character times of no activity on line. This time is calculated from difference of own address and address of last node owning line. The last owner address with B7 and B8 set is used as line release signal. To revitalize communication in case, that last wining node dies before sending of release signal, silence of more than four characters is taken as indication of such situation. All nodes sets interval 20 character times in such case. If node receives any character in this wait interval, it considers line as busy and must wait for release signal or long silence (die of previous master). This scheme lowers possibility of collision and its second benefit is cyclic priority between 16 masters with waiting messages with no decrease of total bandwidth.

To satisfy fully deterministic arbitration between 64 masters, arbitration sequence continues after first break character by sending another three ones in times fully determined by own address of master. Receiving of any character at time of inter-break delay leads to lost of arbitration. Full timing specification is written in figure 3.

2.3 Control Characters

Control characters have bit D8 set and are received by all nodes. They delimit frames and controls bus busy state by bit D7. The sending of an address with bit D7 and D8 set means release of the bus. All values of control characters except destination addresses are selected such way, that their mutual Hamming's distance is at least 2.

Name	Value	Description
DAdr	100h	General address
DAdr	101h .. 164h	Destination node address
uL_Beg	175h	Begin of unaddressed frame
uL_END	17Ch	End of frame
uL_ARQ	17Ah	End with acknowledge request
uL_PRQ	179h	End with proceed request
uL_AAP	176h	End with acknowledge and proceed request
uL_ERR	17Fh	Error without release of bus
LAdr	181h .. 1E4	Release of bus by node
uL_ERR	1FFh	Error, abort and release bus

Next table shows characters used for acknowledge. They are not real control characters (D8=0), but they are important for protocol too and Hamming's distance is selected to 4.

Name	Value	Description
uL_ACK	019h	Acknowledge of frame
uL_NACK	07Fh	Negative acknowledge
uL_WAK	025h	Receiver can probably receive message later, but cannot now

2.4 Commands or Frame Type Codes

Next table shows preferred value ranges of command codes for frames with different processing.

Command Range	Type of Message Processing
00h .. 3Fh	Store to buffer
40h .. 7Fh	Store to buffer without uL_ACK
80h .. 9Fh	Immediate process
A0h .. BFh	Process with additional receive
C0h .. FFh	Process with additional send

Some of the defined commands are enumerated in the next table. Most of them are designed for target system debugging, and that is why, it is necessary to process these frames immediately after receiving and processing must be done in high priority interrupt services. The second part of table summarizes commands/types used for higher level services.

Name	Value	Description
UL_CMD_RES	80h	Reinitialize RS485
UL_CMD_SFT	81h	Test free space in input buffer
UL_CMD_SID	F0h	Send identification
UL_CMD_SFI	F1h	Send amount of free space in input queue
UL_CMD_TF0	98h	End of stepping
UL_CMD_TF1	99h	Begin of stepping
UL_CMD_STP	9Ah	Do step
UL_CMD_DEB	9Bh	Additional debug commands
UL_CMD_SPC	DAh	Send state
UL_CMD_RDM	F8h	Memory read
UL_CMD_WRM	B8h	Memory write
UL_CMD_OI	10h	Standard value for uLan OI
UL_CMD_OIr	11h	Standard type for uLan OI reply
UL_CMD_NCS	7Fh	Network control service
UL_CMD_SNST	C1h	Fast status and connection check

3 uLan driver

ul_drv is the Linux device driver designed to access the uLan network. Today supported hardware is ISA i82510 RS-485 card or simple active converter dongle for standard PC RS-232 ports. The driver version 0.5.5 adds support for PCI card equipped by OX16C950-PCI controller. This version of driver could be compiled for WindowsNT and Windows 2000 as Kernel Mode Driver.

3.1 Install driver for Linux

The "`Makefile`" is a link to "`Makefile-mod`" to compile driver with your kernel configuration. "`Makefile-mod`" expect, that kernel sources are in the "`/usr/src/linux`" directory and that kernel sources and its configuration are of same version as the current running kernel. The kernel source self reference link "`/lib/modules/2.y.z/build`" is used for new kernels. Enter next command in the ul_drv directory to compile and install the module

```
make install
```

The module is installed into "`/lib/modules/x.y.z/misc`" directory or "`/lib/modules/x.y.z-/kernel/drivers/char`".

The driver can control up to nine devices, but only check for 0x3e8 port number is default behavior after "`insmod ul_drv`". Because of driver controls same hardware as the Linux serial driver, it is necessary to disable default kernel driver for same port in case of conflict. It can be done for example by

```
setserial /dev/ttyS2 uart none
```

Port numbers and others parameters can be defined by module parameters. If UDEV nor DEVFS is used, then special character device files must be created to access the driver from programs . Suggested names are

```
mknod /dev/ulan0 c 248 0
mknod /dev/ulan1 c 248 1
ln -sf /dev/ulan0 /dev/ulan
```

"`/dev/ulan`" is default name for client programs. Major number 248 is in experimental range and will be changed in future. This number is defined in ul_hdep.h. If defined as zero, dynamic number is assigned after insmod.

ul_drv module parameters

port=<iop>{,<iop> ...} up to nine IO port numbers separated by comas for up to four uLan devices

irq=<int>{,<int> ...} select for every defined device corresponding IRQ number. Value 0 means autoprobe.

chip=<string>{,<string> ...} optional specification of used controller chip: auto, 82510, 16450, 16950-pci

slot=<slot_spec>{,<slot_spec> ...} optional, used for PnP cards to find **baud** and **my_adr** from slot type (pci), full slot specification (pci00:0c.0) or slot with device interface number specification (pci00:0c.0.1)

baud=<spd>{,<spd> ...} defines transfer baud-rate for every device, default value is 19200

my_adr=<adr>{,<adr> ...} node address of every computer interface as seen from uLan network range is from 1 to 64 default value is 2 for all interfaces

debug=<int> sum of debug flags (FATAL=1, CHIO=2, IRQ=4, MSG=8, FAILS=16, SEQ=32, PORTS=64, FILE=128, FILT=256)

Most of the parameters are optional. The **port** and **baud** are enough for most of ISA or motherboard interfaces. PCI interfaces are matched against parameters in autodetection order. Specification of slot parameter can be used for precision parameters and minor control. Example parameters for the dongle converter plugged to regular COM2 port are below.

```
setserial /dev/ttyS1 uart none
insmod ul_drv port=0x2f8 irq=3 baud=9600 my_adr=2
```

3.1.1 Automatic Driver Startup for Recent Linux Versions

The new Linux module utilities and device management in sontrolled by configuration files stored in the "`/etc/udev/rules.d`" and "`/etc/modprobe.d/ulan`" directories. Next UDEV configuration file ensures, that regular users has access to uLan driver

/etc/udev/rules.d/10-ulan.rules

```
SUBSYSTEM=="ulan",GROUP="users",MODE="0660"
ACTION=="add",DEVPATH=="/class/ulan/ulan0",SYMLINK="ulan"
```

To release standard PC UART/serial port from standard serial driver, add next line to the file as well.

```
KERNEL=="ttyS0",RUN+="/bin/setserial /dev/ttyS0 uart none"
```

If the standard PC UART port is used, the parameters for driver has to be specified. Next file should be createt

/etc/modprobe.d/ulan

```
alias   /dev/ulan*
ul_drv options ul_drv port=0x3f8 irq=4 my_adr=2 baud=19200
```

3.1.2 Automatic Driver Startup for Old Linux Versions

Add next lines to "`/etc/modules.conf`" to enable on-demand module loading

```
alias /dev/ulan* ul_drv
alias char-major-248 ul_drv
options ul_drv port=0x2f8 irq=3 baud=9600 my_adr=2
pre-install ul_drv setserial /dev/ttyS1 uart none
```

The recent versions of the driver supports Linux device filesystem and creates **ulan<x>** devices after driver initialization. Device creation and permissions are controlled by the "`/etc/devfsd.conf`" file.

```
LOOKUP ^ulan$ EXECUTE /bin/ln -s ${mntpnt}/ulan0 ${mntpnt}/ulan
LOOKUP ^ulan.* MODLOAD
REGISTER ^ulan[0-9] PERMISSIONS root.users rw-rw----
```

The first line creates conventional link "`/dev/ulan`" to default interface. Next line enables module autoloading and last changes permission to enable access to ordinal users.

3.2 Install KMD for Windows

uLan driver can be compiled as **WindowsNT/2000** Kernel Mode Driver. Aspects of this version of driver are described in this paragraph. New WMD PnP version of driver (**Windows2000/98**) solves some present limitations of KMD version. Manual installation of current KMD version consists of next steps :

- driver "`ul_drv.sys`" file must be copied to system driver directory "`%WINNT%/system32-/drivers`"
- branch "`ul_drv`" in
  "`HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services`"
  should be added
- next keys in "`ul_drv`" branch are responsible for driver automatic of start
  - `"Type"=dword:00000001`
  - `"Start"=dword:00000002`
  - `"ErrorControl"=dword:00000001`
  - `"DisplayName"="UL_DRV"`
  - `"Group"="port"`
  - `"Tag"=dword:00000001`
- driver needs to know port address and interrupt line of the ISA card or serial port, this information is stored in the "`ul_drv/Parameters`" registry branch. Next base port address and interrupt line apply for default jumpers setting of ISA 82C510 card.
  - `"Port Address"=dword:000003e8`
  - `"IRQ Line"=dword:00000005`
- OX16C954 PCI cards are autodetected by the driver when PCI support is enabled in "`ul_drv\Parameters`"
  - `"ScanforPCI"=dword:00000001`

- next keys in the "`ul_drv\Parameters`" registry branch selects communication and internal driver parameters
  - `"Baud Rate"=dword:00004b00`
  - `"Buffer Size"=dword:00004000`
  - `"My Addr"=dword:00000002`
  - `"Debug"=dword:00000019`

Driver is automatically started after reboot. Driver can be started and stopped from the Device control dialog in **WindowsNT** environment. The Device Manager started from My_Computer ▷ Properties ▷ Hardware For **Windows2000** could be used. View ▷ Show Hidden Devices must be checked and branch Non PnP drivers must be opened to view ul_drv in list of drivers .

There are some limitations of current KMD ul_drv driver version. Driver does not recognize and support more than one **uLan** interface. Main source of this limitation is the use of simple registry "`Parameters`" branch, conflicts with **Windows** internal RS-232 serial driver and motherboard PnP management for serial ports with plugged dongle RS-485 adapter.

Next steps should be done in case of problems :

- check of correct registry parameters
- "`ul_drv\enum`" branch, "`InitStartFailed`" and "`ImagePath`" keys could be deleted
- presence of "`ul_drv.sys`" in "`%WINNT%/system32/drivers`" should be checked
- reboot computer

3.3 WDM Driver for Windows

The WDM driver is designed for **Windows2000** and **Windows98** operating systems. It implements PnP functions. Present version of the driver does not support power management functions. The binary image of WDM version of the driver has been renamed to "`ul_wdm.sys`" to distinguish it from **WindowsNT** KMD version ("`ul_drv.sys`"). Driver can be assigned to one or more PnP standard serial port with RS-485 converter or OX16C954 based PCI cards from device property page of device manager. Next steps could be used to select uLan driver for COM2 serial port.

- prepare driver image ("`ul_wdm.sys`") and installation information ("`ul_wdm.inf`") in some directory. Copy "`ul_wdm.sys`" and "`ul_wdm98.inf`" files to floppy drive in the case of Windows98.
- inspect, that RS-485 converter is plugged to COM2 socket
- open My_Computer ▷ Properties ▷ Hardware ▷ Device Manager
- open branch Ports (COM and LPT) and select Serial Port (COM2) ▷ Properties
- select Driver ▷ Actualization, Installation guide dialog opens
- select Next, then Find optimal driver and Next
- check Look for alternative location and reply right directory where prepared files are located
- check Install one from alternative drivers on next dialog to enable alternative drivers selection dialog and select Next
- choose desired driver, select line with "`- uLan xxxx`" extension to install uLan driver . Same dialog is used to return back to RS-232 driver, when Communication Port (Microsoft) is chosen

- Press Next, driver is started at this point and guide is closed after press to Finish

The COM2 port is then moved to class unknown, when port is assigned to uLan driver. **uLan** communication is possible immediately after port assignment for Windows2000 device manager (**Windows98** requires reboot). When all applications using **uLan** are stopped, the port can be disabled or assigned back to standard RS-232 driver (again RS-232 can be used immediately in **Windows2000** environment). Next Registry branch is responsible for the COM2 port driver assignment

```
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Enum\ACPI\PNP0501\2]
```

The key `"Service"="ul_wdm"` is responsible for driver assignment. The subbranch "`Device Parameters`" stores device specific parameters. Next parameters are recognized by actual version of driver with shown predefined values

- `"uLanBaudrate"=dword:00004b00`
  communication speed, default 19200 Bd
- `"uLanMyAddress"=dword:00000002`
  uLan address of PC computer on uLan network

Installation of **uLan WDM** driver for **ISA** card or **PCI** card is different in case, that device is not recognized automatically by **PnP** manager. In such case Add/Remove Hardware wizard must be used for class Unknown and driver **INF** file location must be specified in **INF** file search dialog.

Registry layout is different and not such clear in case of **Windows98**. The first branch for ports added through Add Hardware wizard is

```
[HKEY_LOCAL_MACHINE\Enum\Root\Unknown\0000]
```

When **PnP COM2** port is used, Bach is

```
[HKEY_LOCAL_MACHINE\Enum\ACPI\*PNP0501\00000002]
```

Above described **uLan** driver parameters are stored in this branch as well, but driver image name and load specification is stored in class branches selected by the `"Driver"` key. Usual storage is

```
[HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\Class\Ports\0001]
```

Next keys in that branch are critical for **uLan** driver loading

```
"DevLoader"="*ntkern"
"NTMPDriver"="ul_wdm.sys"
```

Windows98 are not real operating system and support was added on user request. Use **uLan** driver for **Windows98** system on your own risk and do not expect any usable real or particular help.

3.4 Driver Implementation

The driver is implemented as relatively independent layers and subsystems. Messages are prepared and received in the driver dedicated memory. This memory is divided into blocks with uniform size with atomic allocation routines. When message is being stored into blocks, head of message with couple of data bytes is stored in the first allocated memory block. If all data cannot be stored in the first block, next blocks are allocated and linked together. The message heads are linked

in bidirectional linked lists of messages prepared for sending, processed messages and messages prepared for client notification. These lists or queues are main mechanism for transferring of messages between subsystems.

Link protocol is programmed as finite state machine with state stack, which state routines are executed by interrupt handler. State routine can return positive integer information, negative error notification or zero, which leads to wait for next interrupt. Information or error is used as input parameter when state routine is called. When the state routine wants initiate transfer to another state routine it changes pointer to the actual state routine. If previous state routine returns nonzero value new routine is called immediately, in other case next interrupt invokes new state routine. There is stack of callers of actual state routines which enables to constructs state machine subsystems, which can be used in more places in main state machine loop. Main purpose of this state machine is to send or process messages coming in list of messages prepared for sending and if specified, move these messages onto list of messages prepared for client notification. Received messages are put onto this list too. Subsystem is supervised by timeout handler, which can revitalize communication in case of die of other node. The interrupt and timeout handlers are fully SMP re-entrant.

The state machine subsystem uses pointers to chip driver routines for hardware port manipulation. This is only part dependent on used chip, today 82510, 16450 and OX16C950PCI. These routines can send and receive 9 bit character, connect to RS-485 line by the arbitration sequence, wait for specified time for character and initialize and close port.

File operation subsystem makes interface between OS kernel VFS and client message queues. It enables to prepare single or multi-frame messages and stores notifications of received or processed messages in client's private state structures. This part is heavily operating system dependent.

3.5 Organization of Source Files

Source tree contains directory containing driver kernel source files ("`ul_drv`"), directory with user-space library hiding system dependent kernel driver access routines ("`ul_lib`") and some example utilities ("utils").

3.5.1 Source Files of uLan Driver (ul_drv)

The ("`ul_drv`") directory contains sources of uLan protocol kernel driver. The sources include support for Linux, DOS, WinNT, Win2000/98 and system-less embedded targets.

implementation of 16550 and 82510 chip drivers.includes source of Ox16950 chip driver

generic message processing machine own memory blocks allocation and queues of messages more IOCTL generic code code for client notifications and wakeup

Linux kernel device and DEVFS interface PCI plug and play

library target interface, almost done as stub functions to above generic IOCTL and read and write, enables to link driver directly into application, it can be used under DOS or Linux user-space with IRQ emulation by signals

WinNT target interface, which again uses as most of functions developed for Linux kernel as possible.It can be compiled into PnP Win2000/Win98 WDM driver with included conditionals and support sources for PnP.

ul_drv.h definition of bidirectional linked lists of messages, atomic memory allocations, message flags and driver instance state structure

k_compat.h header file hiding differences between different Linux kernel versions, it is tested from 2.2.xx up-to 2.6.xx (it could work with kernels down-to 1.3 version)

ul_hdep.h definition of hardware dependent functions and macros for all target systems and definition of kernel internal per client state structure

ul_c450.c chip driver for standard PC serial port

ul_c510.c chip driver for the Intel 82C510, ideal UART chip for 9-bit communication with TM empty interrupt, two independent timers for timeouts and other protocol timing, etc.

ul_c950pci.c Ox16C950, 64 byte FIFO, the possibility of automatic transceiver direction control, all events can be reported by interrupts. All these features are fully used by uLan.

ul_cps1.c USB <-> uLan converter support. Does not work on UART state machine level. It is activated when message is stored in ***prep_bll*** and when there is free slot, forwards message to the external converter over USB. It receives sent messages confirmation and incoming messages from converter and stores these in ***proc_bll***.

ul_frame.c sends and receives sequences for frame start and end of frames

ul_ufsm.c state machine. It monitors medium state and when it is not occupied and there is message ready to be sent on ***prep_bll*** queue it starts arbitration and if media arbitration is successful outgoing message processing is started. Message is moved on ***work_bll***. Processed message and messages received from other devices are moved to processed messages queue ***proc_bll***. Messages are then outside of interrupt context announced to the driver clients. As long as they are not processed by clients, they are kept on the ***opan_bll*** queue. In the case of interface which does not use character based state machine (ul_cps1.c - USB), ul_ufsm.c logic is not used and messages are moved from ***prep_bll*** to ***proc_bll*** directly by higher level chip driver.

All the above parts are inherently state machine and therefore are implemented that way. There is "C"-language function representing each state of finite state machine which is responsible for state actions processing and for advancing to next states.

ul_mem.c memory management. Driver allocates for each channel memory buffer divided into pool of constant size blocks. The blocks are allocated for messages as messages are created and grows and due to poll concept, block can be allocated in interrupt context. The first block contains a header with the message control information and fields used for linking the bi-directional list BLL. There are four lists/queues per channel ***prep_bll*** / ***work_bll*** / ***proc_bll*** / ***opan_bll***. The rest of the first block space unused for control information is used as start of data buffer for message. If the message is longer, there is attached a chain of additional blocks, which contain only data.

ul_base.c basic operations used for preparation, initiation of sending, receiving messages, etc. The functions in this file are as far as possible target system independent.

ul_linux.c driver envelope for Linux operating system and character device implementation

ul_inlib.c for linking of applications with driver to use driver code directly in user-space for DJGPP / DOS, system-less targets and Linux user-space test builds

ul_djgpp.c auxiliary functions for DJGPP (interrupts, timing, logging)

ul_wdbase.c common basis for Windows WDM (XP/2k/98) and KMD (NT)

ul_kdmnt.c KMD Windows (NT)

ul_wdent.c desolation and closure for WDM (XP/2k/98)

ul_wdpnp.c PnP support required for WDM

ul_wdpwr.c minimal power management support for WDM

ul_wdusb.c auxiliary functions to hide differences in Windows USB infrastructure implementation

ul_linpci.c general mapping function driver for PCI hardware under Linux

ul_linusb.c general function for Linux USB device enumeration

ul_debug.c debugging support routines

emu_irq.c IRQ emulation of the Linux user

ul_di.c iterator data for iteration through characters and blocks of messages

ul_drv.c sources wrapper

ul_tors.c system independent parts of uLan driver instance constructor and destructor

ul_devtab.c table containing vendor and product identifiers for PnP PCI and USB devices enumeration for Linux and Windows operating systems

ul_tst.c low level test routines

3.5.2 Source Files of uLan Support Library (ul_lib)

The directory ("ul_lib") holds sources for library implementing basic message processing (send, receive, filtering and notification) and some common commands used b applications. Its main purpose is to hide system specific interface to the driver and provide portable uLan network API.

drv_def.h target system dependent parts of user-space driver API

ulan.h target system independent part of API usable under Linux/DOS/NT etc.

ul_l_drv.c kernel driver interface access functions, replaced by direct inclusion of driver into library for DOS target

ul_l_msg.c basic message sending and reception functions

ul_log.c simple logging facilities for messages send and processed by applications

ul_l_oi.c basic primitive functions for higher object uLan layer

3.5.3 Source Files of uLan Utilities (utils)

The sources of command line test utilities and programs which are mostly independent on target systems are located in ("utils") directory.

ul_buftst.c test of hardware of line transceivers

ul_spy.c uLan message monitor

ul_sendhex.c intelhex and binary downloader and uploader

3.6 Driver Components

The driver is implemented in a relatively independent and the subsystems. The individual components of the drivers are:

- General support for the allocation, iteration, filling, handling and reading the messages consisting of one or more frames. This subsystem includes a list of free blocks ***free_blk***, queue of outgoing messages which are ready for processing ***prep_bll***, currently processed message ***work_bll***, queue of already processed or received messages prepared for distribution to operators/clients ***proc_bll*** and the list of messages waiting for completion of processing by the operators ***opan_bll***.
- General support for the creation and cancellation of driver instance and the choice of chip drivers according to interface hardware type.

- Interface for access from the user-space when driver is used on operating systems utilizing memory protection (Linux, Windows NT and Windows WDM driver model) or for integration into the firmware of small embedded devices, without operating system.
- Implementation of the general finite state machine (UFSM) for those cases where the implementation of the protocol uses UART, which handles data at the level of individual characters, whether the hardware FIFO is or is not used. This layer is not used, when the entire frames or messages are transferred to the hardware/device which implements protocol prescribed messages transmit and receive itself (this is case of the host side driver when uLan-USB converter connected to PC is used).
- The chip drivers implement ***ul_call_fnc*** type functions ***fnc_recch*** (receive character function), ***fnc_sndch*** (send characters function), ***fnc_wait*** (waiting for the reception or specified time expire), ***fnc_connect*** (connection to the media), ***fnc_finishtx*** (waiting for end of transmission and bus driver switching for receive), ***fnc_pool*** (test interrupt presence and event type) in the case of generic UFSM use. In the case where the whole messages are transmitted to hardware or interface specific service thread is used, these functions are not used and activation of message processing is achieved by chip driver functions ***fnc_stroke*** invocation.

The course of communication:

Pre-allocated memory divided into blocks of constant length are used for message data storage. These blocks are inserted at single linked list ***free_blk*** after driver initialization. Client/operator opens opens driver by system level open call. During the preparation of messages created by call UL_NEWMSG, the data block for frame header is allocated first. Data are stored first into this block and then additional blocks are allocated and chained after head block to store subsequently written data. The additional frame can be added by UL_TAILMSG after first frame head to to form multi-frame message. When the message is released by call UL_FREEMSG, it is moved to the head of queue of ready for processing messages ***prep_bll*** and if the chip driver/UFSM is not active, is activated by chip functions ***fnc_stroke***. When the previous messages are served and media can be allocated for master operation, message is moved to ***work_bll*** queue. After completion of message prescribed sequence of frames transmit and reception, the message is queued onto ***proc_bll*** queue and bottom half processing is scheduled. Before actual filtering and distribution of message to clients/operators, message is moved to the operators announced queue ***opan_bll***. The operator scans his private receive queue UL_INEPOLL, when it is not empty, accepts the first message frame UL_ACCEPTMSG and possibly its consecutive frame(s) UL_ACTAILMSG, receives data, and finally release the message by call UL_FREEMSG. Registration for messages of operator's interest is achieved by filter specification and addition by call UL_ADDFILT.

3.7 Driver Finite State Machines

The uLan communication protocol processing is implemented as finite state machine. Function of this machine (UFSM) will be described in detail in the following paragraphs. The state transition diagrams are used for better understanding of individual state transitions and calls of the lower level state machine states. Ovals and octagons represent the states of the finite state machines (represented by individual state processing functions in the driver code). Rectangles represent calls/activations of lower level machines. The gray color ovals and octagons represent entry points to lower levels of the machine. The octagons marks states with possibility of returning to a higher level machine. The edges with arrows shows transitions between different states of the machine and lower levels calls.

The highest level state machine controller is schematically illustrated in Figure 4. From the diagram, it is apparent that the central node consists of functions ***uld_drvloop***. This function serves as a sort guide-post. It divides operations into three main directions which are:

- Receiving messages

- Sending messages
- Bus state monitoring

3.7.1 Bus State Monitoring

If no other action is required (send/receive of message) the *functional link* ***UDRV_fnc_recch*** is called. This link will activate specific chip provided function to wait for character receive. The actual function invoked is given by chip support selected during driver instance creation. In all cases, the task of this function is to take a character from the bus and write it into the character buffer in the driver instance state structure. The actual implementation of functions for different architectures will be described later. After capturing the character, processing is moved back to the central node (***uld_drvloop***).

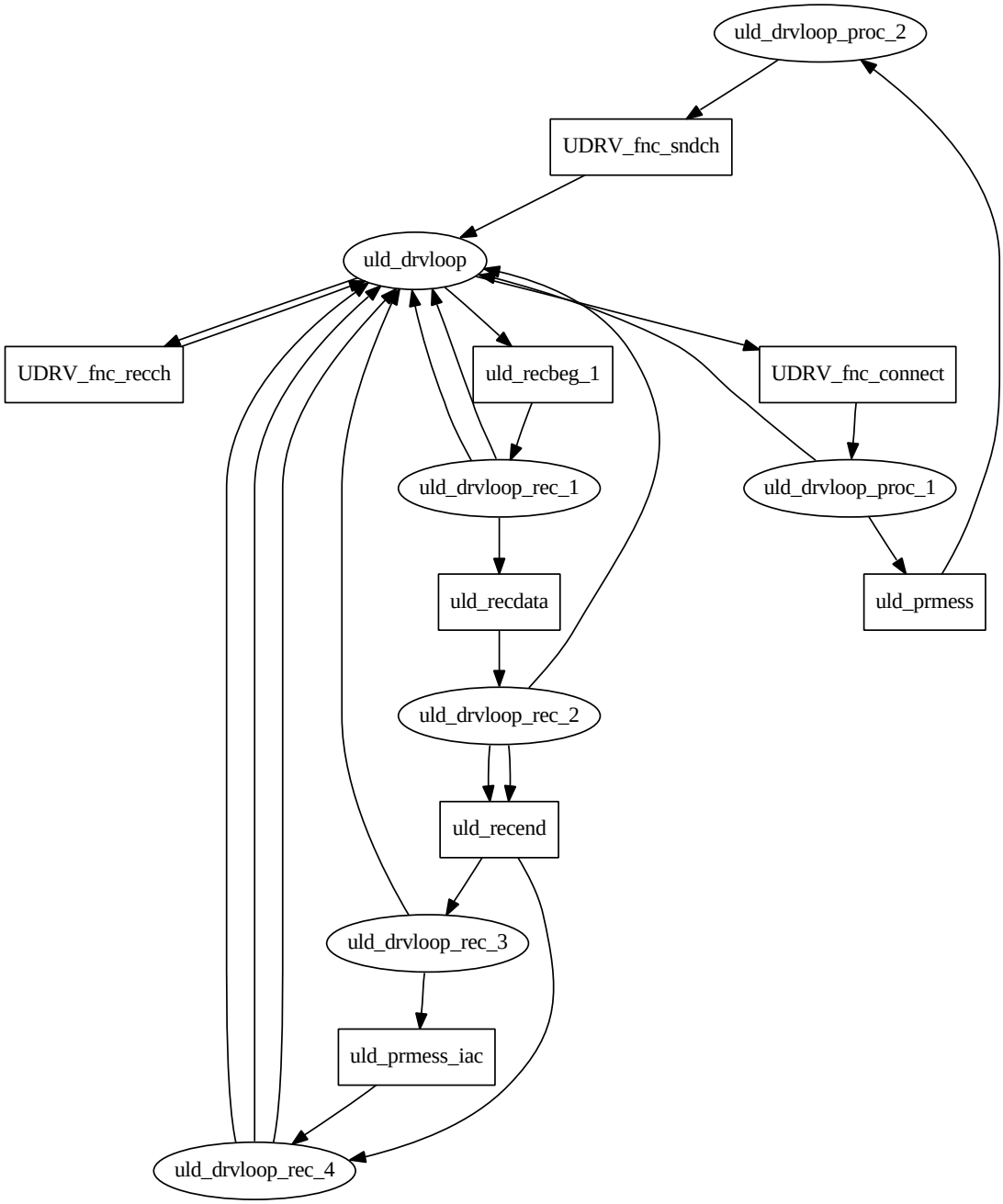

Figure 4: Top-level state machine diagram

3.8 Receiving Messages

If the character (which is located in the character buffer of driver state) with bit **D8** set is received and its code is equal to the own device address or broadcast address (0) or if driver promiscuous mode is enabled, the subroutine to process beginning of the incoming message frame receive is set (function ***uld_recbeg_1***, description can be found below) and then moves into the state ***uld_drvloop_rec_1***.

As ***uld_drvloop_rec_1*** allocates a memory block of type *ul_mem_blk* (message) from ***free_blk*** and fills it with the received frame header information (SAdr, DAdr, CMD). The filled

head block of a message is moved onto ***work_bll*** queue and receipt ion (inactivity) timeout function is set to ***uld_drvloop_rec_error*** to recover from state, when frame sender dies unexpectedly. Then the procedure ***uld_recdata*** (description can be found below) is called which implements receiving of data contained in the message and continues with the ***uld_drvloop_rec_2***.

The ***uld_drvloop_rec_2*** state continues by state ***uld_drvloop_rec_3*** if immediate messages processing/reaction support is compiled in. Otherwise, the state ***uld_drvloop_rec_4*** is entered. Before transition into one of the two possible states, the end of the frame reception ***uld_recend*** (terminating characters, checksum etc.) is processed.

The processing of the ***uld_drvloop_rec_3*** state checks whether reception finished without error and calls subroutine for immediate actions/response processing ***uld_prmess_iac***. The finite state machine advances to the state ***uld_drvloop_rec_4***. In this state, the driver timeout function is set back to an empty action and a stamp is assigned to the message, which serves as a unique identifier for later distribution of messages at higher levels. Finally, the message is moved to the input queue and the distribution of the message to clients is scheduled. The finite state machine returns to the central state (***uld_drvloop***).

3.8.1 Receive Start of Frame

Receive start of frame is illustrated by state diagram in Figure 5.

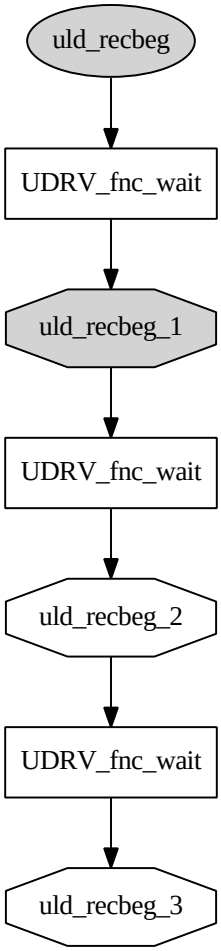

Figure 5: State diagram of frame receive start

This sub-machine begins by calling the function ***UDRV_fnc_wait***. This function is different

from ***UDRV_fnc_recch*** that it timeouts if character is not received within a certain time interval. In the case of timeout, the function generates the timeout activates registered error procedures. If the character is received the transition to ***uld_recbeg_1*** is performed. This state can be alternative entry point to ***uld_recbeg*** if the first character has been already received by some other upper level state machine component. The ***uld_recbeg_1*** check whether received destination address matches this device/module. The destination address is stored into driver instance state and connection flags bit 3 (mask 0x8) is set. If promiscuous mode is enabled and received message does not match module/device address the connection flags bit 4 (mask 0x10) is set as well. Then ***fnc_wait*** is called again and transferred to the state ***uld_recbeg_2*** follows.

The received message source address **SAdr** is stored to the connection parameters and wait for other character follows. After successful character representing **cmd** is receive its value is stored to connection parameters in ***uld_recbeg_3*** state and sub-machine finalizes by returning one level up int calling state machine.

3.8.2 Receive Data

This procedure is sequentially receives data bytes using which are stored in the prepared ***ul_mem_blk*** type structure with use of data iterator as shown in Figure 6. The data receive is terminated if the control character (DB8 = 1) is received or if there is not enough memory to store data.

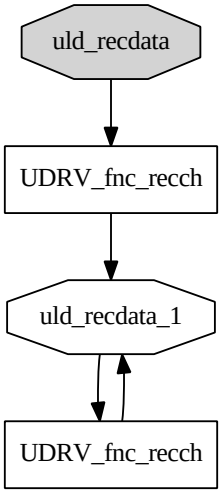

Figure 6: State diagram of frame data receiption

3.8.3 Receiving the End of the Frame

The finalization of frame receive starts in states ***uld_recend*** and ***uld_recend_1*** as shown in Figure 7. The next character reception is initiated until received character or character found in the buffer has ninth bit set (termination character). According to the value of this character (**UL_ARQ**, **UL_PRQ** etc.) connection flags and actual frame flags are set. Then receive of data checksum and state transition to ***uld_recend_2*** state are initialized. If the confirmation of the frame is required and reason to receive frame is not in effect of promiscuous mode, the confirmation character is sent and the state advances to (***uld_recend_3***). If promiscuous mode is enabled, check is performed whether the frame was confirmed by someone else. If not the message will be marked as incorrect (flag ***UL_BFL_FAIL***). If no confirmation is requested then the sub-state machines return control to upper level directly in state ***uld_recend_2***.

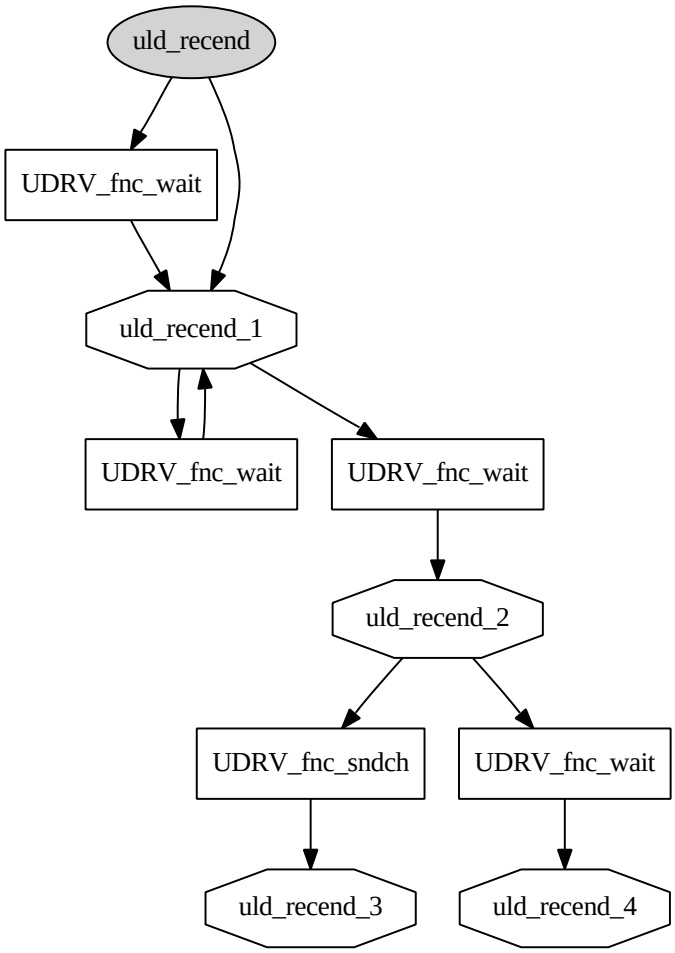

Figure 7: State diagram of the frame end receiption

3.9 Sending messages

If the queue of messages prepared for output/processing (***prep_ bll***) is not empty then the top-level state machine advances from idle state to the process of connecting to the bus using ***UDRV_ fnc_ connect*** (specific chip/hardware related functions beginning UDRV are described later). After the media arbitration and connect sequence is executed, the main finite state machine advances to state ***uld_ drvloop_ proc_ 1***.

The first action for ***uld_ drvloop_ proc_ 1*** state is moving of the first message in the output queue onto the internal working queue (***work_ bll***). The timeout function is set to ***uld_ drvloop_ proc_ error*** handler to protect bus against stale state. Timeout functions will be called if the preset timer overflows. Then the outgoing message processing state machine is activated by call to the ***uld_ prmess*** entry point, which takes care of the message itself and its functions will be explained in the next subsection.

If the timeout or other error is detected during message processing then function ***uld_ drvloop_ proc_ error*** re-attempt to send the message unless the message flag set **UL_BFL_NORE** or the maximum number of transmission attempts (***UL_ RETRY_ CNT***) does not exceed the limit. Otherwise, the message error flag (**UL_BFL_FAIL**) is set and placed it into the input queue which informs the higher protocol layers about inability to send a message to the recipient.

If the transmission of messages is correctly processed, the main state machine reaches state ***uld_ drvloop_ proc_ 2***. In this state, the timeout function is unregistered/set to empty and

bus/media control is relinquish by preparing and sending character with own module/device address and bits **D8** and **D7**. The character prepared in the buffer is sent to the media by invocation ***UDRV_fnc_sndch*** function and media is released. Finally, the message is checked for **UL_BFL_M2IN** flag. If the flag is set, the message is moved onto the input queue (***proc_bll***). Then the higher protocol layers are informed about message which processing status should be examined.

3.9.1 Outgoing message processing

The sub-machine is schematically illustrated in Figure 8.

The message can consist from multiple frames and each frame processing result in sending frame to the bus or reception of frame resulting from immediate action processing in other module/device. The decision about these two main options is taken in ***uld_prmess_frame*** state.

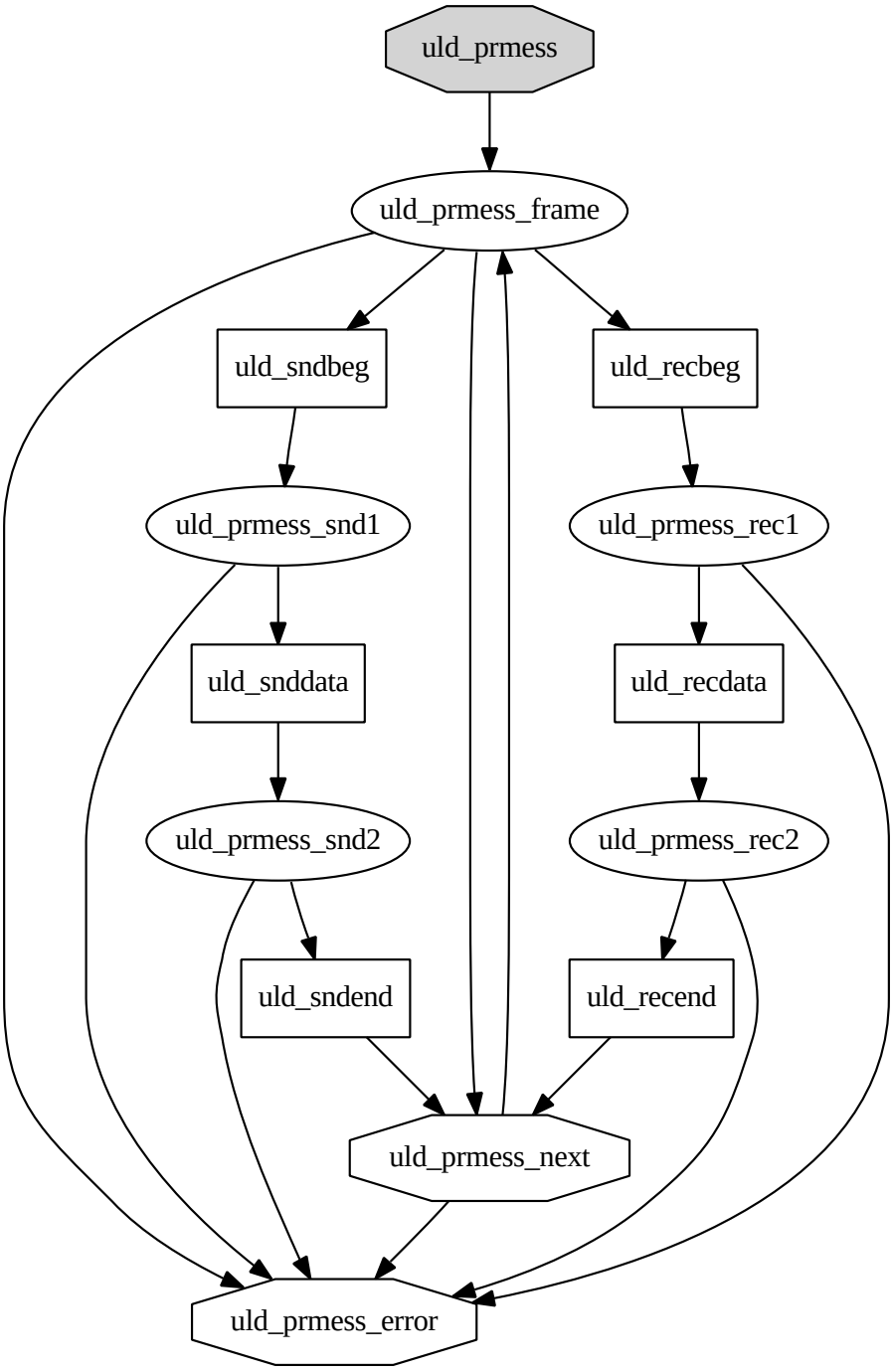

Figure 8: State diagram of outgoing message frames processing

If the current frame is marked for transmission (flag **UL_BFL_SND** is set) then connection state is set according to the frame header and a sub-machine fulfilling frame begin sent (***uld_sndbeg***) is activated. The processing executed by this machine is illustrated in Figure 9. The **DAdr** with **DB8** set is sent first. The **SAdr** and **CMD** follows.

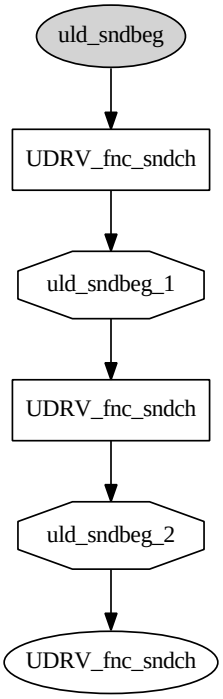

Figure 9: State diagram of frame header transmition

Subsequently, upper level state machine advances into the state ***uld_prmess_snd1***. In this state, the subroutine (***uld_snddata***) is called which is responsible for sending of actual data contained in the frame. The simple scheme of this activity is shown in Figure 10.

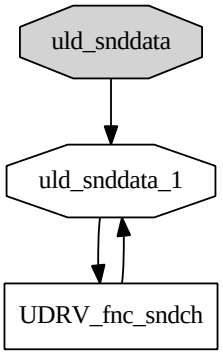

Figure 10: Diagram of the frame data transmition

After the data are sent, upper level state machine advances to the state ***uld_prmess_snd2*** which schedules sending of frame end sequence ***uld_sndend*** according to the connection state buffer. The state machine fulfilling this task is illustrated in Figure 11. The frame termination character (**DB8** bit set) is sent first. Which of the possible termination characters is sent, is controlled by connection flags. Then data integrity xor sum character is sent with **DB8** = 0. If the connection flags do not request that message confirmation from recipient is required then procedure is finished in this state. Otherwise, wait for the arrival of confirmation character is initiated. The cases of missing, incorrect or negative acknowledge are resolved as message processing error described above.

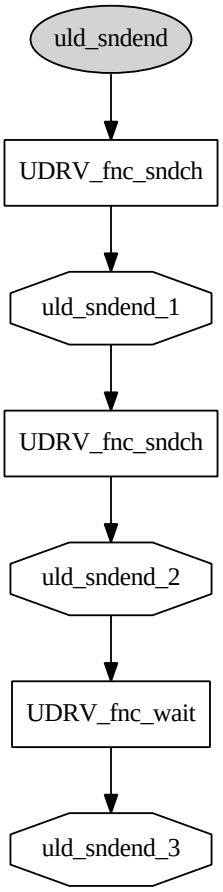

Figure 11: State diagram of the frame end transmition

If frame transmission is successfully completed the state machine lands in state ***uld_prmess_next***, where actual frame header is tested for flag **UL_BFL_TAIL**, which determines whether more frames should be processed in connection to this message. If the answer is yes then next frame processing continues by reenter to state ***uld_prmess_frame***. If the message is not multi-frame one final frame is reached, the subroutine ends there. It is possible that machine lands directly in the ***uld_prmess_next*** state from the state ***uld_prmess_frame***, if message frame does require no action (frame receive or send).

The processing path for frame receive (frame header flag **UL_BFL_REC** set) leads from state ***uld_prmess_frame*** to states ***uld_prmess_rec1*** and ***uld_prmess_rec2*** through the reception of frame header, data and the frame end. Data are stored in this upper layer prepared message frame in this case. The frame receive scenario is taken only for multi-frame messages where previous frame address and command device/module to reply by immediate response frame. The request for uLan network module identification (**uLCo_SID**) is one example for this scenario. The first frame is marked by **UL_BFL_TAIL** and **UL_BFL_SND** flags set which results in its sending to the bus and advancing to the next frame. The following frame (in **uLCo_SID** message block) has **UL_BFL_REC** flag set and its content is filled/replaced by the response frame sent by queried module/device and remains part of the whole message. The message will be processed in the higher layers as sent when it returns to the input queue (the flag **UL_BFL_M2IN** has to be set for that). This communication method is used because of the speed of response, omitting lengthy arbitration access to the bus, and because it can be processed without cooperation from higher protocol layers on the queried module/device. The bus is reserved to one master for the whole operation, which invites the recipient to send reply without arbitration cycle. The message

of this type (**UL_PRQ**) is addressed to provide bidirectional communication for immediate action procedures.

If there is any error detected in message processing, the above described error function is invoked.

3.10 Immediate Actions Processing

This subroutine is called in the receiving part of the machine if the message flag **UL_PRQ** is set. The immediate actions processing state machine is shown in Figure 12. The entry point of this sub-machine is ***uld_prmess_iac***. There is executed testing for possible registered immediate action according to value of **com** (command) field of received frame. If there is registered action for the received **com** value then the next processing is controlled by prescription specified by found immediate action structure. Possible actions are as follows:

- call callback function
- sending data already pre-filled in action buffer as a frame onto bus
- sending of data generated by user-defined function
- reception of the additional frame

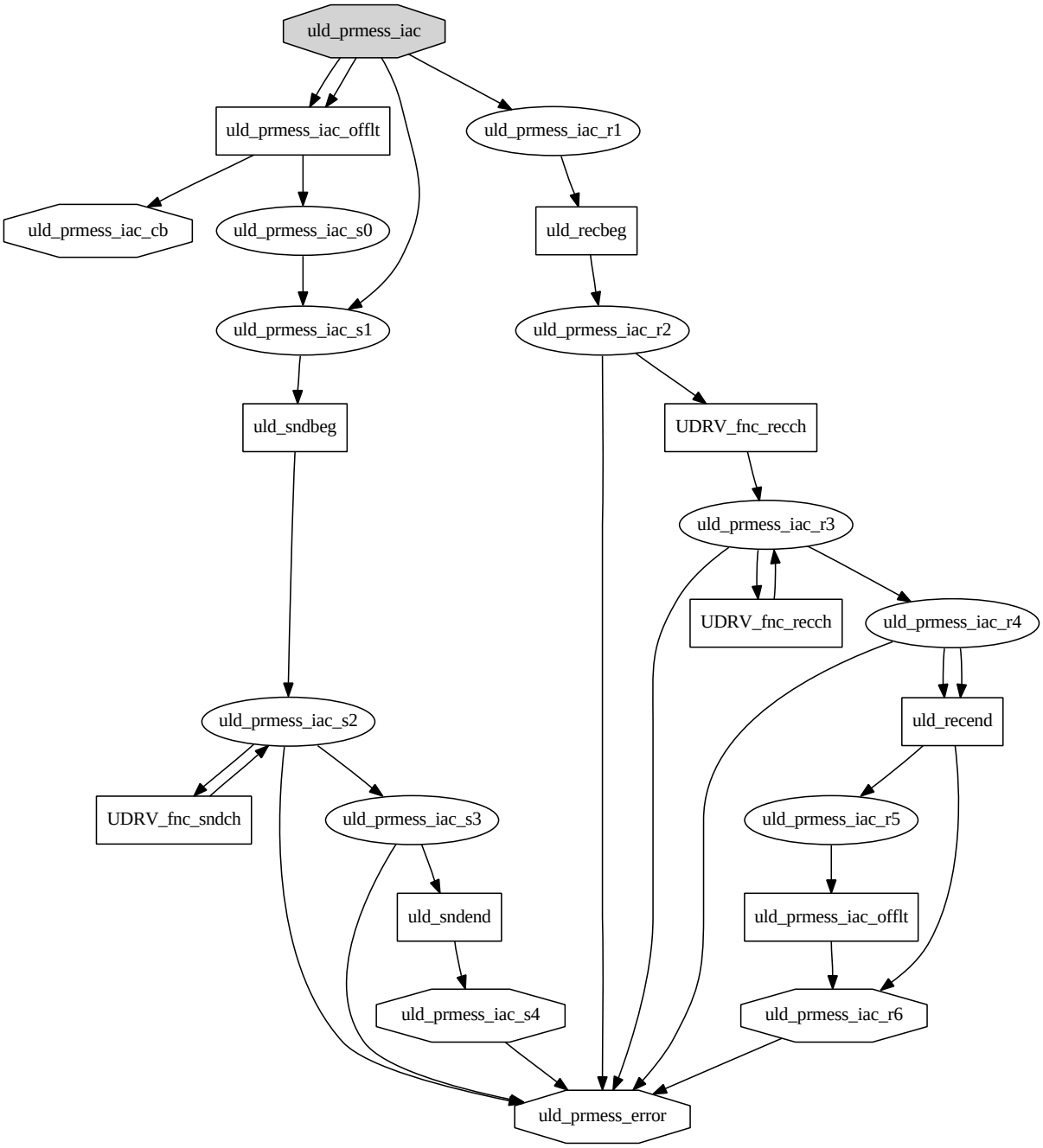

Figure 12: Diagram of the immediate response procedures

3.10.1 Callback functions

If the **UL_IAC_BFL_CB_OFFLT** is set in registered immediate action structure then the processing includes ***uld_prmess_iac_offlt*** state which ensures that transceiver output is disabled by call of chip driver function ***UDRV_fnc_finishtx*** (picture 13). The entire process is completed by calling the function ***uld_prmess_iac_cb***.

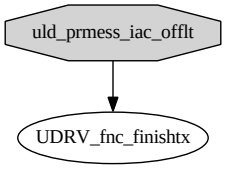

Figure 13: Callback functions

3.10.2 Sending frame on the bus

There are two variants of sending answers on the bus. In the first case (**UL_IAC_OP_SNDBUFF** action operation), the data already prepared in the action buffer during action registration are sent as consequence of processing state ***uld_prmess_iac_s1*** invocation. However, if the **UL_IAC_OP_SND** operation is specified the transceiver output is switched off and user provided callback function is executed in state ***uld_prmess_iac_s0***. This function can prepare data buffer content to be sent and has even access to the previous received frame header and data. In both cases, the actual data transmission begins in the state ***uld_prmess_iac_s1***. This state includes the preparation of the frame header, places 0x7f **DAdr** which corresponds to unspecified recipient target/response begin **UL_BEG**. When frame header is sent, the state advances to ***uld_prmess_iac_s2***. Then frame data are sent according to buffer specification in the action structure. The frame end with type **UL_END** follows.

3.10.3 Receiving frames

This procedure is very similar to the reception of the classic frames with one exception. Received frame data are not stored in any queue but state ***uld_prmess_iac_r3*** processing ensures that data are stored directly into buffer specified during immediate action registration. If for various reasons the frame processing fails or error is indicated the transition to state ***uld_prmess_error*** is executed error function is called.

3.11 Chip drivers

The hardware dependent functions are provided by chip driver support implementations. This design allows to use same driver core for different architectures, chips, providing implementations of an abstract state machine functions to connect to the bus (arbitration including access to the medium), reading and posting character. These functions are set by routine which initializes specific hardware support. There are already implemented drivers for different UART (universal asynchronous receiver/transmitters) types: 82510, 16450, OX16950 and MPC5200. The following subsection describes the concept of these functions on example implementation of these functions for the 82510 UART.

3.11.1 82510 UART Driver

The basic functions of this driver are characterized by state machine drawn in Figure 14.

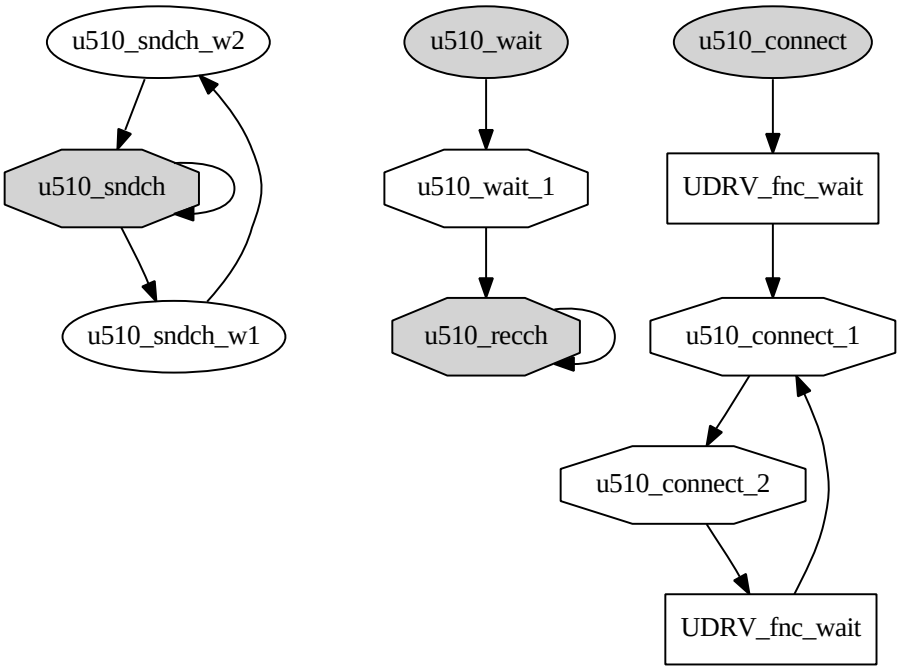

Figure 14: State diagram for 82510 UART chip driver

The 82510 chip character reception (***UDRV_fnc_recch***) is implemented in functions ***u510_recch***. The state diagram for this function consist of only one state. After sub-machine is entered, the check for possible active transmission of some character is checked first. If the character transmission is not finished, chip is instructed to activate interrupt when hardware transmit machine reaches idle state. If it is in idle state already or next interrupt occurs then it is ensured that transceiver output buffer is switched off. Then wait for character reception is initiated. If interrupt arrives or character is already in the receive buffer function stores character value into driver connection state character buffer and returns to upper level state machine. It is responsibility of chip initialization routine to register interrupt. The interrupt acceptance and call to ***u510_pool*** and if is active, calling actual state processing function, is responsibility of generic UFSM state machine implementation.

The function ***u510_wait*** has same responsibility for character reception but adds additional check for delay in character reply and if no character is received within a certain preset period the check in state ***u510_wait_1*** signals this condition and sub-machine returns with **UL_RC_ETIMEOUT** code. If chip signals that there is character received, processing advances to the same ***u510_recch*** function as reception without timeout is implemented.

Sending a character to the bus is provided by sub-machine with the entry point ***u510_sndch***. If the transceiver output to bus is not enabled a loop formed by ***u510_sndch_w1*** and ***u510_sndch_w2*** states and returning back to ***u510_sndch*** is activated. The transmitter output to bus is enabled between the two initial states of this loop and possible remaining input data are flushed. If the character is actually transmitted then processing is suspended until ready for next character condition is signaled by chip interrupt. Then the character is written into transmit holding buffer.

The sub-machine for the bus connect sequence (arbitration the medium) begins by the entry point ***u510_connect***. The time to wait before attempt to connect to bus is calculated from difference between lower bits of self address and last bus holder address and transition to state ***u510_connect_1*** is timed accordingly. If the character has been received during this period, it means that someone else has transmitted to a bus and this arbitration attempt is abandoned. If there is no character or line state detected then transmitter output is switched on and dominant state (logic level 0) is transmitted over one character period, after confirmation about character

transmission end the state machine moves to state ***u510_connect_2***. If all cycles of arbitration sequence are not finished yet, the inactivity/recessive period 1 to 4 characters is computed from value of two bits of own address and next dominant/recessive cycle is entered. This process is carried out three times and two of 6 lower bits of own addresses are processed in each cycle.

3.12 Higher Protocol Layer

- General support for the allocation, iteration, filling, handling and reading the messages consisting of one or more frames.

The layer providing general messages allocation, handling and distribution is placed between the uLan finite state machine (UFSM) and operating system specific driver interface. The implementation allows to run multiple applications over a single driver instance. The services provided by this layer are independent of chip hardware and operating system and and can be used even in system-less environment. The main representatives of this layer are the communication clients/operators related services which are provided through operating system specific system calls to user applications.

3.13 Driver Clients/Operators

Applications participating in uLan communication accesses driver through the objects that are called the driver clients/operators. The client/operator is a structure which represents state and communication interests of given user application driver open instance. The set of general layer functions is independent of used operating (Windows, Linux, system-less environment) and the same set of functions is provided to the application by system calls wrapper library. The client/operator is represented on driver side by **ul_opdata** state structure. The most important fields of the structure are:

- **udrv** - pointer to driver structure to which the client belongs to
- **opnext** and **opprew** - represent pointers to the preceding and the following instance of a list of clients/operators belonging to the same driver instance (hardware/chip device)
- **message** - a pointer to the first frame of actually prepared/processed message
- **data** - the iterator to access data in individual frames of the message
- **recchain** - start of the list of structures/members pointing to received or processed messages which waits for given client/operator processing. Some client specific tags for message are held in these structures as well
- **filtchain** - the purpose of this list is filtering of incoming messages. The list members are of same type members as a **recchain**. The list members holds information about message header parameters which are of interest of client/operator.

3.14 Client/Operator Services

The applications call functions located in uLan driver support library with the handle identifying the client. The handle type is operating system specific and application obtains it as return from ***ul_open*** function call. Essential services are:

3.14.1 Opening driver and registration of the client/operator

The client state is created ***ul_ open*** function call. The function first input parameter specifies requested driver, typically file name of device which has been registered during driver instance initialization. Multiple driver instances utilizing different UART or other uLan bus interface hardware can be used simultaneously. The uLan driver registered device names corresponding to each operating system conventions are shown in Table 1.

Name of equipment	Platform
\\.\UL_DRV*(x)*	Windows
/dev/ulan*(x)*	Linux
NULL	system-less or default driver instance

Table 1: Table name

Table 1 The device name of other than default driver instance includes driver instance/uLan interface (device minor) number *x*. The device numbers are typically assigned according to the order in which uLan interfaces are enumerated but there is possibility to control these numbers assignment under Linux through module parameters. If the device name is omitted (NULL) the default device is chosen which is typical use for case of a single driver instance.

If the driver instance specified by ***ul_ open*** first parameter is found, then client/operator data structure is allocated and corresponding handle is returned to the application.

3.14.2 Closing of the client/operator

This operation is provided by functions ***ul_ close***. First, it removes client remaining client messages and interest filters and then removes the client/operator state structure from the list of diver instance clients. Finally, the state structure memory is returned.

3.14.3 Frame data reading and writing

The ***ul_ read*** function copies data from the currently processed frame client/operator provided buffer. The ***ul_ write*** function works on the same principle with the difference that the direction of data flow is opposite.

3.14.4 Creating messages

The ***ul_ newmsg*** function creates new message and fills its firs frame header according to input parameters. The frame processing flag **UL_BFL_SND** is set and **message** field of client/operator state structure is set to reference this frame. If the message is intended to have multiple frames then consecutive frames are added to message by ***ulan_ tailmsg*** call. This function creates a frame according to provided parameters same way as ***ul_ newmsg*** but also sets the preceding frame of **message** chain as its predecessor.

3.14.5 Release of the message from the client/operator

The ***ul_ freemsg*** first first determines whether the message currently handled by the client has already allocated unique message **stamp**. If the message **stamp** is already assigned then the actual message represent received or processed message and its reference counter is lowered and message is possibly released from memory. Otherwise message is intended for sending/processing and the unique **stamp** identifier is not previously set. The **stamp** is allocated and filled into the message and message is passed for further processing. Furthermore, if the ***UL_BFL_M2IN*** message flag is set then the member for filter list is allocated and its **stamp** field is filled by same value as filled in message, its state is set to **UL_OPST_ONCE** and then it is inserted in client/operator filter chain (**filtchain**). The importance of this behavior will be explained later in

the section devoted to clients. Finally, the message is forwarded to the output queue for further processing by the driver which executes message frame sending and other processing.

3.14.6 Message pickup by application

The application asks driver if there is some message fulfilling declared interests by call to ***ulan_acceptmsg*** function. The messages passing criteria specified by members of **filtchain** are collected in **recchain** queue. The first member of this queue is set-up as actually processed (**message** field of client/operator state) and information about first frame of the message is returned by call. If the message has multiple frames, the ***ulan_actailmsg*** function is used to move on next frame and to acquire attributes of its header.

3.14.7 Adding a filter member in the client/operator filtchain

The application specifies which messages should be delivered to it by filter chain members registration. Adding of members is requested by call to ***ul_addfilt*** function. It allocates and fills a member in accordance with provided input parameters. Messages of client/operators interest may be characterized by first frame header attributes (**DAdr**, **SAdr**, **com** and **stamp**). Then the member filtering method is selected (should trigger only once or multiple times) and member is put on filtchain.

3.14.8 Setting driver attributes

The library also contains functions that directly set driver control attributes. One such function is ***ul_setmyadr*** which is used to set own module/device address by which device is identified in the uLan network. Another example is the function ***ul_setpromode***, which provides activation and deactivation of promiscuous driver mode, in which the all traffic on the bus is is monitored and can be delivered to the appropriate clients/operators. The ***ul_setidstr*** function sets data for immediate response to the reception of identification request.

3.15 Messages Announcing

The received or processed messages are moved onto input queue (***proc_bll***) and further processing is scheduled. The message analysis and distribution to operators is deferred to ***ulan_do_bh*** procedure. This function retrieves messages from input queue one by one and calls function *ulan_proc_arrived* for each of them. The arrival processing function matches first frame header attributes against interests of all operators by comparing attributes (**DAdr**, **SAdr**, **com** and **stamp**) to ones specified in **filtchains** members. Only non-zero value attributes of filter chain members are significant in comparison. The filter chain includes even members responsible for delivering processed messages generated by given client/operator back into its queue for processing result check. If the filter member is found satisfying the above conditions, further action is to deliver message to corresponding client/operator. Message is not copied but filter member structure is directly used or copied as reference to given message and it is queued onto client/operator **recchain** list.

If a filter list member state is **UL_OPST_ONCE** then filter members is directly moved to receive list which effectively removes given filter member from future match testing. If the filter member state is **UL_OPST_FILT**, the copy if filter member is allocated and only that copy provides message reference in client/operators **recchain** and original filter member ensures that all other messages matching criteria will be delivered to given client/operator.

3.16 RS-485 Converter

In figure 15 is shown schematic diagram of simple RS-232 to RS-485 dongle converter. This converter is powered directly from ±12V signals of regular RS-232 port. RTS signal is used for enabling of line driver. CTS signal is used for direct line logic level reading. Interconnection of

TxD and DSR enables timing by sending of zero characters when output driver is disabled. Then modem status interrupt can serve as transmit machine empty interrupt, which is not natively implemented in PC COM hardware.

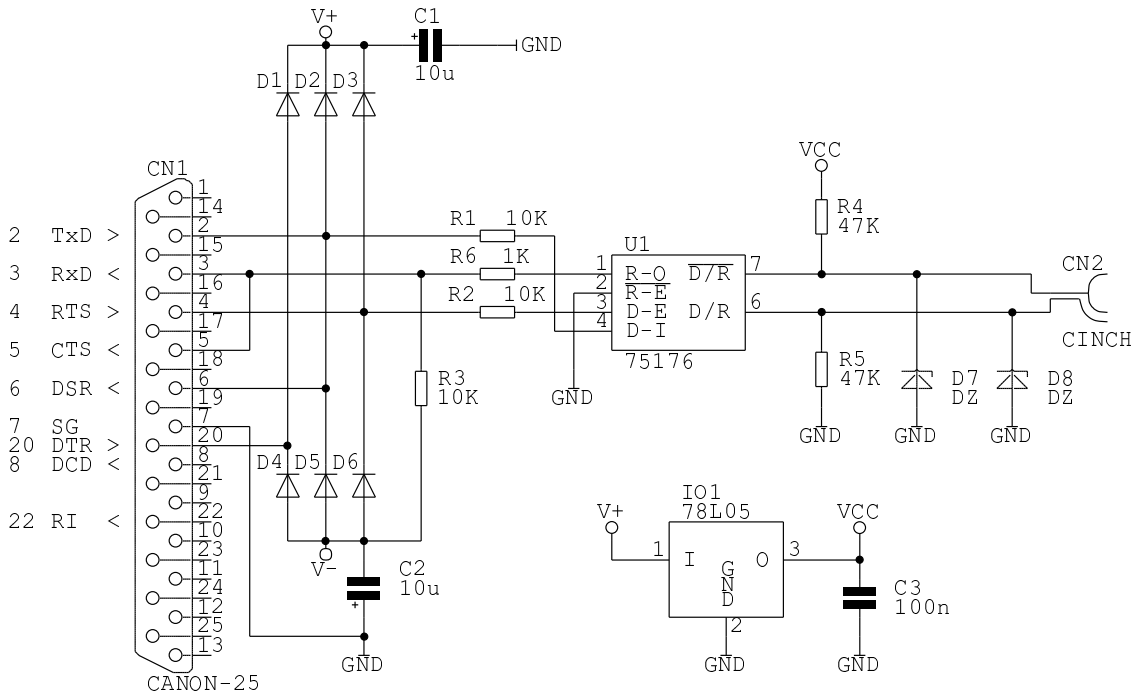

Figure 15: RS-485 Dongle Converter

The converter without external power supply is not well suited for large networks with long cables. But it is elegant solution for networks of up to 10 nodes. The custom ISA card with 82510 is available for large networks. PCI industrial cards with OX16C954 are solution for professional high demanding environment.

4 uLan Interface and Services

4.1 Message Sending and Reception

Every uLan interface (hardware port) is seen as one special character file under **Linux** operating system or system device under **WindowsNT**. All client program operations are accessible through standard `open`, `close`, `read`, `write`, `ioctl` and `select` system calls. The `ioctl` calls are used for preparation and sending of new messages and for selecting, which external messages will be received by associated file handle. The `ul_msginfo` structure is used for all message and frame oriented `ioctl`s.

```
typedef struct ul_msginfo {
        int     dadr;           /* destination address */
        int     sadr;           /* source address */
        int     cmd;            /* command/frame type */
        int     flg;            /* message flags */
        int     len;            /* length of frame */
        unsigned stamp;         /* unique message number */
} ul_msginfo;
```

Next table describes `flg` field of `ul_msginfo`, which requests and reflects state of processing of message or its frames by the driver.

Name	Description
UL_BFL_LOCK	locked message is pointed only once
UL_BFL_MSST	Message must be received by some proces
UL_BFL_M2IN	After successful processing inform sending client
UL_BFL_LNMM	Length of received frame must match expected len
UL_BFL_FAIL	Message cannot be processed - error status
UL_BFL_TAIL	Multiframe message continues by next bll block
UL_BFL_SND	Send this frame
UL_BFL_REC	Receive answer frame into this bll block
UL_BFL_VERL	Verify free space in buffer of destination station
UL_BFL_NORE	Do not try to repeat if error occurs
UL_BFL_REWA	If error occurs do wait with retry
UL_BFL_PRQ	Request immediate processing of frame by receiving station
UL_BFL_ARQ	Request immediate acknowledge by receiving station

A client state description structure is build by the driver and connected to the kernel file description after `open` system call. Client state is used for preparation and reception of messages and stores list of message filters for reception. Client applications manipulates with their respective state through `ioctl` system calls. The client uses standard `read` and `write` system calls for incremental reading of received message frame or filling newly created message frame. Next `ioctls` are defined.

Name	Argument	Ptr data dir	Description
UL_DRV_VER			Returns driver version
UL_NEWMSG	ul_msginfo *ptr	W	Prepare new outgoing message
UL_TAILMSG	ul_msginfo *ptr	W	Add next frame to message
UL_FREEMSG			Free (send) prepared or received message
UL_ACCEPTMSG	ul_msginfo *ptr	R	Accept message from receive queue
UL_ACTAILMSG	ul_msginfo *ptr	R	Accept next frame of multiframe message
UL_ADDFILT	ul_msginfo *ptr	W	Add specification for receiving of messages
UL_ABORTMSG			Abort preparation of message
UL_REWMSG			Return to beginning of first frame of message
UL_STROKE			Restart processing loop of driver
UL_DEBFLG	debug mask		Set kernel driver debug level
UL_HWTEST	subcommand		RS-485 buffer hardware checking

Opening and closing uLan file handle

The file descriptor (handle) is obtained from `open` system call with correct device name parameter. The conventional name for first interface is "`/dev/ulan`" for Linux and "`\\.\UL_DRV`" for WindowsNT based systems.

```
int ul_fd;
ul_fd=open(ul_dev_name, O_RDWR, S_IWRITE | S_IREAD);
if(ul_fd<0)
{ perror("print_nodes : uLan open failed");
```

```
  return -1;
};

/* uLan communication through ul_fd possible */

close(ul_fd);
```

Sending of message or command

New message must be created with module destination address, command type and flags. Data write is optional. Message is send into output queue for transmission after client `UL_FREEMSG` `ioctl`. This `ioctl` returns message unique identifier to the sender of message and for messages with `UL_BFL_M2IN` flag prepares single shot filter which serves for client transmission result notification.

```
int send_command(int ul_fd,int dadr,int cmd,int flg,void *buf,int len)
{ int ret;
  ul_msginfo msginfo;
  memset(&msginfo,0,sizeof(msginfo));
  msginfo.dadr=dadr;
  msginfo.cmd=cmd;
  msginfo.flg=UL_BFL_M2IN|flg;
  ret=ioctl(ul_fd,UL_NEWMSG,&msginfo);
  if(ret<0) return ret;
  if(len)if(write(ul_fd,buf,len)!=len)
  { ioctl(ul_fd,UL_ABORTMSG);
    return -1;
  };
  return ioctl(ul_fd,UL_FREEMSG);
};
```

Well written Linux application takes minimum processor time. Application should call select with list of monitored file descriptors after processing of all available events and data. Code with only one such descriptor for **uLan** driver is presented here.

```
int ul_fd_wait(int ul_fd, int wait_sec)
{
  int ret;
  struct timeval timeout;
  fd_set set;

  FD_ZERO (&set);
  FD_SET (ul_fd, &set);
  timeout.tv_sec = wait_sec;
  timeout.tv_usec = 0;
  while ((ret=select(FD_SETSIZE,&set, NULL, NULL,&timeout))==-1
          &&errno==-EINTR);
  return ret;
}
```

Next example sends command and waits for result of transmission processing. Wait loop is added for case that client has added more message filters before command send or sends some commands without waiting for results. I such case received or processed messages reported before last send command are ignored. Bigger applications should wait for received or processed messages in main event loop and report received or processed messages as events to other objects of application.

```
int send_command_wait(int ul_fd,int dadr,int cmd,int flg,void *buf,int len)
{ int stamp;
  int ret;
  ul_msginfo msginfo;
  stamp=send_command(ul_fd,dadr,cmd,flg,buf,len);
  if(stamp<0) return stamp;
  while(1)
  { ret=ul_fd_wait(ul_fd,10);
    if(ret<=0) return ret?ret:-1;
    ret=ioctl(ul_fd,UL_ACCEPTMSG,&msginfo);
    if(ret<0) return ret;
    ioctl(ul_fd,UL_FREEMSG);
    if(msginfo.stamp==stamp)
    { if(msginfo.flg&UL_BFL_FAIL) return -2;
      else return 1;
    };
  };
};
```

Sending immediate query

Most of field-bus area networks have one specific feature missing and in most cases impossible to implement in LAN or WAN area networks. It is ability to request and receive answer immediately after last byte of query command. This feature is defined and generalized in **uLan** network. Simplest case consisting of master to slave command frame and slave to master immediate reply is discussed here. As in above examples, code is divided into part which prepares and sends message and and result waiting loop. The second part should be implemented as event handling for bigger applications.

```
int send_query(int ul_fd,int dadr,int cmd,int flg,void *buf,int len)
{ int ret;
  ul_msginfo msginfo;
  memset(&msginfo,0,sizeof(msginfo));
  msginfo.dadr=dadr;
  msginfo.cmd=cmd;
  msginfo.flg=UL_BFL_M2IN|flg;
  ret=ioctl(ul_fd,UL_NEWMSG,&msginfo);
  if(ret<0) return ret;
  if(len)if(write(ul_fd,buf,len)!=len)
  { ioctl(ul_fd,UL_ABORTMSG);
    return -1;
  };
  memset(&msginfo,0,sizeof(msginfo));
  msginfo.flg=UL_BFL_REC|UL_BFL_M2IN;
  if(ioctl(ul_fd,UL_TAILMSG,&msginfo)<0)
  { ioctl(ul_fd,UL_ABORTMSG);
    return -1;
  };
  return ioctl(ul_fd,UL_FREEMSG);
};
```

Next function sends query to module dadr with command `cmd` and `lenin` bytes from buffer `bufin`. Destination module receives frame processes command and data and sends immediate reply frame.

Data from reply frame are returned to caller as pointer `*bufout` to malloced memory block. Number of received data bytes is reported as `*lenout`.

```
int send_query_wait(int ul_fd,int dadr,int cmd,int flg,
                    void *bufin,int lenin,void **bufout,int *lenout)
{ int stamp;
  int ret;
  int len;
  ul_msginfo msginfo;
  stamp=send_query(ul_fd,dadr,cmd,flg,bufin,lenin);
  if(stamp<0) return stamp;
  while(1)
  { ret=ul_fd_wait(ul_fd,10);
    if(ret<=0) return ret?ret:-1;
    ret=ioctl(ul_fd,UL_ACCEPTMSG,&msginfo);
    if(ret<0) return ret;
    if(msginfo.stamp==stamp)
    { if(msginfo.flg&UL_BFL_FAIL)
        {ioctl(ul_fd,UL_FREEMSG); return -2;};
      ret=ioctl(ul_fd,UL_ACTAILMSG,&msginfo);
      if(ret<0) {ioctl(ul_fd,UL_FREEMSG); return ret;};
      if(bufout&&lenout)
      { len=msginfo.len;
        if(!*bufout) *bufout=malloc(len);
        else if(*lenout<len) len=*lenout;
        if(read(ul_fd,*bufout,len)!=len)
          {ioctl(ul_fd,UL_FREEMSG); return -3;};
        *lenout=len;
      };
      ioctl(ul_fd,UL_FREEMSG);
      return msginfo.len;
    };
    ioctl(ul_fd,UL_FREEMSG);
  };
};
```

Receiving external messages

Filters must be specified to receive external messages by client. The filter specifies which messages with which source (`sadr`), destination (`dadr`) and command/message type (`cmd`) should be reported to application. Each client could register more filters. To prevent filters duplication in kernel client state, `UL_BFL_NORE` flag could be specified. Then repeat registration of same filter is ignored. Assignment of zero value to some of fields results in ignoring of such field when message is reported to clients.

```
ul_msginfo msginfo;
memset(&msginfo,0,sizeof(msginfo));
msginfo.dadr=filt_dadr;
msginfo.sadr=filt_sadr;
msginfo.cmd=filt_cmd;
ret=ioctl(ul_fd,UL_ADDFILT,&msginfo);
if(ret<0) { printf("add filter failed\n");return ret;};
```

Application interface routines are part of provided **uLan** library. The library includes above discussed routines and simple **uLan** object interface communication layer implementation. uLan

application to library interface is operating system independent and actual version of the library and driver can be compiled for **Linux**, **DOS** and **WindowsNT/2000**.

4.2 Query Module Type

Query for type of module with specified address returns short string describing connected module. Every **uLan** communicating device or module should implement immediate processing of `UL_CMD_SID` command. Query can be used for initial searching for all connected active modules. Next function call with previously opened driver handle returns in case of success allocated buffer filled by null terminated string describing module with `module_adr`. Negative return value indicates error - no reply from module or other problem.

```
ret=ul_send_query_wait(ul_fd, module_adr, UL_CMD_SID, UL_BFL_NORE
        |UL_BFL_PRQ, NULL, 0, (void**)&buf, &buf_len);
```

Example of returned string : "`.mt MDET v0.4a .uP 51x .dy`"
String should conform to next rules :

- it should contain "`.mt`" tag followed by space and one word module or device type
- software version can follow after next space
- other specifications with "`.xx`" tags can follow

Next tags are specified:

- "`.mt`" module type
- "`.mv`" module vendor short name
- "`.uP`" microprocessor family, used for selection of right debugger
- "`.dy`" module supports dynamic address assignment

4.3 Network Control Messages

This one frame message of `UL_CMD_NCS` type is used for changing of module address and other network management purposes. Message processing is not strictly required for all developed modules. Same messages result in sending of reply. Reply is send as regular queued message in **uLan** multi-master environment, no interrupt immediate processing is necessary. Many of commands use module unique four bytes serial number assigned by producer. It is necessary for dynamic address assignment described in next paragraph. Producer is responsible that no pair of all existing modules has same number. Main **uLan** serial number authority is PiKRON Ltd.[5] company. When module wants to use NCS and its serial number could not be assigned, it must report serial number with the most significant byte `SN3` equal to `0xFF`. Random constant values for rest of serial number can sometimes help or be used for experimental devices. Next sub-commands specified by the first data byte are used:

Subcommand	Format after first byte	Description
`ULNCS_RQ_ADDR`	`SN0 SN1 SN2 SN3`	Request for new address
`ULNCS_SET_ADDR`	`SN0 .. SN3 NEW_ADR`	Set new module address
`ULNCS_SID_RQ`		Request identification reply
`ULNCS_SID_RPLY`	`SN0 .. SN3 SID_string`	Reply with SN and identification
`ULNCS_ADDR_NVSV`	`SN0 SN1 SN2 SN3`	Permanent address change

Module can send request for new address after power-up or when its find that it is not in database of dynamic address server. Message can be send to the found server or as broadcast (zero destination address). New address can be send to module at any time by server or on user request. Serial

number check is critical at this phase, it ensures, that two devices with same initial address could be distinguished. Recommended initial address for devices in highly dynamic and changing environment is 0 or 99. The NCS type request for identification has two advantages over basic `UL_CMD_SID`. There is no need for all address scanning, message could be send as broadcast and then all responses could be collected. Reply contains module unique serial number. Returned string is same as for `UL_CMD_SID`. Permanent address change could be useful for static environment with devices which have no keyboard or jumpers for address selection.

4.4 Dynamic Address Assignment

This protocol is mainly defined for environments witch frequent changes, working modules plug and transfer between networks. It demands and uses serial number comparison, implementation of network control services `UL_CMD_NCS` command and immediate status reply command `UL_CMD_SNST`. One dynamic address server must be connected to each network and status check cycle must be repeated. Each cycle first status check message with subcommand `<=3` and broadcast address serves as dynamic server address publication. Server than individually asks all known modules for status (subcommand `16`) and collect this information for other purposes. After connected module receives three times server start cycle message and no status check message, it uses `ULNCS_RQ_ADDR` to request address assignment from server. For case of reconnecting of working module from one to another network, address collision could occur. Next checking prevents such situation. When module receives status query with different serial number, module address is reset to zero and address assignment request is send to the server. Serial number check could be requested as the first command of object interface message as well. This protect reception of module properties writes, reads and commands for critical time window after reconnection of module into different network and reception first status query message.

5 uLan Object Interface Layer

The uLan object interface allows to access (querying and setting) device (node/instrument) objects (properties, variables and services activations in this meaning) organized in dictionary. Each objects has assigned identifier number (OID) by which it can be found in a given directory in object dictionary. The services to list objects accessible for reading and writing are defined. The original design allowed to use same OID with different data type for read and write in theory, but actual use, current design and upper level libraries expect, that same type is used for read and write if both functions are available. A message format is designed to be as short as possible, but does not compromise generality. This approach results in more complex node/instrument profile. The uLan Object Interface enables to store object descriptions directly into devices (nodes/instruments) and provides way to automatically build node profile from the descriptions.

5.1 Object Interface Messages and Basic Services

For communication with Object Interface (OI) in device command `UL_CMD_OISV` (0x10) is used. The general OI message data structure on the network is composed from 3 byte header and serialized OI data.

The OI header bytes are defined to identify request reply correspondence and to specify under which command reply is send:

bcmd	sn	bsn
1 byte	1 byte	1 byte

where:

bcmd back command used as destination command for reply

sn serial number (if unused set to 0, if used then range 0x40 to 0x7f should be used)

bsn back serial number, copied serial number of request into reply

From the accepted request message (msginfo) provides information about command cmd (0x10 for OI) and request source address (adr). The reply is sent to the originator address of request with command value defined by bcmd header field.

The header reply on a OI message is generated by this rules:

where	value is request	value in reply
frame Cmd	cmd	bcmd
data[0]	bcmd	bcmd
data[1]	sn	bsn
data[2]	bsn	sn

After OI message header can be serialized multiple OI statements at the format:

OID number	OI_data
2 bytes LE	data bytes or meta specification (0...n)

OID (object identifier) defines key for access of the object data or service activation

OI_data specifies contents of the OI number.

OID number and OI_data repeats in the data stream as many times as needed. Protocol reserves OID numbers from 1 to 127 for protocol internal objects and services. The range from 128 to 32767 is available for user.

The next OID numbers are defined for protocol services:

Name	OID	hex	Parameters	Description
ULOI_AOID	10	0x0A	<string>	substitutes numeric OID by ASCII name for I_DOII and I_DOIO
ULOI_DOII	12	0x0C	<oid>	query description for writable/in <oid> returned ULOI_DOIIr
ULOI_DOIIr	13	0x0D	<oid> <oid-desc>	description of writable/in <oid>
ULOI_DOIO	14	0x0E	<oid>	query description for readable/out <oid> returned ULOI_DOIOr
ULOI_DOIOr	15	0x0F	<oid> <oid-desc>	description of readable/out object <oid>
ULOI_QOII	16	0x10	<oidfrom> <maxret>	query up-to <maxret> writable/input OIDs list starting from <oidfrom> or 0 for first
ULOI_QOIIr	17	0x11	<oid0>... <oidn><0>	list of writable/in OIDs ended by 0
ULOI_QOIO	18	0x12	<oidfrom> <maxret>	query up-to <maxret> readable/out OIDs list starting from <oidfrom> or 0 for first
ULOI_QOIOr	19	0x13	<oid0>... <oidn><0>	list of readable/out OIDs ended by 0
ULOI_RDRQ	20	0x14	<oid0>... <oidn><0>	object values read request, sequence of (OID[,meta]) groups, activates reply ULOI_RDRQr
ULOI_RDRQr	21	0x15	<oid0><...> ... <oidn><...><0>	reply sequence of (OID,[meta,]value) groups
ULOI_SNCHK	29	0x1D	<SN0 .. SN3>	check module serial number

I_STATUS	30	0x1E		standard OID for 16 bit status read (0 not active, positive busy, negative error)
I_ERRCLR	31	0x1F		standard OID for clear error status

5.1.1 ULOI_AOID

Reference on the OID numbers should be rather specify by self name. This conversion is mainly used for the ULOI_DOIx, where ULOI_AOID switch from value references into ASCII reference.

Network ULOI_AOID references:

ULOI_AOID	OID by name
0x0A 0x00	vs (visible string)

Let's have OID defined by OID_number 220 number has 'TEMP' text description

Network format for regular reference to this object is

OID_number
0xdc 00

Same object access with symbolic ULOI_AOID reference is

ULOI_AOID	name size	name
0x0A 0x00	0x04	0x54 0x45 0x4D 0x50

5.1.2 ULOI_DOII/ULOI_DOIO

ULOI_DOII/ULOI_DOIO numbers are used to get name and data types descriptors of the specific object defined by OID number. Network data format for this commands are defined:

ULOI_DOII/ULOI_DOIO	OID_number
0xC/0xE 0x0	2 bytes LE

Reply data format for ULOI_DOIx numbers are defined:

ULOI_DOIIr/ULOI_DOIOr	OID number	text descriptors (name, type)
0xD/0xF 0x0	2 bytes LE	2 level nested length byte, data block structure

The minimal version of the text descriptors contains:

ND name descriptor

TD type descriptor

The text descriptors encoding follows:

total descriptors len	ND_size	ND_text	TD_size	TD_text
1byte	1byte	ND_size bytes	1byte	TD_size bytes

Example of the data format to get text description of the input OI number 31 is:

Request:

ULOI_DOII	OID_number
0xC 0x0	0x1F 0x00

Reply should be:

ULOI_DOIIr	OID_number	tot. len	ND_size	ND_text	TD_size	TD_text
0xD 0x0	0x1F 0x00	0x9	0x6	0x45 0x52 0x52 0x43 0x4C 0x52	0x1	0x65

Name of the object with OID_number 0x1f is "ERRCLR" and the data type is "e" - command to execute.

5.1.3 ULOI_QOII/ULOI_QOIO

ULOI_QOII/ULOI_QOIO numbers are used to get list of input/output OID numbers. Network data format for this commands are defined:

ULOI_QOII/ULOI_QOIO	start OID_number or 0	max reply count
0x10/0x12 0x0	2 bytes LE	2 bytes LE

Reply data format for ULOI_QOIx numbers are defined:

ULOI_QOIIr/ULOI_QOIOr	n x OID_number	termination
0x11/0x13 0x0	n x 2 bytes LE	0x00 0x00

Example of the data format to get list of input OI numbers is:

ULOI_QOII	start OID_number or 0	max reply count
0x10 0x0	0x00 0x00	0x40 0x00

Reply should be:

ULOI_QOIIr	OID	OID	OID	OID	OID	OID	termination
0x11 0x0	0xC 0x0	0xE 0x0	0x10 0x0	0x12 0x0	0x14 0x0	0x1F 0x0	0x00 0x00

In this example OID numbers 14,16,18,20,31 are listed.

5.1.4 ULOI_RDRQ

The basic object dictionary directory level is used for direct write to the objects in main directory and for acquiring lists and information about these objects. When values of objects should be read, the level is switched into ULOI_RDRQ level in which only OIDs and meta data are serialized. The ULOI parser responses to the switching into read level by generating reply with actual objects values filed in.

The reserved OID number 0 is used for return back to the write direction.

Example of the data format to read value of OID number 230 with type u2:

Request:

ULOI_RDRQ	OID_number
0x14 0x00	0xE6 0x00

Reply should be:

ULOI_RDRQr	OID_number	u2 data read from object
0x15 0x00	0xE6 0x00	0x63 0x00

In this example OID number 230 has value 99 (u2).

Example of the data format to set OID number 230, value 100(u2)

Request:

OID_number	u2 data to write into object
0xE6 0x00	0x64 0x00

Example of the data format to set OID number 231, value 16 (u1) and immediately get this OID value in one data frame is:

Request:

OID_number	u1 data to write	ULOI_RDRQ	OID_number
0xE7 0x00	0x10	0x14 0x00	0xE7 0x00

Reply should be:

ULOI_RDRQr	OID_number	u1 data read from object
0x15 0x00	0xE7 0x00	0x10

5.2 uLOI Data Exchange

All data exchange is using little-endian format. The data length is not encoded in the OI read/write messages to preserve space. The data types and lengths has to be known/retrieved on the uLOI dictionary access client side in advance. The basic concept for data exchange is based on sequences of bytes introduced by OID number and optionally additional meta data (array index, range etc.). Support for multiple directory levels of objects is possible but not in use for now. Sequences of OIDs or OID+data replies are terminated by zero OID (two zero bytes), which returns processing state back to previous level.

The basic level is used for direct write to the objects in main directory and for acquiring lists and information about these objects. When values of objects should be read, the level is switched into ULOI_RDRQ level in which only OIDs and meta data are serialized.

5.3 uLOI Data Types

Basic numeric data types use convention where type base is specified by single character and numeric suffix specifies number of bytes used for type encoding.

code	description	encoded
s1, s2, s4	signed integer	little-endian bytes sequence of corresponding length
u1, u2, u4	unsigned integer	little-endian bytes sequence of corresponding length
f4, f8	floating point number	according to IEEE-754, little-endian, 4-bytes/8-bytes
f2	half float (float16)	
vs, vsXX	visible string in utf-8	the first byte specifies length of the string in bytes (0-127) encoded in following bytes

The display format of the numeric types can be nail down by additional suffixes specified after "/" character. The dot "." is used to specify fixed decimal digits precision for fixed point numeric formats. The specification "u4/.3" specifies 4 byte unsigned format with 3 decimal digits.

The arrays of the objects of the given type are specified by adding of array specification "[]" before type. If the index range is known and fixed than the number of array items can be specified between brackets.

For example "[10]s2" means array of 10 16-bit signed integers.

5.3.1 Visible string

Visible string should defined maximal size of ASCII char in the text description followed after "vs". "vs12" defines string with maximal ASCII size of 12 chars.

Network format for visible string is:

length	text
1 byte (0..127)	length byte(s)

Example of the visible string network format:

0x04	0x41	0x42	0x43	0x44

The example represents encoded text "ABCD"

5.3.2 Arrays

The network format for array access encodes index or index range as meta data and then sequence of array items follows directly as data part of the stream.

The request for array items read contains only meta data part (index or index range) without data. The reply contains index/index range followed by items data sequence.

Index starts from value 0. The highest 2bits in first byte in data part are reserved in this meaning:

bit 7	bit 6	description
0	0	indexed single data item
1	0	indexed range of items
1	1	array command

Indexed data

The network format for for meta-data part of request for indexed data item is:

index
2 bytes LE (0...16363)

In a write write operation or reply containing data, the index is encoded same way and next data bytes follow encoded same way as for scalar value.

Example of the array indexed data communication:

Request (index 1) for array of type u4:

index 1
0x01 0x00

Reply (index 1, data type u4, array[1]=17):

index 1	u4 data for index 1
0x01 0x00	0x11 0x00 0x00 0x00

5.3.3 Indexed range of data

The network format for meta-data part of request for indexed range of data is:

count	index
2 bytes LE (0...16363) | 0x8000	2 bytes LE

Reply for indexed data or data write contains same range specification as for request and then required count of items is encoded according to standard rules. The returned count can be adjusted according to the actual array size. If only one item can be returned, range is encoded according to single data item index specification rules.

Example of the array indexed data communication:

Request (index 1,count 2):

count	index
0x02 0x80	0x01 0x00

Reply (index 1 and 2, data type u1, array[1]=18, array[2]=6):

count	index	u1 data for index 1	u1 data for index 2
0x02 0x80	0x01 0x00	0x12	0x06

Array commands

Not currently used in this protocol version.

6 uLan Connection Network

This subsystem is designed for direct process data (PDO) exchange between devices (nodes/instruments). Each data transfer is identified by connection ID (CID). Design allows to map one or multiple uLOI dictionary objects (properties, variables) as data source or destination for given CID. The mapping is stored directly in devices. Mechanism allows to transfer multiple CID identified data values in single message. Receiver identifies data scope only by CID, no source address or device internal uLOI OID assignment or meta-data format is encoded in PDO messages or directly influence the processing. This allows to connect objects with different OIDs, group multiple objects under single CID, use broadcast to distribute data into multiple destination devices or even use more devices as data source for same CID. When device receives PDO message it process each CID identified data according to configured mapping. CIDs and their respective data for which no mapping is found are simply skipped. There is only one requirement for the data and it is at least compatibility of data types for mapped source and destination OIDs. If destination type is shorter then source, remaining bytes are skipped, counter case is illegal for actual implementation.

6.1 Process Data Messages Format

Command `UL_CMD_PDO` (0x50) is specified for PDO messages. Message format starts with two reserved bytes for future static extensions and one byte follows, which can be used for dynamic PDO messages header extensions in future. These bytes should be send as zero for current protocol version. Each data block is preceded by its CID and data length. Maximal individual data block length is 127 bytes for actual implementation and is encoded in single byte. Format allows future extension to two bytes in future if needed.

Res Lo	Res Hi	Ext len (el)	Ext	CID	data len (dl)	data	CID ...
1 byte	1 byte	1 byte	0..el bytes	2 bytes LE	1 (2) byte	dl bytes	

6.2 Connection ID to Objects Mapping

All configuration/mapping of PDO data source and processing of received PDO messages is done through device objects dictionary and data and meta-data formats are compatible with uLOI layer. Next objects are used for objects to CID specific data mapping.

Name	OID	hex	Parameters	Description
ULOI_PICO	40	0x28	array "[]{u2,u2,u2,u2}"	PDO input CID/OID mapping
ULOI_POCO	41	0x29	array "[]{u2,u2,u2,u2}"	PDO output CID/OID mapping
ULOI_PIOM	42	0x2A	byte array "[]u1"	PDO input/output mapping meta data
ULOI_PEV2C	43	0x2B	array "[]{u2,u2,u2,u2}"	PDO event to CID mapping
ULOI_PDC2EV	44	0x2C	array not specified yet	PDO data change event mapping
ULOI_PMAP _CLEAR	45	0x2D	command "e"	PDO input/output/meta/event mapping clear
ULOI_PMAP _SAVE	46	0x2E	command "e"	PDO input/output/meta/event mapping save

The core part are ULOI_PICO and ULOI_POCO mapping tables, both with same format structure. They are accessible as regular uLOI arrays. Each array entry specifies mapping between CID and object dictionary entries. Simple one to one mappings are specified directly by entry by OID number. Complex mapping can specify offset into block of meta-data byte array instead of direct OID specification. This allows to serialize multiple objects/OIDs data under one CID, add execute command after CID data reception and distribution into uLDOI objects etc. Another possibility is to process same received data by multiple mappings for same CID. The special form to embed 3 bytes (OID + single byte) or 4 bytes (OID + 2 bytes) directly into ULOI_PICO or ULOI_POCO mapping table entry is also supported.

Field	Type	Description
CID	u2	connection identification
Flg	u2	flags specifying mapping type
Meta/OID	u2	byte offset into ULOI_PIOM data or direct OID specification
Meta len	u2	length of meta data or 2 for direct OID specification

Use of Meta/OID field is specified by options set in Flg field.

Symbol	Mask	POCO	PICO	Description
ULOI_CIDFLG_META_LOC	0x0F	y	y	Interpretation of Meta/OID field
ULOI_CIDFLG_META_OID	0x00	y	y	OID specified directly
ULOI_CIDFLG_META_OFFS	0x01	y	y	ULOI_PIOM offset specified
ULOI_CIDFLG_META_EM3B	0x03	y	y	Embedded 3 bytes in Meta and Meta len
ULOI_CIDFLG_META_EM4B	0x04	y	y	Embedded 4 bytes in Meta and Meta len
ULOI_CIDFLG_FIXED	0x10		y	Supply received data by meta
ULOI_CIDFLG_IMMED	0x20	y		Send directly meta-data, no processing
ULOI_CIDFLG_SKIP_LOCAL	0x40	y		Do not process locally
ULOI_CIDFLG_SKIP_SEND	0x80	y		Do not send data

The flags ULOI_CIDFLG_IMMED, ULOI_CIDFLG_SKIP_LOCAL and ULOI_CIDFLG_SKIP_SEND are experimental for actual implementation.

6.3 Events to Process Messages Mapping

The ULOI_PEV2C array specifies, which CID/CIDs identified transfers should be initiated when given event number is activated. One event can be specified multiple times to trigger multiple CID transfers. The ULOI_PEV2C array entry specifies event number to CID mapping and some flags to nail down CID processing.

Field	Type	Description
Evnum	u2	event number triggering transmission
CID	u2	CID to be send
DAdr	u2	destination uLan address, 0 broadcast
Flg	u2	flags influencing processing

Flag	Mask	Description
ULOI_PEV2CFLG_ARQ	0x01	request acknowledge by recipient
ULOI_PEV2CFLG_LOCAL	0x02	use queue for local only processing

References

[1] uLan protocol for RS-485 9-bit network, SourceForge.net project page `http://ulan.sourceforge.net/`

[2] Komunikacni protokol uLan, Pavel Pisa `http://cmp.felk.cvut.cz/~pisa/ulan/dipl_kom.html`

[3] Directory with older versions of uLan Driver for Linux, Pavel Pisa `http://cmp.felk.cvut.cz/~pisa/ulan/`

[4] uLan Driver for Linux, Pavel Pisa `http://cmp.felk.cvut.cz/~pisa/ulan/ul_drv-0.6.0.tar.gz`

[5] PiKRON Ltd., Laboratory Instruments Developers ,
`http://www.pikron.com/`

[6] Stanislav Hrbek, Diploma Thesis, 2009, Praha