EICSS

Embedded Systems

SK336 User Guide

SK336 User Guide

Revision 1.5

Table of Contents

Chapter

1

Introduction

SK336 is a portable and highly optimized framework for networked embedded devices. Main components of the framework are:

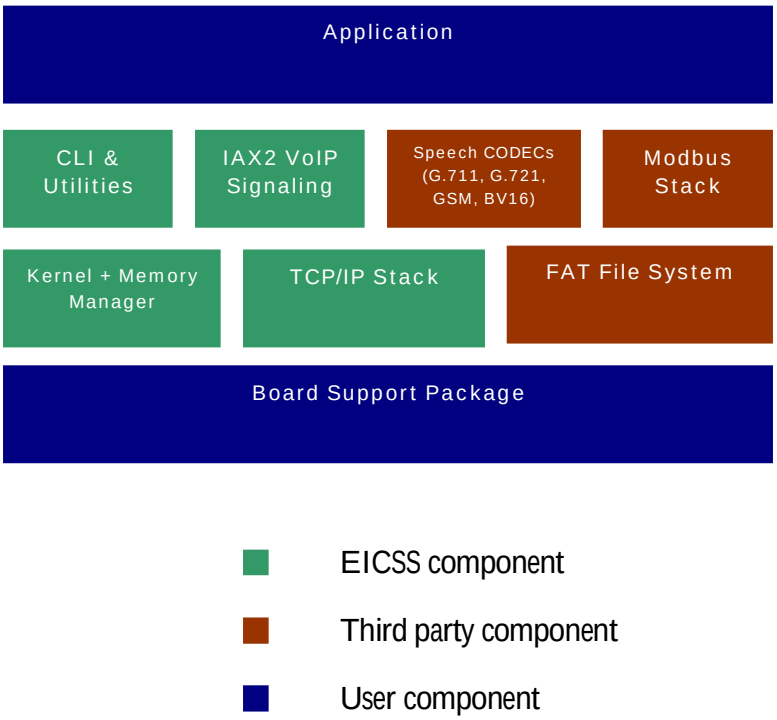
- An innovative kernel
- IPv4 stack
- An optimized implementation of IAX2 signaling protocol
- A comprehensive command-line interface.
- Open-source FAT file system
- Open-source Modbus-TCP stack
- Open-source speech CODECs

Objectives

This document serves as a reference for developers using SK336 embedded development framework for their embedded applications. It provides an insight into directory organization of source base, application examples and API description of different components.

SK336 Architecture

SK336 is highly optimized, modularized and scalable framework as shown in the following figure:



Individual modules can be initialized and used as per application requirements. This allows tailoring the framework for available target memory resources.

Note

Although BSP has been mentioned as a user component, EICSS does provide BSP for selected platforms for evaluation purpose.

Kernel

EICSS has developed a unique mechanism for multi-threading that allows more control over thread execution by designing it as a state machine. Each thread in reality is a regular function. The co-operative multi-threading assumes that a thread completes a step of its processing and yields to other threads by just returning from it. This mechanism has two major advantages over conventional multitasking:

- There is one and only one stack in the system and all threads share that stack space, thus reducing run-time data memory requirements

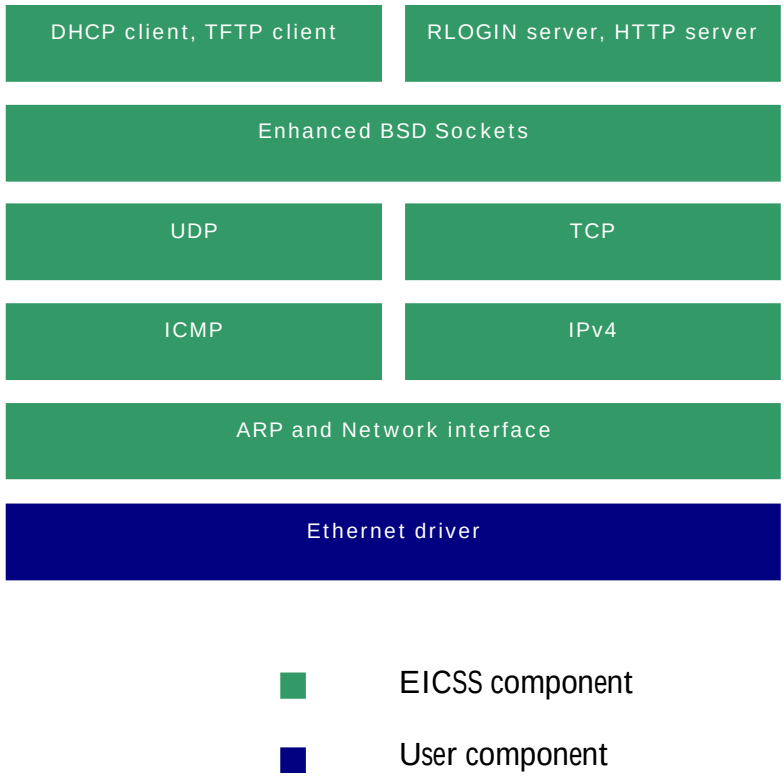
- No protection mechanism is required to access shared resources as threads cannot preempt each other

The kernel provides familiar API to manage threads and inter-thread communication. Threads can delay for desired amount of time or wait for events. All such features result in a highly efficient (and in some more cases, more efficient than conventional real-time operating systems) yet simple and compact multi-threading kernel. Appendix A details complete kernel API including memory manager API.

IPv4 Stack

This is a minimal implementation of IPv4 with emphasis on reducing memory footprint and increasing throughput by avoiding memory copy operations. The stack provides a modified version of subset of BSD style socket API. The modifications allow more control over data flow by allowing API user to use and send data packets through the stack that are passed on to the physical Ethernet interface without any copy operation. Apart from TCP/IP data interface, API is provided to initialize and start networking module. For complete API, please refer to Appendix B.

Following diagram provides an overview of currently implemented components of IPv4 stack.



IAX2 VoIP Signaling

This is an extremely efficient implementation of open-source signaling protocol. It provides comprehensive API to initiate and respond to voice calls and depends on exported functions within the application and BSP for raw voice data and asynchronous event notification. Please refer to Appendix C for API details.

CLI and Utilities

This module provides command line interface over RS232 serial port. The same interface is available remotely via Rlogin (a Telnet implementation is in progress). This module is discussed in detail in Chapter 4. For API details, please refer to Appendix D

Application

In addition to system initialization, application module implements all application specific functionality such as user-interface, application level networking modules etc. These modules can use all services provided by EICSS framework as described above. Samples of initialization module can be found in demo applications.

Board Support Package

SK336 for a particular architecture and platform is shipped complete with basic system level drivers including:

- System timer driver
- UART driver
- EEPROM driver (optional)
- Data FLASH driver (optional)

All user-accessible drivers assume single instance of each device and provide a procedural interface which apparently is a limitation but accompanied device driver manager module allows writing drivers for devices with multiple physical instances on target hardware.

Other peripheral drivers can be developed based on a service agreement or can be added by the application developers. Please refer to chapter 3 for architecture and implementation of low-level Ethernet driver.

Third-party Components

These are highly efficient open-source components and fully integrated with EICSS framework. Most of these components can be used in commercial applications but user should be aware of any patent related issues.

Chapter

2

Writing New Application

Writing new application for target platform either requires framework library for specific platform or framework source code. In the later case, the user of the framework has to write BSP for the target platform.

EICSS demo code can be used as template for writing new application. An application may be organized as multiple source code files but at least one file is required to export `usrMain()` function that gets called after platform specific initializations are complete. Assuming the file name to be `app.c`, following section describe the contents of the file step-by-step.

Including header files

Some basic header files have to be included in the main source file (and all other application level source files) any way:

```
#include "common.h"
#include "board.h"
#include "osmap.h"
```

Following include is required if you plan to use CLI module:

```
#include "cli.h"
```

Following includes are required if you plan to use TCP/IP stack:

```
#include "socket.h"
#include "ethdrv.h"
```

Following include is required for system configuration module:

```
#include "confall.h"
```

Following include is required only for file system module:

```
#include "fs.h"
```

Initializing the system

Before creating user tasks, system should be initialized and brought to stable state as shown in following sequence of function calls:

```
boardInit();
osInit();
```

Following function calls are required only if FLASH based FAT file system is required:

```
flashOpen();
fsInit();
```

Configure the system. This usually means loading configuration information from non-volatile memory and parsing it to system-wide data structures:

```
configInit();
```

Initialize and start networking sub-system:

```
ethdrv.openFunc = EthDevOpen;
ethdrv.closeFunc = EthDevClose;
ethdrv.readFunc = EthDevRead;
ethdrv.writeFunc = EthDevWrite;
ethdrv.controlFunc = EthDevControl;
networkInit(&ethdrv);
networkStart();
```

Initialize CLI:

```
cliInit();
```

At this stage, system is in stable state and user-specific modules can be initialized. After user-level initialization is complete, following system call takes the system to a stable running state:

```
osSchedule();
```

This function should be the last function to be called in sequence as it never returns. All these function calls should be invoked from within following function:

```
void
usrMain(
    void
)
{
    /* All initializations go here! */
}
```

Creating a user-module

For the sake of modularity and clarity, best coding practices should be adapted. It is advisable to create each user module as a separate source code file. The file should export an initialization function that should be called in `usrMain()`.

Since a user-module may create one or more threads, following section describes how to create and start a thread.

Creating and using threads

A thread is an independent entity that needs to run periodically or on-demand to carry out a particular task. This can include reading data from a sensor, turning on/off a device at pre-defined interval or waiting for an event from another module. The kernel module in sk336 provides a comprehensive API to manage threads and inter-thread communication.

Note

The kernel is a non-preemptive kernel and thus requires that each thread release the CPU as soon as it is done with its execution. This essentially means code in a thread be designed and written as an FSM.

Following snippet of code shows how a thread can be created and started:

```
UInt32 targs[4];
UInt32 myThreadID;
UInt32 err;

targs[0] = targs[0];

err = osTaskCreate("MY", MY_THREAD_PRIO, MY_THREAD_STK_SIZE,
MY_THREAD_STK_SIZE, 0, myThread, (void *) Null, &myThreadID);
if(err)
{
    osTaskStart(myThreadID, osTaskStartFlagsStandard, targs);
}
```

Once a thread is created and started, the scheduler starts scheduling it if the thread is not intentionally delaying its own execution or is not waiting for an event from another thread (via message queue). Referring to above code snippet code, the thread `myThread` delays its own execution for 10 system ticks.

```
void myThread( void *parameter )
{

    START_LOOP

        /* Do something */

        /* Waiting for 10 system ticks */
        osTimeDelay(10);

    END_LOOP
}
```

In the above example, `myThread` carries out a specific task and then delays for 10 system ticks making is get scheduled again after 10 system ticks.

In the next example, the same task waits for arrival of a message on a message queue.

```
void myThread(void *parameter)
{
    UInt32 msg_buf[1];

    START_LOOP

        /* Do something */

        /* Wait with infinite timeout for a message */
        err = osQueueReceive(evQueue, 0, 0, msg_buf);
        if(!err)
        {
            /* Message received – do something */
        }

    END_LOOP
}
```

In this example, the thread waits infinitely for a message by specifying infinite timeout when invoking `osQueueReceive` call. This puts the thread to sleep and wakes it up when a message is posted on the queue from another entity in the system.

Chapter

3

Extending Board Support Package

Writing Ethernet driver

Ethernet driver has to export five function calls. These calls are provided to networking module initialization function during system initialization as discussed earlier. For sake of simplicity, we discuss internals of accompanied template driver.

Including header files

Apart from including platform specific header files, following includes are required:

```
#include "common.h"
#include "osmap.h"
#include "config.h"
#include "socket.h"
#include "usrconf.h"
#include "mman.h"
#include "ethdrv.h"
#include "utils.h"
```

Data structures

A data structure needs to declared and defined to hold data private to the driver:

```
typedef struct ethdrv_inst_vars_t {
    UInt16 state;
    UInt32 instance;
    UInt16 base;
    UInt32 rx_thread_id;
    UInt32 tx_thread_id;
    UInt16 rx_errors;
    UInt16 rx_length_errors;
    UInt16 rx_dropped;
    UInt16 rx_packets;
    UInt16 rx_bytes;
    UInt16 rx_fifo_errors;
    UInt16 rx_over_errors;
    UInt16 multicast;
```

```
    Eaddr  hwAddress;
    dCallbackSetup_t callbackSetup;
} EthDrvInstVars_t, *pEthDrvInstVars_t;
```

EthDevOpen

Open function is responsible for creating an instance of the driver:

```
UInt32
EthDevOpen(
   UInt32* instance,
   UInt16 unit
)
{
    pEthDrvInstVars_t ivp;

    ivp = (pEthDrvInstVars_t) Malloc(sizeof(EthDrvInstVars_t));

    if(ivp == Null) {
        return ETHDRV_NOTENOUGHMEM;
    }

    memset(ivp, 0x0, sizeof(EthDrvInstVars_t));

    *instance = (UInt32) ivp;

    return ERR_OK;
}
```

EthDrvClose

Closes specified instance of the driver. In most embedded applications, the devices are not closed and this call is usually a stub:

```
UInt32
EthDevClose(
    UInt32 instance,
    UInt16 unit
)
{
    UInt32 err = ERR_OK;
    pEthDrvInstVars_t ivp;

    ivp = (pEthDrvInstVars_t) instance;

    return err;
}
```

EthDrvControl

This call performs all operations other than data transfer. These include initializing the device, starting/stopping the device and creating threads/ISRs to handle data I/O etc:

```
UInt32
```

```
EthDevControl(
    UInt32 instance,
    UInt16 unit,
    pdControlArgs_t args
)
{
    pEthDrvInstVars_t ivp;
    UInt32 targs[4];
    UInt32 err;
    pdCallbackSetup_t dcbSetup;

    targs[0] = targs[0];

    ivp = (pEthDrvInstVars_t) instance;

    switch(args->command) {
    case DD_INIT:
        ivp->instance = instance;

        lEthDevReset(ivp, True);

        err = osTaskCreate("NRX", RX_PRIO, RX_STACK_SIZE,
RX_STACK_SIZE, 0, rxThread, (void *) ivp, &ivp->rx_thread_id);
        if(err) {
            return err;
        }

        err = osTaskCreate("NTX", TX_PRIO, TX_STACK_SIZE,
TX_STACK_SIZE, 0, txThread, (void *) ivp, &ivp->tx_thread_id);
        if(err) {
            return err;
        }

        ivp->state = nisUp;
        break;
    case DD_INSTALL_CALLBACK:
        dcbSetup = (pdCallbackSetup_t) args->params;
        ivp->callbackSetup.callbackContext = dcbSetup->callbackContext;
/* Context to use when calling callbacks */
        ivp->callbackSetup.rxPacketReport  = dcbSetup->rxPacketReport;
        ivp->callbackSetup.txPacketDemand  = dcbSetup->txPacketDemand;
        break;
    case DD_START:
        osTaskStart(ivp->rx_thread_id, 0, targs);
        osTaskStart(ivp->tx_thread_id, 0, targs);
        break;
    case DD_STOP:
        osTaskSuspend(ivp->rx_thread_id);
        osTaskSuspend(ivp->tx_thread_id);
        break;
    default:
        break;
    }

    return ERR_OK;
}
```

EthDevRead

This is functional interface for reading data from device and should be implemented as a stub as functional interface is not used by upper layers.

```
UInt32
EthDevRead(
    UInt32 instance,
    UInt16 unit,
    void* data
)
{
    return ERR_OK;
}
```

EthDevWrite

This is functional interface for writing data to device and should be implemented as a stub as functional interface is not used by upper layers.

```
UInt32
EthDevWrite(
    UInt32 instance,
    UInt16 unit,
    void* data
)
{
    return ERR_OK;
}
```

Transferring data to/from upper layer

Real data transfer from/to upper layer takes place in receive and transmit threads (in polled-mode drivers) or ISRs (in asynchronous mode drivers). In polled mode, for example, `rxThread` periodically polls device registers, reads data from device RX FIFO when data is available, packetizes it and sends it over to upper layer:

```
static void
rxThread(
    void *pdata
)
{
    pEthDrvInstVars_t ivp;
    UInt32 err;
    pEthernetPacket_t ep;
    void *packet;

    ivp = (pEthDrvInstVars_t) pdata;

    START_LOOP

        if(isRxDataAvailable())
        {
            /* Create new packet */
```

```
            err = packetCreate(&packet, L_PACKET);
            if(err)
            {
                ivp->rx_dropped++;
            }
            else
            {
                ep = (pEthernetPacket_t)
netGetPacketPayloadPtr(packet);
                /* Fill-in the packet with received data */
                ep->size = readData(&ep->header, 1518);
                /* Report packet to upper layer */
                err = ivp->callbackSetup.rxPacketReport(ivp-
>callbackSetup.callbackContext, packet);
                if(err)
                {
                    packetDestroy(packet);
                }
                ivp->rx_packets++;
                ivp->rx_bytes += ep->size;
            }
        }

    END_LOOP
}
```

Similarly, `txThread` demands data from upper layer, checks for enough space in TX FIFO and writes packet data to the FIFO to send out the packet:

```
static void
txThread(
    void *pdata
)
{
    pEthDrvInstVars_t ivp;
    void* packet;
    UInt32 err;

    ivp = (pEthDrvInstVars_t) pdata;

    START_LOOP
        /* Get packet from upper layer */
        err = ivp->callbackSetup.txPacketDemand(ivp-
>callbackSetup.callbackContext, &packet);
        if(err) {
            if(packet) {
                packetDestroy(packet);
            }
            CONTINUE;
        }

        /* Send packet */
        lEthDevWrite((UInt32) ivp, 0, packet);

        /* Release packet */
```

```
        packetDestroy(packet);
    END_LOOP
}
```

Chapter

4

CLI

CLI module provides interactive command processing over serial interface. These commands can be divided into two categories:

- System level commands
- Application level commands

System level commands

System level commands are already implemented within framework and facilitate system configuration and monitoring. Following is a list of all system level commands:

```
[Network]
ifconfig [static/dhcp] <ip> <subnet mask> <gateway> <dns server>
setmac <12 digit hex string>

[File System]
mkfs admin
get <ip> <file name>
ren <current name> <new name>
ls
rm <file name>
cat <file name>

[Statistics]
netstat
qstat
tstat

[Others]
quit/logout
```

Application level commands

These commands are application specific and are supposed to be implemented in application module in an exported function named `processAppCommand`:

```
void
processAppCommand(
    Int argc,
```

```
    char argv[MAX_PARAMS][MAX_TOKEN_SIZE]
)
{
    UInt8 writeback = False;

    if(argc)
    {
        if(strcmp(argv[0], "help") == 0)
        {
        }
        else if(strcmp(argv[0], "copyrights") == 0)
        {
            printf("EICSS\r\n");
        }
        else if(strcmp(argv[0], "quit") == 0)
        {
            exit(0);
        }
    }

    /* Write current config to EEPROM/FLASH */
    if(writeback == True)
    {
        /* Gather all three type of framework config
         * to application config and update
         * FLASH/EEPROM
         */
        configSave();
    }
}
```

CLI module calls the function and passes command and command arguments as a list of strings and total number of arguments (including the command itself).

Note

If CLI feature is not required, CLI module shouldn't be initialized during system initialization and empty stub should be exported as application command processor.

Appendix A

Kernel API Reference

osInit

Initialize kernel to known state including allocation of resources. This call also initializes memory manager.

Include

```
osmap.h
```

Prototype

```
UInt32 osInit(void)
```

Parameters

```
None
```

Returns

```
ERR_OK

OS_ERR_NOTENOUGHMEM
```

osExit

Halt application execution. This function should only be called on an irrecoverable error.

Include

```
osmap.h
```

Prototype

```
void osExit(void)
```

Parameters

```
None
```

Returns

```
Never
```

osTaskCreate

Create a new thread. The thread will be created in dormant state and will remain in that state until started explicitly by a osTaskStart/osTaskResume call.

Include

```
osmap.h
```

Prototype

```
UInt32 osTaskCreate(char *name, UInt16 prio, UInt16 sstack, UInt16 ustack,
UInt16 flags, void (*start_addr)(void *pdata), void *udata, UInt32 *tid)
```

Parameters

```
name         I     Thread name

prio         I     Thread priority (valid only if underlying RTOS supports
it)

sstack       I     System stack size in bytes

ustack       I     User stack size in bytes

flags        I     Thread specific flags

start_addr   I     Address of function that gets executed

udata        I     Arguments to pass to thread when executing it

tid          O     Thread ID used to refer to the thread subsequently
```

Returns

```
ERR_OK

OS_ERR_NOTENOUGHMEM
```

osTaskStart

Start the thread. This lets the scheduler keep the thread in execution loop till the thread sleeps explicitly or waits for an event.

Include

```
osmap.h
```

Prototype

```
UInt32 osTaskStart(UInt32 tid, UInt32 mode, UInt32 targs[4])
```

Parameters

```
tid          I     Thread ID

mode         I     Start mode (not used)
```

```
targs       I     Arguments to pass on to the thread when starting (not
used)
```

Returns

```
ERR_OK

OS_ERR_INVALIDTASKID
```

osTaskDestroy

Terminate the thread. The thread is removed from scheduler list of threads until created again using osTaskCreate call.

Include

```
osmap.h
```

Prototype

```
UInt32 osTaskDestroy(UInt32 tid)
```

Parameters

```
tid         I     Thread ID
```

Returns

```
ERR_OK

OS_ERR_INVALIDTASKID
```

osTaskSuspend

Suspend thread execution indefinitely. Thread execution can be resumed by calling osTaskStart or osTaskResume. Please note that if current thread attempts to suspend itself, it looses its execution immediately.

Include

```
osmap.h
```

Prototype

```
UInt32 osTaskDestroy(UInt32 tid)
```

Parameters

```
tid         I     Thread ID
```

Returns

```
ERR_OK
```

```
OS_ERR_INVALIDTASKID
```

osTaskResume

Resume execution of thread.

Include

```
osmap.h
```

Prototype

```
UInt32 osTaskResume(UInt32 tid)
```

Parameters

```
tid          I      Thread ID
```

Returns

```
ERR_OK

OS_ERR_INVALIDTASKID
```

osTimeDelay

Makes calling thread execution delayed for specified number of ticks.

Include

```
osmap.h
```

Prototype

```
void osTimeDelay(UInt32 ticks)
```

Parameters

```
ticks        I      Number of timer ticks to delay for.
```

Returns

```
None
```

osQueueCreate

Create a FIFO message queue.

Include

```
osmap.h
```

Prototype

```
UInt32 osQueueCreate(char *name, UInt16 count, UInt16 flags, UInt32 *qid)
```

Parameters

```
name        I     Queue name

count       I     Queue length

flags       I     Queue specific flags

qid         O     Queue ID used to refer to the queue subsequently
```

Returns

```
ERR_OK

OS_ERR_NOTENOUGHMEM
```

osQueueDestroy

Destroy specified message queue. The queue should be empty before being deleted.

Include

```
osmap.h
```

Prototype

```
UInt32 osQueueDestroy(UInt32 qid)
```

Parameters

```
qid         I     Queue ID
```

Returns

```
ERR_OK

OS_ERR_INVALIDQUEUEID

OS_ERR_QUEUENOTEMPTY
```

osQueueReceive

Get a message pending on the queue in FIFO fashion.

Include

```
osmap.h
```

Prototype

```
UInt32 osQueueReceive(UInt32 qid, UInt32 flags, UInt32 timeout, UInt32
msg_buf[1])
```

Parameters

```
qid          I     Queue ID

flags        I     Message reception specific flags (not used)

timeout      I     Time in timer ticks to wait for a message. A timeout
value 0 means wait forever. All other timeout values are ignored and the
call returns with error

msg_buf      I     Message to receive
```

Returns

```
ERR_OK

OS_ERR_INVALIDQUEUEID

OS_ERR_QUEUEEMPTY

OS_ERR_TIMEOUT
```

osQueueSend

Push a message to specified FIFO queue.

Include

```
osmap.h
```

Prototype

```
UInt32 osQueueSend(UInt32 qid, UInt32 msg_buf[1])
```

Parameters

```
qid          I     Queue ID

msg_buf      I     Message to send. A reference to a larger message can be
sent as pointer to the message
```

Returns

```
ERR_OK

OS_ERR_INVALIDQUEUEID

OS_ERR_QUEUEEFULL

OS_ERR_NOTENOUGHMEM
```

osTimeSet

Set current system date and time. The time will be advanced by an RTC device (if available) or by RTC emulation.

Include

```
osmap.h
```

Prototype

```
UInt32 osTimeSet(pTimeInfo_t time)
```

Parameters

```
time        I     Pointer to data structure holding date and time-of-day
values
```

Returns

None

osTimeGet

Get current system date and time. If hardware doesn't provide RTC device and RTC emulation is not enabled either, values in the data structure may be invalid.

Include

```
osmap.h
```

Prototype

```
UInt32 osTimeSet(pTimeInfo_t time)
```

Parameters

```
time        I     Pointer to data structure holding date and time-of-day
values
```

Returns

```
None
```

osTicksPerSecond

Get system timer tick rate.

Include

```
osmap.h
```

Prototype

```
UInt32 osTicksPerSecond(void)
```

Parameters

```
None
```

Returns

System timer tick rate

osSchedule

Start multi-threading. This should be the last call after initializing all system resources as it never returns.

Include

osmap.h

Prototype

```
void osSchedule(void)
```

Parameters

None

Returns

None

osYield

Let other threads get chance to execute without leaving current thread context.

Include

osmap.h

Prototype

```
void osYield(void)
```

Parameters

None

Returns

None

Memory Manager API

Malloc

Allocate specified number of bytes from heap. The memory allocated with this call is supposed to be held by the caller for lifetime of application and cannot be returned to heap by any means.

Include

mman.h

Prototype

```
void *Malloc(UInt32 size)
```

Parameters

```
size        I     Chunk size in bytes
```

Returns

```
Valid pointer to allocated chunk on success

NULL on failure
```

packetCreate

Allocate a data packet from specified memory pool. The data packets are supposed to be used for zero-copy transfer of data to/from network interface by application.

Include

```
mman.h
```

Prototype

```
UInt32 packetCreate(void **packet, UInt16 type)
```

Parameters

```
packet      I/O   Pointer to place pointer to allocated packet in. Points
to NULL on failure to allocate packet of specified type.

type        I     Packet type. Valid pool types are S_PACKET (for small
packet) and L_PACKET (for large packet). Actual packet sizes are
application specific.
```

Returns

```
ERR_OK on success

An error code other than ERR_OK on failure. In most cases, checking for
NULL value in *packet should be enough.
```

Note

```
Pointer to packet returned by this call should not be manipulated directly.
Please see networking API for description of appropriate calls that
translate packet pointer to desired packet payload pointer.
```

packetDestroy

De-allocate specified data packet. The packet is returned to corresponding memory pool and available for subsequent allocation.

Include

mman.h

Prototype

```
void packetDestroy(void *packet)
```

Parameters

packet	I	Pointer to the packet being de-allocated

Returns

None

Note

An attempt to de-allocate an invalid packet pointer results in system shutdown indicating an irrecoverable error.

Appendix B

Networking API

networkInit

Initialize all components of IPv4 stack and start it. Some of application level components are also started including DHCP client, DNS client, HTTP server and RLOGIN server.

Include

```
socket.h
```

Prototype

```
UInt32 networkInit(pEthDrv_t ethdrv)
```

Parameters

```
ethdrv      I     Pointer to data structure holding pointers to exported
functions of low level networking driver
```

Returns

```
ERR_OK on success

A number of error code other than ERR_OK on failure
```

netGetIP

Get current interface IP (whether static or DHCP acquired).

Include

```
socket.h
```

Prototype

```
void netGetIP(IPaddr ipAddress)
```

Parameters

```
ipAddress    I/O    Octet array holding current IP address
```

Returns

```
None
```

netGetSubnetMask

Get current interface subnet mask.

Include

```
socket.h
```

Prototype

```
void netGetSubnetMask(IPaddr subnetMask)
```

Parameters

```
subnetMask   I/O   Octet array holding current subnet mask
```

Returns

```
None
```

netGetGateway

Get current gateway IP address.

Include

```
socket.h
```

Prototype

```
void netGetGateway(IPaddr gateway)
```

Parameters

```
gateway      I/O   Octet array holding current gateway IP address
```

Returns

```
None
```

netGetServer

Get current server IP address as reported by DHCP. May not be a valid IP address if DHCP is not enabled.

Include

```
socket.h
```

Prototype

```
void netGetServer(IPaddr ipServer)
```

Parameters

```
ipServer     I/O   Octet array holding server IP address
```

Returns

```
None
```

netGetDNS

Get current DNS server IP address

Include

```
socket.h
```

Prototype

```
void netGetDNS(IPaddr dnsServer)
```

Parameters

```
dnsServer    I/O    Octet array holding DNS server IP address
```

Returns

```
None
```

netGetUdpPayloadPtr

Get UDP payload pointer in specified data packet.

Include

```
socket.h
```

Prototype

```
void *netGetUdpPayloadPtr(void *packet)
```

Parameters

```
packet       I      Pointer to data packet
```

Returns

```
Pointer to UDP payload
```

netGetTcpPayloadPtr

Get TCP payload pointer in specified data packet.

Include

```
socket.h
```

Prototype

```
void *netGetTcpPayloadPtr(void *packet)
```

Parameters

```
packet      I     Pointer to data packet
```

Returns

```
Pointer to TCP payload
```

netGetRawPayloadPtr

Get raw IP payload pointer in specified data packet.

Include

```
socket.h
```

Prototype

```
void *netGetRawPayloadPtr(void *packet)
```

Parameters

```
packet      I     Pointer to data packet
```

Returns

```
Pointer to raw IP packet payload
```

netGetPacketPayloadPtr

Get packet payload pointer in specified data packet.

Include

```
socket.h
```

Prototype

```
void *netGetPacketPayloadPtr(void *packet)
```

Parameters

```
packet      I     Pointer to data packet
```

Returns

Pointer to packet payload pointer

Note

This call allows using data packets as general purpose dynamic memory. To avoid memory leak, packet payload and not the packet pointer itself should be manipulated. Payload size can be acquired by calling netGetPacketPayloadSize().

netGetUdpPayloadSize

Get available space in bytes for UDP payload.

Include

```
socket.h
```

Prototype

```
UInt16 netGetUdpPayloadSize(void *packet)
```

Parameters

```
packet      I     Pointer to data packet
```

Returns

```
UDP payload size
```

netGetTcpPayloadPtr

Get available space in bytes for TCP payload

Include

```
socket.h
```

Prototype

```
UInt16 netGetTcpPayloadSize(void *packet)
```

Parameters

```
packet      I     Pointer to data packet
```

Returns

```
TCP payload size
```

netGetRawPayloadSize

Get available space in bytes for raw IP payload.

Include

```
socket.h
```

Prototype

```
UInt16 netGetRawPayloadPtr(void *packet)
```

Parameters

```
packet      I     Pointer to data packet
```

Returns

```
Raw IP packet payload size
```

netGetPacketPayloadSize
Get available space in bytes for general purpose use.

Include

```
socket.h
```

Prototype

```
UInt16 netGetPacketPayloadPtr(void *packet)
```

Parameters

```
packet      I     Pointer to data packet
```

Returns

```
Packet payload size
```

Sockets API

Following is a list of supported BSD style socket calls:

```
socket
bind
listen
connect
sendto
recvfrom
closesocket
inet_addr
inet_ntoa
ioctlsocket
setsockopt
gethostbyname
gethostname
```

To support zero-copy data transmission over TCP/IP stack, socket API has been slightly modified, especially `sockaddr_in` data structure:

```
struct sockaddr_in {
    short   sin_family;
    u_short sin_port;
    u_short reserved;
    struct  in_addr sin_addr;
    void    *dptr;
    char    sin_zero[8];
```

```
};
```

As seen above, `dptr` of type void has been added to this structure. The pointer can be used by the application to pass data packet through the stack without copying data at any point resulting in an extremely efficient data propagation through the stack. When using zero-copy feature, the parameter that would normally specify user data should be set to NULL in calls such as `send()`, `sendto()`, `recv()` and `recvfrom()`. On the other hand, a valid buffer pointer would override zero-copy operation and ignore packet pointer in `dptr` member of `sockaddr_in` data structure.

Appendix C

IAX2 Signaling API

ippInitMessaging

Initializing signaling module to known state.

Include

```
ippapi.h
```

Prototype

```
UInt32 ippInitMessaging(void)
```

Parameters

```
None
```

Returns

```
ERR_OK

OS_ERR_NOTENOUGHMEM

IPP_ERR_NETWORK_RESOURCE

IPP_ERR_RECEIVE_MESSAGE
```

ippStartMessaging

Start signaling module.

Include

```
ippapi.h
```

Prototype

```
UInt32 ippStartMessaging(void)
```

Parameters

```
None
```

Returns

```
ERR_OK

OS_ERR_INVALIDTASKID
```

ippSetMediaFormat

Declare supported media formats. This bitmap shall be used to negotiate best media type to use for an incoming or outgoing call.

Include

```
ippapi.h
```

Prototype

```
void ippSetMediaFormat(UInt32 format)
```

Parameters

```
format      I     Media format bitmap (one format per bit)
```

Returns

```
None
```

ippRegister

Force registration process with all configured servers.

Include

```
ippapi.h
```

Prototype

```
void ippRegister(void)
```

Parameters

```
None
```

Returns

```
None
```

ippStartCall

Start a new outgoing call.

Include

```
ippapi.h
```

Prototype

```
UInt32 ippStartCall(char *ext, UInt32 *context)
```

Parameters

```
context      O     Call context for future call reference

ext          I     Extension to call (ASCII text)
```

Returns

```
ERR_OK

OS_ERR_NOTENOUGHMEM
```

ippStopCall

Terminate an ongoing call.

Include

```
ippapi.h
```

Prototype

```
void ippStopCall(UInt32 context)
```

Parameters

```
context      I     Call context
```

Returns

```
None
```

ippAcceptCall

Accept an incoming call.

Include

```
ippapi.h
```

Prototype

```
UInt32 ippAcceptCall(UInt32 context)
```

Parameters

```
context      I     Call context
```

Returns

```
ERR_OK

OS_ERR_NOTENOUGHMEM
```

ippRejectCall

Reject an incoming call.

Include

```
ippapi.h
```

Prototype

```
void ippRejectCall(UInt32 context)
```

Parameters

```
context      I      Call context
```

Returns

```
ERR_OK

OS_ERR_NOTENOUGHMEM
```

ippHoldCall

Put a current call on hold.

Include

```
ippapi.h
```

Prototype

```
void ippHoldCall(UInt32 context)
```

Parameters

```
context      I      Call context
```

Returns

```
None
```

ippUnholdCall

Take a call out of hold making it currently active call.

Include

```
ippapi.h
```

Prototype

```
void ippUnholdCall(UInt32 context)
```

Parameters

```
context      I      Call context
```

Returns

None

ippSendSMS
Send SMS.

Include

ippapi.h

Prototype

```
UInt32 ippSendSMS(char *ext, char *message)
```

Parameters

ext	I	Extension to send message to
message	I	Message body (ASCII text)

Returns

None

IAX2 Signaling Callbacks

ippEventCB
This callback should be implemented by application to listen to asynchronous events from signaling module.

Include

ippapi.h

Prototype

```
void ippEventCallback(UInt32 event, UInt32 args)
```

Parameters

event	I	Event to report (see below for event details)
args	I	Any arguments related to the event

Returns

None

Note

Following are supported events:

```
IPP_CB_EV_CALL_ESTABLISHED          args = context

IPP_CB_EV_INCOMING_CALL             args = context

IPP_CB_EV_ERROR                     args = context

IPP_CB_EV_BUSY                      args = context

IPP_CB_EV_RINGBACK                  args = context

IPP_CB_EV_HANGUP                    args = context

IPP_CB_EV_CLIENT_REGISTERED         args = none

IPP_CB_EV_CLIENT_DEREGISTERED       args = none

IPP_CB_EV_CALL_DROPPED              args = context

IPP_CB_EV_SMS                       args = Pointer to message
```

codecGetFullFrame

This callback is called to request a PCM coded 16-bit voice buffer that can be compressed and sent out.

Include

```
ippapi.h
```

Prototype

```
void *codecGetFullFrame(void)
```

Parameters

```
None
```

Returns

```
Pointer to buffer holding 16-bits PCM voice samples

NULL if no buffer available
```

codecGetEmptyFrame

This callback is called to request an empty buffer that can be filled in with decompressed 16-bit coded PCM voice data.

Include

```
ippapi.h
```

Prototype

```
void *codecGetEmptyFrame(void)
```

Parameters

None

Returns

Pointer to buffer that can hold 16-bit PCM voice samples

NULL if no buffer available

codecPutFullFrame

This callback is called to report a decompressed 16-bit PCM coded voice buffer.

Include

ippapi.h

Prototype

```
void codecPutFullFrame(void *buffer)
```

Parameters

Buffer	I	Pointer to buffer containing 16-bit PCM voice samples

Returns

None

codecPutEmptyFrame

This callback is called to recycle a voice buffer whose contents have been compressed and transmited.

Include

ippapi.h

Prototype

```
void codecPutEmptyFrame(void *buffer)
```

Parameters

buffer	I	Pointer to buffer to release

Returns

None

Appendix D

CLI and Utilities API

cliInit

Initialize command line interface to know state and start CLI thread.

Include

`cli.h`

Prototype

```
UInt32 cliInit(void)
```

Parameters

`None`

Returns

`ERR_OK`

`OS_ERR_NOT_ENOUGHMEM`

`OS_ERR_INVALIDTASKID`

cliStart

Start CLI module.

Include

`cli.h`

Prototype

```
void cliStart(void)
```

Parameters

`None`

Returns

`None`

cliStop

Stop CLI module.

Include

```
cli.h
```

Prototype

```
void cliStop(void)
```

Parameters

```
None
```

Returns

```
None
```

parseCmdLine

Parse a string and extract individual tokens and total number of tokens. This is a utility function that may be used to parse strings read from other input sources such as files.

Include

```
cli.h
```

Prototype

```
Int parseCmdLine(char *s, char argv[MAX_PARAMS][MAX_TOKEN_SIZE])
```

Parameters

```
s           I     Pointer to string to parse

argv        I/O   Array to hold parsed tokens. MAX_PARAMS number of tokens
each of size MAX_TOKEN_SIZE are allowed.
```

Returns

```
Number of tokens parsed
```

processAppCommand

This callback should be implemented by application if it wishes to process commands other than system commands. A stub is required even if application doesn't wish to process commands.

Include

```
cli.h
```

Prototype

```
void processAppCommand(Int argc, char argv[MAX_PARAMS][MAX_TOKEN_SIZE])
```

Parameters

argc I Number of tokens corresponding to the command

argv I/O Array to hold parsed tokens. MAX_PARAMS number of tokens each of size MAX_TOKEN_SIZE are allowed. First token is the command name.

Returns

Number of tokens parsed