freeRTOS & lwIP

for ZYNQ (and Zedboard)

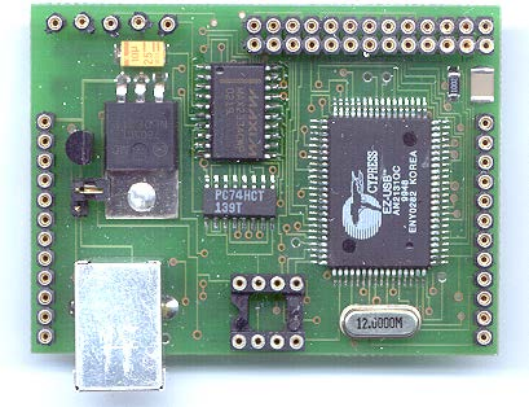
Dr. Heinz Rongen
Forschungszentrum Jülich GmbH
Zentralinstitut Systeme der Elektronik (ZEA-2)
H.Rongen@fz-juelich.de

µController (The small ones …)

- **Single Chip Solutions / Applications**
- Internal Memory: Flash / RAM (program / data memory)
- Internal I/O
  - Digital I/O: Ports, Timer, Counter, PWM, …
  - Analog I/O: ADC‘s, DAC‘s, …
  - **Communication: I2C, SPI, RS232, USB, …**

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

- **Typically: No Networking / Ethernet:**
  - TcpIP Stack to complex to implement

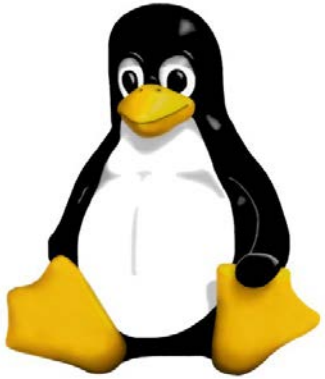


Big brothers .. Using Operating Systems

(Processor Systems)

- **Processor Systems: Multiple Chip Solutions**
- External Memories
- External I/O‘s

- **Operating system support**
  - **Windows, embedded Linux, …**
- **Full features Networking / Ethernet**

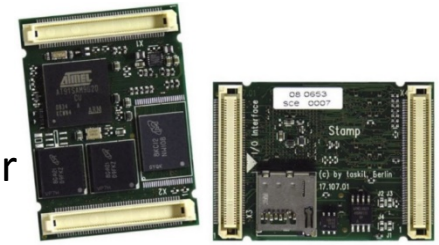


Embedded Linux

www.taskit.de

Stamp 926

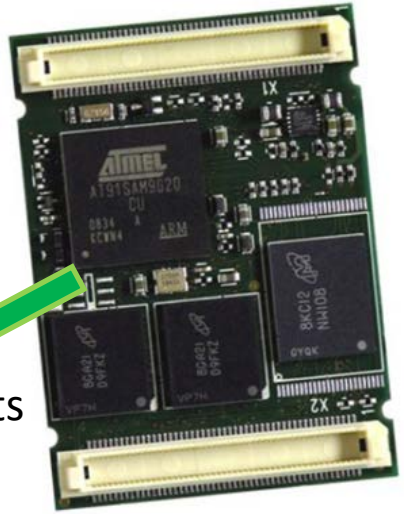
- ARM9
- 16 MB Flash
- 32 MB SDRAM
- SSD Karte
- LCD/TFT Controller
- **Ethernet**



The gap …

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

- **Powerful**
- **OS Support**
  - Multi-Tasking
  - Real-time
- **Ethernet**



ZYNQ OS support

- **Zynq-7000 supports a comprehensive collection of operating systems to suit your system needs**
  - Open Source Linux
    - Xilinx provides a freely downloadable Linux solution
    - Xilinx Git Repository
    - Xilinx acquired PetaLogix
  - Open Source Android
    - Freely downloadable Android 2.3 solution
  - Open Source FreeRTOS
    - Light-weight real-time OS
    - Used in applications that demand deterministic and real-time responsiveness to events in the system

Operating System (OS) Considerations

- **Bare-Metal 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

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

GUI based Operating Systems

Linux

- open-source operating system
- used in many embedded designs
- Linux is not a "real" real-time operating system
- Full-featured operating system
    - Memory Management Unit (MMU)
    - Full support of all standard interfaces
    - Network, USB, …
    - and File-System

→ High complexity

Real-Time Operating Systems

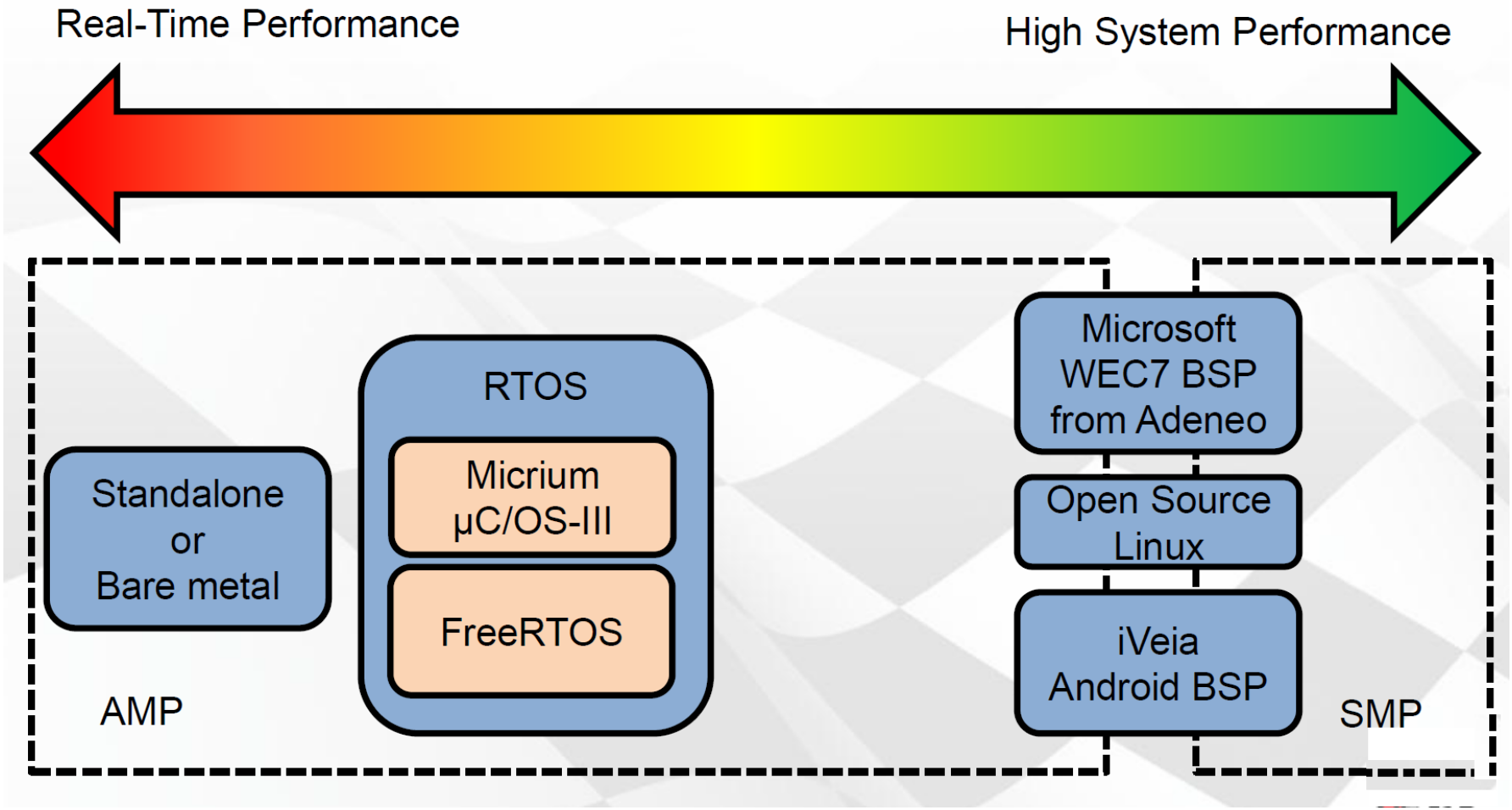
RTOS

- 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

Realtime needed

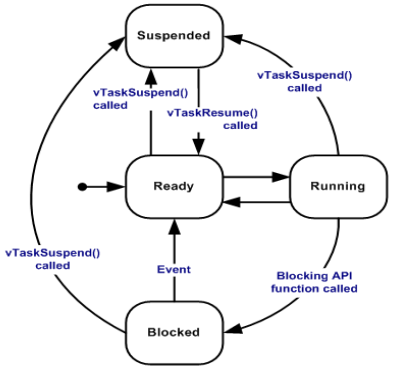
- **What are your software application requirements?**



Solution: freeRTOS

de.wikipedia.org/wiki/FreeRTOS

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



Valid task state transitions

This on

- Atmel µController
- PIC µController
- ....
- and **ZYNQ**

lwip.wikia.com/wiki/LwIP_Wiki

IP (Internet Protocol)
ICMP (Internet Control Message Protocol)
IGMP (Internet Group Management Protocol)
UDP (User Datagram Protocol)
TCP (Transmission Control Protocol)
BSD Berkeley-like socket API
DNS (Domain names resolver)
SNMP (Simple Network Management Protocol)
DHCP (Dynamic Host Configuration Protocol)
PPP (Point-to-Point Protocol)
ARP (Address Resolution Protocol) for Ethernet

→ in ca. 40 KB code und 8 KB Ram

Characteristics of freeRTOS (Operating System)

- **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 multiple tasks
- Utilities to view CPU utilization, stack utilization etc.

Supported CPUs (Ports):

- http://www.freertos.org/RTOS_ports.html

Repository (Library) for freeRTOS and lwIP

- A stand-alone board support package (BSP) is a library generated by the Xilinx SDK that is specific to a hardware design.
- It contains initialization code for bringing up the ARM CPUs in ZYNQ and also contains software drivers for all available ZYNQ peripherals.

The freeRTOS Repository

- The FreeRTOS port extends the stand-alone BSP to also include FreeRTOS source files
- After using this port in a Xilinx SDK environment, the user gets all the FreeRTOS source files in a FreeRTOS BSP library.
- This library uses the Xilinx SDK generated stand-alone BSP library.

Programming styles

Standalone:

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

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 ()
{
   Init_A();
   while (1)
   {
       do_A();
   }
}
```

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

```
void Task_C ()
{
   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

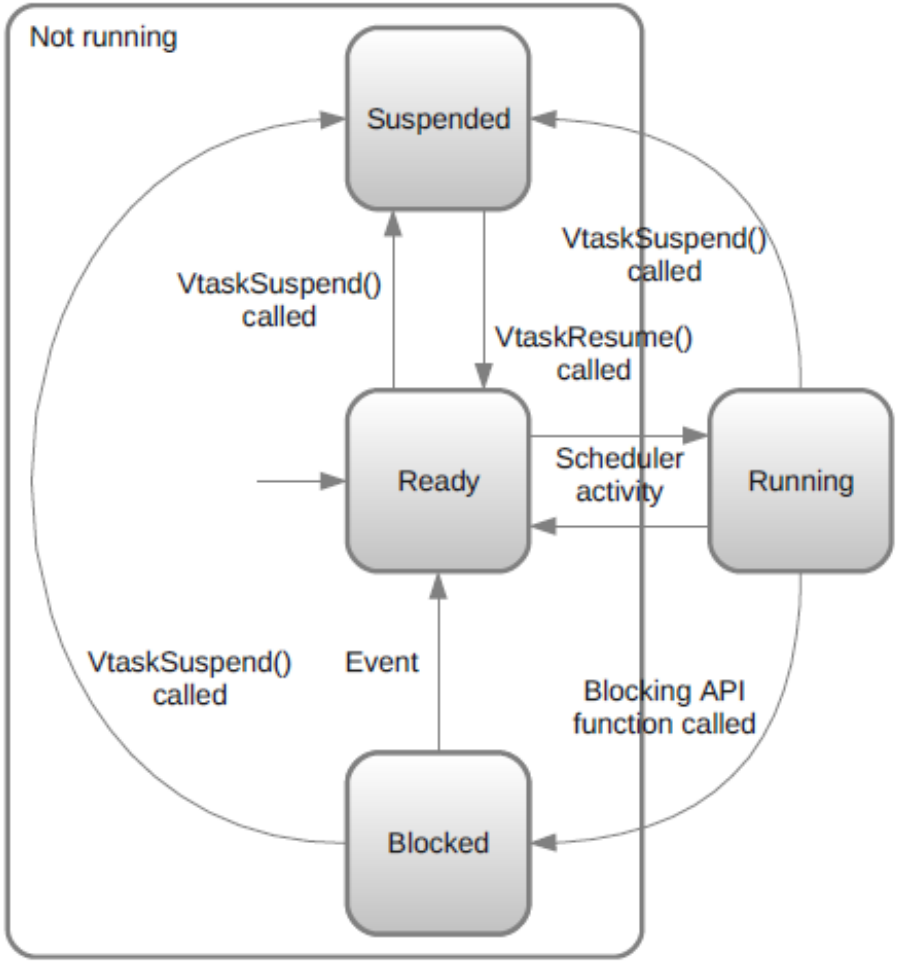
- state is set "Ready"

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)



Stack Memory

The MAIN Function

- The main function in FreeRTOS based project creates tasks.
- FreeRTOS will let you multi-task based on your tasks and their priority.

- Remember that a "task" is simply a "function" name of type: **void my_task(void* p)**

Each task has his own Stack:
Every variable you declare or memory allocate uses memory on the stack.

- The stack size of a task depends on the memory consumed by its local variables and function call depth.
- Please note that if your task (or function) uses printf, it consumes around 1024 bytes of stack.
- **At minimum however, you would need at least 256 bytes + your estimated stack space above**.
- If you don't allocate enough stack space, your CPU will run to an exception and/or freeze.

Creating a Task

The Task function itself:

```
void ATaskFunction( void *pvParameters )
{
   // do initilisation
   while (1)
   {
        // Task execution code
   }
}
```

Install 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
       unsigned short uxPriority,            // Priority
       xTaskHandle*   pxCreatedTask  );      // Pointer to receive
                                             //  Task handle
```

Simple Main and Task

- Simple task example that prints a message once a second.
- **vTaskStartScheduler()** never returns and FreeRTOS will begin servicing the tasks at this point.
- Every task must have an **infinite loop and NEVER EXIT**.

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

void main(void)
{
  xTaskCreate  (hello_world_task, „TestTask", 512, NULL, 1, NULL);

  vTaskStartScheduler();

 // never comes hers
}
```

FreeRTOS Configuration

in FreeRTOSConfig.h (http://www.freertos.org/a00110.html)

```
/* Basic Setup */
#define configUSE_PREEMPTION      1
#define configUSE_IDLE_HOOK       0
#define configUSE_TICK_HOOK       0
#define configCPU_CLOCK_HZ        ( 48000000 )                        // CPU CLK Generator
#define configPBA_CLOCK_HZ        ( 24000000 )                        // Peripheral Bus CLK
#define configTICK_RATE_HZ        ( ( portTickType ) 1000 )           // RTOS tick = 1 ms
#define configMAX_PRIORITIES      ( ( unsigned portBASE_TYPE ) 8 )    // Priorities
#define configMINIMAL_STACK_SIZE  ( ( unsigned portSHORT ) 256 )
#define configTOTAL_HEAP_SIZE     ( ( size_t ) ( 1024*25 ) )
#define configMAX_TASK_NAME_LEN   ( 20 )                              // limit the strings
...

/* Include / Exclude some API function. */
#define INCLUDE_vTaskPrioritySet            1
#define INCLUDE_uxTaskPriorityGet           1
#define INCLUDE_vTaskDelete                 1
#define INCLUDE_vTaskCleanUpResources       0
#define INCLUDE_vTaskSuspend                1
#define INCLUDE_vTaskDelayUntil             1
#define INCLUDE_vTaskDelay                  1
#define INCLUDE_xTaskGetCurrentTaskHandle   1
#define INCLUDE_xTaskGetSchedulerState      0
...
```

SDK: Done by modify BSP Configuration

Controlling Tasks

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

- If 2 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.

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.

- Remember that each function given above takes a parameter that decides how long you are willing to sleep. You could use this parameter as a timeout. For example, your logic may be: "I'm going to wait 1 second to receive something from the queue, otherwise I will <do whatever>".

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;
}
```

Delays: Give the next Task the chance to run …

void vTaskDelay(const TickType_t xTicksToDelay);

Delay a task for a given number of ticks

- Remember: in freeRTOSConfig.h: Tickrate = 1000 → 1 tick = 1 ms

void vTaskDelayUntil(TickType_t *pxPreviousWakeTime, const TickType_t xTimeIncrement);

Delay a task until a specified time.

```
void vTaskFunction(void * pvParameters )
{
TickType_t xLastWakeTime;

// Initialise the xLastWakeTime variable with the current time.
  xLastWakeTime = xTaskGetTickCount();

  for( ;; )
  {
     vTaskDelayUntil( &xLastWakeTime, 100 );      // Wait for the next cycle.
      // Perform actions here.
  }
}
```

Stop the scheduler

One method to create a critical section:

- prevent a task from preempting it
- but let interrupts to do their job

→ Stopping the scheduler , or stopping all other tasks.

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

```
// Create a critical section: Example
// Write the string to stdout
// Suspending the scheduler as a method of mutual exclusion. */
...

   vTaskSuspendAll();
   {
     printf( "%s", pcString );
     fflush( stdout );
   }
   xTaskResumeAll();

...
```

Software Timers: xTimerCreate

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

```
timers.h
  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
```

Example:

```
#include <timers.h>

TimerHandle_t xTimer;

void vTimerCallback ( TimerHandle_t pxTimer )                          // Timer callback
function
{
static cnt = 0;
//  id = (long) pvTimerGetTimerID(pxTimer); // Which timer expired?
  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();
}
```

Hand-on: Project 1: Simple FreeRTOS
Create a Zynq PS (Programmable System) for freeRTOS

- 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

The next step is to create a new block diagram. Select the "Add IP" link to add the first block,

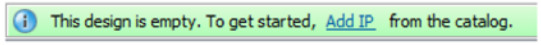
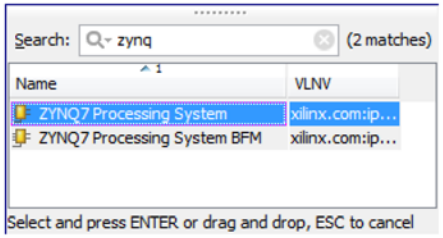
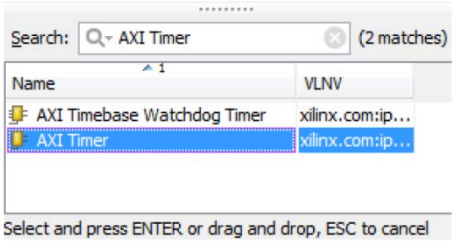


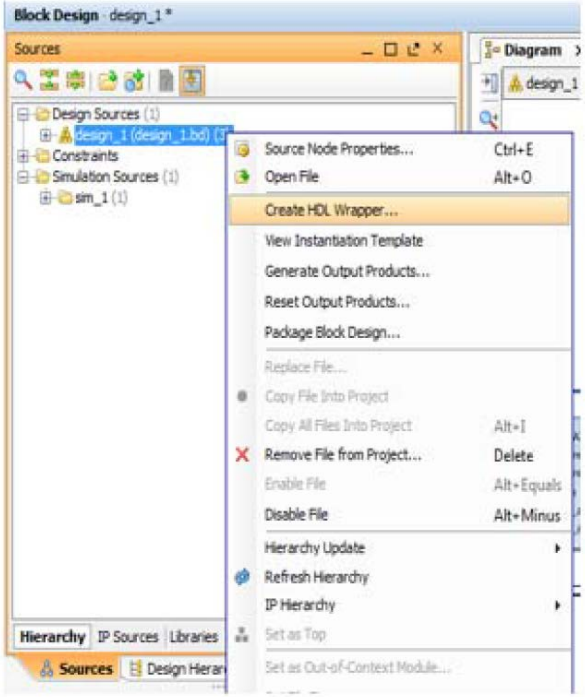
Figure 5. Add IP Block

Select the ZYNQ processing system and "Run Block Automation".

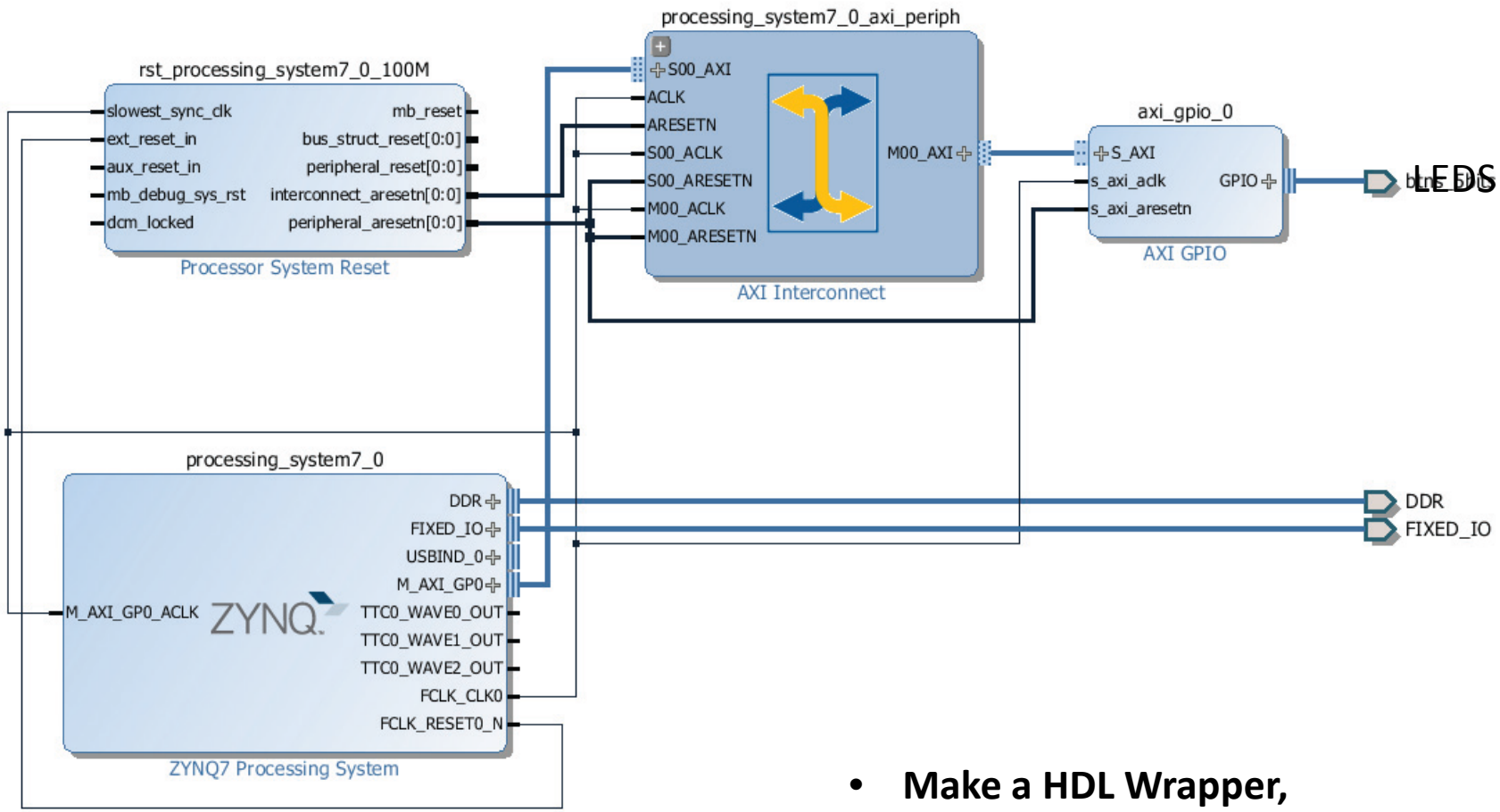




Then „Create Wrapper“



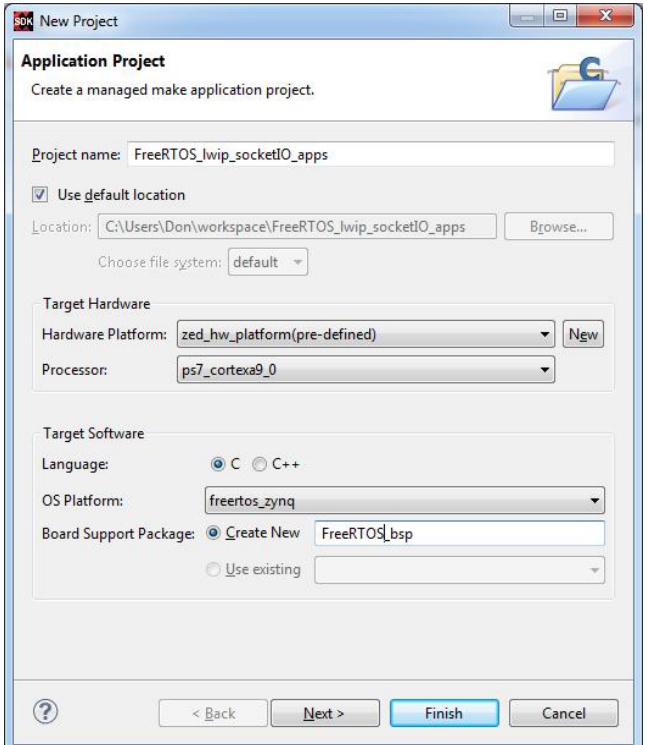
Project 1: A simple freeRTOS example



- **Make a HDL Wrapper,**
- **Implement the Design,**
- **Generate Bitstream,**
- **Export to SDK and launch SDK**

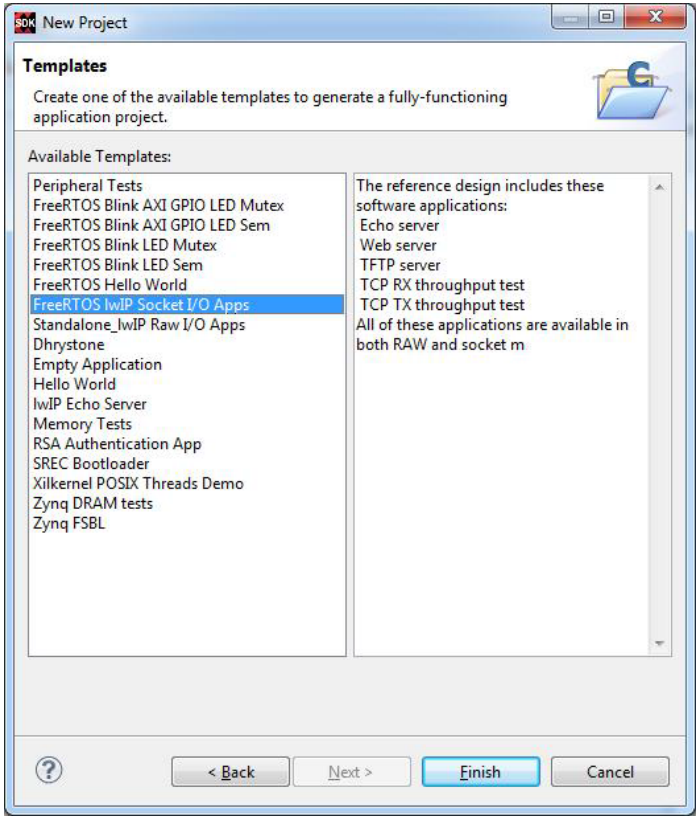
SDK: New Project

- Choose **File -> New -> Application Project** -> Select.
- New Application Project window opens up. Provide Project Name.
- Under Target Hardware tab, choose a Hardware Platform from dropdown list against Hardware Platform attribute. (Choice can be made to use pre-defined hardware platforms or create new hardware project, say zed_hw_platform)
- Choose the processor for which the application should be targeted.
- Under Target Software tab, select **freertos_zynq** as OS Platform. Name for Board Support Package will be populated based on the application project name.
- Accept the default or edit and ... **Click Next**.

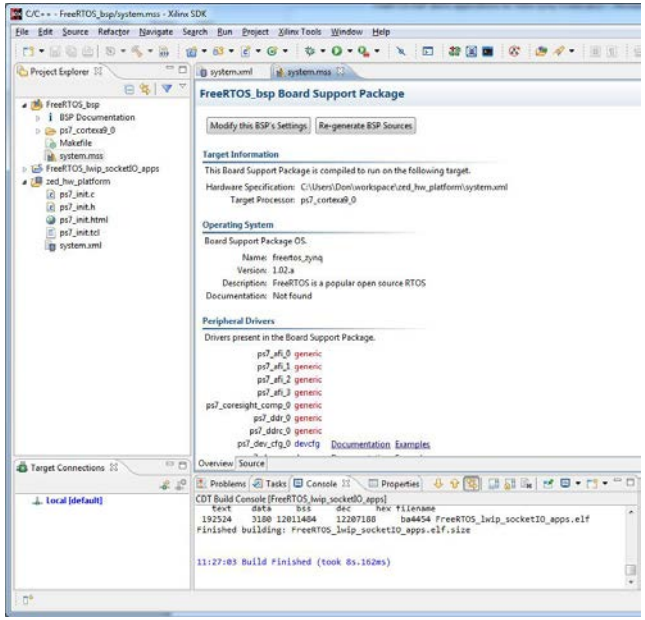


Select a Application template

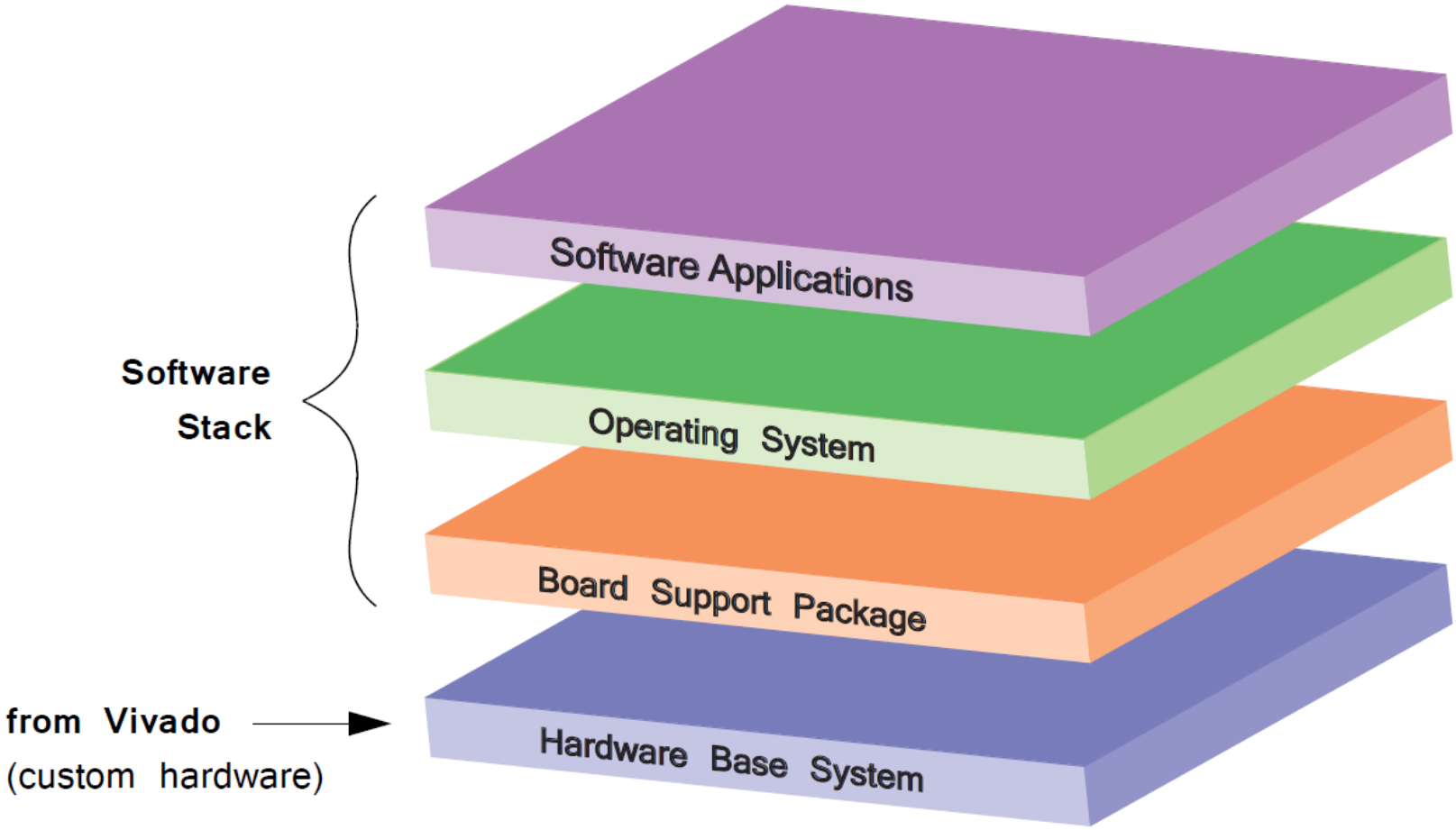
- Select any of the available FreeRTOS applications
- For first example: **“FreeRTOS Hello World”**
- Click Finish.



- SDK project will be created



Software Stack



First freeRTOS example

```
int tick=0;
```

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

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

  }
}
```

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

   //   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;
}
```

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

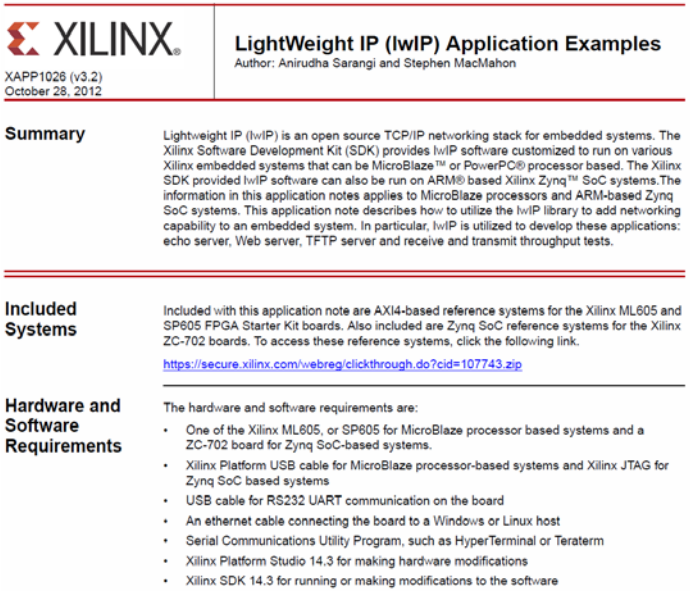
Default Base-Address: look in **xParameters.h** (0x4120_0000)

freeRTOS and lwIP in a FPGA

lwIP is a small independent implementation of the TCP/IP protocol. The focus of the lwIP TCP/IP implementation is to reduce resource usage while still having a full scale TCP. This makes lwIP suitable for use in embedded systems with tens of kilobytes of free RAM and room for around 40 kilobytes of code ROM.

Main features include:

- Protocols: IP, IPv6, ICMP, ND, MLD, UDP, TCP, IGMP, ARP, …
- DHCP client, DNS client, …
- SNMP agent
- APIs:
  - Berkeley-alike socket API
  - specialized APIs for enhanced performance,
- Applications: HTTP(S) server, SNTP client, SMTP(S) client, ping, TFTP, …



XILINX

XAPP1026 (v3.2)
October 28, 2012

LightWeight IP (lwIP) Application Examples
Author: Anirudha Sarangi and Stephen MacMahon

Summary

Lightweight IP (lwIP) is an open source TCP/IP networking stack for embedded systems. The Xilinx Software Development Kit (SDK) provides lwIP software customized to run on various Xilinx embedded systems that can be MicroBlaze™ or PowerPC® processor based. The Xilinx SDK provided lwIP software can also be run on ARM® based Xilinx Zynq™ SoC systems.The information in this application notes applies to MicroBlaze processors and ARM-based Zynq SoC systems. This application note describes how to utilize the lwIP library to add networking capability to an embedded system. In particular, lwIP is utilized to develop these applications: echo server, Web server, TFTP server and receive and transmit throughput tests.

Included Systems

Included with this application note are AXI4-based reference systems for the Xilinx ML605 and SP605 FPGA Starter Kit boards. Also included are Zynq SoC reference systems for the Xilinx ZC-702 boards. To access these reference systems, click the following link.

https://secure.xilinx.com/webreg/clickthrough.do?cid=107743.zip

Hardware and Software Requirements

The hardware and software requirements are:

- One of the Xilinx ML605, or SP605 for MicroBlaze processor based systems and a ZC-702 board for Zynq SoC-based systems.
- Xilinx Platform USB cable for MicroBlaze processor-based systems and Xilinx JTAG for Zynq SoC based systems
- USB cable for RS232 UART communication on the board
- An ethernet cable connecting the board to a Windows or Linux host
- Serial Communications Utility Program, such as HyperTerminal or Teraterm
- Xilinx Platform Studio 14.3 for making hardware modifications
- Xilinx SDK 14.3 for running or making modifications to the software

Project 2: lwIP

Install additional BSP

- Copy "HrFreertos_lwip_echo" to
  - **x:\Xilinx\SDK\2016.2\data\embeddedsw\lib\sw_apps**

- Re-Start SDK

- Create a SDK project (freeRTOS) based on this template

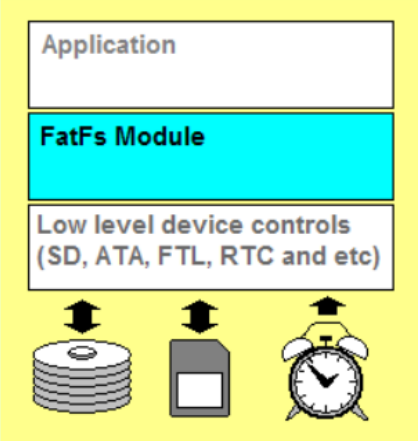
- Check for MAC address
  - IP Address
  - Tasks

- Test
  - Web Server
  - Echo Server

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.

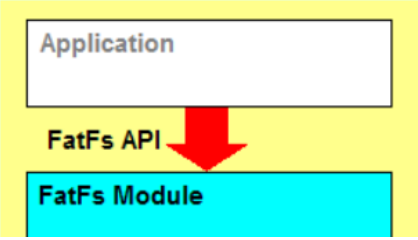


Features

- 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

Enable XilFatFS in the BSP

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


static 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;
}
```

```
Res = f_open (&hFile, "PixCalib.bin", FA_CREATE_ALWAYS | FA_WRITE );
if (Res)
{
          printf("SD: WRITE Pixel Calib Matrix: Open failed \n\r");
          return 0;
}


Res = f_write (&hFile, p, len, &wr);
Res = f_close(&hFile);
```