ul_drv - uLan RS-485 Communication Driver

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

2nd December 2002

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.

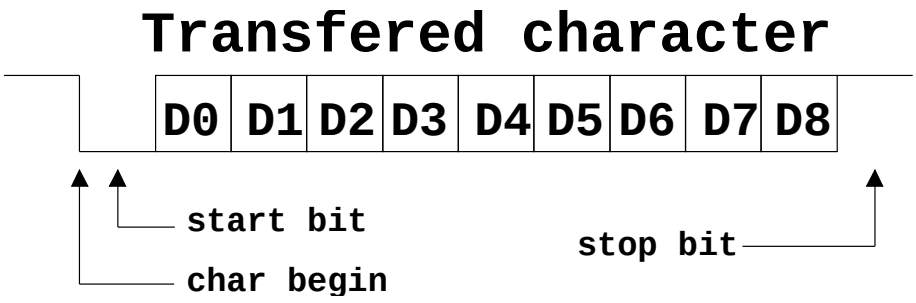

Figure 1: 9-bit Character Format

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.

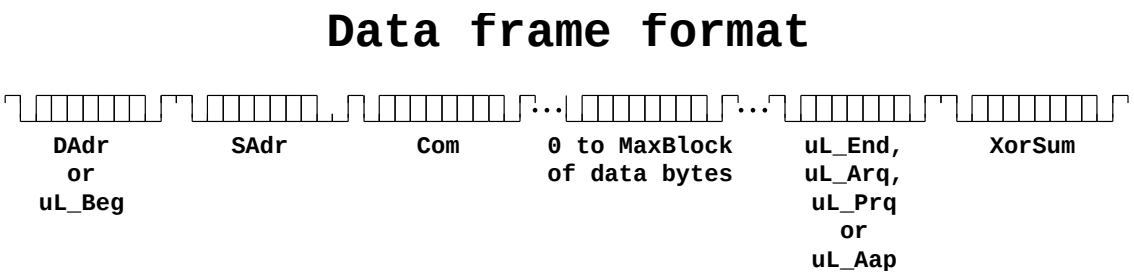

Figure 2: uLan Frame Format

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.

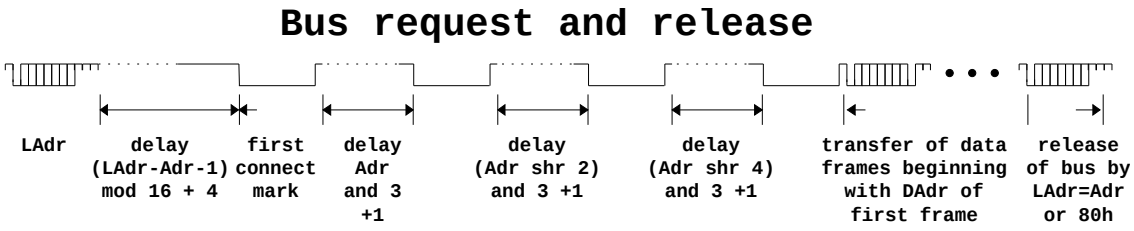

Figure 3: uLan Access Arbitration

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. 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
```

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 informations ("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

Instalation 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 vizard 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 throught Add Hardware vizard is

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

When **PnP COM2** port is used, brach 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 automata 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 automata subsystems, which can be used in more places in main automata loop. Main purpose of this automata 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 reentrant.

The automata 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 clients' 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_drv`") and some example utilities ("`ul_drv`").

ul_drv	directory containing sources of kernel driver (Linux, DOS, WinNT, Win2000/98)
k_compat.h	header file hiding differences between different Linux kernel versions, it is tested from 2.2.xx up-to 2.5.xx (it could work with kernels down-to 1.3 version)
ul_drv.h	declaration of bidirectional linked lists of messages, atomic memory allocations, message flags and driver state structure (header is almost target system independent)
ul_hdep.h	definition of hardware dependent functions and macros for all target systems and definition of kernel internal per client state structure
ul_drv.c	implementation of 16550 and 82510 chip drivers. includes source of Ox16950 chip driver generic message processing automata 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. this file is really long and should be divided into more parts, but there could be problems with make commands under WinNT
ul_lib	library implementing basic message processing (send, receive, filtering and notification) and some common commands for applications
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
utils	command line test programs independent on target systems
ul_buftst.c	test of hardware of line transceivers
ul_spy.c	uLan message monitor
ul_sendhex.c	intelhex and binay downloader and uploader

3.6 RS-485 Converter

In figure 4 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.

The converter without external power supply is not well suited for large networks with long

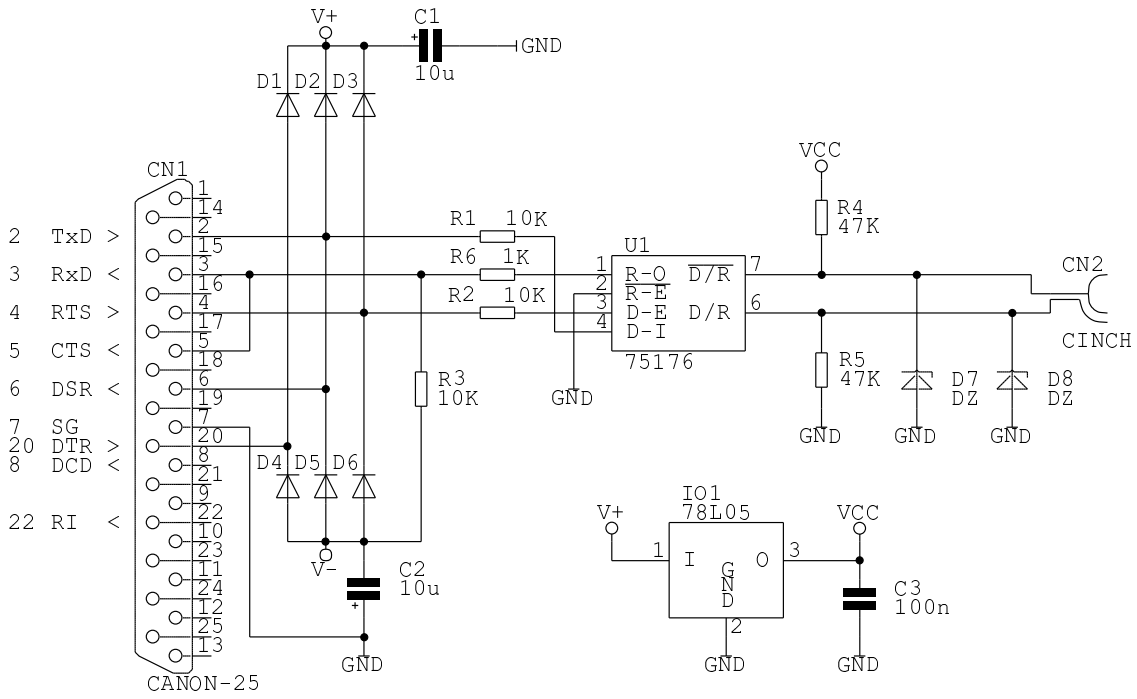

Figure 4: RS-485 Dongle Converter

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 proffesional 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 succesfull 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 imediate processing of frame by receiving station
UL_BFL_ARQ	Request imediate 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 `ioctl`s 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 beggining of first frame of message
UL_STROKE			Restart proccessing 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 fieldbus 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 results in sending of reply. Reply is send as regular queued message in **uLan** multimaster environment, no interrupt immediate processing is necessary. Many of commands uses 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.[4] 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 subcommands 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 is designed to enable flexible setting and querying of node/instrument variables/objects and for sending of commands. 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 nodes/instruments and provides way to automatically build node profile from the descriptions.

Name	OID	Parameters	Description
I_AOID	10	<string>	Substitues numeric OID by ASCII name for I_DOII and I_DOIO
I_DOII	12	<oid>	Query description for writeable/in <oid> returned I_DOII_R
I_DOIIr	13	<oid><oiddesc>	Description of writeable <oid>
I_DOIO	14	<oid>	Query description for readable/out <oid> returned I_DOIO_R
I_DOIOr	15	<oid><oiddesc>	Description of readable <oid>
I_QOII	16	<oid><maxret>	Query up-to <maxret> writeable/input OIDs list startingfrom <oid> or 0 for first
I_QOIIr	17	<oid0>... <oidn><0>	List of writeable OIDs ended by 0
I_QOIO	18	<oid><maxret>	Query up-to <maxret> readable/output OIDs list startingfrom <oid> or 0 for first
I_QOIOr	19	<oid0>... <oidn><0>	List of readable OIDs ended by 0

I_RDRQ	20	<oid0>... <oidn><0>	Query for values of OIDs
I_RDRQr	21	<oid0><...> ... <oidn><...><0>	Reply with acquired values of OIDs
I_SNCHK	29	<SN0 .. SN3>	Check module serial number
I_STATUS	30		Standard OID for 16 bit status
I_ERRCLR	31		Standard OID for clear error status

References

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

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

[3] uLan Driver for Linux, Pavel Pisa `http://cmp.felk.cvut.cz/~pisa/ulan/ul_drv-0.6.3.tar.gz`

[4] PiKRON Ltd., Laboratory Instrumets Developers , `http://www.pikron.com/`