Programmer's Guide

CoBox

Version 6500
05-17-2007

LANTRONIX®

Contents

Before starting… 7

General 8

Hardware Overview 10

Programming Environment 18

Demo Sample Programs 91

Ethernet Frame Handling 93

DSTni Chipset Loading 96

CoBOS Standard UDP Handlers 98

Before starting…

Before starting, it is recommended to read the *User's Manual* of the particular hardware platform you will use in development. Once you thoroughly understand how to use, configure, and load firmware onto the device, you can begin software development. The programmer should also become very familiar with the standard functionality of the device.

Hardware requirements

You need a PC with two serial ports and a connection to a local area network (LAN). In some cases, a single serial port will suffice. The best setup is to have a second PC for a separate connection to the second serial port of the device, as well as for independently capturing network packets. You will also need the correct RS-232 serial cables with the corresponding connectors DB-9 or DB-25 and network RJ-45 cable. The appropriate power supply will also be required.

Software requirements

On your PC, you must have MS-DOS or MS-Windows (NT, 2000, XP, etc) with access to the Command Prompt or DOS prompt. Your computer must also have a LAN connection with TCP/IP protocol. The best approach is to have static IP addresses on both your PC and your device server – ask your LAN administrator for details. All necessary software for developing a custom program for the CoBox is available with the CPK (**C**oBox **P**rogramming **K**it), *except* the necessary compiler.

You may also need some additional software programs. To analyze a serial port's data flow, you will need to use a terminal program. If you do not have another preference, you can use the standard Windows Hyperterminal. Configure the terminal program the same as your Cobox's serial port's default – 9600 Baud, 8 data bits, no parity bit, and 1 stop bit.

You must also have a network terminal program. For TCP connection, you can use the standard Windows Telnet. For UDP connection, Windows has no terminal client. Lantronix has provided an additional utility UDPCBXTest.exe as part of the CPK.

Next, to download your program into your CoBox, you will need any TFTP client. Use the tftp.exe of standard Windows NT or another client that you are comfortable using. For monitoring network activity, you may want some sort of network "sniffer". However this particular utility is **not** included in the CPK, you would have to provide one of your own means. (Various network analyzers can easily be found on the internet)

For compiler tool information, see the Programming Environment section.

General

CPK Introduction

The CoBox Programmers Kit is a specific set of libraries and utilities produced by developers at Lantronix. The main purpose of the CPK is for the development of Lantronix Device Servers. The underlying operating system of the CPK is CoBOS, or **CoB**ox **O**perating **S**ystem.

The CPK is an internal tool of Lantronix, and is **NOT** readily available. Your protection of this software is required.

Occasionally customer requirements drive the need for custom applications. For those specific cases, Lantronix has allowed customers to write their own application upon the Lantronix CoBox family. You must remember that your future programs can only use available C-functions in the CPK. This set of functions is constantly changing and improving by the developers of Lantronix. You should understand the sample DEMO-programs are written for training and teaching purposes and are only a basis for your future programs.

CoBox Family

The CoBox family of products is microprocessor-based platforms, designed to exchange data between a serial device and the network. However, this is only the tip of the iceberg. In reality, on the CoBox base, you can do almost anything you want, because it is based on a general-purpose microprocessor. You can write programs that transform CoBox into a mini WEB-server, or an intelligent controller of different digital devices, or a modem bridge/router for connecting remote networks, etc. Of course, applications need to consider the available hardware support of the chosen delivery platform.

CoBOS Introduction

CoBOS is the name given to this Lantronix proprietary operating system. CoBOS is a cooperative multi-tasking operating system. CoBox is designed with small memory footprint requirements in mind. Custom programs are linked together with CoBOS and supporting libraries to create one single program (OS included). The resultant image is loaded into flash memory.

Upon boot, the low-level boot loader is executed. The loader inspects the hardware, finds a loadable image, loads the image, and begins execution of the image. On some hardware, the image is executed from flash, while on platforms containing the Lantronix DSTni-LX or EX processor; the image is loaded into RAM and then executed.

Once the firmware starts executing, the IP stack is initialized, and other low level tasks are started. Lastly, the function newmain() is called. This is the start of all custom applications. The HTTP server and remote configuration tasks are typically started in newmain(). Also, the serial ports are normally initialized here as well.

CoBOS Tasks

The CoBox has a cooperative multitasking operating system. New tasks can be started with the spawn() function. The task control block (TCB) structure (see tcbdef.h) maintains process specific data, as well as a pointer to the next TCB. Control of the processor is released back to the kernel by calling the nice() function. Nice() will then continue execution of the following task in a circular list fashion. Tasks, or processes should be careful to release the CPU at least once every second, or the watchdog timer will reboot the CoBox. Also note, that as long as ‘this’ process is running, others are not. All blocking I/O functions imply a nice(). For example, waiting for a character to arrive via getch().

The actively running process is described in ActPro (Active Process), which is a pointer to the currently running TCB.

CoBOS Serial Channel Control

Each of the CoBox's serial channels is assigned a channel control block (CCB). The CCB structure (see ccbdef.h) maintains channel specific data and configuration for the assigned port, along with its associated buffers (or FIFOs). Configuration of the CCB is required for serial communications. Typical settings include Baud Rate, Parity, Databits, Stopbits, flow control, and interface mode. When a task is spawned, it can be associated with a particular serial channel. The InitLocalChan() function, then pushes the configuration in the CCB on to the CPU's UART for the associated channel. This associated stream is assigned to the ActPro->CCB_Ptr or the "active process's CCB pointer". Furthermore, the associated stream's FIFOs are also assigned to the ActPro->IO_Ptr or the active process's IO pointer. Please note, the newmain() process is associated with the first serial port (AllCCB[0]) at startup.

Steps to writing and checking CoBox programs

1. Write or edit your program by any interface/editor;
2. Make the program using Borland C tools and convert it to ROM-file by the e2i.exe utility (we usually use a batch file t.bat or m.bat included in CPK);
3. Download the ROM-file into CoBox by TFTP client. Use the table titled "Firmware Support of various Products and Password" for the correct destination.

   /* for example: tftp.exe –i <CoBox_IP> PUT <program_name>.rom <destination> */

4. If (necessary) set up new CoBox's parameters in SETUP by Telnet or Terminal;

   /* for example: telnet.exe <CoBox_IP> 9999 */

5. Start the necessary utilities for testing (Telnet, COM1/2 Terminals, UDP-client, etc.);
6. If (necessary) power off/on to reset CoBox, go to step 1.

Hardware Overview

Hardware

Chipsets

Model	Micro100	SDS-1101	SDS-2101	UDS-1100	UDS-2100	Xpress DR+ (W)
Code Image	m100.rom	sds1101.rom	sds2101.rom	uds1100.rom	uds2100.rom	drig.rom (dr_mrv.rom)
CPU	Lantronix DSTni-LX001 48MHz	Lantronix DSTni-EX 48-88MHz	Lantronix DSTni-EX 48-88MHz	Lantronix DSTni-EX 48-88MHz	Lantronix DSTni-EX 48-88MHz	Lantronix DSTni-EX 48-88MHz
Network Controller	CPU	CPU	CPU	CPU	CPU	CPU
Serial Controller	CPU	CPU	CPU	CPU	CPU	CPU
EEPROM	None	2 Kbytes	2 Kbytes	2 Kbytes	2 Kbytes	2 Kbytes
RAM	256 Kbytes	256 Kbytes	256 Kbytes	256 Kbytes	256 Kbytes	256 Kbytes
Flash PROM	512Kbytes serial flash	2048 Kbytes	2048 Kbytes	2048 Kbytes	2048 Kbytes	2048Kbytes

Model	Xport-01 (LX)	Xport-03 (EX)	WiPort NR	Matchport BG	WiPort (B/G) (opt2)	WiBox (B/G)
Code Image	xpt.rom	xptex.rom	fpt.rom	mpt_bg.rom	wpt_mrv.rom	wbx_mrv.rom
CPU	Lantronix DSTni-LX001 48MHz	Lantronix DSTni-EX 48-88MHz	Lantronix DSTni-EX 48-88MHz	Lantronix DSTni-EX 48-88MHz	Lantronix DSTni-EX 48-88MHz	Lantronix DSTni-EX 48-88MHz
Network Controller	CPU	CPU	CPU	CPU	CPU	CPU
Serial Controller	CPU	CPU	CPU	CPU	CPU	CPU
EEPROM	None	None	2 Kbytes	2 Kbytes	2 Kbytes	2 Kbytes
RAM	256 Kbytes	256 Kbytes	256 Kbytes	256 Kbytes (1.25MB)	256 Kbytes (1.25MB)	256 Kbytes
Flash PROM	512Kbytes serial flash	512Kbytes serial flash	2048Kbytes	2048Kbytes (4096Kbytes)	2048Kbytes (4096Kbytes)	2048Kbytes

CPU Register Usage

Register	Usage
AX	General purpose register use
BX	
CX	
DX	
SI	Source index
DI	Destination index
DS	Data segment
SS	Stack segment
CS	Code Segment
ES	Extra Segment
IP	Instruction pointer

Memory Maps

DSTni Based Products

Memory Block (24-bit address)	WiBox, WiPort BG, WiPort NR, Matchport BG, UDS1100, UDS2100, SDS2101, SDS2102, DR+, DR+W	WiPort opt 2	XPort-01 & 03, Micro-100
FFFFFF FF0000	Boot code (reserved 64KB) & EX Loader	Boot code (reserved 64KB) & EX Loader	Boot code (reserved 64KB)
FEFFFF FE0000	WEB19	WEB51	Not Used
FDFFFF FD0000	WEB18	WEB50	
FCFFFF FC0000	WEB17	WEB49	
FBFFFF FB0000	WEB16	WEB48	
FAFFFF FA0000	WEB15	WEB47	
F9FFFF F90000	WEB14	WEB46	
F8FFFF F80000	WEB13	WEB45	
F7FFFF F70000	WEB12	WEB44	
F6FFFF F60000	WEB11	WEB43	
F5FFFF F50000	WEB10	WEB42	
F4FFFF F40000	WEB9	WEB41	
F3FFFF F30000	WEB8	WEB40	
F2FFFF F20000	WEB7	WEB39	
F1FFFF F10000	WEB6	WEB38	
F0FFFF F00000	WEB5	WEB37	
EFFFFF EF0000	WEB4	WEB36	
EEFFFF EE0000	WEB3	WEB35	
EDFFFF ED0000	WEB2	WEB34	

ECFFFF EC0000	WEB1	WEB33	
EBFFFF EB0000	Firmware Image Bank 2 (Storage)	WEB32	
EAFFFF EA0000		WEB31	
E9FFFF E90000		WEB30	
E8FFFF E80000		WEB29	
E7FFFF E70000		WEB28	
E6FFFF E60000		WEB27	
E5FFFF E50000	Firmware Image Bank 1 (Storage)	WEB26	
E4FFFF E40000		WEB25	
E3FFFF E30000		WEB24	
E2FFFF E20000		WEB23	
E1FFFF E10000		WEB22	
E0FFFF E00000		WEB21	
DFFFFF DF0000	Mirrored 2MB (E00000 – FFFFFF)	WEB20	
DEFFFF DE0000		WEB19	
DDFFFF DD0000		WEB18	
DCFFFF DC0000		WEB17	
DBFFFF DB0000		WEB16	
DAFFFF DA0000		WEB15	
D9FFFF D90000		WEB14	
D8FFFF D80000		WEB13	
D7FFFF D70000		WEB12	
D6FFFF D60000		WEB11	
D5FFFF D50000		WEB10	

D4FFFF D40000		WEB9	
D3FFFF D30000		WEB8	
D2FFFF D20000		WEB7	
D1FFFF D10000		WEB6	
D0FFFF D00000		WEB5	
CFFFFF CF0000		WEB4	
CEFFFF CE0000		WEB3	
CDFFFF CD0000		WEB2	
CCFFFF CC0000		WEB1	
CBFFFF CB0000		Firmware Image Bank 2 (Storage)	
CAFFFF CA0000			
C9FFFF C90000			
C8FFFF C80000			
C7FFFF C70000			
C6FFFF C60000			
C5FFFF C50000		Firmware Image Bank 1 (Storage)	
C4FFFF C40000			
C3FFFF C30000			
C2FFFF C20000			
C1FFFF C10000			
C0FFFF C00000			
BFFFFF B00000	Mirrored 2MB (E00000 – FFFFFF)	Mirrored 4MB (C00000 – FFFFFF)	
AFFFFF A00000			
9FFFFF 900000	Mirrored 2MB (E00000 – FFFFFF)		

8FFFFF 800000			
7FFFFF 500000	Not Used	Not Used	
4FFFFF 400000		External RAM 1MB	
3FFFFF 040000		Not Used	
03FFFF 030000	RAM3 (Firmware Image, Executing)	RAM3 (Firmware Image, Executing)	RAM3 (Firmware Image, Executing)
02FFFF 020000	RAM2	RAM2	RAM2
01FFFF 010000	RAM1 (Network Buffers)	RAM1 (Network Buffers)	RAM1 (Network Buffers)
00FFFF 000000	RAM0	RAM0	RAM0

RAM3

CoBOS application programs execute from RAM3.

RAM2

Application programs can use the RAM2 (64 Kbytes of RAM) if the image is NOT executing from RAM2. This area must be accessed by far pointers.

RAM1 (64 Kbytes)

Address range	Description
FFFF E800	Ethernet Receive Chain
E7FF E000	2KB Free space
DFFF D000	TCP Buffer (1st TCPAlloc for tcp14.lib), free space with tcp12 and tcpip.lib
CFFF C000	TCP Buffer (2nd TCPAlloc for tcp14.lib), free space with tcp12 and tcpip.lib
BFFF B000	TCP Buffer (1st TCPAlloc for tcp12.lib, 3rd for tcp14), free space with tcpip.lib
AFFF 8000	TCP Buffer (2nd, 3rd, and 4th TCPAlloc for tcp12.lib, 4th, 5th and 6th for tcp14), free space with tcpip.lib
7FFF 7000	TCP Buffer (1st TCPAlloc for tcpip.lib)
	TCP Buffer (Nth TCPAlloc)
0FFF 0000	TCP Buffer (8th, 12th, or 14th TCPAlloc based on lib used)

RAM0 (lower 64 Kbytes)

Address range	Description
FFFF FC00	1KB Main Task Stack
FBFF F000	Ethernet Transmit Chain
EFFF C800	Common Ethernet receive buffers (~10KB)
C7FF 0100	Initialised & uninitialized data
00FF 0000	Interrupt vectors

XPort & Micro-100 Serial Flash Page Map

The XPort 512KB serial flash is divided into 264 byte pages.

Page	Byte offset range	Description
557	540591 147048	WEB1-6
308	147047 81312	Firmware Image (Storage) – 2nd 64KB
261	81311 68904	Backup Firmware – 12KB
257	68903 67848	Backup Configuration Data - Reserved
7	68903 1848	Firmware Image (Storage) – 1st 64KB
5	1847 1320	Firmware header and 2nd stage loader
1	1319 264	Hardware and configuration settings
0	263 0	Reserved – EX MAC address

TFTP Firmware area

In V6 and above, the firmware upload procedure now writes directly to flash.

WARNING:
If the variable *fw_stat* is not equal to 0 the firmware and web page upload procedure is running and attempting to write flash memory.

Example:

```
if (fw_stat) {
/* Stop accessing flash memory */
}
```

Programming Environment

Operating System

MS-DOS, Microsoft Windows (95, 98, NT, 2000).

Directory Structure

To avoid problems with older MS-DOS software, do not use more than eight characters for filenames. This is an example directory tree for CoBox development. Please note, starting with V6, each product now has it's own working sub-directory. This subdirectory scheme removes the need for a specific suffix being added to the created object files.

```
C:\Source

    └── CoBox

          ├── cpk430                 (Projects using kernel 4.3)
          ├── cpk450                 (Projects using kernel 4.5)
          ├── cpk500                 (Projects using kernel 5.0)
          ├── cpk520                 (Projects using kernel 5.2)
          ├── cpk550                 (Projects using kernel 5.5)
          ├── cpk551                 (Projects using kernel 5.51)
          └── cpk580                 (Projects using kernel 5.8)
          └── cpk6101                (Projects using kernel 6.1.0.1)
          └── cpk6500                (Projects using kernel 6.5.0.0)
                ├── Bin              (r2h, web2cob, utilities, etc.)
                ├── Doc              (Documentation files)
                ├── Inc              (Include files)
                ├── Lib              (Libraries)
                ├── SNMP             (SNMP include files)
                ├── Stdf             (Standard setup)
                ├── TCP_UDP_terminal (Encryption tools)
                ├── UDP_terminal     (UDP tester)
                └── Demo[x]          (Demo projects)
                      ├── xpt(s)     (XPort-01 product (small build))
                      ├── xptex(s)   (XPort-03 product (small build))
                      ├── wbx_mrv    (WiBox product)
                      └── wpt_mrv    (WiPortproduc)
```

Environment Variables

You should add the **bin** directory to your search path. Change the environment settings on Windows NT/2000 or add this line at the end of C:\autoexec.bat:

```
PATH=C:\Source\CoBox\cpk6500\bin;%PATH%
```

Compiling

Compiler

- Borland Turbo C Version 5.2. Warning: new Borland compilers do not support 16-bit CPUs.
- Small memory model
  (64 Kbytes code, 64 Kbytes data, max. file size 128 Kbytes). E2I will inform you if the image is too large.

Libraries

Library name	Contents
crstub	Stub without encryption
crypt[2]	Encrypting and decrypting
Drig, drw	Xpress DR+ specific functions, wireless
fpt	WiPort NR specific functions
kern100	Kernel functions
kernMAC	Kernel functions for XPort
m100	Micro 100 specific functions
Romlib	Memory and String functions
Parfl	Parallel Flash functions
Mrv[8385]	WiPort Marvell radio interface
serfl[ex]	Serial Flash functions [EX based]
snmp[m]	SNMP functions [XPort]
std	Standard setup functions
stubs	Stub functions
supp[WPA]	Radio encryption support
tcpip, tcp12, tcp14	TCP/IP functions (handling 8, 12 or 14 connections simultaneously)
uds21	UDS-2100 & SDS-2101 specific functions
vds100	UDS-1100 & SDS-1101 specific functions
web[m]	HTTP and web server functions [XPort]
web_fs[m]	Web file system functions [XPort]
WiBox	WiBox specific functions
WiPort	WiPort specific functions
XPort	XPort specific functions
XPortEX	EX-based Xport specific functions

Makefile

The make process consists of two Makefiles and product subdirectories. Each demo directory contains it's own Makefiles. Please review them in detail. **You MUST define BCBIN in the Makefile.**

Version File (VERSION)

The Version file contains the software version number as a four digits on the first line (e.g. 6.5.0.1 stands for version 6.5.0.1). Each demo project contains a Version file, which can be customized, as desired, for each build.

6.5.3.4

Linker File (.LK)

Each demo directory contains product type subdirectories. Each subdirectory contains the link directives file for that product. You will need to modify this file if you add additional .obj files to the build process.

Batch Files (.BAT)

Three sample batch files, which can be used for any of the demo projects, are provided in the bin directory as reference.

- t.bat   Make whole project (make all).
- m.bat   Make whole project and wait for key after each page (make | more).
- s.bat   Cleanup all generated files (make clean).

Make Commands

Note: The optional "s" character will force the make process to attempt to build a 64KB module (will not use two TEXT segments).

- make all   Make whole project (make all).
- make dr_mrv.rom   Make Xpress DR+ wireless image only.
- make drig[s].rom   Make Xpress DR+ image only.
- make fpt[s].rom   Make WiPort-NR image only.
- make m100[s].rom   Make Micro-100 image only.
- make mpt_bg.rom   Make Matchport BG image only.
- make sds1101[s].rom   Make SDS-1101 image only.
- make sds2101[s].rom   Make SDS-2101 image only.
- make xpt[s].rom   Make XPort-01 image only.
- make xptex[s].rom   Make XPort-03, and XPort-485 image only.
- make wpt_mrv.rom   Make WiPort (B/G radio) image only.
- make wbx_mrv.rom   Make WiBox (B/G radio) image only.
- make u2100[s].rom   Make UDS-2100 image only.
- make u1100[s].rom   Make UDS-1100 image only.
- make clean   Cleanup all generated files

Firmware Support of various Products and Password

There are various ROM images available from Lantronix. Some images support multiple platforms while others are very specific. Please choose the right firmware file (.rom) according to the following table (optionally an ‘s’ may be appended for a 64KB ROM image):

Product	ROM file	Destination
Micro-100	M100.ROM	4M
SDS-1101	SDS1101.ROM	D3
SDS-2101	SDS2101.ROM	D4
UDS-1100	UDS1100.ROM	U3
UDS-2100	UDS2100.ROM	U4
Xpress DR+	DRIG.ROM	R1
Xpress DR+W	DR_MRV.ROM	R2
XPort-01 (LX)	XPT.ROM	X4
XPort-03 (EX)	XPTEX.ROM	X5
WiPort-NR	FPT.ROM	FX
WiPort(B/G)	WPT_MRV.ROM	W6
WiBox(B/G)	WBX_MRV.ROM	W7
Matchport BG	MPT_BG.ROM	W8

See e2i documentation in chapter Utilities for destination details.

Restrictions

- Do not use a lot of stack, stack memory is limited! Bigger buffers should be defined as global variables.
  Stack memory for the main() task is FFC0h…FFFFh = 1024 bytes.
- Don’t place static structures onto the stack. You should define them as global variables.
- Dynamic memory allocation like malloc() is not supported.
- Memory usage is limited to C800h bytes. Add vectors + _data + cdata + const + _bss + extdata + stack, which is the actually used memory. The result has to be lower than C800h. Information can be read from of the screen output or the map file (example below):

```
 Start  Stop   Length Name               Class

 00000H 0A99AH 0A99BH _TEXT              CODE
 0A9A0H 0AA1FH 00080H VECTORS            DATA
 0AA20H 0B13BH 0071CH _DATA              DATA
 0B13CH 0B13CH 00000H CDATA              DATA
 0B13CH 0B13CH 00000H CONST              CONST
 0B13CH 10179H 0503EH _BSS               BSS
10180H 1018FH 00010H EXTDATA
10190H 1019FH 00010H STACK              STACK
```

  Compiler and linker do **not** detect an overflow.

- Operations using 32 bits are not supported. e.g. the command

```
var >> 16
```

is OK but

```
var >> 17
```

cannot be used.

- Floating point arithmetic is not supported.
- Functions assigned to pointers **MUST** be declared globally, NOT static.

Programming

Multitasking

The CoBox' round robin multitasking is controlled with four interrupts in the following priorities:

1. Serial interfaces
2. Timer
3. Network interface
4. Standard

Priority 1 is the highest priority. That means that e.g. the network event can interrupt the standard event.

CoBox' multitasking occurs only when you call the **nice ()** function. Remember to insert this function in any longer loop, otherwise the watchdog will reset the CoBox after approximately 1 second. **nice ()** is also called from some internal functions (typically IO functions like getch()).

Watchdog

The watchdog is a hardware timer that resets the CoBox if it is not triggered regularly. The timeout varies from 700 to 1300 mS depending on CPU clock speed.

How to Send a Ping

The below sample will send 10 ping requests to 65.33.232.134, with a 30mS timeout.

```
#include <memory.h>
BYTE p[4] = {65,33,232,134};

int ping(BYTE *ping_ip, WORD cnt, WORD to);
extern WORD icetim, iceseq;

demo()
{
      ping(p, 10, 30);
}

int ping(BYTE *ping_ip, WORD cnt, WORD to)
{
      AD_T a;
      int i, j;

      memset(&a, 0, sizeof(AD_T));
      memcpy(&a.ipa, ping_ip, 4);
      iceseq = 0;
      for (i = 0; i < cnt; i++) {
           if (icmp_out(&a, 8+12, 0x0008, i)) {
                return(0);
           }
           j = (WORD) ticks;
           while (((WORD) ticks - j) < to) {
                nice();
                if (iceseq) {
                    printf("Seq %3u time %ums\r\n", iceseq, (WORD) ticks - icetim);
                    iceseq = 0;
                }
           }
      }
      return ((WORD) ticks - icetim); /* return last ping time */
}
```

NOTE: icmp_out can only effectively send a ping, other requests will have a 0'd ICMP payload.

```
int icmp_out (AD_T *a, WORD len, WORD code, WORD seq)
    a - pointer to AD_T address structure with IP address filled in, other fields 0
    len - must be 8 plus the ping data length (typically 12) == 8+12 or 20
    code - ICMP type field - 0x08 for Echo Request
    seq - ICMP sequence number
```

TCP Connections

Example:

```
TCP_t *t;
BYTE InBun[128 + 8];   /* include 8 additional bytes for the FIFO control block */
BYTE OutBuf[128 + 8]; /* include 8 additional bytes for the FIFO control block */
t                  = TCPAlloc();  /* Allocate TCP Structure */
t->r.StCall        = ChanS2NoTel; /* ChanS2() selects telnet automatically */
t->r.RcvCall       = ChanRcv;
t->RcvFifo         = FifoInit(InBuf, 128); /* FIFO size, w/o the control block */
t->XmitFifo        = FifoInit(OutBuf, 128);/* FIFO size, w/o the control block */

TCPOpen(0, t, 10001);      /* Passive open to port 10001 */

while(t.State == ESTABLISHED) {
/* Send and receive data */
}

T_Discon(t);       /* Close connection */
```

Connection States

The State variable in the TCP structure indicates the current connection state (see tcp.h for states).

Example:

```
if(t->State == ESTABLISHED) {
/* Connection established */
}
if(t->State == LISTEN) {
/* Passive connection is waiting for connect from foreign host */
}
```

How to open, close and re-open sockets

In a typical environment CoBOS has 8 handles available (tcpip.lib) for Socket connections (unless using the 12 or 14 network connection library, tcp12 or tcp14). Depending on your system those could be utilized for:

- Web Server
- Telnet connection
- for user application

If your application is required to ‘open, close and re-open’ connections, you have to make sure that you are not using a new connection each time. If you do, you will find that the CoBox is re-booting once you try to open up the 9th (13th or 15th) connection.

The proper way would be to re-use the handle after T_Discon(xxx) finishes. The handle is still valid.

Example:

```
/* Initialize the TCP structure */
t            = TCPAlloc();
t->r.StCall  = ChanS2NoTel;
t->r.RcvCall = ChanRcv;
t->RcvFifo   = FifoInit(inbuf, 512);
t->XmitFifo  = FifoInit(outbuf, 512);

/* Set the IP Address */
t->a.Ip[0]   = pBIPAddresstoSendto[0];
t->a.Ip[1]   = pBIPAddresstoSendto[1];
t->a.Ip[2]   = pBIPAddresstoSendto[2];
t->a.Ip[3]   = pBIPAddresstoSendto[3];
t->a.Port    = 18245;

for(;;)
{
/* Open an active TCP connection on an available port */
WConnectionOpen = TCPOpen(1, t, 0);
/* Send Data */
    /* ... */

/* Disconnect the current network connection*/
T_Discon(t);
}
```

The network stack will send the TCP data in an ordinarily fashion. In some cases, the programmer may wish to request that the packet be sent now. This may be accomplished by ORing the tcp structures s member sflg with 2 (t->s.sflg |= 2).

UDP Data Transfer

Send block as UDP packet:

```
main() {
                   ...
udp_send(target, 1234, 1234, sendbuf, strlen(sendbuf));
    /*       |       |     |     |        |                */
    /*       |       |     |     |        Length           */
    /*       |       |     |     Buffer                    */
    /*       |       |     To port                         */
    /*       |       From port                             */
    /*       Destination IP address, 0 for broadcast       */
                   ...
}
```

Receive UDP packets on port 1234:

```
BYTE       bbuf[300];
Int        buflen;

main() {
           ...
udp_register(1234, rcvr);
           ...
}

/* rcvr: Demo UDP receive subroutine, called by kernel */
/* Parameters:                                         */
/*     buf     UDP content received                    */
/*     len     Length of UDP content                   */
/*     bflg    True if block came in by broadcast      */
/*     xip     Source IP address (pointer)             */
/*     a       Source address structure (pointer)      */
/*     from    From port                               */
/*     to      Destination port                        */
void rcvr(buf, len, bflg, xip, a, from, to)
BYTE   *buf, *xip;
AD_T   *a;
Int    len, bflg;
WORD   from, to;
{
    if(len && (len < 300)) {
        memcpy(bbuf, buf, len);
        buflen = len;
    }
}
```

Queues / FIFOs

Function

Serial and TCP queues are handled as FIFOs. The FIFO size must be a power of 2 (i. e. 256, 512, 1024,…) **PLUS** 8 bytes. Initialization of the FIFO is done by using the **FifoInit()** function and the power of 2 size (see TCP example above). The IOCall function is available for accessing the FIFO.

FIFO Structure

FIFO Control Block				**Data Block**
Input pointer IP	Output pointer OP	Mask (size – 1)	Base (Pointer to 1st byte of data block)	FIFO Data
WORD	WORD	WORD	WORD	0…size-1

If IP is equal to OP the FIFO is empty.

If IP is equal to OP – 1 the FIFO is full.

OP – IP = Number of characters stored in FIFO.

Incoming data is stored to IP’s address. Then IP is increased and points to the next available cell.

The byte below the OP cannot be used. In this case IP would be equal to OP and this would indicate an empty FIFO. A 128 byte FIFO can only store 127 bytes!

Timer

The internal CoBox timer is represented in global variables time and ticks. 1000 ticks is equal to 1 second. The timer resolution for V6 and above is **1 mS** for all platforms. That means the timer value (ticks) will be updated every 1 millisecond. This could change in the future.

Example:

```
#define wtime      ((WORD) time)
#define sticks     ((WORD) ticks)

demo() {
    DWORD ctime;   /* Current time in ms */
    DWORD stime;   /* Current time in s */

ctime = sticks;
stime = wtime;

/* Disable interrupts to prevent timer task from changing contents between the
reading of the two variables */
disable();
ctime = ticks;
stime = time;
enable();
}
```

LED Control

The LED states are controlled by corresponding global variables of type WORD:

LED	Standard function	Variable
Green	Channel 1	BlinkGWord[0]
Yellow	Channel 2	BlinkGWord[1]
Red	Error	BlinkRWord

The bits of the variables are used to control the LED's with a clock rate of 0.25 seconds. So the pattern 0xCCCC, which is binary

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0

will result in a 50% duty cycle LED blinking with a period time of one second.

After executing bit 0 the pattern starts again with bit 15 and so the whole sequence will circle within a 4 second period.

Any changed variable contents will be activated with the start of a new period. The function **BlinkReset()** will start the period immediately.

Example:

```
BlinkRWord    = 0x0000; /* Red LED: OFF */
BlinkGWord[0] = 0xFFFF; /* Green LED: always ON */
BlinkGWord[1] = 0x0005; /* Yellow LED: 2 flashes every 4 s */
BlinkReset();           /* Set values immediately */
```

Tasks

Each task has a Task Control Block (TCB). New tasks may be started with the spawn() function.

```
BYTE my_tcb_stack[256];
int my_task(void);

spawn(my_task, my_tcb_stack, sizeof(my_tcb_stack), 0, “my_task”);
```

Streams

Each task has an associated stream. This stream may be the serial port, a TCP FIFO, or not associated. IO functions (like printf, getch, etc) operate on the associated stream.

HTTP Server Control

The http server is started via:

spawn(WebProcess, kalloc(WEBSSIZE), WEBSSIZE, 0, “Web1”);

Multiple instances of the http server may be started.

When an http GET request occurs, the memory areas are checked for the file name starting in WEB0, then proceeding to WEB1, Web2, etc. The first matching file will then be sent back.

CPK applications may register a callback routine for a specific request by calling WebMethRegister().

Since the callback routine is called by the WebProcess(), the callback’s associated IO streams are the TCP connection FIFOs. This makes it simple to respond to the request by using printf(), or IOCall().

Example:

```
int my_callback(WCT *w, char *path, char *hdr);
WebMethRegister((“POST”, my_callback, “test.cgi”);
…
int my_callback(WCT *w, char *path, char *hdr)
{
--- Do processing of request ---
--- Read from the FIFO to get POST form tags ---
printf(“HTTP/1.0 200 Document follows\r\n”);
printf(“Content-type: text/html\r\n”);
printf(“\r\n”);
printf(“<html><body>”);
printf—other html information—
printf(“</body></html>”);
```

Hardware Detection

Processor Type

The variable *HW.cpu* contains the currently used processor type:

HW.cpu	Processor
0	NEC V.40
1	AMD 188ES
2	DSTni-LX-001 (186 core) & DSTni-EX

CPU Clock

The variable *HW.cpuclk* contains the currently used processor clock frequency in Mhz.

HW.cpuclk	Clock Frequency
10	10 Mhz
20	20 Mhz
25	25 Mhz
48	48 Mhz (DSTni-LX)
88	88 Mhz (DSTni-EX Hi-Performancemode)

EEPROM Type

The variable *HW.eeprom* contains the currently used EEPROM type:

HW.eeprom	EEPROM
0	93C46
1	24LC02, 24LC04, 24LC16, AT45DB041B (XPort)

Debugging

There are several methods to help you debug your CPK application.

Serial Port

If you need some debugging information you can use the 2nd serial port (if available). Just open a new channel in main.c:

```
#if DEBUG
int    Chan2Stack[200];
extern WORD dioptr, sioptr;

Chan2() {
    extern WORD dioptr;
    ActCCB->V24_speed = 0x02; /* 9600 bps */
    ActCCB->V24_mode  = 0x4c; /* RS232, 8N1 */
    InitLocalChan();
    dioptr = ActPro->IO_Ptr;      /* This stores an I/O pointer;
                                     necessary for access and
                                     reference by the main process */

    while(1) {
        putstr(“\n\rCoBox demo template - DEBUG Port\n\r”);
        Monitor();        /* Start ProMon for debug channel */
    }
}
#endif
…
newmain()
{
…
#if DEBUG
    spawn(Chan2, Chan2Stack, sizeof(Chan2Stack), 1, “C2”);
    sdelay(500); /* Wait for proper init of channel 2 */
#endif
…
}
```

To make it simpler you can define two functions:

```
void startdebug(void) {
    ActPro->IO_Ptr = dioptr;
}

void stopdebug(void) {
    ActPro->IO_Ptr = sioptr;
}
```

For output something to the debug port you simply switch the standard output:

```
#if DEBUG
startdebug();
putstr(“\n\rInit UDP receiver...”);
stopdebug();
#endif
```

Syslog

If there is a syslog server in your network you can use the **syslog()** function for sending debug information from the CoBox to the server.

Example:

```
memset(smtp_s.logmsg, 0, 513); /* Clear old message */
strcpy(smtp_s.logmsg, “Debug information”);
syslog((WORD)(LOG_NOTICE + LOG_LOCAL7), smtp_s.logmsg);
```

UDP

Simply call the **udp_send()** function to send debugging information. Use either a network sniffer for receiving and displaying the packets or send it to another UDP receiver program. (If your only using a sniffer, it might make sense to broadcast the packet.)

ProMon

ProMon 3.0 can be started with the **Monitor()** function and allows some simple debugging. See the “Debug Functions” section.

Telnet

Open a telnet session to the debug port 9998.

Example:

```
int telnetdebugStack[200];
static BYTE inbuf[128 + 8], outbuf[128 + 8];
WORD telnetptr;
TelnetDebug() { register TCP_t *t; extern int ChanS2(), ChanRcv();
    t              = TCPAlloc();
    t->r.StCall    = ChanS2;
    t->r.RcvCall   = ChanRcv;
    t->RcvFifo     = FifoInit(inbuf, 128);
    t->XmitFifo    = FifoInit(outbuf, 128);
    ActPro->IO_Ptr = &(t->RcvFifo);

    TCPOpen(0, t, 9998); /* Open passive connection to port 9998 */
    while(1) {
        sdelay(50);
        t->State = LISTEN;

        while(t->State != ESTABLISHED) nice();
        Monitor();     /* Start ProMon, disconnect with Quit */
        T_Discon(t);
    }
}

newmain() {
    ...
    spawn(TelnetDebug, telnetdebugStack, sizeof(telnetdebugStack), -1,
« TelnetDeb »);
    sdelay(500);
    ...
}
```

How to upgrade a project from 4.3 to 4.5

Changes in the programming environment

One major change is an upgrade from Borland C 2.0 to 3.1. This will result in a lot of error messages when compiling an existing 4.3 program.

This is due to the fact, that this compiler recognizes more possibly wrong syntax. Additionally this requires prototypes for each function. The appropriate prototypes can be found in the include files in the inc directory. Below are some items , which have to be added/changed at a minimum.

main.c

#include kernel.h

#include io.h

setpar.c

#define SETUPVAR Setup

#include io.h

int SetParStart(int d);

How to upgrade a project from 4.5 to 5.0

Changes in the programming environment

CPK5 includes support for a new processor family used by Lantronix, the DSTni-LX. This version of the CPK can be used to build software for all the CoBox and UDS family of products. As such, substantial changes were made to the main.c, Makefile, and linker files. Please review these files for your current project requirements.

main.c

Add:

```
/* network driver declaration */
#ifdef N0
extern int N0_DRV;
#endif
#ifdef N1
extern int N1_DRV;
#endif
#ifdef N2
extern int N2_DRV;
#endif
#ifdef N3
extern int N3_DRV;
#endif
#ifdef ND
extern int ND_DRV;
#endif
```

Add to newmain()

```
/* network driver initialization */
#pragma warn -eff
#ifdef N0
        N0_DRV;
#endif
#ifdef N1
        N1_DRV;
#endif
#ifdef N2
        N2_DRV;
#endif
#ifdef N3
```

```
        N3_DRV;
#endif
#ifdef ND
        ND_DRV;
#endif
#pragma warn +eff
```

Support Removed

Support for putint(), puthex(), and delay() has been removed in this release. Please use printf() and sdelay() as substitute routines.

Include Ordering

Include io.h before ip.h.

Include ip.h before tcp.h or udp.h

How to upgrade a project from 5.0 to 5.2

Changes in the programming environment

CPK520 includes support for a new Lantronix product family, the XPort. This version of the CPK can be used to build software for all the CoBox, UDS and XPort family of products. As such, several changes were made to the main.c, setpar.c, tools.c, Makefile, and linker files. Please review these files for your current project requirements. If your target platform is XPort, several additions have been made for that support.

main.c

Changes to VersionInit().

setpar.c

Changes in setpar.c:

new *baudratestrings[]
230400 is now valid for DSTni-LX platforms
0 disables the the serial port

XPort does not support RS-485 modes

Function parameters within setpar.c routines may have changed.

#define COBOX added to support long DHCP names

Support Removed from tools.c

Support for longdiv(), lmod(), lmul(), atoi(), a2toi(), a2toh() has been moved into the kernel.

Support for spri(), putCRLF() has been removed, in favor of printf().

Include file changes

Include 'kernel' directory has been removed. Files moved into inc.

Include the following files for XPort builds

```
#include "..\XPort\bitsXP.h"
#include "..\serfl\ECtypes.h"
#include "..\serfl\serflash.h"
```

```
#include <digio.h>
```

Include the following files for DinRail builds

```
# include <digio.h>
```

New include file bldFlags.h has been added to all the demos. This contains product specific compile options and should be included in all application source files.

How to upgrade a project from 5.2 to 5.5

Changes in the programming environment

CPK550 includes support for a two new Lantronix family products, the Micro-100 and the UDS-200. This version of the CPK can be used to build software for all the CoBox, UDS and XPort family of products. As such, changes were made to the bldflags.h and Makefile, along with new linker files for the two new products. Please review these files for your current project requirements.

XPort timer change

XPort and all other DSTni-LX based products now utilize a 1mS timer, while AMD and NEC based products utilize a 5mS timer.

Protocol changes to port 0x77F0

Port 0x77f0 now supports UDP, and the underlying protocol has changed. See the GPIO Control Interface document for this new protocol definition.

XPort & Serfl include directories moved

The Xport & serfl directories are now located below the inc directory.

How to upgrade a project from 5.5 to 5.51

Changes in the programming environment

CPK551 includes support for a three new Lantronix family products, the SDS-1100, SDS-2100 and the XPort EX. This version of the CPK can be used to build software for all the CoBox, UDS and XPort family of products. As such, changes were made to the bldflags.h and Makefile, along with new linker files for the three new products. In addition, four new libraries have been added. Please review these files for your current project requirements.

Main.c change

Due to kernel reorganization, you MUST add one additional declaration. Changes were made to the demo main.c files to reflect this change. Please add the following line in your project:

```
BYTE hls[75];
```

Setpar.c change

The XPort EX has the ability to run at serial speeds of up to 920Kbps. However, setting this baud rate also requires changing the CPU clock speed. These changes are made in Setup record 3. Setup record 3, also contains other kernel specific values. Avoid using record 3 in your project.

XPort & Serfl include directories moved

The Xport & serfl directories are now located below the inc directory.

How to upgrade a project from 5.51 to 5.8

Changes in the programming environment

CPK580 includes support for a two new Lantronix family products, the WiPort and the WiBox. This version of the CPK can be used to build software for all the CoBox, UDS and XPort family of products. As such, changes were made to the bldflags.h and Makefile, along with new linker files for the two new products. In addition, new libraries have been added. Please review these files for your current project requirements.

Main.c change

Additional include files were added to main.c to support the two new wireless products. The byte-order of *firmwarecheck* was changed.

Setpar.c change

Additional include files were added to main.c to support the two new wireless products, along with RS-485 support in the XPort. The setup menu now includes these new options.

WiFi.c added

WiFi.c contains the standard setup configuration dialog for the wireless interface. It is located in the stdf directory.

Version file change

The version file format changed, and as such, you **must** use the new e2i.exe provided.

Other changes

Added documentation as to “How to send a ping”.

Added documentation to CoBOS Ethernet frame handling.

Fixed a bug in putchar() of tools.c.

Micro-100 now uses virtual IO (pios). Added support in VersionInit via bldflags.

Added SetServicePort() to set port number for a specific service (HTTP, SMTP).

Changed Makefiles to include relative path to E2I.EXE.

How to upgrade a project from 5.8 to 6.1

Changes in the programming environment

CPK6101 includes **support** for **only** the Lantronix DSTni based family of products. This version of the CPK can **not** be used to build software for the older CoBox, Micro, Mini, UDS-10 or any other NEC, AMD or InnvoASIC CPU based product. A **major** change in V6 is the ability to support ROM images that are **greater than 64KB** in size. As such, changes were made to the flash file systems, bldflags.h, Makefile, and the directory structure; along with new linker files for the products. In addition, the libraries have been changed. Please review these files for your current project requirements. In order to build the larger than 64KB ROM images, you **must** have TASM.EXE.

There is **no direct** firmware conversion from V5.8 to V6.1 for some products. You **must** load an intermediate ROM image that understands the new flash file system layout. After the “upgrade” process, you may simply use the normal tftp process to reflash the firmware. **Be aware** that the new flash process writes directly to flash and will take longer (possibly 20 seconds or more) than the old process which cached the data first.

The V6 build process will attempt to relocate pieces of code into RAM2 and RAM3. In order to make this happen, the linker will need to make multiple passes. The make process will find the far links and create the required "proxy" code to access those far modules.

The product "small" builds (ie: xptexs.rom) will attempt to keep the single ROM image as 64KB. The normal build (xptex.rom), will attempt to locate **your** application in RAM2 while the rest of the code resides in RAM3. You can change the location of certain code pieces by manipulating the Makefile.

Main.c change

Changes were made to support additional security settings.

GLOBAL BYTE ethmode is added to support Ethernet Mode (duplex, and speed) selection.

MTU size is now defined in VersionInit().

Setpar.c change

Changes were made to factory_defaults(), now called default_setup().

Record 1 will now be reset.

Additional setup menu options were added to support the security settings. Please review these changes.

Demo.c change

Pointers to functions **MUST** be declared in global space (**NOT** statically defined).

Changes were made to demo.c to assit in automatic testing.

Other changes

WPA is now supported.

The radio firmware of the WiPort is **no** longer a separate file. The radio firmware is now integrated into the WiPort ROM image.

WPT.ROM and WBX.ROM no longer exist. These have been replaced by WPT_AGR.ROM and WBX_AGR.ROM. New additions WPT_MRV.ROM and WBX_MRV.ROM are added to support the new WiPort's B/G radio.

Addition support was added for new products (UDS-1100, WiPort G, and DRIG).

How to upgrade a project from 6.1 to 6.5

Changes in the programming environment

CPK6500 includes support for same devices as in CPK6101 except Agere radio based products (WBX_AGR, & WPT_AGR) have been dropped. New support has been added for the UDS2100, SDS1101, SDS2101, Matchport BG and the Xpress DR+ wireless. It is no longer possible to build a 64KB ROM image for the wireless devices. All wireless products now use a version of crypt, crypt2 and suppwpa libraries which have portions of their code located in TXT1 which will limit your available CODE space. In order to build the larger than 64KB ROM images or any wireless product, you **must** have TASM.EXE.

The V6 build process will attempt to relocate pieces of code into RAM2 and RAM3. In order to make this happen, the linker will need to make multiple passes. The make process will find the far links and create the required "proxy" code to access those far modules. During the first pass of the linker it is **normal** to see FIXUP errors. These errors should not be seen on the final link pass.

The product "small" builds (ie: xptexs.rom) will attempt to keep the single CODE segment as 64KB. The normal build (xptex.rom), will attempt to locate **your** application in TXT1 (RAM2) while the rest of the code resides in TEXT (RAM3). You can change the location of certain code pieces by manipulating the Makefile.

SNMP functions referenced from the MIB should be built with G_FLAGS (located in TEXT not TXT1).

Main.c change

Changes were made to support additional network interfaces which includes two new external references. VersionInit() was changed to add support for arp cache timeout and multiple network interfaces.

Setpar.c change

All instances of putcst_pde were changed to putst_pde.

A bug was fix in the baudratestrings array.

defaultWiFiSettings() now takes two parameters. Source code is available in stdf\wifi.c.

Demo.c change

All instances to ChanS2() have been changed to ChanS2NoTel() so telnet mode is no longer the default in the demo projects. All references to GChanS2() have been removed.

SNMP change

The entry point for your SNMP mib has changed from priv_mib to data_priv_mib.

Other changes

Part of 802.11i is now supported on the wireless products.

All references to \n\r have been changed to \r\n in the demos.

Additional Notes

Library Functions

All library functions are compiled into libraries. To use them, you must include the related library into your project.

Tools

All tools are available as source code. You can either include `tools.c` into your project or copy the functions you need into your own source.

Input and Output Functions

FlushIn

Description: Clears input buffer of associated stream.

Location: `kern100.lib, kernMAC.lib`

Prototype: `io.h`

Syntax: int **FlushIn**(void)

Parameter: None

Return value:

fprintf

Description: Print formatted into a FIFO.

Location: `kern100.lib, kernMAC.lib`

Prototype: `io.h`

Syntax: int **fprintf**(WORD *FIFO, const char *format, d1, …, dx)

Parameter: *FIFO = pointer to FIFO
format = format string
d1…dx = data to be printed

See **printf()** for format variables table.

Return value: 0 = OK
-1 = Stream not open (null-pointer)

getch

Description: Get one char of associated stream.

Location: `kern100.lib`

Prototype: `io.h`

Syntax: char **getch**(void)

Parameter: None

Return value: Received char (blocking)

get_int

Description: Get integer value from a string. Get_int() will skip over leading characters that are less than 0x20.

Location: `kern100.lib, kernMAC.lib`

Prototype: `io.h`

Syntax: char ***get_int**(BYTE *buf, WORD *value)

Parameter: *buf = pointer to string
*value = pointer to value

Return value: Pointer to char terminating the value
0 = No value parsed

get_ips

Description: Get IP address from a string. Get_ips() will skip over leading characters that are less than 0x20.

characters that are less than 0x20.

Location: `kern100.lib, kernMAC.lib`

Prototype: `io.h`

Syntax: char ***get_ips**(char *buf, BYTE *ip)

Parameter: *buf = pointer to string
*ip = pointer to IP address (4 bytes)

Return value: Pointer to character terminating the string
Null pointer = Error

gethex

Description: Read hexadecimal value while optionally printing the current value. If v is 0, the contents of vl will be printed as a prompt. The new value will be stored in vl. However, if vl is NULL, no value will be printed and the new value will be stored in v.

Location: `kern100.lib, kernMAC.lib`

Prototype: `io.h`

Syntax: char **gethex**(WORD *v, WORD *vl)

Parameter: *v = value
*vl = last value

Return value: char value

gethex8

Description: Read one hexadecimal byte while optionally printing the current value. If v is 0, the contents of vl will be printed as a prompt. The new value will be stored in vl. However, if vl is NULL, no value will be printed and the new value will be stored in v.

Location: `kern100.lib, kernMAC.lib`

Prototype: `io.h`

Syntax: int **gethex8**(BYTE *v, BYTE *vl)

Parameter: *v = value
*vl = last value

Return value: Last input character

getint

Description: Read integer value while optionally printing the current value. If v is 0, the contents of vl will be printed as a prompt. The new value will be stored in vl. However, if vl is NULL, no value will be printed and the new value will be stored in v.

Location: `kern100.lib, kernMAC.lib`

Prototype: `io.h`

Syntax: int **getint**(WORD *v, WORD *vl)

Parameter: *v = value
*vl = last value

Return value: Value

getint8

Description: Read one integer byte value while optionally printing the current value. If v is 0, the contents of vl will be printed as a prompt. The new value will be stored in vl. However, if vl is NULL, no value will be printed and the new value will be stored in v.

Location: `kern100.lib, kernMAC.lib`

Prototype: `io.h`

Syntax: int **getint8**(BYTE *v, BYTE *vl)

Parameter: *v = value
*vl = last value

Return value: Value

getip

Description: Read a 4 bytes IP address as 4 decimal values. The current address bytes are printed out in decimal and can be used as a value when pressing **<**Enter**>** or a point.

Location: `kern100.lib, kernMAC.lib`

Prototype: `io.h`

Syntax: void **getip**(BYTE *p)

Parameter: *p = pointer to IP address

Return value: None

getstr

Description: Read a string with echo. ATTENTION, the string must have a size of at least (maxlen + 1) !!!

Location: `kern100.lib, kernMAC.lib`

Prototype: `io.h`

Syntax: int **getstr**(char *buf, int maxlen)

Parameter: buf = pointer to string
maxlen = max. string length

Return value: TRUE = if chararacters are placed in buf
FALSE = otherwise

getyn

Description: Read boolean value from input. If only <Enter> is pressed the default value is used.

Location: `kern100.lib, kernMAC.lib`

Prototype: `io.h`

Syntax: int **getyn**(int default)

Parameter: default = 1 if Y, 0 if N

Return value: 1 if Y, 0 if N

getynt

Description: Read boolean value with writing default.. If only <Enter> is pressed the default value is used.

Location: `kern100.lib, kernMAC.lib`

Prototype: `io.h`

Syntax: int **getynt**(int default)

Parameter: default = 1 if Y, 0 if N

Return value: 1 if Y, 0 if N

kbhit

Description: Check and return number of bytes available for reading from associated stream.

Location: `kern100.lib, kernMAC.lib`

Prototype: `io.h`

Syntax: int **kbhit**(void)

Parameter: None

Return value: Number of characters available for reading.

OutBuf

Description: Check and return number of bytes available for writing to the associated stream.

Location: `kern100.lib, kernMAC.lib`

Prototype: `io.h`

Syntax: int **OutBuf**(void)

Parameter: None

Return value: Number of characters available for writing.

printf

Description: Print formatted to associated stream.

Location: `kern100.lib, kernMAC.lib`

Prototype: `io.h`

Syntax: int **printf**(const char *format, d1, …, dx)

Parameter: format = format string
d1…dx = data to be printed

Return value: 0 = OK
-1 = Error

Following format string variables are supported:

Type	Format string	Remarks
unsigned int	%nu	n is single digit length
signed int	%nd	n is single digit length
hex	%nx	n is single digit length
char	%c	single char
string	%ns	n is single digit length
time (dword)	%T	Time in ms is argument, prints as xxx.yyy (y seconds fractions)
pointer (long)	%P	Prints pointer as xxxx:yyyy
IP address	%nA	With n == 3, fixed format, without n no leading zeroes
Hardware address	%nH	With n != 0, xx:yy:zz format, with n == 0 xxyyzz format, hardware address is BYTE * parameter, IF PARAMETER IS (BYTE *) 0, the own address is printed
Serial number	%S	Lantronix serial number (7 digits)
Software version	%nV	Lantronix UDS Software Version, format Vx.y or x.ybz. If parameter n =! 0, include release date (yymmdd)

putch

Description: Send one character on associated stream via sendblk.

Location: `kern100.lib, kernMAC.lib`

Prototype: `io.h`

Syntax: void **putch**(BYTE c)

Parameter: c = character to send

Return value: None

putcstr

Description: Writes a string constant to associated stream *(1)*. To output the same string again simply call the function (or **putcstn()**) only with the label *(2)*.

This function saves memory because all strings are stored only once. The strings are converted into the files `texte.asm` and `texte.h` by the `filt` program.

Location: `kern100.lib, kernMAC.lib`

Prototype: `io.h`

Syntax: *(1)* void **putcstr**(`T_TXT1/*"Demo project"*/`)

(2) void **putcstr**(`T_TXT1`)

Parameter: Label [and string].

Return value: None

putcstn

Description: Writes a string constant to associated stream with leading CRLF *(1)*. To output the same string again simply call the function (or **putcstr()**) only with the label *(2)*.

This function saves memory because all strings are stored only once. The strings are converted into the files `texte.asm` and `texte.h` by the `filt` program.

Location: `kern100.lib, kernMAC.lib`

Prototype: `io.h`

Syntax: *(1)* void **putcstn**(`T_TXT1/*‟Demo project‟*/`)

(2) void **putcstn**(`T_TXT1`)

Parameter: Label [and string].

Return value: None

putyn

Description: Write ‘Y‘ or ‘N‘ depending on parameter.

Location: `kern100.lib, kernMAC.lib`

Prototype: `io.h`

Syntax: void **putyn**(int i)

Parameter: i = 1 for ‘Y‘ or 0 for ‘N‘

Return value: None

sendblk

Description: Send data on serial interface with automatic interface recognition (RS232/RS485).
This is not a library function but defined in tools.c.

Location: `tools.c`

Prototype:

Syntax: void **sendblk**(BYTE *sb, WORD len)

Parameter: *sb = send buffer
len = number of chars to send

Return value: None

sprintf

Description: Print formatted into a string.

Location: `kern100.lib, kernMAC.lib`

Prototype: `io.h`

Syntax: int **sprintf**(char *string, const char *format, d1, …, dx)

Parameter:
*string = pointer to string
format = format string
d1…dx = data to be printed

See **printf()** for format variables table.

Return value:
0 = OK
-1 = Error

Format Conversions

a2toh

Description: Converts two-digit hex values in ASCII notation to integer.

Location: `kern100.lib, kernMAC.lib`

Prototype: `kernel.h`

Syntax: BYTE pascal **a2toh**(char *p)

Parameter: *p = hex string to convert

Return value: Integer value of hex string

a2toi

Description: **atoi**() with only two digits.

Location: `kern100.lib, kernMAC.lib`

Prototype: `None`

Syntax: pascal **a2toi**(char *p)

Parameter: *p = two-digit decimal value

Return value: Integer value of p

atoi

Description: Convert an ASCII string to integer.

Location: `kern100.lib, kernMAC.lib`

Prototype: `kernel.h`

Syntax: WORD pascal **atoi**(char *p)

Parameter: *p = integer value as ASCII string

Return value: Integer value of p

decodeBase64

Description: Convert a base64 encoded string into a byte array.

Location: `kern100.lib, kernMAC.lib`

Prototype:

Syntax: void decodeBase64(BYTE *chBase64, BYTE*chStr)

Parameter: *chBase64 = pointer to Base64 encoded string
*chStr = pointer to array to receive the decoded string

Return value: None

encodeBase64

Description: Converts a byte array into a base64 encoded string.

Location: `kern100.lib, kernMAC.lib`

Prototype:

Syntax:

```
void encodeBase64(BYTE *bStr, WORD bLen,
BYTE *chBase64
```

Parameter: *bStr = pointer to source string to be encoded

bLen = number of BYTES to encode

*chBase64 = pointer to array to receive the encoded array

Return value: None

Time Functions

get_trand

Description: Get timer random value, used for random seed.

Location: `kern100.lib, kernMAC.lib`

Prototype: `kernel.h`

Syntax: WORD **get_trand**(void)

Parameter: None

Return value: Timer value

sdelay

Description: Wait for a specified time period. During delay time the multitasking is enabled.

Location: `kern100.lib, kernMAC.lib`

Prototype: `kernel.h`

Syntax: void **sdelay**(WORD t)

Parameter: t = delay time in ms

Return value: None

MsGet

Description: Get actual seconds fraction.

Location: `kern100.lib, kernMAC.lib`

Prototype: `kernel.h`

Syntax: DWORD **MsGet**(void)

Parameter: None

Return value: Actual timer value in ms (seconds fraction)

Math Functions

lmod

Description: Mod long with int.

Location: `kern100.lib, kernMAC.lib`

Prototype: `kernel.h`

Syntax: WORD **lmod**(DWORD lx, WORD y)

Parameter: lx
y

Return value: lx%y

lmul

Description: Multiplication of long with int to long: long*int = long.

ATTENTION:
The call of the function must have the following syntax:

```
resultlong = lmul(longvalue, intvalue);
```

The function then creates two nibbles out of the word. ***That means defined with three parameters but called with two!***

Location: `kern100.lib, kernMAC.lib`

Prototype: `kernel.h`

Syntax: DWORD **lmul**(WORD ll, lh, cons)

called as

DWORD **lmul**(DWORD lw, WORD cons)

Parameter: ll, lh = the two words of the long value
cons = the integer value

Return value: lllh*cons

longdiv

Description: Div long with int.

Location: `kern100.lib, kernMAC.lib`

Prototype: `kernel.h`

Syntax: DWORD **longdiv**(DWORD lx, WORD y)

Parameter: lx
y

Return value: lx/y

String Functions

sprl

Description: Simulates something like a **sprintf**(). Prints a long value into a string. The string length is always 10 chars.

Location: `tools.c`

Prototype: None

Syntax: BYTE ***sprl**(BYTE *p, DWORD val)

Parameter: *p = where to put string (len must be min. 10)
val = integer value for conversion

Return value: Pointer to end of created string

strchr

Description: Search for first appearance of a character in a string.

Location: `romlib.lib`

Prototype: `string.h`

Syntax: char ***strchr**(char *s, int c)

Parameter: *s = string
c = character

Return value: Pointer to c in string

strcpy

Description: This copies characters from the string *from* (up to and including the terminating null character) into the string *to*. Like **memcpy()**, this function has undefinded results if the strings overlap.

Location: `romlib.lib`

Prototype: `string.h`

Syntax: void **strcpy**(char *to, char *from)

Parameter: *to = first string
*from = second string

Return value: None

strcmp

Description: The **strcmp** function compares the strings *s1* and *s2*.

Location: `romlib.lib`

Prototype: `string.h`

Syntax: int **strcmp**(char *s1, char *s2)

Parameter: *s1 = string 1
*s2 = string 2

Return value: < 0 if s1 < s2
== 0 if s1 and s2 are equal
> 0 if s1 > s2

strncmp

Description: The **strncmp** function compares the first *len* characters of the strings *s1* and *s2*.

Location: `romlib.lib`

Prototype: `string.h`

Syntax: int **strcmp**(char *s1, char *s2, unsigned len)

Parameter: *s1 = string 1
*s2 = string 2
len = number of chars to compare

Return value: < 0 if s1 < s2
== 0 if first *len* bytes of s1 and s2 are equal
> 0 if s1 > s2

strlen

Description: Get string length.

Location: `romlib.lib`

Prototype: `string.h`

Syntax: int **strlen**(char *s)

Parameter: *s = string

Return value: String length.

Memory Functions

MBufInit()

Description:	This function initialises up to 32 1KB pool buffers in RAM0
Location:	`kern100.lib, kernMAC.lib`
Prototype:	`mbuf.h`
Syntax:	void **MBufInit**(void)
Parameter:	None.
Return value:	None.

MBufGet()

Description:	This function allocates one buffer from the pool.
Location:	`kern100.lib, kernMAC.lib`
Prototype:	`mbuf.h`
Syntax:	void ***MBufGet**(void)
Parameter:	None.
Return value:	Pointer to 1KB buffer

MBufFree()

Description:	This function frees the buffer back to the pool.
Location:	`kern100.lib, kernMAC.lib`
Prototype:	`mbuf.h`
Syntax:	void **MBufFree**(void *buf)
Parameter:	buf = buffer to free.
Return value:	None.

memset

Description:	This function copies the value of *c* into each of the first *size* bytes of the object beginning at *block*.
Location:	`romlib.lib`
Prototype:	`memory.h`
Syntax:	void **memset**(BYTE *block, int c, WORD size)
Parameter:	*block = buffer c = value size = number of bytes to fill
Return value:	None.

memcpy

Description: This function copies *size* bytes from the object beginning at *from* into the object beginning at *to*. The behavior if this function is undefined if the two arrays *to* and *from* overlap.

Location: `romlib.lib`

Prototype: `memory.h`

Syntax: BYTE **memcpy**(BYTE *to, BYTE *from, WORD size)

Parameter: *to = buffer 1
*from = buffer 2
size = number of bytes to copy

Return value: The value returned by **memcpy()** is the value of *to*.

memcmp

Description: Compares the first *len* bytes of two blocks.

Location: `romlib.lib`

Prototype: `memory.h`

Syntax: int **memcmp**(const void *to, const void *from, WORD len)

Parameter: *to = buffer 1
*from = buffer 2
len = number of bytes to compare

Return value: < 0 if buffer 1 < buffer 2
== 0 if first *len* bytes of buffer 1 and buffer 2 are equal
> 0 if buffer 1 > buffer 2

movedata

Description: This function copies *len* bytes from source address to destination address over different segments (far copy).

Location:

Prototype: `memory.h`

Syntax: void **movedata**(WORD fseg, WORD from, WORD tseg, WORD to, WORD len)

Parameter: fseg = source segment
from = source address
tseg = destination segment
to = destination address
len = number of bytes to copy

Return value: None

UDP Functions

udp_register

Description: Set function to be called for incoming data on this port number. This function will handle the incoming datagram.

Example:

udp_register(1234, (PTF) rcvr)

Prototype for receiver function:

rcvr(BYTE *buf, int len, int bflg, BYTE *xip, AD_T *a, WORD from, WORD to)

Parameters:

buf	UDP content received
len	Length of UDP content
bflg	True if block came in by broadcast
xip	Source IP address (pointer)
a	Source address structure (pointer)
from	From port
to	Destination port

Location: `tcpip.lib`

Prototype: `udp.h`

Syntax: **udp_register**(WORD port, PTF funct)

Parameter: port = port number
funct = pointer to receiver function for this port number

Return value: Always 1

udp_reregister

Description: Change or delete a registered function for a specific port number.

Location: `tcpip.lib`

Prototype: `udp.h`

Syntax: int **udp_reregister**(WORD port, PTF funct)

Parameter: port = port number to change
funct = pointer to new function for this port number or NULL for removing the actual function

Return value: Always 1

udp_send

Description: Send a buffer contents using UDP.

Location: `tcpip.lib`

Prototype: `udp.h`

Syntax: void **udp_send**(BYTE *ipaddr, WORD srcport, WORD destport, BYTE *buf, WORD len)

Parameter: *ipaddr: IP address of target, 0 (NULL) for broadcast
srcport: Source port number
destport: Destination port number

*buf: Buffer
len: Number of bytes to send

Return value: None

udp_sehw

Description:	Send a buffer contents using UDP.
Location:	`tcpip.lib`
Prototype:	`udp.h`
Syntax:	int udp_sehw(AD_T *adr,WORD srcport, WORD destport, BYTE *buf,int len)
Parameter:	adr: filled in address structure of destination srcport: Source port number destport: Destination port number *buf: Buffer len: Number of bytes to send
Return value:	Always 0

TCP Functions

TCPAlloc

Description:	Allocate a TCP structure.
Location:	`tcpip.lib, tcp12.lib, tcp16.lib`
Prototype:	`tcp.h`
Syntax:	TCP_t t = **TCPAlloc**(void)
Parameter:	None
Return value:	TCP structure

TCPOpen

Description:	Open a TCP connection.
Location:	`tcpip.lib, tcp12.lib, tcp16.lib`
Prototype:	`tcp.h`
Syntax:	int **TCPOpen**(int mode, TCP_t *t, int port)
Parameter:	int mode = 0 for passive connection, 1 for active connection *t = TCP structure port = TCP port number to open. When mode = 1 (active connection) the port can be zero, then every connection gets a unique port number.
Return value:	0 = OK -1 = active open failed

TcpWriteNB

Description: TCP Write without FIFO.

Location: `tcpip.lib, tcp12.lib, tcp16.lib`

Prototype: `tcp.h`

Syntax: WORD **TcpWriteNB**(TCP_t *t, void far *s, WORD len)

Parameter: *t = TCP structure

*s = data buffer

len = number of bytes to send

Return value: Number of bytes written

T_Discon

Description: Close TCP connection.

Location: `tcpip.lib, tcp12.lib, tcp16.lib`

Prototype: `tcp.h`

Syntax: void **T_Discon**(TCP_t t)

Parameter: T = TCP structure

Return value: None

ChanRcv

Description: TCP connection receiver. ChanRcv is called when data has been received on connection. The standard function places the data into the receive FIFO.

Location: `tcpip.lib, tcp12.lib, tcp16.lib`

Prototype: None

Syntax: int ChanRcv(TCP_t *t, BYTE *buf, int len)

Parameter: T = pointer to a TCP_t structure
*buf = pointer to a receive buffer
len = number of bytes received

Return value: > int, but returns nothing... This should be changed to void.

ChanS2

Description: TCP connection status function. ChanS2 is call upon five different conditions or states of the TCP connection. The standard function will enable Telnet protocol.

Location: kern100.lib, kernMAC.lib

Prototype: None

Syntax: WORD ChanS2(TCP_t *t, int function, int option)

Parameter: t = pointer to a TCP_t structure
> function = connection function state
> option = function option

Return value: Always 1

Function	Option	Meaning
1	Unused	Upon receipt of first TCP SYNC packet. Return 1 to accept connection, 0 to deny.
2	Unused	Upon state being reset to LISTEN - connection ended.
3	N/A	Reserved
4	int	Upon acceptance of the connection, and switching to ESTABLISHED. Option is 0 for incoming, 1 for outgoing connection.
5	Unused	Used when t->RcvFifo is NULL. This call needs to return the size of the buffer available for incoming TCP packets. The tcp stack will advertize this as the 'window size'.

To disable telnet functionality, you'll need to write your own TCP Channel Status function. For example :

```
int GChanS2(TCP_t *t, int typ, int mode)
{
        if (typ==4) {
                t->TelBits=0; /* Not a telnet connection */
        }
        return(1);
}
```

Or use ChanS2NoTel().

Configurable Pin Functions

defaultCP_settings

Description: Set Default Configurable Pin Settings into S_tmpRec7

Location: `<prod>.lib`

Prototype: `bitsXP.h`

Syntax: void defaultCP_settings(void);

Parameter: None

Return value: None

SaveCPsettings

Description: Save settings from S_tmpRec7 into working DIO structure

Location: `<prod>.lib`

Prototype: `bitsXP.h`

Syntax: void SaveCPsettings(void);

Parameter: None

Return value: None

dio_vbit_init

Description: Initialize low level PIO settings

Location: `<prod>.lib`

Prototype: `digio.h`

Syntax: void dio_vbit_init(void);

Parameter: None

Return value: None

dio_vbit_in

Description: Set configurable pin to input

Location: `<prod>.lib`

Prototype: `digio.h`

Syntax: void dio_vbit_in(WORD pin);

Parameter: pin = USER1, USER2 or USER3

Return value: None

dio_vbit_out

Description: Set configurable pin to output, and set state

Location: `<prod>.lib`

Prototype: `digio.h`

Syntax: void dio_vbit_out(WORD pin, WORD val);

Parameter: pin = USER1, USER2 or USER3
val = 0 or 1

Return value: None

dio_vbit_read

Description: Read configurable pin

Location: `<prod>.lib`

Prototype: `digio.h`

Syntax: int dio_vbit_read(WORD pin);

Parameter: pin = USER1, USER2 or USER3

Return value: int = 0 or 1

dio_vbit_reset

Description: Reset configurable pin

Location: `<prod>.lib`

Prototype: `digio.h`

Syntax: void dio_vbit_reset(WORD pin);

Parameter: pin = USER1, USER2 or USER3

Return value: None

dio_vbit_set

Description: Set configurable pin

Location: `<prod>.lib`

Prototype: `digio.h`

Syntax: void dio_vbit_set(WORD pin);

Parameter: pin = USER1, USER2 or USER3

Return value: None

dio_vbit_setres

Description:	Set or reset configurable pin
Location:	`<prod>.lib`
Prototype:	`digio.h`
Syntax:	void dio_vbit_setres(WORD pin, WORD val);

Parameter:	pin = USER1, USER2 or USER3
	val = 0 or 1

Return value:	None

Web Functions

SetServicePort

Description: Set the port number for a particular service

Location: `kernel.lib`

Prototype: `kernel.h`

Syntax: void SetServicePort(WORD srv, WORD port_number);

Parameter: srv = service
port_number = service socket port number

Return value:

WebMethRegister

Description: Add a callback method for web requests

Location: `web.lib`

Prototype: `web.h`

Syntax: int WebMethRegister(char *meth, int (*f)(), char *path);

Parameter: meth = http method (POST, GET, HEAD)
f = callback function (must be declared global)
path = relative path following http://<ip_address>/

Return value:

Example:

```
WebMethRegister(“GET”, my_callback, “call_my_callback.cgi”);
int my_callback(WCT *w, char *file, char *hdr);
```

DNS Functions

dns_resolve

Description: Returns the IP address for a given hostname.

Location: `tcpip.lib, tcp12.lib, tcp16.lib`

Prototype: `ip.h`

Syntax: DWORD **dns_resolve**(char *hostname)

Parameter: *hostname = name to resolve
The name server IP address has to be set in the IP structure.

Return value: IP address, 0.0.0.0 if error occurred

Example:

```
ip.ns[0] = 194; /* set the name server IP address */
ip.ns[1] = 39;
ip.ns[2] = 78;
ip.ns[3] = 11;
*((DWORD *) ip.addr) = dns_resolve(“jl232.pronet.de”);
printf(“IP address is: %A\n\r”, ip.addr);
```

Multitasking Functions

kill

Description: Terminate a process.

Location: `kern100.lib`

Prototype: None

Syntax: void **kill**(TCB *proc_tcb)

Parameter: *proc_tcb = TCB of process to kill

Return value: None

nice

Description: Allow changing of the Task Control Block.

Location: `kern100.lib`

Prototype: `kernel.h`

Syntax: void **nice**(void)

Parameter: None

Return value: None

reset

Description: Reset device immediately.

Location: `kern100.lib`

Prototype: `kernel.h`

Syntax: void **reset**(void)

Parameter: None

Return value: None

spawn

Description: Start a new process

Location: `kern100.lib`

Prototype: `kernel.h`

Syntax: void **spawn**(void *start, BYTE *newtcb, int size, int Chan, char *Name);

Parameter:

*start = process name (function)
*newtcb = Stack
size = Stack size
Chan = 0: using first serial interface
= 1: using second serial interface
= -1: no serial interface used
*Name = Process name shown in process table

Return value: None

FIFO Control

FifoInit

Description: Initialize a FIFO.

Location: `kern100.lib`

Prototype: `io.h`

Syntax: WORD **FifoInit**(BYTE *buf, int size)

Parameter: size = size of FIFO. Must be a power of 2!

buf = memory used for storing the FIFO. buf must be at least size + 8 bytes large as it contains also the FIFO control block.

Return value: Pointer to FIFO.

IOCall

Description: The IOCall function accesses the queue p and executes function f. Depending on function f additional paramters might be needed.

Location: `kern100.lib`

Prototype: `io.h`

Syntax: int **IOCall**(f, p [,x])

Parameter: f = Function
p = Address of a pointer to FIFO (see Example)
x = Additional parameters

Defined functions:

f	x	Function
0	none	Check and return number of bytes available for reading
1	none	Get one byte out of queue
2		*NOT IMPLEMENTED*
3	none	Clear buffer
4	none	Check and return free buffer space available for storing
5	byte	Store one byte into queue
6	buf, len	Store many bytes from buf in queue. **IOCall()** returns when whole buffer is sent.
7	byte	"Unget", stuff byte back to top of queue
8		*NOT IMPLEMENTED*
9	none	Buffer size, return value is "mask", which is equ. to size-1

Return value: See table.

Example:

Each process – and thus channel control block – has two FIFO's for input and output functions defined. The addresses of these FIFO's are stored in a structure, to which a pointer is held in the TCB (task control block):

ActPro->IO_Ptr	points to a array, containing two pointers to FIFO's
ActPro->IO_Ptr[0]	the input queue address
ActPro->IO_Ptr[1]	the output queue address

Clear the output queue of the associated stream. This can be either a serial or a TCP stream.

```
IOCall(3, (BYTE *) &ActPro->IO_Ptr[1]);
IOCall(3, (BYTE *) &t->XmitFifo);
```

LED Control

BlinkReset

Description:	Use updated LED variables immediately.
Location:	`kern100.lib`
Prototype:	`kernel.h`
Syntax:	void **BlinkReset**(void)
Parameter:	None
Return value:	None

error

Description:	Red LED is on, yellow LED shows error code. After 20 seconds the device is resetting.
Location:	`kern100.lib`
Prototype:	`kernel.h`
Syntax:	void **error**(WORD w)
Parameter:	w = BlinkGWord[1]
Return value:	None

Serial Port Control

InitLocalChan

Description: Init of channel specific setup, interrupt controlled drivers are activated with full support (flow control, etc.), buffers are flushed, hardware "unlocked", status reset.

To switch serial interface specifics (i. e. flow control method) the settings have to be put in the channel control block (CCB) and **InitLocalChan()** must be called for the changes to take effect. Buffers will be flushed.

Location: `kern100.lib`

Prototype: `io.h`

Syntax: void **InitLocalChan**()

Parameter: None

Return value: None

InitLocalIO

Description: General serial interface reset, setup of vectors, hardware detection. Should only be used at bootup time. Serial "debug driver" is initialized on first channel, polled I/O.

Location: `kern100.lib`

Prototype: `io.h`

Syntax: void **InitLocalIO**()

Parameter: None

Return value: None

lio_cts

Description: Set state of CTS line. Accesses currently active interface.

Location: `kern100.lib`

Prototype: `io.h`

Syntax: void **lio_cts**(int state)

Parameter: TRUE or FALSE

Return value: None

lio_dcd

Description: Set state of DCD line. Accesses currently active interface.

Location: `kern100.lib`

Prototype: `io.h`

Syntax: void **lio_dcd**(int state)

Parameter: TRUE or FALSE

Return value: None

lio_tx

Description:	Set transmit state of RS485/2-wire interface.
Location:	`kern100.lib`
Prototype:	`io.h`
Syntax:	void **lio_tx**(int state)
Parameter:	state = 0: disable, 1: enable
Return value:	None

lio_rts

Description:	Get state of RTS line. Accesses currently active interface.
Location:	`kern100.lib`
Prototype:	`io.h`
Syntax:	int **lio_rts**(void)
Parameter:	None
Return value:	Returns, always True, at the moment

lio_cok

Description:	Get state of DTR line. Accesses currently active interface
Location:	`kern100.lib`
Prototype:	None
Syntax:	int **lio_cok**(void)
Parameter:	None
Return value:	TRUE or FALSE

lio_rva

Description:	Set state of RVA pin.
Location:	`kern100.lib`
Prototype:	None
Syntax:	void **lio_rva**(int state)
Parameter:	TRUE or FALSE
Return value:	None

LIOObuf

Description: Get empty state of the transmit register

Location: `kern100.lib`

Prototype: Io.h

Syntax: WORD **LIOObuf**()

Parameter: None

Return value: 0 if empty, non zero otherwise

LioBrk

Description: Send a break signal. LioBrk sends a break on the channel defined by the ccb.

Location: `kern100.lib`

```
uds.lib: for the AMD platforms(UDS10,
Mini REv2, Micro, FL, ...)

ec1.lib: for DSTNILX platfoms (UDS100
only at the moment)

cbx.lib: for V40 platforms (E2, DR1)
```

Prototype: `io.h`

Syntax: void LioBrk(CCB *ccb)

Parameter: ccb = pointer to channel control block

Return value: None

EEPROM Functions

These functions read and write data into ‘setup or configuration’ memory. Typically, the programmer will change values in Setup[], then use StoreCMOS() to save these values. If you do not use StoreCMOS() and use EE_Write() directly, be sure you **do not** write over the first 6 bytes of ‘setup’ memory (kernel dependent information is stored there). All of these functions are automatically indexed into page 1 on the XPort. However, it **is** possible to overwrite the firmware image.

EE_Read

Description:	Read block out of EEPROM and validate checksum.
Location:	`kern100.lib, kernMAC.lib`
Prototype:	`kernel.h`
Syntax:	pascal **EE_Read**(BYTE *buf, WORD adr, WORD len)
Parameter:	buf = buffer to store content adr = address in EEPROM len = number of bytes to read Checksum is checked but not stored in buf!
Return value:	0 = ok, 1 = checksum error, 2 = memory error

EE_Write

Description:	Write block to EEPROM and calculate checksum.
Location:	`kern100.lib, kernMAC.lib`
Prototype:	`kernel.h`
Syntax:	pascal **EE_Write**(BYTE *buf, WORD adr, WORD len)
Parameter:	buf = buffer to store content adr = address in EEPROM len = number of bytes to write 2 bytes more are stored at the end of buf to contain the checksum!
Return value:	0 = ok, 2 = memory error

StoreCMOS

Description:	Write setup array to EEPROM.
Location:	`kern100.lib, kernMAC.lib`
Prototype:	`kernel.h`
Syntax:	void pascal **StoreCMOS**(void)
Parameter:	None
Return value:	None

Flash Functions

These functions are used to write data into the WEB locations. On XPort, these functions automatically index to the correct page.

CopyEEPR

Description: Copy 64 K EEPROM contents into flash memory.

Location:

```
kern100.lib, kernMAC.lib

   uds.lib: for the AMD platforms(UDS10,
   Mini REv2, Micro, FL, ...)

   ec1.lib: for DSTNILX platfoms (UDS100
   only at the moment)

cbx.lib: for V40 platforms (E2, DR1)
```

Prototype: `flash.h`

Syntax: void **CopyEEPR**(void)

Parameter: None

Return value: None

flsh_clr

Description: Clear flash page.

Location:

```
kern100.lib, kernMAC.lib

   uds.lib: for the AMD platforms(UDS10,
   Mini REv2, Micro, FL, ...)

   ec1.lib: for DSTNILX platfoms (UDS100
   only at the moment)

cbx.lib: for V40 platforms (E2, DR1)
```

Prototype: `flash.h`

Syntax: void **flsh_clr**(WORD ofs, WORD page)

Parameter: page = page number to clear
ofs = offset ignored

Return value: None

flsh_pgm

Description: Program flash memory.

Location: `kern100.lib, kernMAC.lib`

```
   uds.lib: for the AMD platforms(UDS10,
   Mini REv2, Micro, FL, ...)

   ec1.lib: for DSTNILX platfoms (UDS100
   only at the moment)

cbx.lib: for V40 platforms (E2, DR1)
```

Prototype: `flash.h`

Syntax: void **flsh_pgm**(WORD dstofs, WORD dstseg, WORD srcofs, WORD srcseg, WORD count)

Parameter: dstofs = destination offset
dstseg = destination segment
srcofs = source offset
srcseg = source segment
count = number of pages to program (must be 0 for 64 K)

Return value: None

flsh_typ

Description: Get flash type.

Location: `kern100.lib, kernMAC.lib`

```
   uds.lib: for the AMD platforms(UDS10,
   Mini REv2, Micro, FL, ...)

   ec1.lib: for DSTNILX platfoms (UDS100
   only at the moment)

cbx.lib: for V40 platforms (E2, DR1)
```

Prototype: `flash.h`

Syntax: WORD **flsh_typ**(WORD ofs, WORD page)

Parameter: page = page number
ofs = offset

Return value: Flash type

Flash File System Functions

The following functions operate on files stored in flash memory.

get_file_curr_pos

Description:	Get file read location.
Location:	`serfl.lib, parfl.lib`
Prototype:	`filesys.h`
Syntax:	DWORD **get_file_curr_pos** (int handle)
Parameter:	handle = handle to previously opened file.
Return value:	File read pointer's current position, or -1 on error

get_file_len

Description:	Get file len.
Location:	`serfl.lib, parfl.lib`
Prototype:	`filesys.h`
Syntax:	DWORD **get_file_len** (int handle)
Parameter:	handle = handle to previously opened file.
Return value:	Length of file, or -1 on error

get_file_start_pos

Description:	Get file start location.
Location:	`serfl.lib, parfl.lib`
Prototype:	`filesys.h`
Syntax:	DWORD **get_file_start_pos**(int handle)
Parameter:	handle = handle to previously opened file.
Return value:	Offset to start of file, or -1 on error

r_close

Description:	Close a file..
Location:	`serfl.lib, parfl.lib`
Prototype:	`filesys.h`
Syntax:	int **r_close**(int handle)
Parameter:	handle = handle to previously opened file.
Return value:	always 0

r_open

Description: Opens a file.

Location: `serfl.lib, parfl.lib`

Prototype: `filesys.h`

Syntax: int **r_open**(BYTE far *fname)

Parameter: fname = filename

Return value: valid index to file handle, or -1 on error.

r_read

Description: Read from a file.

Location: `serfl.lib, parfl.lib`

Prototype: `filesys.h`

Syntax: WORD **r_read**(int handle, BYTE far *buf, WORD len)

Parameter: handle = handle to previously opened file.

buf = buffer to hold data.

len = maximum number of byte to read.

Return value: number of byte read, -1 on error

set_file_curr_pos

Description: Set file read location.

Location: `serfl.lib, parfl.lib`

Prototype: `filesys.h`

Syntax: int **set_file_ curr_pos** (int handle, DWORD offset)

Parameter: handle = handle to previously opened file.

offset = offset into the file

Return value: 0 = success

-1 = error

Random Generator Functions

rand

Description:	Get random number.
Location:	`kern100.lib, kernMAC.lib`
Prototype:	`random.h`
Syntax:	WORD **rand**(void)
Parameter:	None
Return value:	Random number (unsigned)

srand

Description:	Set random seed
Location:	`kern100.lib, kernMAC.lib`
Prototype:	`random.h`
Syntax:	void **srand**(WORD seed)
Parameter:	seed = random seed
Return value:	None

Encryption Functions

tf_byte_stream

Description: This function can encrypt or decrypt any number of bytes.

Location: `crypt.lib, crstub.lib`

Prototype: `2fish.h`

Syntax: void pascal **tf_byte_stream**(tf_block *tmpb, tf_key_struct *keystr, BYTE *input, int nBytes, BYTE *output, BYTE *pstat, enum tf_stream_mode mode)

Parameter: tmpb = temporary buffer used for decryption
keystr = key structure
input =buffer containing data to be decrypted
nBlocks = number of blocks to decrypt
output = buffer to store decrypted data
pstat = stores current state
mode = one out of: tf_ofb, tf_cfb_e, tf_cfb_d

Return value: None

tf_block_decrypt

Description: Decrypt a ciphered block.

Location: `crypt.lib, crstub.lib`

Prototype: `2fish.h`

Syntax: void pascal **tf_block_decrypt**(tf_block *tmpb, tf_key_struct *keystr, tf_block *input, int nBlocks, tf_block *output)

Parameter: tmpb = temporary buffer used for decryption
keystr = key structure
input =buffer containing data to be decrypted
nBlocks = number of blocks to decrypt
output = buffer to store decrypted data

Return value: None

tf_block_encrypt

Description: Encrypt a plain block.

Location: `crypt.lib, crstub.lib`

Prototype: `2fish.h`

Syntax: void pascal **tf_block_encrypt**(tf_block *tmpb, tf_key_struct *keystr, tf_block *input, int nBlocks, tf_block *output)

Parameter: tmpb = temporary buffer used for encryption
keystr = key structure
input =buffer containing data to be encrypted
nBlocks = number of blocks to encrypt
output = buffer to store encrypted data

Return value: None

tf_key_prep

Description: Key pre-processing. Needs to be done only once per key.

Location: `crypt.lib, crstub.lib`

Prototype: `2fish.h`

Syntax: void pascal **tf_key_prep**(tf_key_struct *keystr, void *key, BYTE keyLen)

Parameter: keystr = pointer to key structure
key = pointer to key string
keyLen = key length in bits (e.g. tf_key_128)

Return value: None

Debug Functions

Monitor

Description: Start ProMon.

Location: `kern100.lib, kernMAC.lib`

Prototype: `kernel.h`

Syntax: void **Monitor**(void)

Parameter: None

Return value: None

Command	Function
D	Dump memory. `D segment:offset`
E	Edit memory contents. `E segment:offset data`
F	Fill memory `F segment:offset,length data`
C	Clear flash page. `C segment`
H, ?	Help. Print command list.
I	Port input. `I port`
M	Move memory. `M 0000:source_offset,length->destination_offset`
O	Port output. `O port data`
P	Print process table.
Q	Quit ProMon.
T	Print timer.
V	Verify memory. Compares page 1 (0000:1000) with page 8 (0000:8000)
W	Switch to word mode prefix.
!	Echo mode.

syslog

Description: Send a message to a syslog server.

Location: `tools.c`

Prototype: `./syslog.h`

Syntax: void **syslog**(WORD level, char *message)

Parameter: level: priority level + facility level (see syslog.h)
message: pointer to message to be send.

Return value: None

SNMP

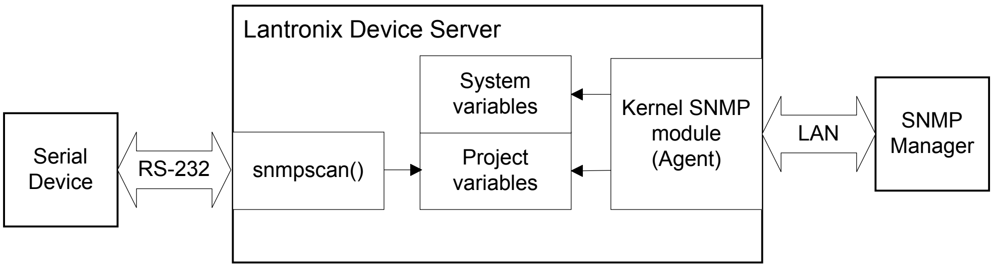
Introduction

Since it was developed in 1988, the Simple Network Management Protocol has become the de facto standard for internetwork management. Because it is a simple solution, requiring little code to implement, vendors can easily build SNMP agents to their products. SNMP is extensible, allowing vendors to easily add network management functions to their existing products. SNMP also separates the management architecture from the architecture of the hardware devices, which broadens the base of multivendor support. Perhaps most important, unlike other so-called standards, SNMP is not a mere paper specification, but an implementation that is widely available today.

For more detailed information on the SNMP protocol please read the file: SNMP - Simple Network Management Protocol.htm in the directory demo11.

CoBox SNMP structure

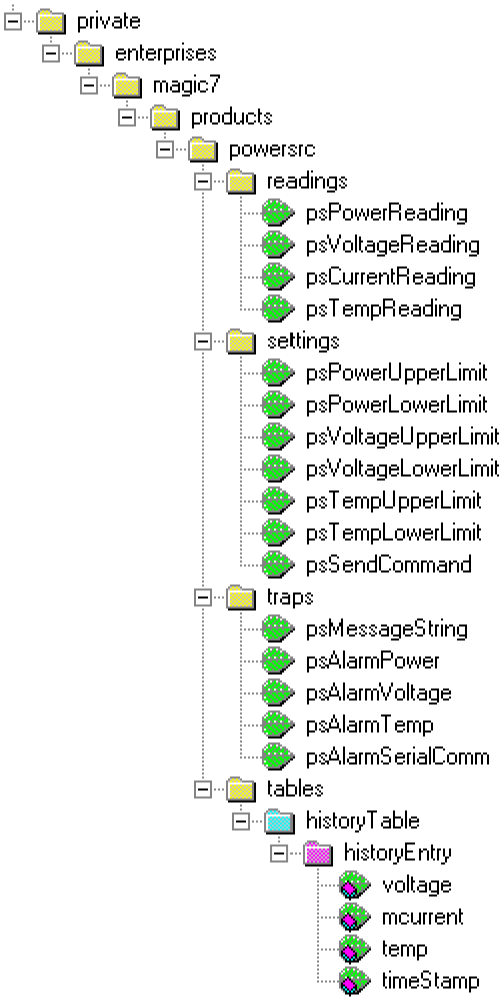
The kernel, as it comes in this development environment, contains the basic SNMP implementation for the Ethernet interface as it is found in almost any, Ethernet enabled device. It supports the storage of the IP settings and keeps track of several counters for network traffic. In picture 1 this is indicated as the System variables block. For the demosnmp example we have expanded the default SNMP functionality with a customized part, the Project variables block, which is described in a so-called ‘private MIB’. This private MIB is the definition of the extra information, which can be read or written by the SNMP manager. It has to be compiled to become understandable for the SNMP manager and , in the device server we have to implement a structure to hold the extra data in such a way that it is accessible by the kernel SNMP module.



picture 1 - **CoBox SNMP block diagram**

The example is meant to be the SNMP-management interface for a power supply. The power supply will report its temperature, output voltage and current in regular intervals to the serial port. The device server takes the voltage and current readings, multiply them to get delivered power and compare all the values against predefined limits. If one of them is out of range an appropriate SNMP-trap will be sent to the SNMP manager. The function **snmpscan()** handles all the incoming serial data from the power supply. The name of this handler function can be changed by the user in main.c and scan.c.

SNMP requests are processed by the kernel depending on the private MIB description. The file DEMOSNMP.MIB contains the text version of the private MIB while Picture 2 shows a graphic representation of it.



```
private
  enterprises
    magic7
      products
        powersrc
          readings
            psPowerReading
            psVoltageReading
            psCurrentReading
            psTempReading
          settings
            psPowerUpperLimit
            psPowerLowerLimit
            psVoltageUpperLimit
            psVoltageLowerLimit
            psTempUpperLimit
            psTempLowerLimit
            psSendCommand
          traps
            psMessageString
            psAlarmPower
            psAlarmVoltage
            psAlarmTemp
            psAlarmSerialComm
          tables
            historyTable
              historyEntry
                voltage
                mcurrent
                temp
                timeStamp
```

picture 2 - Private MIB for power supply control

SNMP environment

In the file snmp.c some static variables are defined for the SNMP environment.

Enterprise ID

A private MIB must contain a unique enterprise ID. For the example we have used a fake company called MAGIC7 with an enterprise ID of 7777. This enterprise ID must be declared as static variable enterpriset[]:

```
BYTE enterpriset[ ] = { 6, 7, 0x2b, 6, 1, 4, 1, 188, 97 };
```

This is the MIB tree iso.org.dod.internet.private.enterprises.magic7.

The first character is always 6, second character is length of object ID excl. length, and the third byte is 0x2b. The next characters represent *dod.internet.private.enterprises* which is OID 6.1.4.1.

Calculation of the enterprise id (7777 is taken as an example):

Given number: 7777 = 1E61(hex) = 0001 1110 0110 0001
Use seven bits, fill with zero: = 0011 1100 0110 0001
Set first bit to 1: = 1011 1100 0110 0001 = BC61 (hex)
= 188, 97 (dec)

SNMP object

Use the global variable S_obj[] to point at the start of the MIB tree handled by the device server:

GLOBAL BYTE S_obj[ ] = { 9, 0x2b, 6, 1, 4, 1, 188, 97, 1, 2 };

In our examples this is the MIB tree iso.org.dod.internet.private.enterprises.magic7.products.powersrc.

Traps variables

For each generated trap we have defined an OID. This OID will be used when generating the trap message, which can also hold optional measurement values.

Example:

```
BYTE onbvar[] = {
/* 1st char is always 6; 2nd length of objid excl. length */
                  6, 8, 0x2b, 6, 1, 2, 1, 1, 1, 0,           /* System Description
*/
                  6, 8, 0x2b, 6, 1, 2, 1, 1, 3, 0                     /* SysUpTime
*/
}; /* object-id for standard-traps */
BYTE onbvar2[] = {
6, 12, 0x2b, 6, 1, 4, 1, 188, 97, 1, 2, 3, 1, 0  /* Error-String */
}; /* object-id for customer specific Trap 2 */
BYTE onbvar3[] = {/* 1st char is always 6; 2nd length of objid excl. length */
6, 12, 0x2b, 6, 1, 4, 1, 188, 97, 1, 2, 3, 1, 0, /* Error-String */
6, 12, 0x2b, 6, 1, 4, 1, 188, 97, 1, 2, 1, 0, 0  /* Error-Value  */
/* This byte must be set to passed variable ---^      */
}; /* object-id for customer specific Trap 3 */
```

The OID onbvar[] is used with the coldstart trap, onbvar2[] is used with the serial communication traps while onbvar3[] is used when sending an alarm trap message.

The function ps_trap() is called from various places in snmpscan() and each time different parameters are passed causing the appropriate data to be included in the trap message.

```
void ps_trap( int typ, int spec )
{
int rc;
if( !*( ( DWORD * ) ( Setup + 22 ) ) )
/* check for SNMP mgr IP */
return;
else
    {
if( typ < 6 )
        {
/* Standard traps */
scon.inpptr = onbvar;
scon.inpend = onbvar + sizeof( onbvar );
        }
else
        {
/* Customer specific trap */
switch( spec )
            {
case 2:
scon.inpptr = onbvar2;
scon.inpend = onbvar2 + sizeof( onbvar2 );
break;
case 3:
scon.inpptr = onbvar3;
scon.inpend = onbvar3 + sizeof( onbvar3 );
break;
default:
scon.inpptr = onbvar;
scon.inpend = onbvar + sizeof( onbvar );
break;
            }
        }
    }

rc = snmp_trap( enterpriset, typ, spec ); /* create trap message */
if( rc > 0 )
    {
if( Setup[ 22 ] )
udp_send( Setup + 22, 162, 162, outbuf, rc );
if( Setup[ 26 ] )
udp_send( Setup + 26, 162, 162, outbuf, rc );
if( Setup[ 30 ] )
udp_send( Setup + 30, 162, 162, outbuf, rc );
    }
}
```

custom routine for sending traps

In this example the traps will be send to up to three manager IP addresses. A specific OID will be send as trap depending on the spec variable.

Checking access rights

Access rights are processed by the function **snmp_acheck()**. The access will be permitted depending on IP address and read/write community name of SNMP manager.

If no custom **snmp_acheck()** function is provided a default function will be used and access will always be granted.

```
int snmp_acheck(unsigned char *ip, char *community, WORD comlen)
{
int i;
BYTE IPflag=0;

/* Check IP address to be one of the defined SNMP managers, */
/* and check SNMP community name otherwise exit with no access */
IPflag = 0;                           /* set IP flag to false     */
for(i=0 ; i<3; i++) {
if ( PS.trapadr[i][0] != 0 )
if ((memcmp( ip, PS.trapadr[i], 4)) == 0) {
IPflag=1;                       /* set IP Flag to true      */
      }
  }


if (IPflag == 0)                      /* all tests failed?        */
  {
return(1);                          /* access denied            */
  }


  /*
Check SNMP Community
if community is Write Community - read and write access  - return (0)
  if community is Read  Community - read only access       - return (2)
  if community not known - no access                       - return (1)
we need to check the write community first, this is because if the
read and the write communities have the same name and read is checked
first, write requests are blocked as a read only access is returned
  */

if(strncmp(community, PS.wt_community, comlen) == 0)    /* write ? */
  {
return(0);                                 /* read-write access  */
  }
else
  {
if(strncmp(community, PS.rd_community, comlen) == 0)   /* read ? */
    {
return(2);                                 /* read-only access */
    }
else                                         /* then: not known  */
    {
return(1);                                 /* access denied    */
    }
  }
}
```

custom routine for checking access rights

Private MIB

MIB tree definition

The private MIB tree inside the CoBox, as defined in `snmpmib.h`, must exactly match the MIB file `DEMOSNMP.MIB` that is used in the MIB browser. The example shows the beginning of a MIB file with corresponding part in CoBox source.

You don’t have to use the same names inside the MIB and `priv_mib` but this will help understanding the structure.

MIB file :

```
magic7        OBJECT IDENTIFIER ::= { enterprises 7777 }

products      OBJECT IDENTIFIER ::= { magic7 1 }

powersrc      OBJECT IDENTIFIER ::= { products 2 }

readings      OBJECT IDENTIFIER ::= { powersrc 1 }
settings      OBJECT IDENTIFIER ::= { powersrc 2 }
traps         OBJECT IDENTIFIER ::= { powersrc 3 }
tables        OBJECT IDENTIFIER ::= { powersrc 4 }
```

Private MIB tree :

```
asm priv_mib:    dw 1, 7777
asm dw S_LEER                , 7776
asm dw S_SEQU                , magic7

asm magic7:      dw 1, 1
asm dw S_SEQU                , products

asm products:    dw 1, 2
asm dw S_LEER                , 1
asm dw S_SEQU                , powersrc

asm powersrc:    dw 1, 4
asm dw S_SEQU                , readings
asm dw S_SEQU                , settings
asm dw S_SEQU                , traps
asm dw S_SEQU                , tables
```

The enterprise name is magic7 and the enterprise identifier is 7777. The header file has to declare a structure with 7776 empty entries (`S_LEER`) and one for the enterprise ID. `S_SEQU` points to the enterprise `magic7`, which contains only one entry `products`. `products` contains ***two*** entries (`dw 1,` **2**): one is empty and the 2nd points to `powersrc`. `powersrc` itself contains ***four*** entries (`dw 1,` **4**).

Read-only variables

Until now only the MIB structure is defined. The next part shows the definition of some read-only variables.

MIB file:

```
psPowerReading OBJECT-TYPE
                   SYNTAX      INTEGER(0..400000000)
                   ACCESS      read-only
                   STATUS      mandatory
DESCRIPTION “Power output”
::= { readings 1 }
psVoltageReading OBJECT-TYPE
                   SYNTAX      INTEGER(0..65535)
                   ACCESS      read-only
                   STATUS      mandatory
DESCRIPTION “Voltage output”
::= { readings 2 }
psCurrentReading OBJECT-TYPE
                   SYNTAX      INTEGER(0..65535)
                   ACCESS      read-only
                   STATUS      mandatory
DESCRIPTION “Current output”
::= { readings 3 }
psTempReading OBJECT-TYPE
                   SYNTAX      INTEGER(0..65535)
                   ACCESS      read-only
                   STATUS      mandatory
DESCRIPTION “Temperature”
::= { readings 4 }
```

Private MIB tree:

```
asm readings: dw 1, 4
asm dw S_INT + S_LONG + S_RAM, PS.PowerReading
asm dw S_INT +          S_RAM, PS.VoltageReading
asm dw S_INT +          S_RAM, PS.CurrentReading
asm dw S_INT +          S_RAM, PS.TempReading
```

Variable type descriptors

Symbol	Description
S_SEQU	Sequence, directs to another label
S_LEER	Number of empty fields
S_RAM	Field is stored in RAM
S_ROM	Field is stored in ROM
S_OCTSTR	Field is a zero terminated string
S_INT	Field is an integer value
S_LONG	Field is a long value
S_TIPADR	Field is an IP address
S_TCTR	Field is a counter
S_TGAUGE	Field is a gauge
S_TTICK	Field is a timer
S_AFUN	Field will be handled by following function (put '_' in front)

Read/write variables

By adding the S_SET to the variable type it becomes writable, meaning that the value of it can be changed directly from the SNMP browser (manager). It is also possible to attach a separate function to handle variable changes. In this way extra checking and validation can be added.

Private MIB tree:

```
asm Settings: dw 1, 7
asm dw S_INT + S_LONG + S_RAM + S_SET     , PS.PowerUpperLimit
asm dw S_INT + S_LONG + S_RAM + S_SET     , PS.PowerLowerLimit
asm dw S_INT + S_RAM + S_SET               , PS.VoltageUpperLimit
asm dw S_AFUN                             , _voltage_Lower
asm dw S_INT + S_RAM + S_SET              , PS.TempUpperLimit
asm dw S_INT + S_RAM + S_SET              , PS.TempLowerLimit
asm dw S_AFUN                             , _send_command
```

Example function for handling a read/write variable, see comments:

```
voltage_Lower(op) WORD *op;
{
if ( ( scon.flags & FLAG_SET ) )                /* Setting ? */
    {
if ( ( scon.intval < 300) || (scon.intval > 500 ) )
        {
return S_badValue;
        }
if( scon.flags & FLAG_PASS )            /* just checking */
        {
return 0;
        }
else                                        /* now do it */
        {
PS.VoltageLowerLimit = (WORD) scon.intval;
return 0;
        }
    }
else
    {
if (scon.flags & FLAG_INCR)
        {
if (scon.objend == op )
            {
*scon.objend++ = 0;
            }
        }
if (scon.objend != op + 1 || op[0] != 0)
        {
return S_NextEntry;
        }
return (snmp_leaf(S_INT + S_RAM, &(PS.VoltageLowerLimit)));
    }
}
```

Tables

MIB definition for a table:

```
asm Tables:       dw 1, 1
asm dw S_SEQU, historyTable

asm historyTable: dw 0, 1
asm dw S_SEQU, historyEntry

asm historyEntry: dw 2, 4
asm dw S_AFUN, _history_table
```

This function is called to handle a request on the table:

```
history_table( WORD * op )
{
WORD i;
if( ( scon.flags & FLAG_SET ) &&
( !( scon.flags & FLAG_PASS ) ) ) return S_readOnly;
if( scon.flags & FLAG_INCR )
    {
/* if increment */
if( scon.objend == op )
        {
i = *op = 1;
scon.objend++;
        }
else
        {
i = *op + 1;
        }
*op = i;
    }
else
{                              /* non incremental */
i = *op;
    }

scon.objend = op + 1;
if( i > PS_HIST )
    {
op[ -1 ]++;
i = *op = 1;
    }

switch( op[ -1 ] )
    {
case 1: /* Volt */
snmp_leaf( S_INT  + S_RAM, &PS.hist[ i ].volt );
break;
case 2: /* Amp */
snmp_leaf( S_INT  + S_RAM, &PS.hist[ i ].amp );
break;
case 3: /* Temp */
snmp_leaf( S_INT + S_RAM, &PS.hist[ i ].temp );
break;
case 4: /* TimeStamp */
snmp_leaf( S_TTICK + S_RAM  + S_LONG,
&PS.hist[ i ].tstamp );
break;
default:
return S_noSuchName;
    }

return 0;
}
```

Utility Programs

cbxfilt

Program Description

The goal of **cbxfilt.exe** is to save memory, storing strings only once.

cbxfilt.exe extracts text lines from the source file(s) and creates two new files `texte.asm` and `texte.h`, which are included into the project.

First the program searches for lines containing **putcstn()** or **putcstr().**

```
putcstn(T_TXT1/*‟Demo project‟*/)
```

Then **cbxfilt.exe** creates a table with label (T_TXT1) and corresponding string (Demo project). If the same string is used next time, you only need to write the label and the corresponding text will be printed:

```
putcstn(T_TXT1)
```

Command Syntax

cbxfilt file1 [file2 file3…]

r2h

Program Description

r2h Converts CoBox .ROM files into Intel hex format .HEX.

Command Syntax

r2h romfile

e2i

Program Description

e2i converts linker output <Project name>.EXE into .ROM file <Project name>.ROM.

Command Syntax

e2i Name Type Target Version

Name: Project name
Type: TFTP password
Target: cbx, uds, ec1, xpt
Version: version filename

Example:

e2i demo 3Q ec1 myVers

Web2CoB

Program Description

Web2CoB is a command line utility that collects files from a given directory and puts it into one COBOX.COB file. After uploading the COBOX.COB file to a CoBox via TFTP into memory areas WEB0…WEB6 it can be used as a CoBox web server directory.

WEB0 is located in RAM and loose its contents after a reset. WEB1…WEB6 is stored in the flash memory.

When an http request occurs the memory areas are checked for the file name starting in WEB0. The first matching file will then be send back.

Command Syntax

```
Web2CoB [/o <output file>] [/d <directory>]
```

Output file: Optional parameter for output file name. Default file name is `cobox.cob`.

Directory: Optional parameter for source directory. Default is the current directory.

Structure of .COB File

Entry	Length [Bytes]	Remarks
Magic	4	Magic is always "CoB1"
File 1 directory entry:		
Length of file name (1)	1	0 = end of directory, nothing follows. ⇒ max. length of file name is 255 chars
File length (1)	2	Max. file length is 64 Kbytes
File Start Position (1)	4	Relatively to start of .COB file
File name (1)	Depends on length entry (1)	Contains full path name in valid web syntax following http://<server address>/ e.g.: pic/hires/ltx_logo.jpg
File 2 directory entry:		
Length of file name (2)	1	
File length (2)	2	
File Start Position (2)	4	
File name (2)	Depends on length entry (2)	
…	…	Repeat until last directory entry
File (1)	Depends on length entry (1)	
File (2)	Depends on length entry (2)	
…	…	Repeat until last file entry

Demo Sample Programs

Introduction

Several sample programs are provided as part of this kit. The samples are to provide examples of commonly required functionality.

In each demo project, there are several source files.

Main.c is a common block of all demos. Main.c contains all needed procedures for initializing so-called "process" that is a main feature of CoBox's operation system. The WebProcess() is launched from main.c for those demos requiring web services.

Demo.c is where most of the functionality changes take place.

Setpar.c is where changes to *setup menu* are made.

Tools.c is a collection of *nice to have* utilities, and is common across all demos. By default these utilities are undefined with #ifdef statements.

Below is a description of each demo's functionality.

Demo 1

In this version, the program is only a so-called "hello world" program that demonstrates a very basic functionality. In this case, it is a simple template of the necessary infinite loop with the **nice ()** function.

Demo 2

In this version, the program begins the exchange of UDP packets in a classic serial tunnel (port number 1234).

Demo 3

In this version, the program adds the use of the setup menu, and setup data array to hold remote socket information.

Demo 4

In this version, the program adds the exchange of TCP packets to the above demo project.

Demo 5

In this version, the program joins together the two FIFO for simple data handling.

Demo 6

In this version, the program adds the ability to use passive or active TCP connections (listen or connect).

Demo 7

In this version, the program adds DNS resolver functionality.

Demo 8

In this version, the program changes the serial port reading for non-buffered IO operations.

Demo 9

In this version, adds TwoFish encryption to LTX, DLX and U200.

Demo 10

This is a complete program change and only supports the Xport and WiPort. This demo is used to control the configurable pins of the Xport & WiPort. This demo includes a Java based applet, which can be used to manipulate the pins.

Demo 11

This program is an example of implementing a private SNMP MIB.

Demo 12

This program is an example of implementing a cgi callback through the HTTP server. To use this demo, you must tftp the cobox.cob file to WEB1. Then, use a browser to connect to the CoBox… http://<ip address>/test.html. After submitting your query, you'll have 10 seconds to input serial data as the response. (A loopback connector would work.)

Browsing to http://<ip address>/testjs.html is an example of using Java Script in an application. In this example, the serial response is placed into a Java Script variable.

Demo13

This program is an example of implementing SMTP, which is a basic way to send mail.

Demo14

This program is an example of implementing Rijndael encryption. This demo is for encrypted Xports, SDS and Micro-100s only, and may not be in all kits.

Demo15

This program is an example of implementing a SNTP client. This is a simple way of receiving network time. NOTE: this demo has not had extensive testing performed.

Demo16

This program is an additional example of implementing a SNMP private MIB. NOTE: this demo has not had extensive testing performed.

Ethernet Frame Handling

Inbound Frame Processing

Upon reception of Ethernet frames, CoBOS removes the frames from the Ethernet ring buffer, and places them into a section of RAM reserved for incoming frames. When the IP Process task is in the run state, it inspects the buffer for inbound frames and handles them according to the Ethernet type field.

Under normal processing, the Ethernet type field is tested for IP or ARP. If neither condition is true, the packet is passed to a default packet handler (pkt_default() – if it's defined).

ARP Handler

If this is an ARP packet, CoBOS will act on it locally. If it's not a locally handled packet it is passed to a default arp handler (pkt_defarp() – if defined).

IP Handler

IP packets have a larger processing procedure. If the packet is IP addressed to the CoBox (or a broadcast or multicast), the packet continues processing; otherwise it is passed to the default IP handler (pkt_defip() – if defined).

Only three types of IP packets are handled: ICMP, UDP and TCP. Processing within these sections is controlled by the global parameter **tc_para**.

ICMP Handler

If the packet is ICMP, and (tc_para & 2) is true, call pkt_defip(), otherwise handle the packet locally. After local processing of ICMP, if (tc_para & 8) is true, then call pkt_defip().

UDP Handler

If the packet is UDP, and a handler is registered (udp_register()), call the handler. Then if (tc_para & 1) is true and it's a broadcast, multicast or no port handler is registered call pkt_defip().

TCP Handler

If the packet is TCP and (tc_para & 4) is true, call pkt_defip(), otherwise handle the packet locally.

After processing ICMP, UDP or TCP, if the packet is a broadcast or multicast, call pkt_defip(), if it was not previously called.

If pkt_defip or pkt_defarp are not set, pkt_default() is called in it place - if it's set.

Finally, pkt_bridge() is called for all packets if set.

Packet Handler Syntax

Hooks into the network stack are available via the use of four functions and one control switch. These functions are normally NULL, but if defined, will be called by the network stack at different points during processing of the Ethernet frame. Each Ethernet frame is preceeded by a length field (see e_hdr_t structure below). Each packet handler is called with a pointer to specific data within the frame.

```
typedef struct e_hdr_t {
    int len;            /* length */
    BYTE to[6];         /* to address */
    BYTE from[6];       /* from address */
    WORD type;          /* ethernet type, 0x800-IP 0x806-ARP */
} e_hdr_t;


void *pkt_default(BYTE *rb)
    rb - pointer to e_hdr_t structure

void *pkt_bridge(BYTE *rb)
    rb - pointer to e_hdr_t structure

void *pkt_defarp(BYTE *rb)
    rb - pointer to the "to" field of the e_hdr_t structure

void *pkt_defip(BYTE *rb, WORD b)
    rb - pointer to IP header in the Ethernet frame.
    b  - broadcast flag
        1 - broadcast
        2 – multicast
```

To use the bridge functionality for example:

```
extern void (*pkt_bridge)();
void bridge_default(BYTE *rb);

demo()
{
     pkt_bridge = bridge_default;
     …
}

void bridge_default(BYTE *rb)
{
     /* Insert your handler code here */
}
```

Outbound Frame Processing

Overview

You can send raw Ethernet frames by passing a complete frame to the Ethernet controller. To perform this functionality correctly, a three step process is required. Your program should allocate the buffer, send the buffer, and then free the buffer.

GetSendBuf

Description: This function return a pointer to a transmit buffer. This function will block until one is available.

Location: `<platform specific>.lib`

Prototype: `nethw.h`

Syntax: BYTE *GetSendBuf(void);

Parameter: None

Return value: Returns a pointer to a transmit Ethernet buffer.

FreeSendBuf

Description: This function frees reference to the transmit buffer.

Location: `<platform specific>.lib`

Prototype: `nethw.h`

Syntax: void FreeSendBuf(BYTE *buf);

Parameter: Buf – ponter to transmit buffer returned by GetSendBuf()

Return value: None

_send_block

Description: Lists a buffer available to the Ethernet controller for transmit.

Location: `<platform specific>.lib`

Prototype:

Syntax: int _send_block(BYTE *buf, WORD len, WORD interface);

Parameter: buf – ponter to transmit buffer returned by GetSendBuf()
len – number of bytes to transmit
interface – always 0

Return value: Always 0

`For example:`

```
{
    BYTE *buf;

    buf = GetSendBuf();
    …                /* Store bytes into the buffer */
    _send_block(buf,  length, 0);
    FreeSendBuf(buf);
}
```

DSTni Chipset Loading

Introduction

Lantronix uses both the DSTni-LX and the DSTni-EX chipsets in some of its products. The on-chip boot loaders are different. Both loaders will attempt to find a valid bootable image from the serial port, parallel flash and SPI interface (serial flash), in that order. The DSTni-EX will also attempt to boot over the network using a BOOTP / TFTP sequence. The first valid image found will be loaded.

The DSTni-LX will inspect the serial port at 115200, 8, N, 1 for the serial download signature. If the signature is found, the serial binary file is loaded directly to segment 0x0008. The DSTni-EX performs the same way, except inspects the port at 57600.

Parallel flash is inspected on 64KB boundaries starting at segment address 0xFF00. If a valid image is found, the header describes the size, load location and entry point of the image (checksum validation is performed).

Serial flash is inspected at page 5 for a valid header. The header describes the size, load location and entry point of the image (checksum validation is performed).

A network boot image must also contain a valid header as above.

CoBOS Loading

Serial Flash Devices

XPort, Micro-100 (CPK580) and earlier than V6

Serial flash page 5 contains a small "intermediate loader". The main 64KB CoBOS image is stored at page 6+. A damaged intermediate loader (checksum error of either the intermediate loader or the CoBOS image) will cause the DSTni-EX to attempt a network boot. A DSTni-LX will spin on the serial port for download.

Load Process:

1. The on-chip loader copies the image described in the header to the load location described in the header (segment 0x0008 for XPort and Micro-100).
2. On-chip loader turns over control to the copied image (intermediate loader).
3. The "intermediate loader" copies the CoBOS image to RAM3.
4. The "intermediate loader" turns over control to CoBOS in RAM3.

XPort, Micro-100, (CPK6100) and V6 or later

Serial flash page 5 & 6 hold the small "intermediate loader". The first 64KB of the CoBOS image is stored at page 7. A damaged header or checksum error of the intermediate loader image will cause the DSTni-EX to attempt a network boot. A DSTni-LX will spin on the serial port for download.

Load Process:

1. The on-chip loader copies the image described in the header to the load location described in the header (segment 0x0008 for XPort and Micro-100).
2. On-chip loader turns over control to the copied image (intermediate loader).
3. The "intermediate loader" inspects page 7 for a valid CoBOS image and copies the first 64KB from page 7 to RAM3.
4. The "intermediate loader" will then copy any remaining bytes from page 308 to RAM2.
5. The "intermediate loader" turns over control to CoBOS in RAM3.

If the intermediate loader load is valid and no valid CoBOS image exists, the intermediate loader will invalidate itself and reboot.

Parallel Flash Devices

WiPort, WiBox, UDS-100, SDS, Xpress-DR (CPK580) and earlier than V6

Parallel flash segment 0xFF00 holds the header and a small "intermediate loader". The main 64KB CoBOS image is stored at segment 0x8000. A damaged header or checksum error of the intermediate loader will cause the DSTni-EX to attempt a network boot (wired interface only). A DSTni-LX (Xpress-DR, SDS or UDS-100) will spin on the serial port for download.

Load Process:

1. The on-chip loader turns over control to the "intermediate loader" image described in the header.
2. The "intermediate loader" searches for a valid CoBOS image in each 64KB flash segment starting at 0xFE00 and ending at 0x8000 (working down).
3. The "intermediate loader" turns over control to the first valid CoBOS image found.

WiPort, WiBox, UDS-100, SDS, Xpress-DR (CPK6100) and V6 or later

The boot and load process is the same as describe for the V5.8.0.1 devices above.

V6+ however uses two banks of six 64KB pages for CoBOS image storage. These two banks are known as the "executing bank" and the "upgrade bank".

The first bank can start on any 1MB boundary, and the second bank is located six 64KB pages above it. WEB 1 is located six 64KB pages above bank 2.

For example:

Bank 1 at 0xE00000 (segment 0xE000)
Bank 2 at 0xE60000 (segment 0xE600)
WEB1 at 0xEC0000 (segment 0xEC00)

When the executing bank is assigned to bank 1, bank 2 is the upgrade bank. When bank 2 is the executing bank, bank 1 is the upgrade bank. TFTP upgrades write directly to the flash upgrade bank. After successful checksum and flashing of the upgrade image, the V6+ device will invalidate the CoBOS image in the executing bank and reboot.

Notes:

1. Device Installer's firmware recovery procedure of WiPort W4 or WiBox W3 will erase twelve 64KB pages of flash (both banks) effectively erasing segments 0x8000 – 0x8B00 inclusive.
2. It is possible to flash a 64KB executable image into the WEB areas. Once this is done, that image will be executed upon reboot since it will be found first. This will cause the logical flash layout to be different than documented. The ONLY recovery method is to erase the parallel flash.
3. It is also possible to AU flash a V5.8 image on top of V6. The safest method would be to append the radio firmware to the rom image before the flashing. The new V5.8 rom image would be stored in either segment 0x8000 or 0x8600 which "could" cause a different logical flash layout.
4. Changing to or from V6+ will require a reload of the Web Pages also.

CoBOS Standard UDP Handlers

These are the standard functions we associate with the listed UDP port numbers.

7	UDP_Echo
68	dhcpr
69	tf_recv
161	snmp_input
1023	dns_reply
0x77F0	GPIOUDPrec
0x77FE	fw_recv