Introduction to

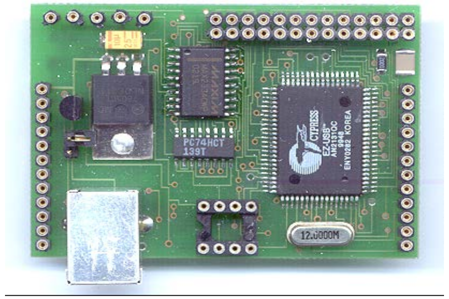
Real-time Operating System for Xilinx ZYNQ SoC

freeRTOS & lwIP & FatFS

Dr. Heinz Rongen
Forschungszentrum Jülich GmbH
Systeme der Elektronik (ZEA-2)
H.Rongen@fz-juelich.de

Motivation for OS

µController vs. PC



Embeeded System (µController, single Chip Solutions)
- Internal Memory: Flash/RAM (program / data memory)
- Internal I/O

- **„Bare-Metal“ Programmierung**
  Without the support of a Operating system

- **Communication**
  Serial RS232 (no Ethernet)

- **Data storage**
  EEPROMs (limited size, no Filesystem)

PC Processor Systems: (Multiple Chip Solutions)
- External Memories
- External I/O‘s

- **Operating systems**
  Windows, embedded Linux, …

- **Communication**
  Full featured Networking / Ethernet

- **Data storage**
  Hard discs, File-System, …

The gap …

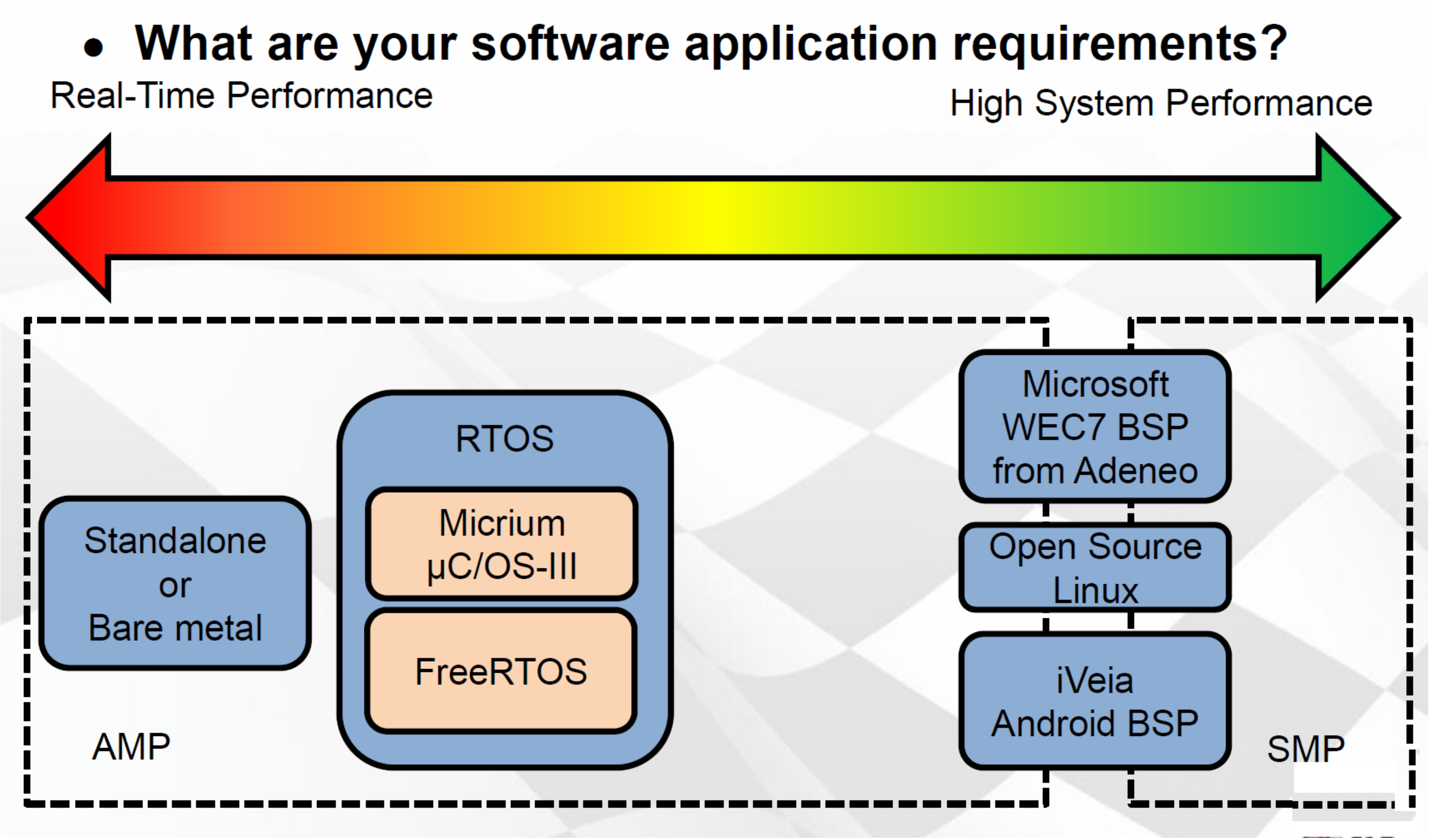
µController
‚Baremetal programming‘

PC
‚Operating system‘ based programming

- Simple „**single Chip**“ Hardware
- As „Building-Block“ for own HW developments

- **todays µC are powerful** (ARM processor@1GHz)
- **OS Support**
  - Multi-Tasking
  - Real-time
- **Ethernet**
- **Mass storage, Filesystem**

Multi-Tasking / Real-time / Graphical user Interface
What you need ?



Operating System (OS) Considerations

Bare-Metal / Standalone System

- Software system without an operating system
- Best deterministic behavior (no overhead, fastest interrupt response, …)
- No support of advanced features (no driver layer, no networking, USB, …)

→ **Minimal complexity**

Real-Time Operating Systems RTOS

- deterministic time behaviour
- predictable response time
- For timing sensitive applications
- Multitasking Support
  Static Task links,
  all Task code in image
- Tcp/IP Stacks available

→ **Medium complexity**

GUI based Operating Systems

Linux (Windows, ..)

- open-source operating system
- used in many embedded designs
- Full-featured operating system
  - Memory Management Unit (MMU)
  - Full support of all standard interfaces
  - Network, USB, …
  - and File-System
- no "real-time" behaviour

→ **High complexity**

As processing speed has continued to increase for embedded processing, the overhead of an operating system has become mostly negligible in many system designs.

ZYNQ OS support

https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842118/Zynq-7000+AP+SoC+Operating+Systems

https://developer.arm.com/solutions/os

Solution: freeRTOS, lwIP, FatFS

Operating System + Ethernet TCP/IP + Filesystem

FatFS

de.wikipedia.org/wiki/FreeRTOS

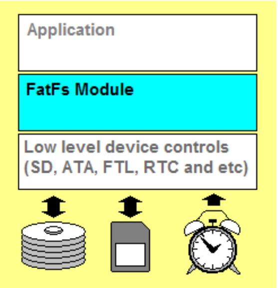
- **Open-Source-Echtzeitbetriebssystem**
- **for embedded MicroControllers**
- **Multitasking fähig**
- präemptive und cooperativer Scheduler

lwip.wikia.com/wiki/LwIP_Wiki

- **full scale TCP protocol stack**
- **small memory footprint**
- **for embedded systems, µC**
- **Open Source (C Code)**

http://elm-chan.org/fsw/ff/00index_e.html

- **FatFs is a generic FAT/exFAT filesystem**
- **for µC embedded systems**
- **Open Source (C Code)**



This on
- µController
- ….
- and **ZYNQ** **(ARM µC)**

and the best: all included in Xilinx SDK Libraries

UG643: OSLIB_RM.pdf

SDK provides a variety of

- packages, including drivers
- Libraries,
- board support packages,
- and complete operating systems

XILINX

UG643 (v2014.1) April 1, 2014

OS and Libraries Document Collection

Summary

The Software Development Kit (SDK) provides a variety of Xilinx® software packages, including drivers, libraries, board support packages, and complete operating systems to help you develop a software platform. This document collection provides information on these. Complete documentation for other operating systems can be found in the their respective reference guides. Device drivers are documented along with the corresponding peripheral documentation. The documentation is listed in the following table; click the name to open the document.

Table 1-1: OS and Libraries Document Collection Contents

Document Name	Summary
LibXil Standard C Libraries	Describes the software libraries available for the embedded processors.
Standalone (v.4.0)	The Standalone platform is a single-threaded, simple operating system (OS) platform that provides the lowest layer of software modules used to access processor-specific functions. Some typical functions offered by the Standalone platform include setting up the interrupts and exceptions systems, configuring caches, and other hardware specific functions. The Hardware Abstraction Layer (HAL) is described in this document.
Xilkernel (v6.0)	Xilkernel is a simple embedded processor kernel that can be customized to a large degree for a given system. Xilkernel has the key features of an embedded kernel such as multi-tasking, priority-driven preemptive scheduling, inter-process communication, synchronization facilities, and interrupt handling. Xilkernel is small, modular, user-customizable, and can be used in different system configurations. Applications link statically with the kernel to form a single executable.
LibXil Memory File System (MFS) (v2.0)	Describes a simple, memory-based file system that can reside in RAM, ROM, or Flash memory.
lwIP 1.4.0 Library (v2.0)	Describes the SDK port of the third party networking library, Light Weight IP (lwIP) for embedded processors.
LibXil Flash (v4.0)	Describes the functionality provided in the flash programming library. This library provides access to flash memory devices that conform to the Common Flash Interface (CFI) standard. Intel and AMD CFI devices for some specific part layouts are currently supported.
LibXil Isf (v4.0)	Describes the In System Flash hardware library, which enables higher-layer software (such as an application) to communicate with the Isf. LibXil Isf supports the Xilinx In-System Flash and external Serial Flash memories from Atmel (AT45XXXD), Intel (S33), and ST Microelectronics (STM) (M25PXX).
LibXil FFS (v2.0)	Xilffs is a generic FAT file system that is primarily added for use with SD/eMMC driver. The file system is open source and a glue layer is implemented to link it to the SD/eMMC driver. A link to the source of file system is provided in the PDF where the file system description can be found.

https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/oslib_rm.pdf

freeRTOS

Dr. Heinz Rongen
Forschungszentrum Jülich GmbH
Systeme der Elektronik (ZEA-2)
H.Rongen@fz-juelich.de

Characteristics of freeRTOS (Operating System)

Born in 2003 and initially conceived for microcontrollers

- Really light
- Really simple: the core of the O.S. are just 3 C files
- Minimal processing overhead
- Ported to a large number of architectures
- Open Source MIT license

- **FreeRTOS is a "Embedded Operating System" for**
- **Embeedded MicroController**
- Software that provides multitasking facilities.
- FreeRTOS allows to run multiple tasks
- and has a simple scheduler to switch between tasks.

FreeRTOS features:

- Priority-based multitasking capability
- Queues to communicate between multiple tasks
- Semaphores to manage resource sharing between tasks
- Utilities to view CPU utilization, stack utilization etc.

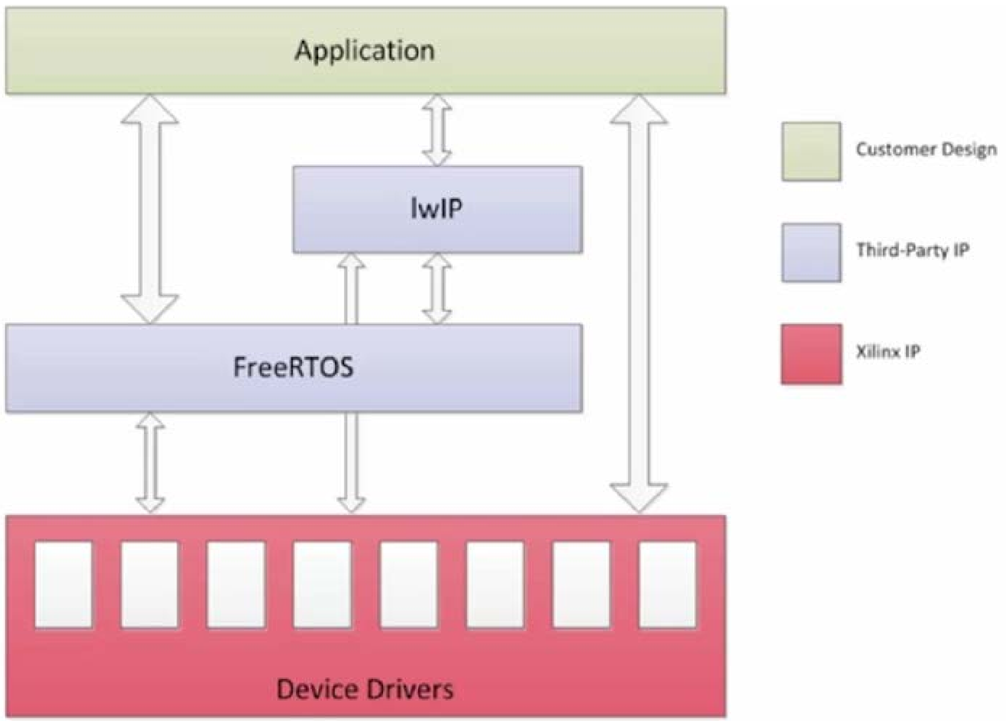
More information at: www.FreeRTOS.org

Supported CPUs (Ports): http://www.freertos.org/RTOS_ports.html

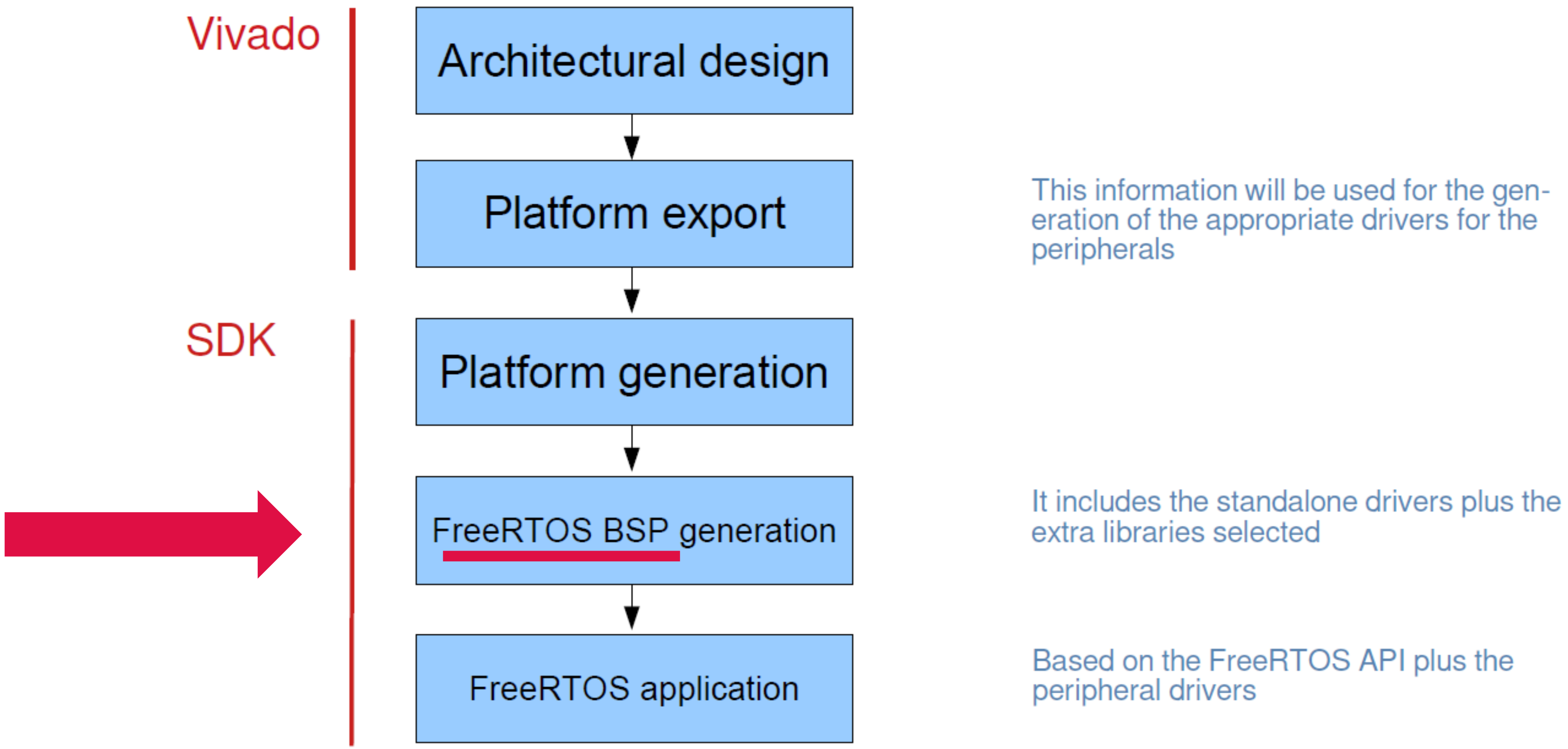
FreeRTOS in Xilinx environment

- **FreeRTOS completely integrated in Xilinx Software Development Flow (SDK, VITIS)**
- **Provided as a BSP**

- **Extension of the standalone BSP**
  - Includes the O.S. runtime
- All low level drivers can be directly used
- Optional extensions:
  - Filesystem
  - Network
  - ..



FreeRTOS design flow



FreeRTOS Configuration

through a header file: FreeRTOSConfig.h

(http://www.freertos.org/a00110.html)

```
#define configUSE_PREEMPTION 1

#define configUSE_MUTEXES 1

#define INCLUDE_xSemaphoreGetMutexHolder 1

#define configUSE_RECURSIVE_MUTEXES 1

#define configUSE_COUNTING_SEMAPHORES 1

#define configUSE_TIMERS 1

#define configUSE_IDLE_HOOK 0

#define configUSE_TICK_HOOK 0

#define configUSE_DAEMON_TASK_STARTUP_HOOK 0

#define configUSE_TICKLESS_IDLE 0
#define configTASK_RETURN_ADDRESS    NULL
#define INCLUDE_vTaskPrioritySet              1
#define INCLUDE_uxTaskPriorityGet             1
#define INCLUDE_vTaskDelete                   1
#define INCLUDE_vTaskCleanUpResources         1
#define INCLUDE_vTaskSuspend                  1
#define INCLUDE_vTaskDelayUntil               1
#define INCLUDE_vTaskDelay                    1
#define INCLUDE_eTaskGetState                 1
#define INCLUDE_xTimerPendFunctionCall        1
#define INCLUDE_pcTaskGetTaskName             1
#define configMAX_API_CALL_INTERRUPT_PRIORITY (18)
```

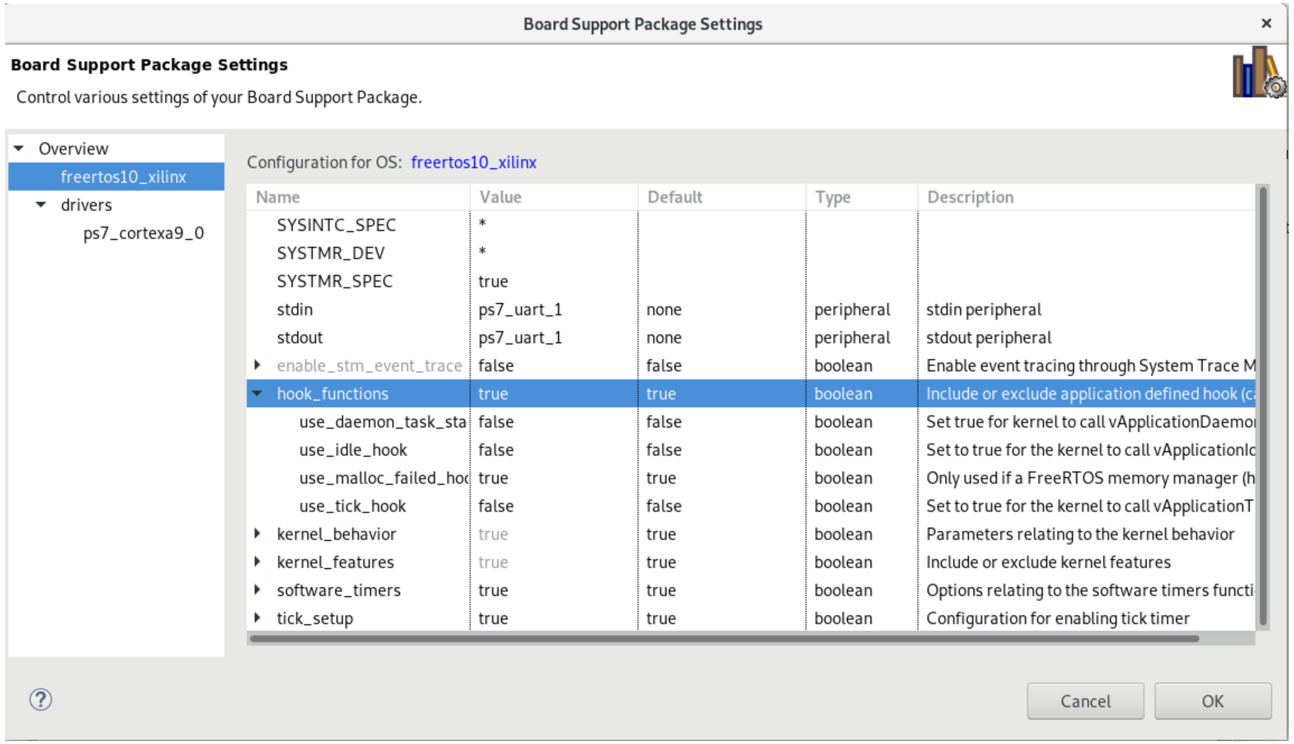
Tasks can be interrupted by others with higher priority

This will include a timer service task

Hooks are used to trigger the execution of functions upon the happening of certain events

Some functionality can be optionally included/excluded from the core of the O.S.

by the mss file in the FreeRTOS BSP generated in the SDK



Board Support Package Settings

Board Support Package Settings

Control various settings of your Board Support Package.

Overview / freertos10_xilinx / drivers / ps7_cortexa9_0

Configuration for OS: freertos10_xilinx

Name	Value	Default	Type	Description
SYSINTC_SPEC	*			
SYSTMR_DEV	*			
SYSTMR_SPEC	true			
stdin	ps7_uart_1	none	peripheral	stdin peripheral
stdout	ps7_uart_1	none	peripheral	stdout peripheral
enable_stm_event_trace	false	false	boolean	Enable event tracing through System Trace M
hook_functions	true	true	boolean	Include or exclude application defined hook (c
use_daemon_task_sta	false	false	boolean	Set true for kernel to call vApplicationDaemo
use_idle_hook	false	false	boolean	Set to true for the kernel to call vApplicationI
use_malloc_failed_hoo	true	true	boolean	Only used if a FreeRTOS memory manager (h
use_tick_hook	false	false	boolean	Set to true for the kernel to call vApplicationT
kernel_behavior	true	true	boolean	Parameters relating to the kernel behavior
kernel_features	true	true	boolean	Include or exclude kernel features
software_timers	true	true	boolean	Options relating to the software timers functi
tick_setup	true	true	boolean	Configuration for enabling tick timer

Cancel OK

Programming styles - STANDALONE

Standalone/Baremetal: Based on one superloop

```
void main ()
{
   Init_all();
   while (1)
   {
      do_A();
      do_B();
      do_C();
   }
}
```

Programming styles – Multitasking OS

freeRTOS

```
void main()
{
        xTaskCreate (Task_A, ….);
        xTaskCreate (Task_B, ….);
        xTaskCreate (Task_C, ….);
        xTaskStartScheduler ();
}
```

- The main program only initializes the needed tasks and starts the scheduler.
- After this the tasks (in this example 3 Tasks) are now working in parallel.
- Each Task can have his own initializing part.
- Finally each tasks operates in a own while loop, given the feeling of having several main programs in parallel.

```
void Task_A (void *p)
{
   Init_A();
   while (1)
   {
     do_A();
   }
}
```

```
void Task_B (void *p)
{
   Init_B();
   while (1)
   {
     do_B();
   }
}
```

```
void Task_C (void *p)
{
   Init_C();
   while (1)
   {
     do_C();
   }
}
```

A Task

- Task’s are parallel operating MAIN routines
- is a simple C function
- a pointer to parameters (void *) as input
- Creates a forever loop (while (1))
- The tasks are controlled by the Scheduler (freeRTOS internal function)

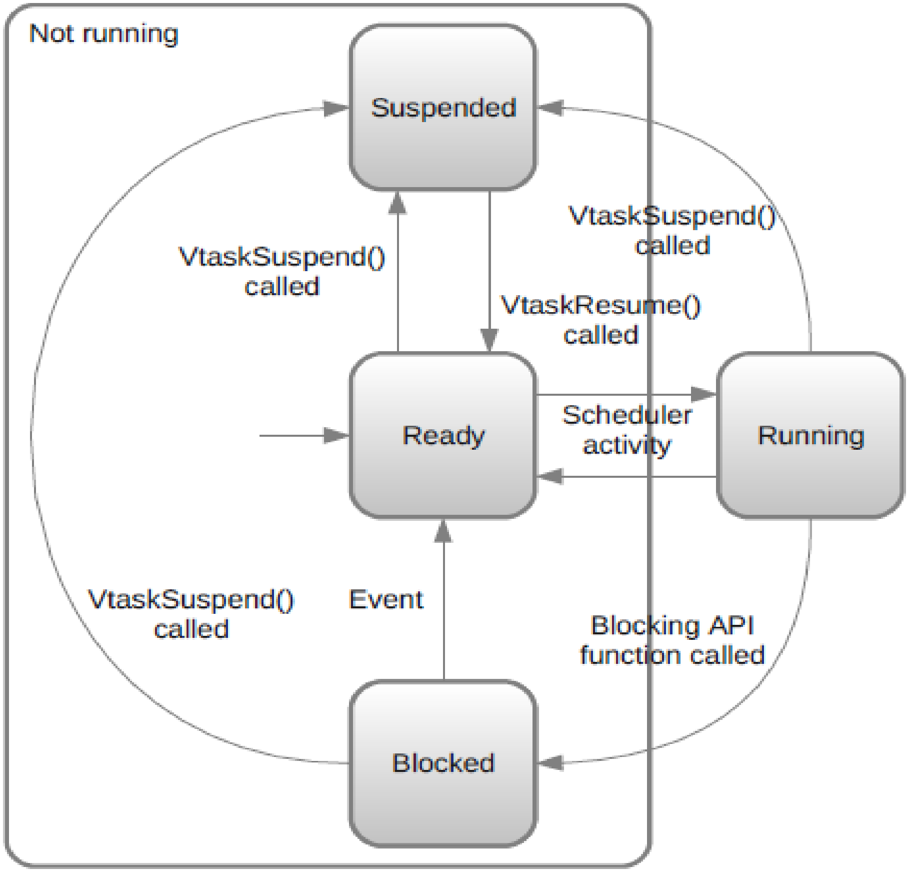
A task can be preempted (swapped out)
- because of a more priority task
- because it has been delayed (call to vTaskDelay())
- because it waits for a event (semaphore, …)

When a task can run
- state is set “Ready”

Task will start (swapped in) when
- No more priority task running at this time
- No Delay or Blocking condition

Finally
- a call to vTaskSuspend() stopps the Task at all
- vTaskResume() brings him back to the scheduler)

```
void my_task(void* p)
{
  while (1)
  {
    .....
  }
}
```



Creating a Task

The Task function itself:

```
void task_MySampleTask( void *p )
{
int xx[32];
   // do initilization
   while (1)
   {
        // Task execution code
   }
}
```

Remark for Stacksize:

- Each task has his own Stack
- Local variables and function calls are using stack
- ‘printf’, consumes around 1024 bytes of stack
- Running out of Stack will crash freeRTOS (freeze)
- **Allocate minimum 1024 bytes for Stack**

Create the Task (in main.c):

```
portBASE_TYPE xTaskCreate (
        pdTASK_CODE    pvTaskCode,              // pointer to the Task
        char*          pcName,                  // String: name of Task
        unsigned short usStackDepth,            // Stacksize                  ???
        void *         pvParameters,            // pointer to Parameters      (can be NULL)
        unsigned short uxPriority,              // Priority                   (default 2)
        xTaskHandle*   pxCreatedTask  );        // Pointer to receive Task handle (can be NULL)
```

A typical freeRTOS application

```
int main ( void )
{
// do needed Platform initialization

// Now we deal with RTOS. Create the Tasks and start the scheduler

//   1) Start LED 1 toogle
       xTaskCreate (Task_LEDS, (signed char*) "LEDs", 64, NULL, 1, NULL);

//   2) Start SWITCH
       xTaskCreate (Task_SWITCH, (signed char*) "SWs", 64, NULL, 1, NULL);

//   3) Start LCD-Anzeige
      xTaskCreate (Task_LCD, (signed char*) "LCD", 1024, NULL, 1, NULL);


// Finally: Start FreeRTOS
      vTaskStartScheduler();

// Will only reach here if there was insufficient memory to create the idle task
      return 0;
}
```

Simple ‚main‘ and first task

Simple task example that prints a message once a second:

- **xTaskCreate (..)** → creates a Task
  - Every task has an infinite loop
  - Works as independent main programs
- **vTaskStartScheduler (..)** → Starts the OS scheduler and FreeRTOS will begin servicing the tasks
- **vTaskDelay (..)** → Give up CPU time and allow others task to run

```
void task_HelloWorld (void* p)
{
   while(1)
   {
         printf("Hello World!");
         vTaskDelay(1000);
   }
}

void main(void)
{
  xTaskCreate  (task_HelloWorld, „HelloW", 1024, NULL, 1, NULL);

  vTaskStartScheduler();
   // never comes here
}
```

Controlling Tasks

- Task with highest priority will run first, and never give up the CPU until it sleeps

- If two or more tasks with the same priority do not give up the CPU (they don't sleep), then FreeRTOS will share the CPU between them (time slice).

- Here are some of the ways you can give up the CPU:

  - **vTaskDelay (..)** This simply puts the task to "sleep„. You decide how much you want to sleep.
  - **vTaskDelayUntil (..)** Delay a task until a specified absolute time.

  - **xQueueSend (..)** If the Queue you are sending to is full, this task will sleep (block).
  - **xQueueReceive (..)** If the Queue you are reading from is empty, this task will sleep (block).
  - **xSemaphoreTake (..)** You will sleep if the semaphore is taken by somebody else.

Stop a Task / Scheduler

Task:

- A Task can stop himself or also other tasks

```
// Example:  Stopping a task
...
   vTaskDelete (NULL);
```

Scheduler:

- one method to create a critical section:
    - prevent a task from preempting it
    - but let interrupts to do their job

→ Stopping the scheduler (= stopping all other tasks)
→ Do code in ‘critical section’
→ Restart the scheduler

Notice: No FreeRTOS API functions can be called when the scheduler is stopped !

```
// Example:  Create a critical section
...
   vTaskSuspendAll();
   {
     printf( "%s", pcString );
     fflush( stdout );
   }
   xTaskResumeAll();

...
```

Software Timers

```
TimerHandle_t xTimerCreate (
      const char * const pcTimerName,                   // name of the timer
      const TickType_t xTimerPeriod,                    // Period in ticks (ms)
      const UBaseType_t uxAutoReload,                   // FALSE=OneShoot / TRUE=Repeat
      void * const pvTimerID,                           // ID number
      TimerCallbackFunction_t pxCallbackFunction );     // Timer Callback function
```

```
#include <timers.h>

TimerHandle_t xTimer;

void vTimerCallback ( TimerHandle_t pxTimer )                    // Timer callback function
{
static cnt = 0;
  cnt++;
  if (cnt > 10) xTimerStop( pxTimer, 0 );
}


void main( void )
{
   xTimer = xTimerCreate ("Tim1", 200, TRUE, (void*)1, vTimerCallback );  // Install the timer
   xTimerStart (xTimer, 0 );                                                   // Start timer

...  // do more (Create other tasks …..)
  vTaskStartScheduler();
}
```

See: http://www.freertos.org/FreeRTOS-Software-Timer-API-Functions.html

Project 1: simple freeRTOS example

```
#include „freeRtos.h“
#include „task.h“
int tick=0;
```

```
void Task_LED (void* p)
{
  while (1)
  {
     Xil_Out32 (adrGPIO, tick);
     vTaskDelay (100);
  }
}
```

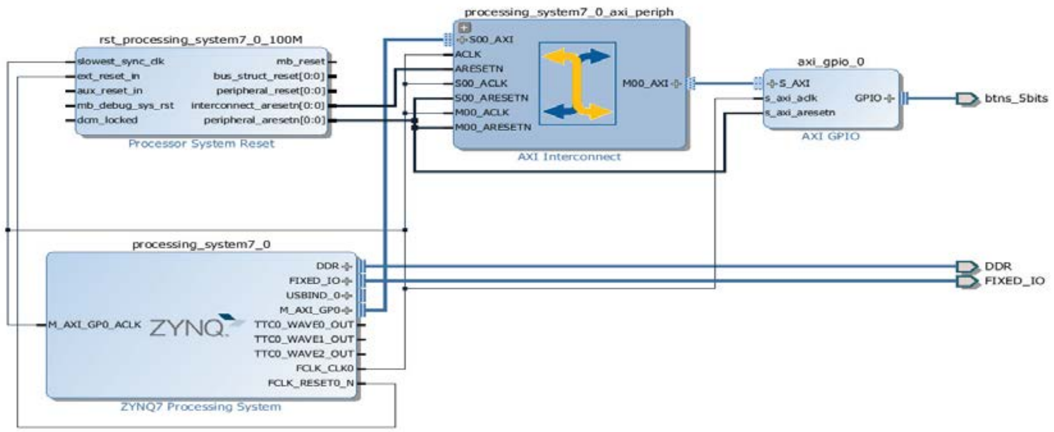
```
void Task_Print (void* p)
{
  while (1)
  {
     printf („Tick is %d \n“, tick);
     vTaskDelay (500);
     tick++;
  }
}
```

```
int main ( void )
{
   //   1) Start LED 1 toogle
   xTaskCreate (Task_LED,  (signed char*) "LEDs", 1024, NULL, 1, NULL);

   //   2) printf
   xTaskCreate (Task_Print, (signed char*) “Print", 1024, NULL, 1, NULL);

   // Finally: Start FreeRTOS
   vTaskStartScheduler();

// Will only reach here if there was insufficient memory to create the idle task
   return 0;
}
```

Demonstration 1: First (simple) FreeRTOS application

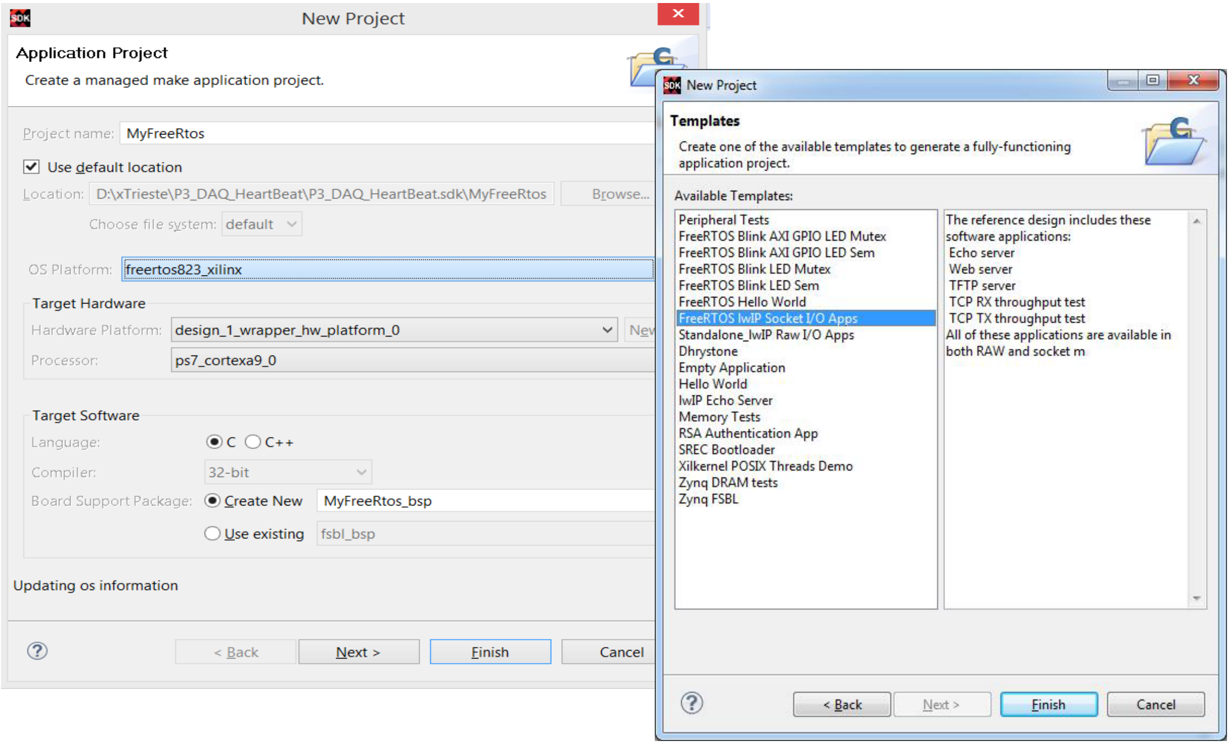
1) VIVADO Block Design

- Configure a Vivado project with the IPI (IP Block Integrator)
- Add the "ZYNQ7 Processing System"
  - (use the standard MIO Signals (PS Multiplexed IO))
- Add a GPIO to the LEDs
- Make a HDL Wrapper,
- Implement the Design,
- Generate Bitstream,
- Export to SDK and launch SDK

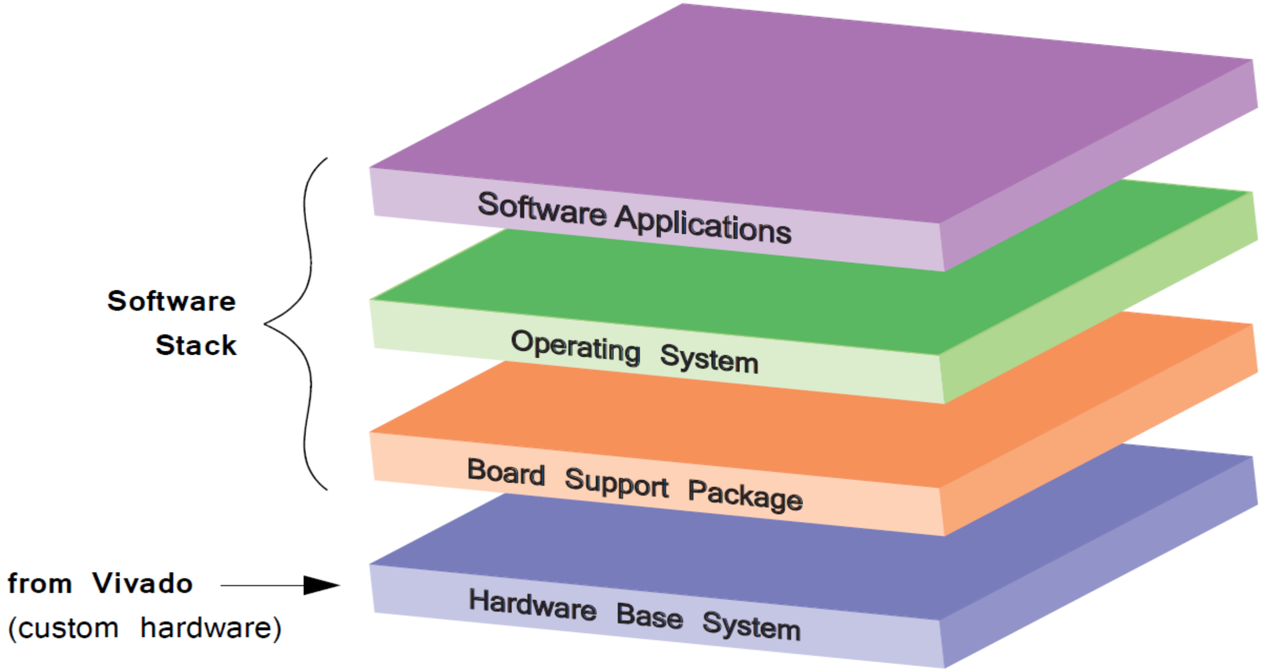


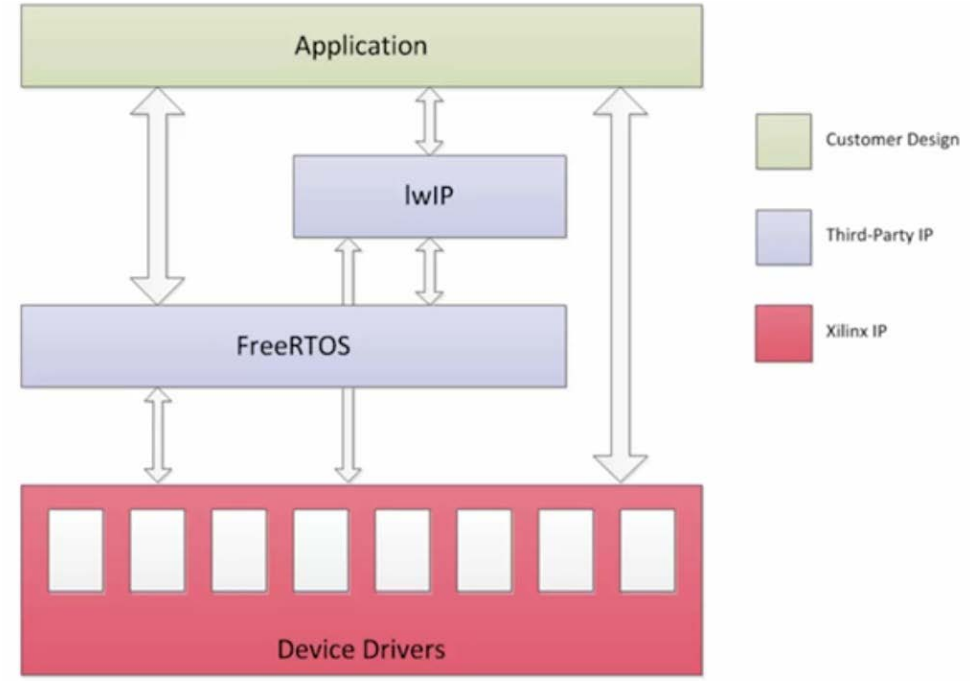
2) SDK Application

- File -> New -> Application Project
- Provide Project Name
- Target Hardware: Choose the Hardware Platform
- **OS Platform: freertosXXX_Xilinx**
- Click Next
- For first example: "FreeRTOS Hello World"
- Click Finish



Appendix: Software Stack





lwIP

TCP/IP Ethernet-stack for freeRTOS

Dr. Heinz Rongen
Forschungszentrum Jülich GmbH
Systeme der Elektronik (ZEA-2)
H.Rongen@fz-juelich.de

The TCP/IP Stack

lwIP stands for ‘Lightweight IP’

- full scale TCP protocol stack
- small memory footprint (for embedded systems, µC)
- Open Source (C Code)

Supports a large number of protocols and APIs

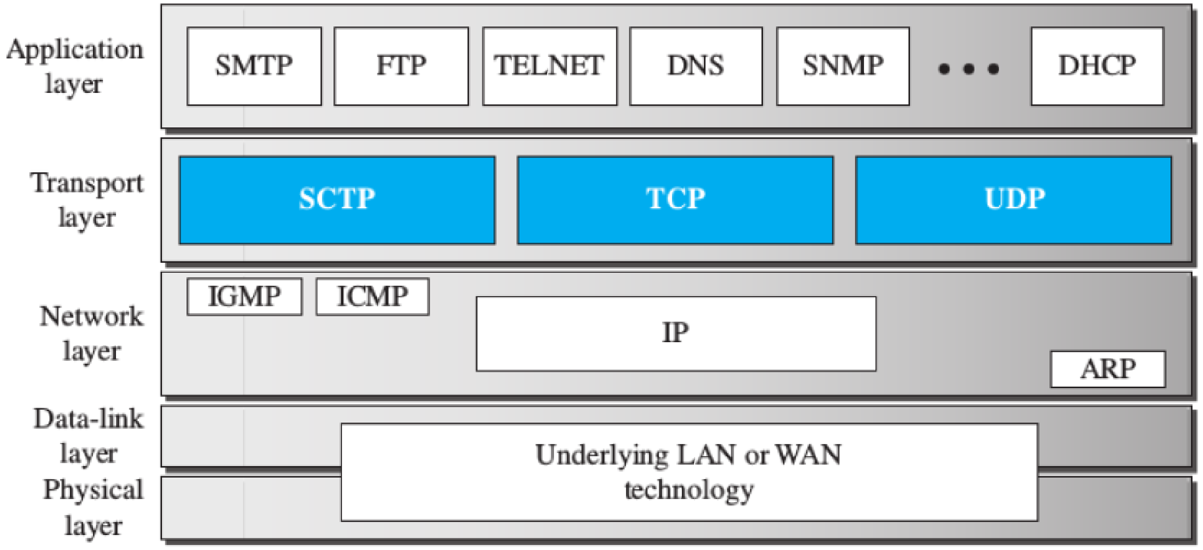
- TCP Transport Control Protocol
- UDP User Datagram Protocol
- IP Internet Protocol
- ICMP Internet Control Message Protocol
- ARP Address Resolution Protocol
- DHCP Dynamic Host Configuration Protocol
- Raw API and Berkeley sockets (requires an multitasking O.S.)

Included in Xilinx SDK

- includes driver for Xilinx Ethernet driver
- XAPP1026 is the reference application note

Application level

- HTTP(S) server, SNTP client,
  SMTP(S) client, ping, TFTP, …



The network design is organized as a layer stack.

- Each layer provides a set of services to the upper layer and requires services from the lower layer.

http://www.zedboard.org/content/xapp1026-zedboard

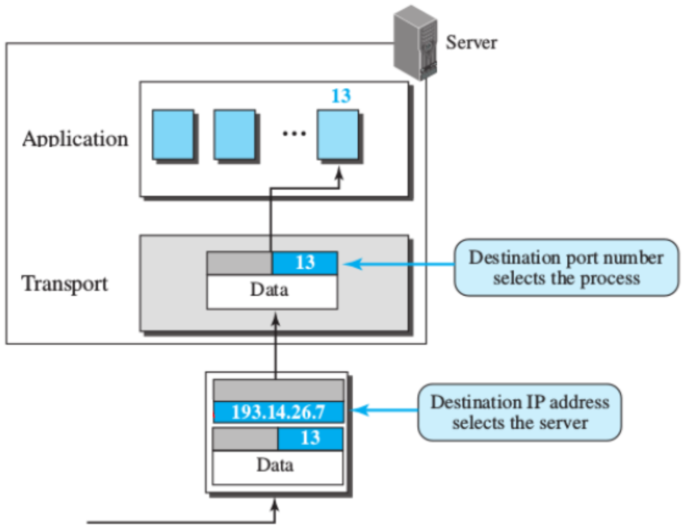
BSD Sockets

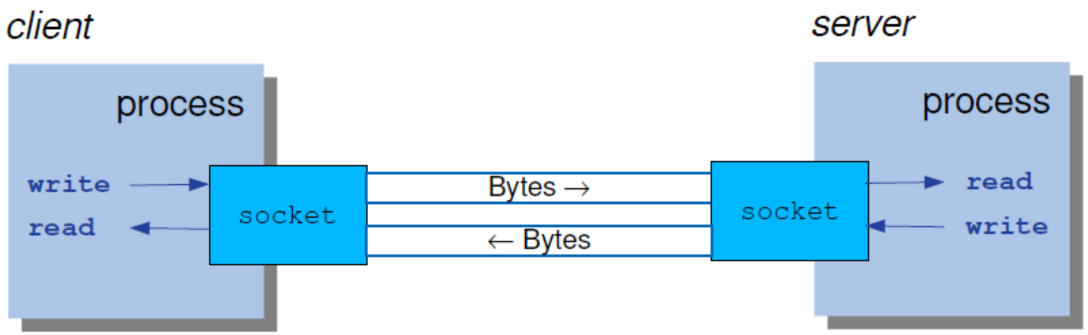
BSD Sockets (Berkeley sockets | POSIX sockets)

- de facto standard API
- Basic abstraction for network programming
- Combination of **IP address + port**
- Inter-process communication

→ use „LwIP Socket API“

`lwip_socket(AF_INET, SOCK_STREAM, 0)`





SETUP & STARTUP OF THE NETWORK

Basic template for freeRTOS & lwIP:

- Start a *“**task_StartEthernet**”*
  - Initializes lwip
  - Configures a network interface
  - Start the interface and a reception task
  - Installs any other network tasks
    - Here: Web-Server, Echo-server
  - Finally the start up task deletes himself

After initialization several threads are active:

- Reception task
- Web-server
- Echo-server

```
int main()
{
    printf ("Start Ethernet \n\r");
    xTaskCreate (task_StartEthernet,  (char*)"Start_Eth", 2048, NULL, DEFAULT_THREAD_PRIO, NULL);

    vTaskStartScheduler();
    while(1);
    return 0;
}
```

```
void task_StartEthernet(void *p)
{
struct netif *netif;
struct ip_addr ipaddr, netmask, gw;

/* the mac address of the board. this should be unique per board */
unsigned char mac_ethernet_address[] = { 0x00, 0x0a, 0x35, 0x00, 0x01, 0x02 };

    lwip_init();
    printf ("Start Ethernet2 \n\r");

    netif = &server_netif;

    /* initialize IP addresses to be used */
    IP4_ADDR(&ipaddr,  192, 168,   2,  2);
    IP4_ADDR(&netmask, 255, 255, 255,  0);
    IP4_ADDR(&gw,      192, 168,   2,  1);

    /* print out IP settings of the board */
    print("\r\n\r\n");
    print("-----lwIP Socket Mode Demo Application ------\r\n");

    print_ip_settings(&ipaddr, &netmask, &gw);
    /* print all application headers */

    /* Add network interface to the netif_list, and set it as default */
    if (!xemac_add(netif, &ipaddr, &netmask, &gw, mac_ethernet_address, PLATFORM_EMAC_BASEADDR))
    {
        xil_printf("Error adding N/W interface\r\n");
        return;
    }
    netif_set_default(netif);
    netif_set_up(netif);  // specify that the network if is up

    // start packet receive thread - required for lwIP operation
    sys_thread_new ("xemacif_input_thread", (void(*)(void*))xemacif_input_thread, netif, 2048, DEFAULT_THREAD_PRIO);

//**************************************************

    sys_thread_new("httpd", web_application_thread, 0,  2048, DEFAULT_THREAD_PRIO);

    sys_thread_new("echod", echo_application_thread, 0,  2048,  DEFAULT_THREAD_PRIO);

    vTaskDelete(NULL);
    return;
}
```

UDP

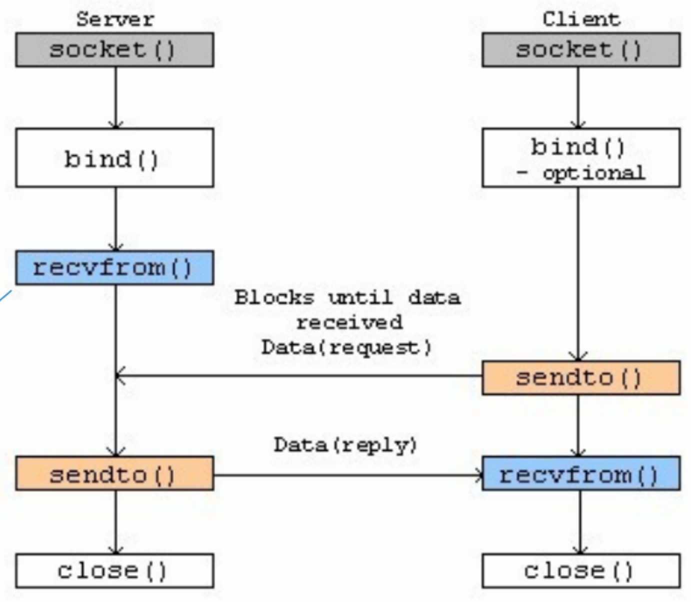
Unreliable protocol

- **No error control**
- **corrupted packets are ignored**
- **No flow control (Speed)**

But:

- **Extremely simple (minimum overhead)**
- **the fastest way (lowest latency)**

- UDP socket Programming flow



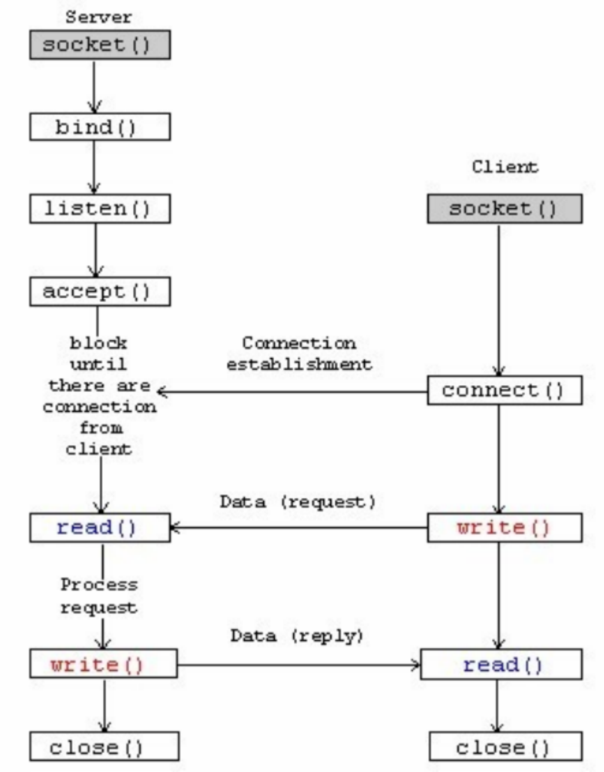

```
void echo_application_thread()
{
        int sock, new_sd;
        struct sockaddr_in address, remote;
        int size;

        int RECV_BUF_SIZE = 2048;
        char recv_buf[RECV_BUF_SIZE];
        int n, nwrote;

        if ((sock = lwip_socket(AF_INET, SOCK_DGRAM, 0)) < 0)
                return;

        address.sin_family = AF_INET;
        address.sin_port = htons(echo_port);
        address.sin_addr.s_addr = INADDR_ANY;

        if (lwip_bind(sock, (struct sockaddr *)&address, sizeof (address)) < 0)
                return;

        if ((n = read(sock, recv_buf, RECV_BUF_SIZE)) < 0) {
                xil_printf("%s: error reading from socket %d, closing socket\r\n",
                                __FUNCTION__, sock);
        }
```

Used in applications where loss of some part of the information can be tolerated, example Video Streaming/conference

TCP

- Connection-oriented protocol
- Reliable, Error free (correction)
  - Retransmission of lost or corrupted packets

- Complex protocol with multiple phases
  - higher latency, lower throughput
  - Connection control



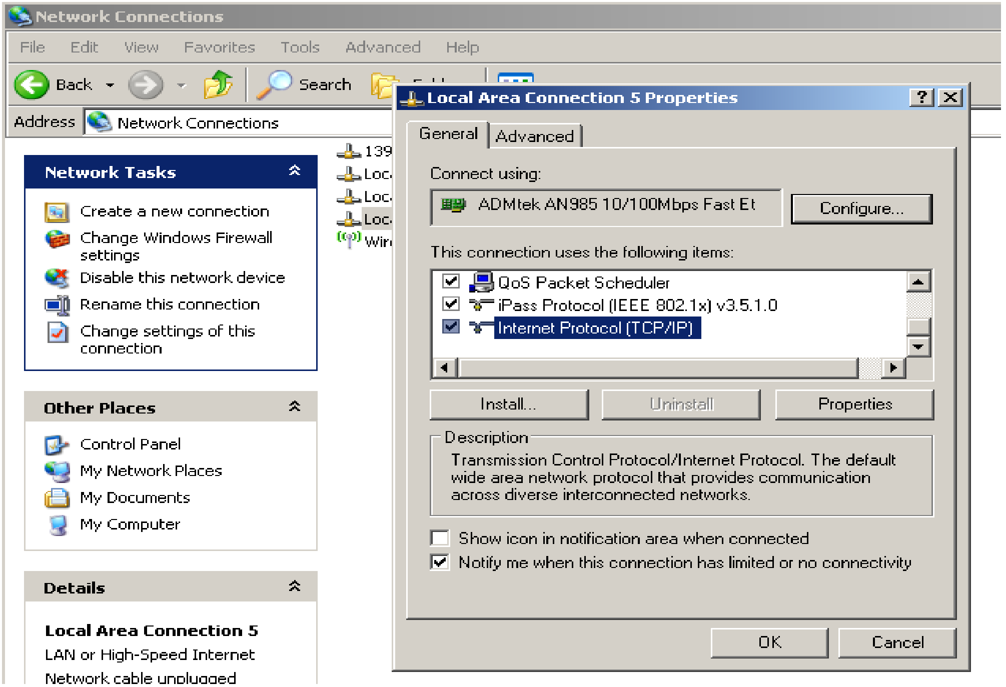
Used when loosing information can't be tolerated.
Example: HTTP, E-mail, binary Data, …

```
void echo_application_thread()
{
        int sock, new_sd;
        struct sockaddr_in address, remote;
        int size;

        if ((sock = lwip_socket(AF_INET, SOCK_STREAM, 0)) < 0)
                return;

        address.sin_family = AF_INET;
        address.sin_port = htons(echo_port);
        address.sin_addr.s_addr = INADDR_ANY;

        if (lwip_bind(sock, (struct sockaddr *)&address, sizeof (address)) < 0)
                return;

        lwip_listen(sock, 0);

        size = sizeof(remote);

        while (1) {
                if ((new_sd = lwip_accept(sock, (struct sockaddr *)&remote, (socklen_t *)&size)) > 0) {
                        sys_thread_new("echos", process_echo_request,
                                (void*)new_sd,
                                THREAD_STACKSIZE,
                                DEFAULT_THREAD_PRIO);
                }
        }
}
```

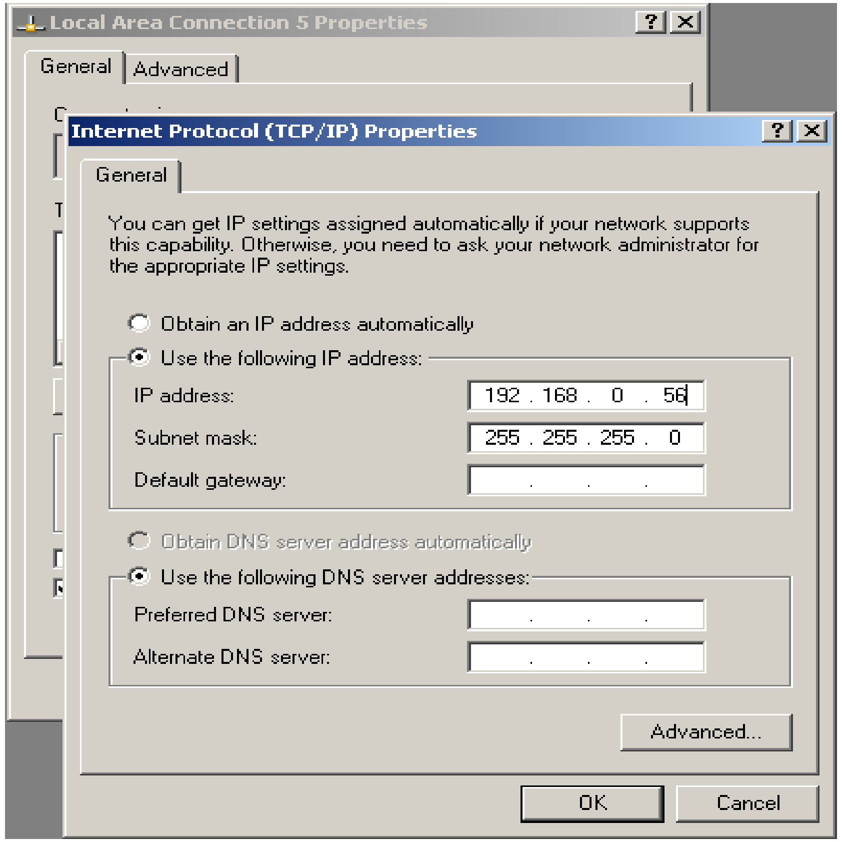
```
/* thread spawned for each connection */
void process_echo_request(void *p)
{
  int sd = (int)p;
  int RECV_BUF_SIZE = 2048;
  char recv_buf[RECV_BUF_SIZE];
  int n, nwrote;

  while (1) {
    /* read a max of RECV_BUF_SIZE bytes from socket */
    if ((n = read(sd, recv_buf, RECV_BUF_SIZE)) < 0) {
      xil_printf("%s: error reading from socket %d, closing socket\r\n", __FUNCTION__, sd);
      break;
    }

    /* break if client closed connection */
    if (n <= 0)
      break;

    /* handle request */
    if ((nwrote = write(sd, recv_buf, n)) < 0) {
      xil_printf("%s: ERROR responding to client echo request. received = %d, written = %d\r\n",
                 __FUNCTION__, n, nwrote);
      xil_printf("Closing socket %d\r\n", sd);
      break;
    }
  }

  /* close connection */
  close(sd);
  vTaskDelete(NULL);
}
```

Setup PC for communication

Control Panel → Network connections → Properties



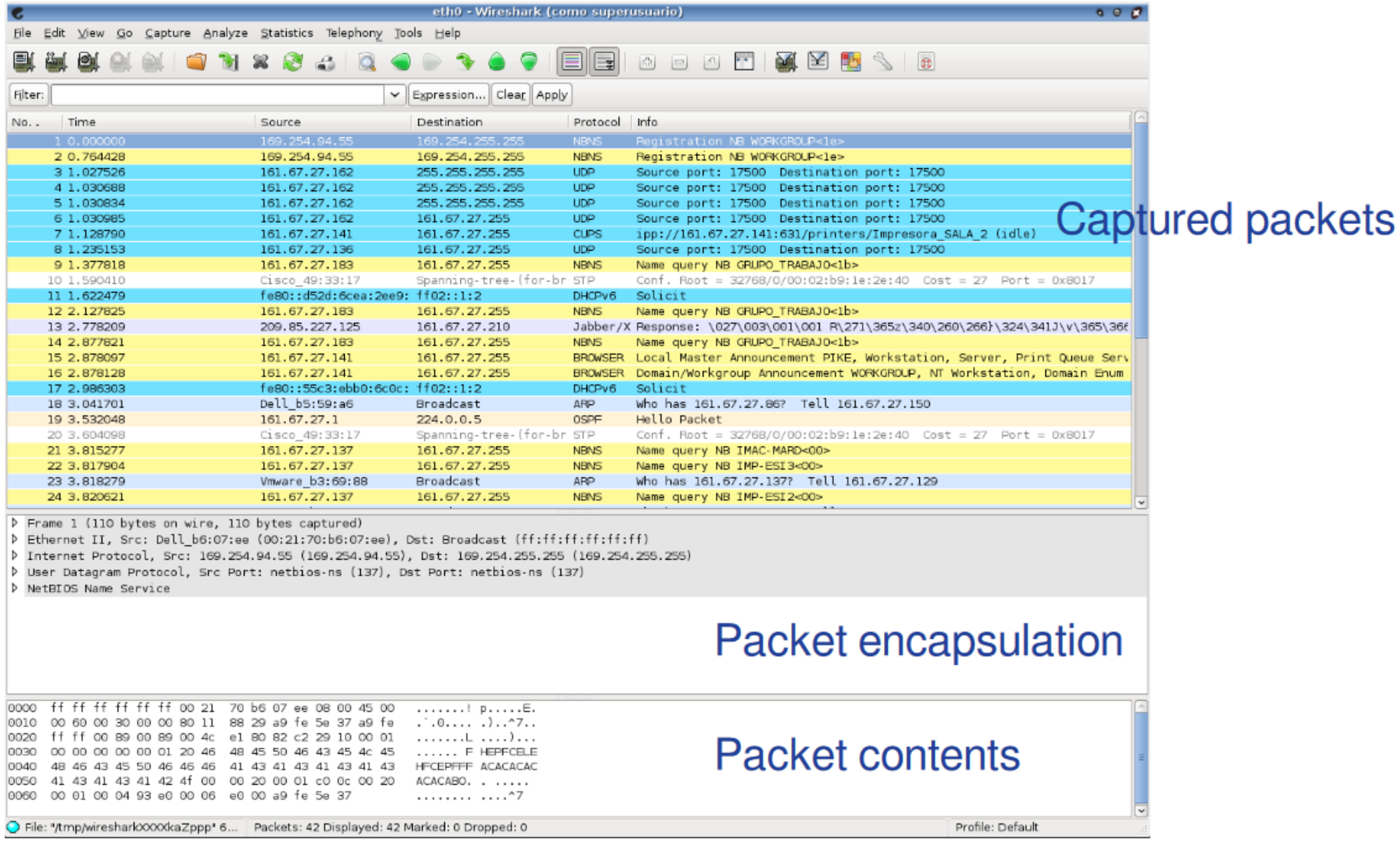
Set PC TCP/IP to address within same subnet:



TROUBLESHOOTING

Wireshark

- widely-used network protocol analyzer.
- see what’s happens on your network
- Filters to select specific packets
- Live capture / Offline analysis
- Multi-Protocol
  - Decryption of packets headers



FatFS

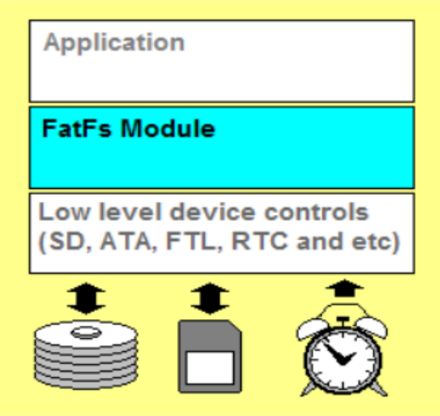
http://elm-chan.org/fsw/ff/00index_e.html

Dr. Heinz Rongen
Forschungszentrum Jülich GmbH
Systeme der Elektronik (ZEA-2)
H.Rongen@fz-juelich.de

FAT File System

FatFs - Generic FAT Filesystem Module

FatFs is a generic FAT/exFAT filesystem module for small embedded systems. The FatFs module is written in compliance with ANSI C (C89) and completely separated from the disk I/O layer. Therefore it is independent of the platform. It can be incorporated into small microcontrollers with limited resource, such as 8051, PIC, AVR, ARM, Z80, RX and etc. Also Petit FatFs module for tiny microcontrollers is available here.

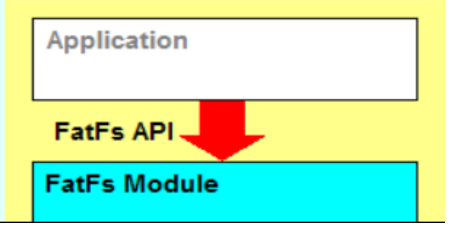


Features

- DOS/Windows compatible FAT/exFAT filesystem.
- Platform independent. Easy to port.
- Very small footprint for program code and work area.
- Various configuration options to support for:
  - Multiple volumes (physical drives and partitions).
  - Multiple code pages including DBCS.
  - Long file name in OEM code or Unicode.
  - exFAT filesystem.
  - Thread safe.
  - Fixed or variable sector size.
  - Read-only, optional API, I/O buffer and etc...

Application Interface

FatFs provides various filesystem functions for the applications as shown below.



- File Access
  - f_open - Open/Create a file
  - f_close - Close an open file
  - f_read - Read data from the file
  - f_write - Write data to the file

FatFS programming example

```
//-- Fat-File System
#include "ff.h"


FATFS      fatfs;
TCHAR      *Path = "0:/";


int SD_Mount (void)
{
FRESULT Res;
           Res = f_mount (&fatfs, Path,1);
           if (Res != FR_OK)
           {
                      printf ("SD: Mount failed\n\r");
                      return 0;
           }
           printf ("SD mounted\n\r");
           return 1;

}

int SD_Unmount(void)
{
           f_mount(NULL, Path, 1);
           SDMounted = 0;
           return 1;
}
```

```
void Write_SDcard (void)
{
int Res, wr;
FIL* hFile;
     Res = f_open (&hFile, "data.bin", FA_CREATE_ALWAYS | FA_WRITE );
     if (Res)
     {
          printf("SD: Open failed \n\r");
          return 0;
     }
     Res = f_write (&hFile, Data, 1024*2, &wr);
     Res = f_close (&hFile);
     return 1;
}
```

```
void Read_SDcard (void)
{
int Res, rd;
FIL* hFile;
     Res = f_open (&hFile, "data.bin", FA_READ );
     if (Res)
     {
          printf("SD: Open failed \n\r");
          return 0;
     }
     Res = f_read (&hFile, Data, 1024*2, &rd);
     Res = f_close (&hFile);
     return 1;
}
```

Links / Demonstration

Links

freeRTOS Documentation:

http://www.freertos.org

FreeRTOS API documentation
http://www.freertos.org/a00106.html

Books:
- FreeRTOS Reference Manual.pdf
- FreeRTOS_Melot.pdf

lwIP Documentation:

http://www.nongnu.org/lwip/

Two Application Program's Interfaces (APIs)

- Netconn API: http://lwip.wikia.com/wiki/Netconn_API
- Socket API: http://pubs.opengroup.org/onlinepubs/007908799/xnsix.html
  (compatible to posix- / BSD-sockets)

Demonstration 2

REALTIME DATA ACQUISITION SYSTEM BASED ON FREERTOS & LWIP

Annex: Accessing memory mapped Hardware-Devices

Using BSP driver functions:

```
#include "xparameters.h"
#include "xgpio.h"

#define LED_CHANNEL     1
#define SW_CHANNEL      2

XGpio Gpio;             /* The Instance of the GPIO Driver */

            // GPIO Initialisation:
            XGpio_Initialize                    (&Gpio, XPAR_AXI_GPIO_0_DEVICE_ID);    //=0
            XGpio_SetDataDirection              (&Gpio, LED_CHANNEL,    0xFFFFFF00);   // 0 = Outputs
            XGpio_SetDataDirection              (&Gpio, SW_CHANNEL,     0xFFFFFFFF);   // 1 = Inputs

            //GPIO Data:
            Data = XGpio_DiscreteRead           (&Gpio, SW_CHANNEL);
            XGpio_DiscreteWrite                 (&Gpio, LED_CHANNEL, Data);
```

Using direct I/O functions:

```
#include "xparameters.h"

#include “Xil_io.h”


                 Xil_Out32 (Addr, Value)

                 Xil_In32  (Addr)
```

Pointer usage:

```
#include "xparameters.h“
            *(u32*)Addr = Value;
```

AXI GPIO Registers

Registers

There are four internal registers in the AXI GPIO design as shown in Table 4. The memory map of the AXI GPIO design is determined by setting the C_BASEADDR parameter. The internal registers of the AXI GPIO are at a fixed offset from the base address and are byte accessible.

Table 4: **Registers**

Base Address + Offset (hex)	Register Name	Access Type	Default Value (hex)	Description
C_BASEADDR + 0x00	GPIO_DATA	Read/Write	0x0	Channel 1 AXI GPIO Data Register
C_BASEADDR + 0x04	GPIO_TRI	Read/Write	0x0	Channel 1 AXI GPIO 3-state Register
C_BASEADDR + 0x08	GPIO2_DATA	Read/Write	0x0	Channel 2 AXI GPIO Data Register
C_BASEADDR + 0x0C	GPIO2_TRI	Read/Write	0x0	Channel 2 AXI GPIO 3-state Register

3-State Register: 1 = Input 0=Output

Default Base-Address: look in **xParameters.h** (0x4120_0000)

Change Echo Server to “Data-Server”

```
u16_t echo_port = 1111;

short int Data[1024];
int tick = 0;

void GetMyDAQ_Data (void)
{
int i;
       for (i=0; i<1024; i++)
              Data[i] = i + tick*33;
       tick++;
}
```

```
GetMyDAQ_Data ();

nwrote = write (sd, Data, 1024*sizeof(short));

//nwrote = write (sd, recv_buf, n);  // handle request by sending data back
```

Labview Client for Visualization