FreeRTOS Operating system and lwIP for SoC

1st Mesoamerican Workshop on Reconfigurable X-ray Scientific Instrumentation for Cultural Heritage

Luis Guillermo García Ordóñez

IAEA International Atomic Energy Agency
Atoms for Peace and Development

IUPAP Centenary YEARS 1922 - 2022

Gobierno de la República Guatemala
Secretaría Nacional de Ciencia y Tecnología

Carolina Academia Coactemalensis Inter Cæteras Orbis Conspicua

ICTP The Abdus Salam International Centre for Theoretical Physics

Outline

- Firmware development for microcontrollers
  - Bare Metal,
  - O.S. Based Embedded Systems
- FreeRTOS
  - Motivation for using FreeRTOS
- FreeRTOS in the Zynq
  - Integration of FreeRTOS on Xilinx hardware
- What is lwIP
- Practical applications (UDMA)

The Bare-metal approach:

Meet the superloop:

- Forever loop that sequences the set of tasks
- Polled or interrupt-based I/O
- Typical in standalone implementations

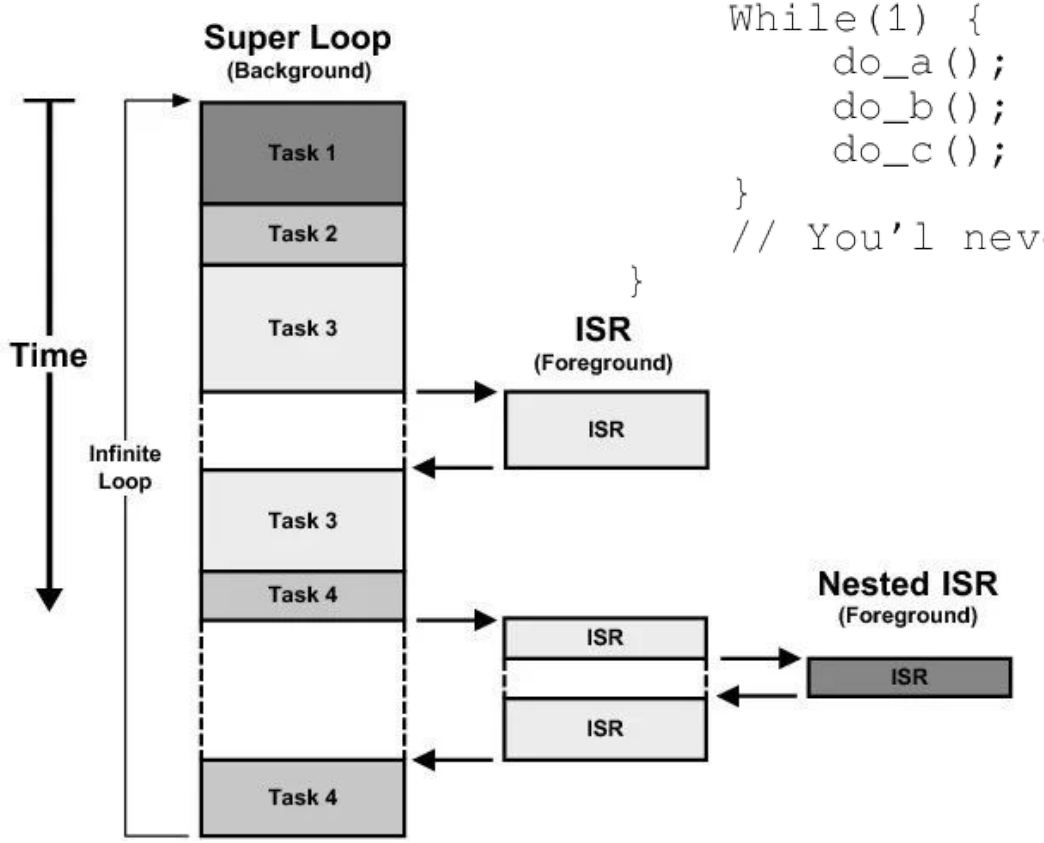
Pros:

- Simple
- No OS overhead

Cons:

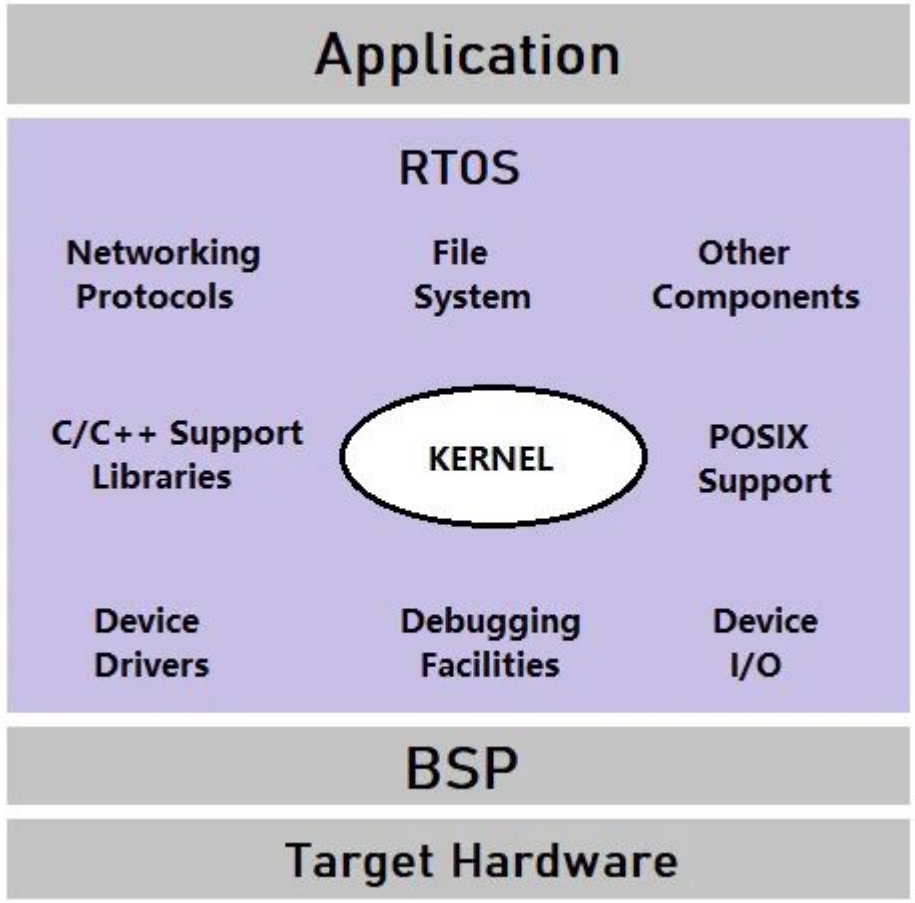
- Difficult to scale up (low number of tasks)
- Difficult to balance time and tasks priorities

```
int main() {
    init_system();
    …
    While(1) {
        do_a();
        do_b();
        do_c();
    }
    // You'l never get here
}
```



O.S. Based Embedded Systems

- Multi-threaded: multiple threads spawn to carry out multiple tasks concurrently
- Each task has different priority and timing requirements
- The operating system provides some hardware abstraction layer
- Extra services, such as a filesystem, network stack, ...
- **Pros:**
  - More modular architecture
  - Tasks can be pre-empted. Avoid priority inversion
- **Cons:**
  - More complex and extra overhead
  - Higher memory requirements
  - Thread execution is difficult to test
  - Usually not Determinist….. But



O.S. Based Embedded Systems

FreeRTOS

O.S. vs Real Time O.S. (RTOS):

Simple OS has a non-deterministic response time to external events, **RTOS however replies to all the external activities in minimal and**

- Born in 2003 and initially conceived for microcontrollers
  - Really light
  - Platform-Independent, Modular and Simple
  - Minimal processing overhead
    - FreeRTOS IRQ dispatch 10-50 cycles aprox.*
    - Embedded Linux IRQ dispatch = 100 cycles aprox.
    - Ported to a large number of architectures (e.g. ARM, AVR, RISC-V, and MicroBlaze)
- Currently is Amazon the company that stewards the development of the O.S.
- Open Source MIT license
- More information at www.FreeRTOS.org

*For ARM Cortex-M, it may vary depending of the architecture

Which I should choose?

Bare Metal

- An application that does not require task/thread preemption
- Where real-time deadline is not a requirement (deterministic behavior)
- Low-level applications which do not have large memory to fulfill the need of an operating system.
- When you don’t want to use third-party firmware and drivers to interfere with your application
- A low-cost application which can access all the registers of the hardware.

RTOS

- An application that needs task preemption and where interrupts and tasks need to be prioritized .i.e. hard real-time deadline requirement
- High-level application where the computing cost of the project is not a big deal.
- High memory usage and efficient processing are required.
- Applications in which modularity is an important point to be followed and code redundancy should be minimum.

FreeRTOS ecosystem of products:

- Amazon FreeRTOS for IoT devices
- Network communication stack
- Command Line Interface
- SSL/TLS security
- File systems (e.g. FAT32)

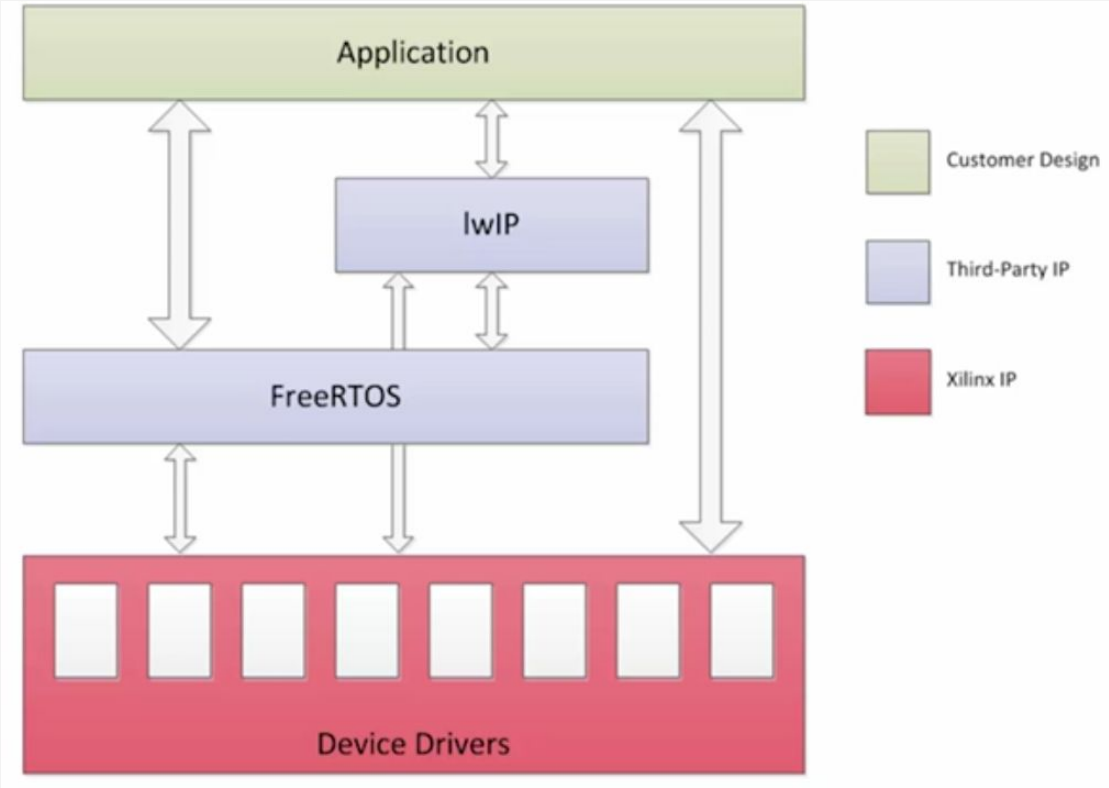
FreeRTOS in Xilinx tools

FreeRTOS completely integrated in Xilinx Software Development Flow
Provided as a BSP:

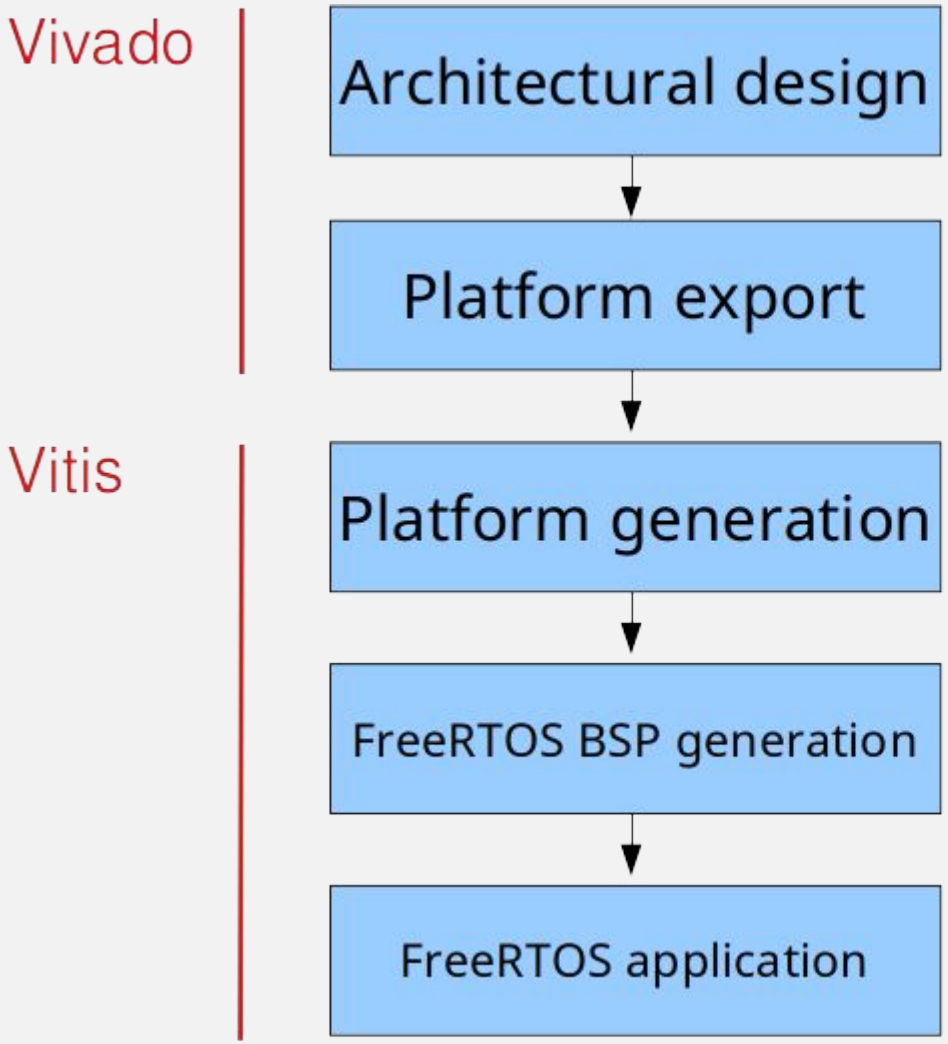
- Extension of the standalone BSP
  - All low level drivers can be directly used
- Includes the O.S. runtime

Optional extensions:

- Filesystem
- Network
- ...



FreeRTOS Design Flow



This information will be used for the generation of the appropriate drivers for the peripherals

It includes the standalone drivers plus the extra libraries selected

Based on the FreeRTOS API plus the peripheral drivers

- Through a header file: FreeRTOSConfig.h

```
#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)
```

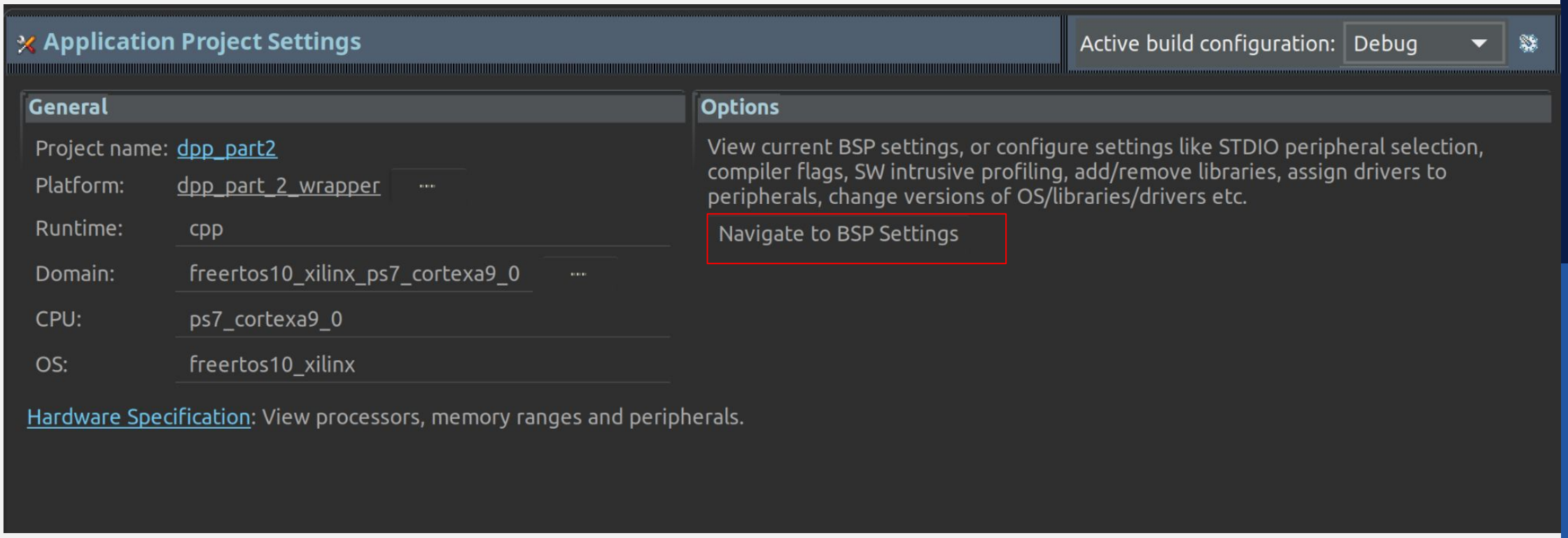
Tasks can be interrupted by others with higher priority

This will include a timer service task

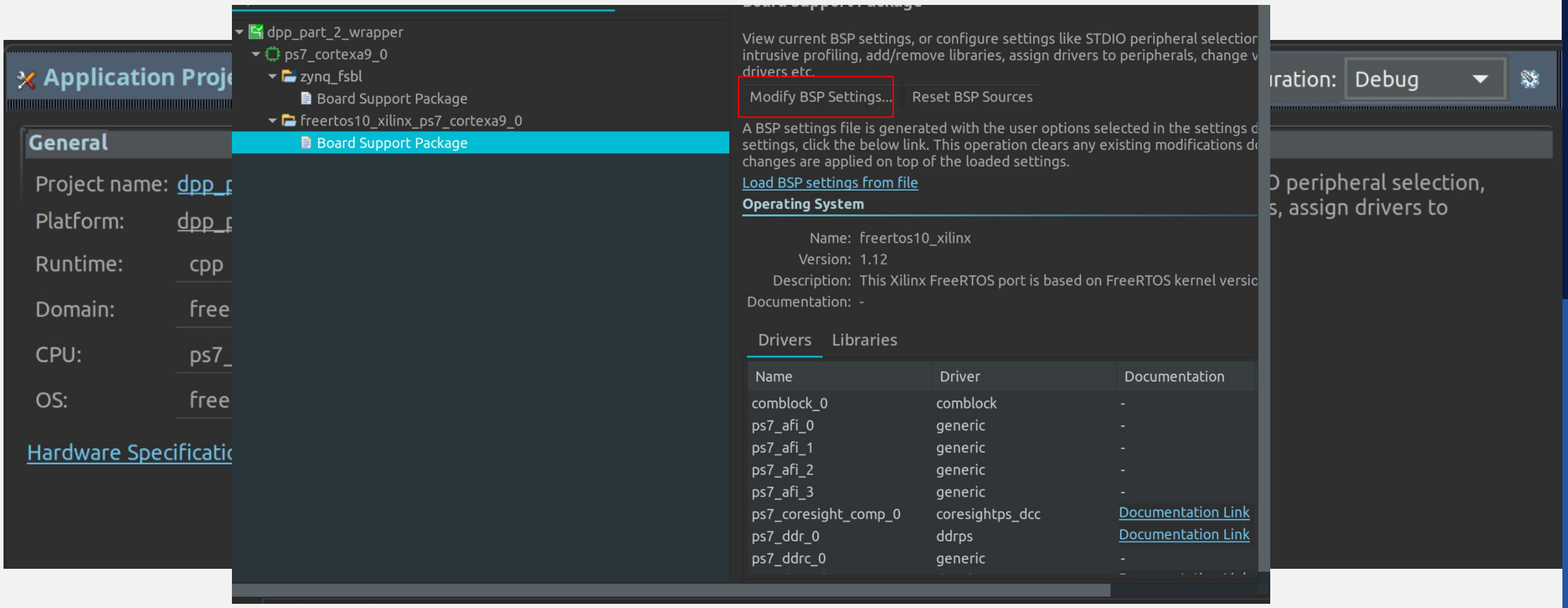
Hooks are used to trigger the execution of functions upon the happening of certain events

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

Xilinx configuration is through the mss file in the FreeRTOS BSP generated in Vitis

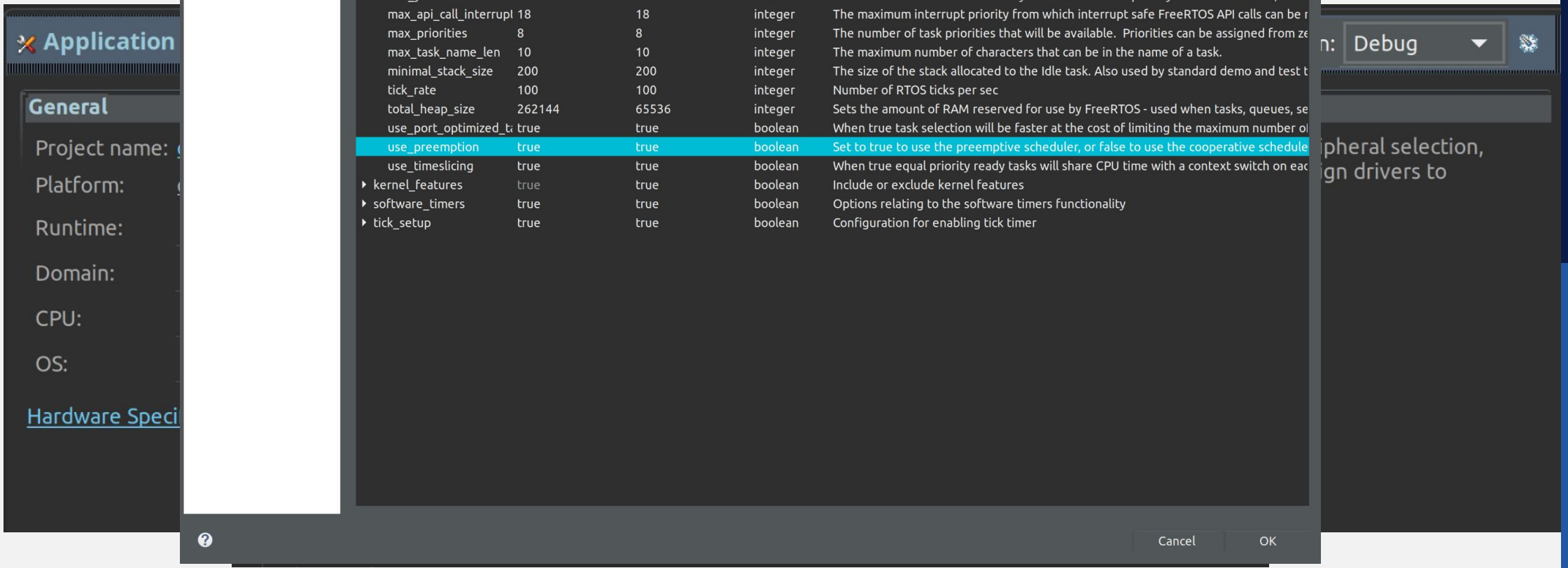


Xilinx configuration is through the mss file in the FreeRTOS BSP generated in Vitis

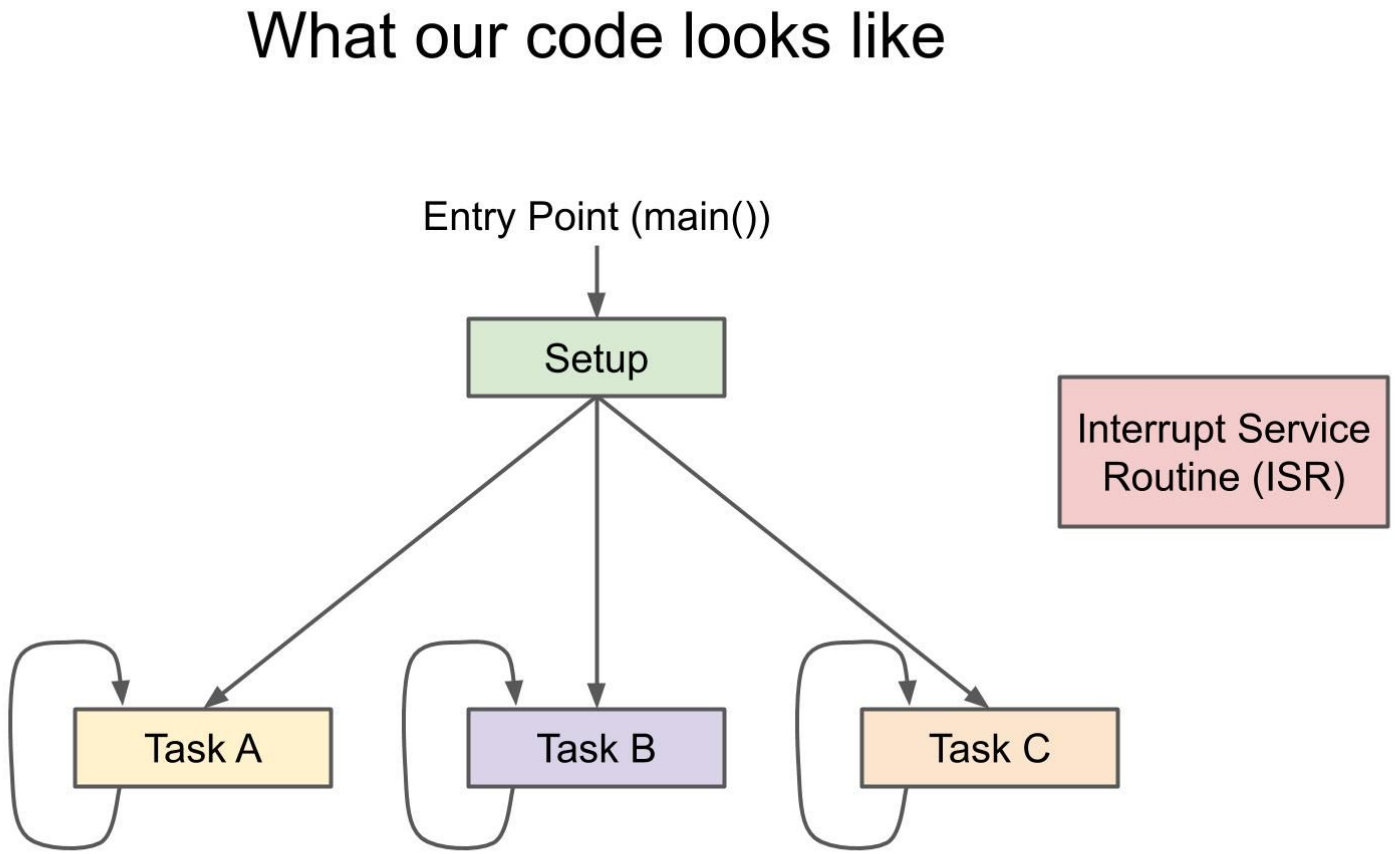


FreeRTOS C

Xilinx … e in the F…

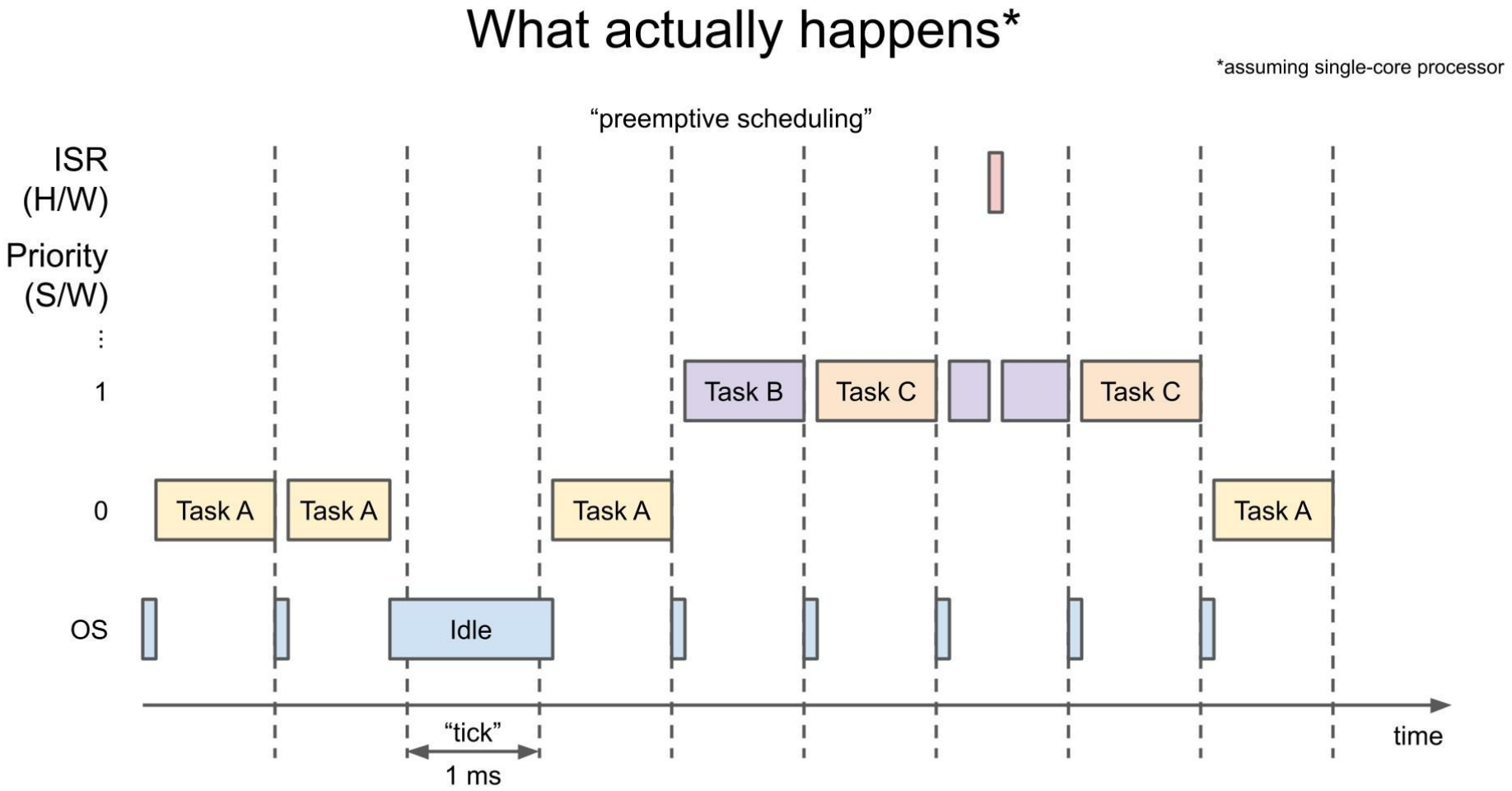


FreeRTOS Tasks



https://www.digikey.com/en/maker/projects/introduction-to-rtos-solution-to-part-3-task-scheduling

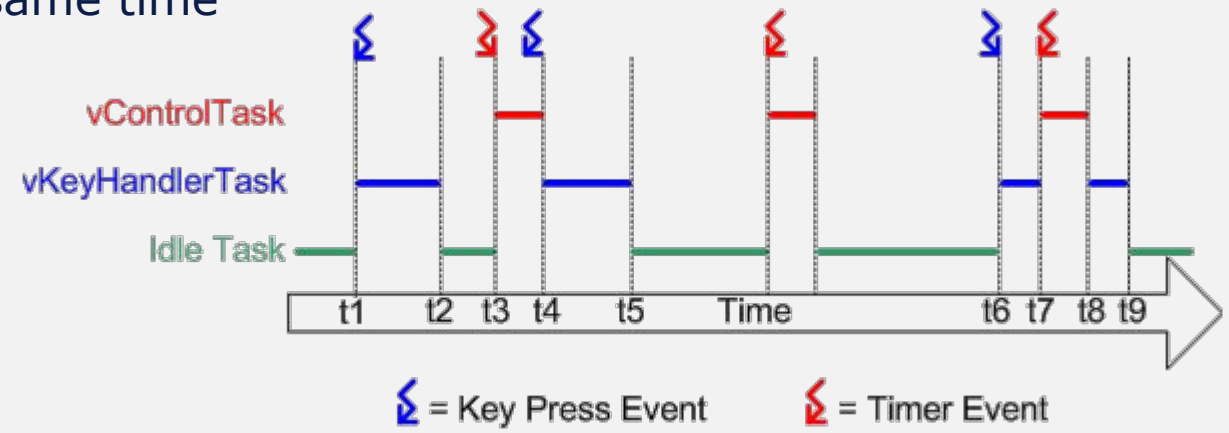
FreeRTOS Tasks



https://www.digikey.com/en/maker/projects/introduction-to-rtos-solution-to-part-3-task-scheduling

FreeRTOS tasks

- Every thread of execution is a task
- Tasks are independent between them. They have their own execution context (memory)
- Tasks are never called from the program
- Tasks are executed by the FreeRTOS scheduler **depending on their priorities and as a response to events**
- Only one task active at the same time
- Tasks never return
- There's a special IDLE task
  - No need to create it



A typical FreeRTOS application will look like this

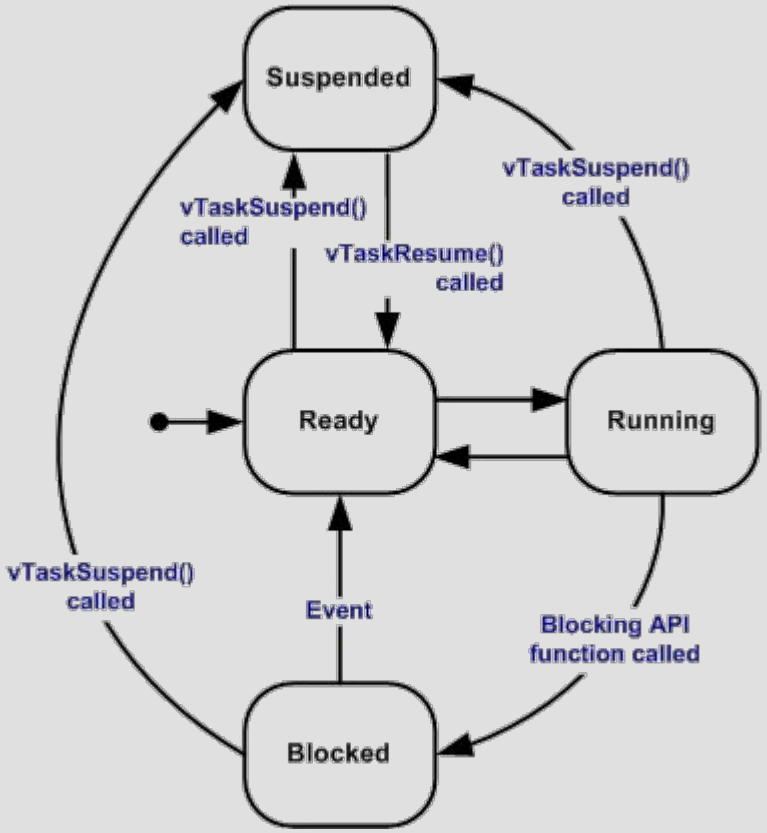
```
void main()
{
  xTaskCreate (Task_A, ….);
  xTaskCreate (Task_A, ….);
  xTaskCreate (Task_A, ….);
  xTaskStartScheduler ();
}
```

```
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();
  }
}
```

FreeRTOS task model



Tasks can be in different states of execution

Ready:
- When the task can be selected for execution, but is kept waiting since the CPU is busy with another task (depends on priority – next slide)

Running :
- Really executing the code

Blocked:
- Waiting for something:
  - An event. (e.g. a message has been received in a queue)
  - `vTaskDelay()` has been called so a certain time must pass.

Suspended:
- After calling `vTaskSuspend()`
- Can later be resumed using `xTaskResume()`

FreeRTOS priorities

- Tasks have priorities, used to the scheduler to select the most urgent one
- The range of different priorities is configurable in `"FreeRTOSConfig.h"`
  `configMAX_PRIORITIES`
- Tasks can change their own priority, as well as the priority of other tasks.
- The IDLE task is the one with the lowest priority `"task.h"`
  `tskIDLE_PRIORITY ( ( UBaseType_t ) 0U )`
- The FreeRTOS scheduler is preemptive:
  - If a task with a higher priority that the actual one is READY, then the RUNNING one will be evicted and moved to the READY state, while the former will start the execution

FreeRTOS tasks creation

Tasks are modelled after normal C functions e.g.

```
static void My_Task( void *myParameters );
```

- void return:
  - And remember in fact they should never return
- void pointer for arguments. Can be later casted to the right type

Since not called, they must be registered (created) into the scheduler

- The IDLE task is created automatically (special case)

Can also be destroyed at run-time

Some related functions:

- xtaskCreate()
- xtaskDelete()

Task creation

In order to create a Task:

```
BaseType_t xTaskCreate(TaskFunction_t pxTaskCode,
                       const char * const pcName,
                       const configSTACK_DEPTH_TYPE usStackDepth,
                       void * const pvParameters,
                       UBaseType_t uxPriority,
                       TaskHandle_t * const pxCreatedTask
```

pxTaskCode: pointer to the function that really implements the task
pcName: name assigned, mainly used for debug purposes
usStackDepth: refers to the local memory assigned to the task

- The **configMINIMAL_STACK_SIZE** parameter set in the `FreeRTOSConfig.h` configuration file

pvParameters: since no parameters are sent to the task
uxPriority: priority assigned to the task.

- This constant is defined as the minimum possible priority
- The lowest the number, the lowest the priority

pxCreatedTask: task handler

- From my past slide: `static void My_Task( void *myParameters );`

Task creation example

```
xTaskCreate(My_Task_Code,
(const char *) const “My_Task_Name”,
            configMINIMAL_STACK_SIZE,
            &myParameters,
             tskIDLE_PRIORIRTY,
             &My_Task
```

Hello World

3. Once the scheduler is started, functions will be executed depending on the scheduling policy

1. **main** function is normally used to create at least one task

2. The scheduler is a never-ending loop, so the program should never get to this point

```
#define TASK_NAME "HelloTask"
#define TASK_STACKDEPTH 1000
#define TASK_PRIORITY 1
#define TASK_PARAMETER NULL
#define TASK_HANDLE NULL


void sayHello( void *pvParameters )
{
  while (1) {
    printf("hello\n");
    vTaskDelay(1000 / portTICK_RATE_MS);
  }
}

int main( void )
{
  xTaskCreate( sayHello, TASK_NAME, TASK_STACKDEPTH,
               TASK_PARAMETER, TASK_PRIORITY, TASK_HANDLE );
  vTaskStartScheduler();

  printf("Something wrong\n");
  return 0;
}
```

“sayHello” task activation:

```
#define TASK_NAME "HelloTask"
#define TASK_STACKDEPTH 1000
#define TASK_PRIORITY 1
#define TASK_PARAMETER NULL
#define TASK_HANDLE NULL


void sayHello( void *pvParameters )
{
  while (1) {
    printf("hello\n");
    vTaskDelay(1000 / portTICK_RATE_MS);
  }
}

int main( void )
{
  xTaskCreate( sayHello, TASK_NAME, TASK_STACKDEPTH,
               TASK_PARAMETER, TASK_PRIORITY, TASK_HANDLE );
  vTaskStartScheduler();

  printf("Something wrong\n");
  return 0;
}
```

Once the scheduler is started, the task becomes ready

Since it’s the only task apart from the *IDLE* one (always present) it will be scheduled to *RUN*.

There are no other tasks but the IDLE one, with lower priority, so the task is always chosen to *RUN*.

But when the task executes vTaskDelay to force a waiting time, it becomes *BLOCKED*, waiting for the time to pass

Once the time has passed,

- The task will be moved to the *READY* state
- The *IDLE* task (priority 0) will be evicted
- The **sayHello** task will move to *RUN*

FreeRTOS Task Communication

Two mechanisms:

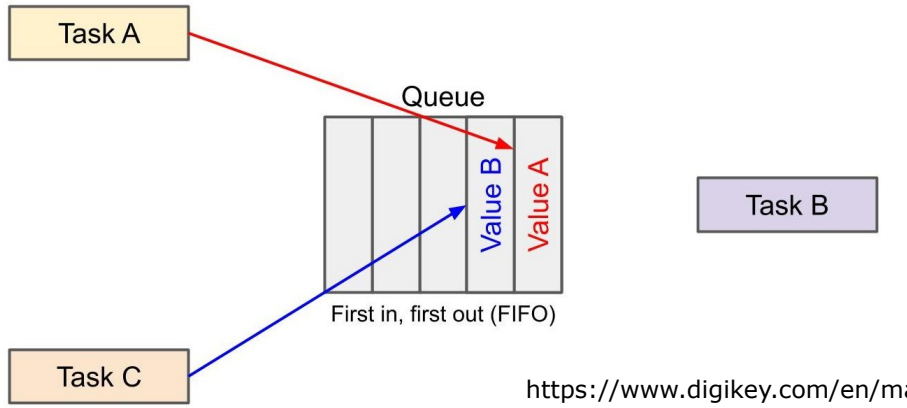
- **Global variables** which can be read from all tasks
- **Queues** as the main mechanism for inter-task communic

Queues:

- Asynchronous model of communication based on a FIFO
- Data can written to both the head and tail of the queue
- Arbitrary size and depth, but **defined at compile time**
- Items are passed by value → not zero copy
- Access can be blocking or non-blocking

Global variables and their risks

- The global variable is shared by all tasks
- Access control should be managed by the programmer
  - Since processes can be evicted, the state can be inconsistent
- E.g.:
- One process writes and another reads: Ok
- Two processes write
  - You may assume wrong states
  - Need for explicit synchronization mechanisms such as locks



https://www.digikey.com/en/maker/projects/introduction-to-rtos-solution-to-part-5-freertos-queue-example/72d2b361f7b94e0691d947c7c29a03c9

FreeRTOS queues

Queue creation:

```
xQueueHandle xQueueCreate (unsigned portBASE_TYPE uxQueueLength,
                           unsigned portBASE_TYPE uxItemSize)
```

Queue data insertion at the back of the queue:

```
portBASE_TYPE xQueueSendToBack (xQueueHandle xQueue,
                                const void * pvItemToQueue,
                                portTickType xTicksToWait)
```

If `xTicksToWait` is 0 it will return immediately if full otherwise it will wait

Data insertion at the front of the queue:

```
portBASE_TYPE xQueueSendToFront (xQueueHandle xQueue,
                                 const void * pvItemToQueue,
                                 portTickType xTicksToWait)
```

Data extraction:

```
portBASE_TYPE xQueueReceive (xQueueHandle xQueue,
                             void * pvBuffer,
                             portTickType xTicksToWait)
```

The producer-consumer example

```
xQueueHandle queue;

void producer( void *pvParameters ) {
  int value = 0;

  while (1) {
    xQueueSendToBack(queue, &value, 0);
    value++;
    vTaskDelay (1000 / portTICK_RATE_MS);
  }
}

void consumer( void *pvParameters ){
  int value;

  while (1) {
    xQueueReceive(queue, &value, portMAX_DELAY);
    printf("value received: %d\n", value);
    vTaskDelay (1000 / portTICK_RATE_MS);
  }
}


int main( void ) {
  queue = xQueueCreate(100, sizeof(int));

  xTaskCreate( producer, P_TASK_NAME, TASK_STACKDEPTH, TASK_PARAMETER,
           TASK_PRIORITY, TASK_HANDLE );
  xTaskCreate( consumer, C_TASK_NAME, TASK_STACKDEPTH, TASK_PARAMETER,
           TASK_PRIORITY, TASK_HANDLE );
  vTaskStartScheduler();
}
```

Queue declaration

If the queue is full, it will return immediately

Blocking read

Queue creation with limited size

Be careful with priorities

Another Exampl

Vitis Implementation

```c
int main()
{
    sys_thread_new("main_thrd", (void(*)(void*))main_thread, 0,
                   THREAD_STACKSIZE,
                   DEFAULT_THREAD_PRIO);
    vTaskStartScheduler();
    while(1);
    return 0;
}

void network_thread(void *p)…

int main_thread()
{
#if LWIP_DHCP==1
    int mscnt = 0;
#endif

    /* initialize lwIP before calling sys_thread_new */
    lwip_init();

    /* any thread using lwIP should be created using sys_thread_new */
    sys_thread_new("NW_THRD", network_thread, NULL,
        THREAD_STACKSIZE,
            DEFAULT_THREAD_PRIO);

#if LWIP_DHCP==1…
#endif
    vTaskDelete(NULL);
    return 0;
}
```

Another Exampl

Vitis Implementation

```
int main()
{
    sys_thread_new("main_thrd", (void(*)(void*))main_thread, 0,
                   THREAD_STACKSIZE,
                   DEFAULT_THREAD_PRIO);
    vTaskStartScheduler();
    while(1);
    return 0;
}

void network_thread(void *p)…

int main_thread()
{
#if LWIP_DHCP==1
    int mscnt = 0;
#endif

    /* initialize lwIP before calling sys_thread_new */
    lwip_init();

    /* any thread using lwIP should be created using sys_thread_new */
    sys_thread_new("NW_THRD", network_thread, NULL,
        THREAD_STACKSIZE,
            DEFAULT_THREAD_PRIO);

#if LWIP_DHCP==1…
#endif
    vTaskDelete(NULL);
    return 0;
}
```

Another Exampl

Vitis Implementation

```
int main()
{
    sys_thread_new("main_thrd", (void(*)(void*))main_thread, 0,
                   THREAD_STACKSIZE,
                   DEFAULT_THREAD_PRIO);
    vTaskStartScheduler();
    while(1);
```

```
sys_thread_t sys_thread_new( const char *pcName, void( *pxThread )( void *pvParameters ), void *pvArg, int iStackSize, int iPriority )
{
xTaskHandle xCreatedTask;
portBASE_TYPE xResult;
sys_thread_t xReturn;

    xResult = xTaskCreate( pxThread, ( const char * const) pcName, iStackSize, pvArg, iPriority, &xCreatedTask );

    if( xResult == pdPASS )
    {
        xReturn = xCreatedTask;
    }
    else
    {
        xReturn = NULL;
    }

    return xReturn;
}
```

```
#endif
    vTaskDelete(NULL);
    return 0;
}
```

Another Exampl

Vitis Implementation

Wait, what is this, Network?

```
int main()
{
    sys_thread_new("main_thrd", (void(*)(void*))main_thread, 0,
                   THREAD_STACKSIZE,
                   DEFAULT_THREAD_PRIO);
    vTaskStartScheduler();
    while(1);
    return 0;
}

void network_thread(void *p)…

int main_thread()
{
#if LWIP_DHCP==1
    int mscnt = 0;
#endif

    /* initialize lwIP before calling sys_thread_new */
    lwip_init();

    /* any thread using lwIP should be created using sys_thread_new */
    sys_thread_new("NW_THRD", network_thread, NULL,
        THREAD_STACKSIZE,
            DEFAULT_THREAD_PRIO);

#if LWIP_DHCP==1…
#endif
    vTaskDelete(NULL);
    return 0;
}
```

Another Example

Vitis Implementation

UDMA?

IP ADDRESS?

```
void network_thread(void *p)
{
    struct netif *netif;
    /* the mac address of the board. this should be unique per board */
    unsigned char mac_ethernet_address[] = { 0x00, 0x0a, 0x35, 0x00, 0x01, 0x02 };
    ip_addr_t ipaddr, netmask, gw;
#if LWIP_DHCP==1
    int mscnt = 0;
#endif

    netif = &server_netif;

    xil_printf("\r\n\r\n");
    xil_printf("-----UDMA Server------\r\n");

#if LWIP_DHCP==0
    /* initialize IP addresses to be used */
    IP4_ADDR(&ipaddr,  192, 168, 1,  10);
    IP4_ADDR(&netmask, 255, 255, 255, 0);
    IP4_ADDR(&gw,      192, 168, 1,  1);
#endif

    /* print out IP settings of the board */

#if LWIP_DHCP==0
    print_ip_settings(&ipaddr, &netmask, &gw);
    /* print all application headers */
#endif

#if LWIP_DHCP==1
    ipaddr.addr = 0;
    gw.addr = 0;
    netmask.addr = 0;
#endif
```

lwIP (lightweight IP)

Lightweight IP (lwIP)

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

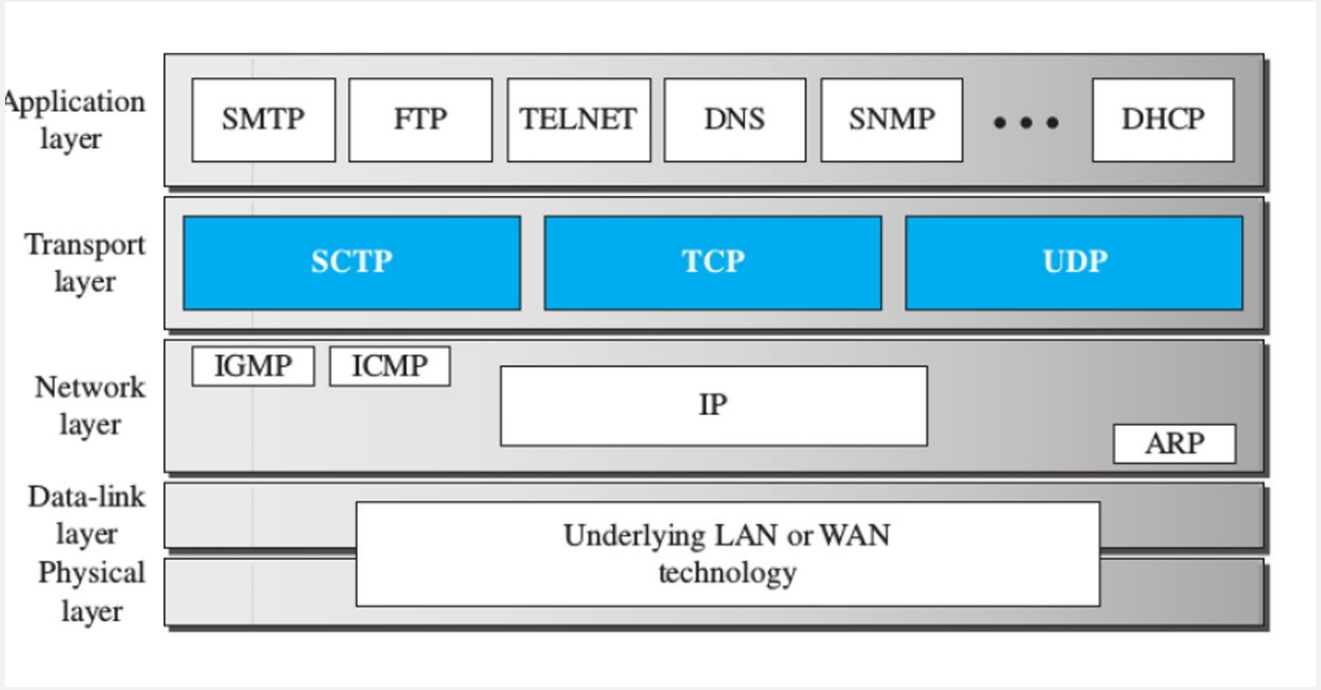
Supports a large number of protocols and APIs

• 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 Vitis

Application level

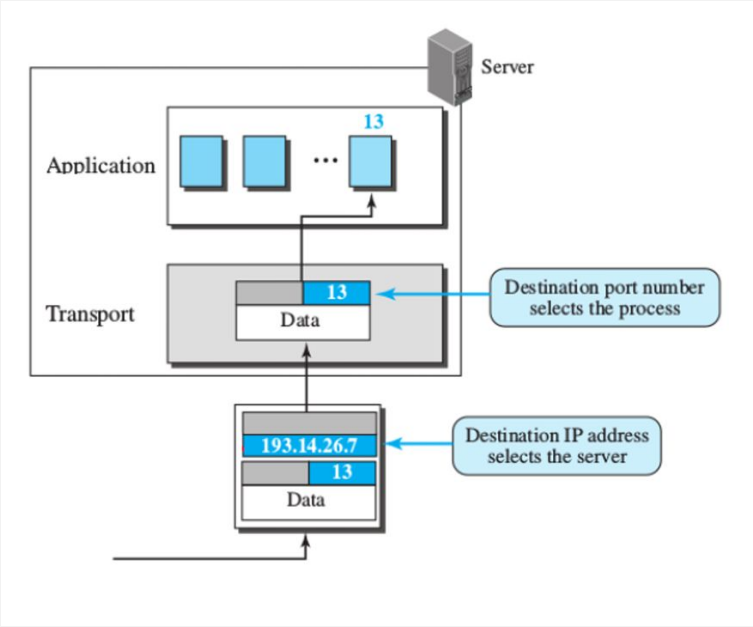
• HTTP(S) server, SNTP client, SMTP(S) client, ping, TFTP, …



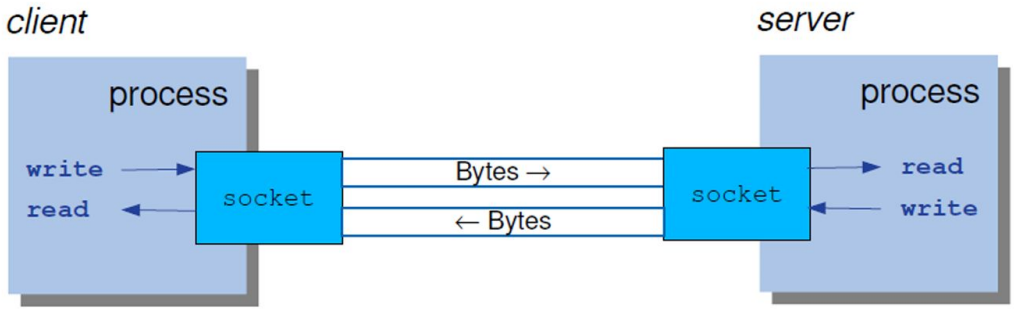
The network design is organized as a layer stack.
• Each layer provides a set of services to the upper layer and requires services from the lower layer.

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 Soc




```
lwip_socket(AF_INET, SOCK_STREAM, 0)
```



lwIP

Integration with freeRTOS

It is easier to understand with an example:

Start a task (e.g. `network_thread`)

1. Initializes lwip
2. Configures a network interface
3. Start the interface and a reception task
4. Install any other network tasks
   a. (In this example: `udma_application_thread`)
5. Finally the start up task deletes itself.

After initialization two threads are active:

- Reception task
- UDMA application

```
sys_thread_new("NW_THRD", network_thread, NULL,
    THREAD_STACKSIZE,
        DEFAULT_THREAD_PRIO);
```

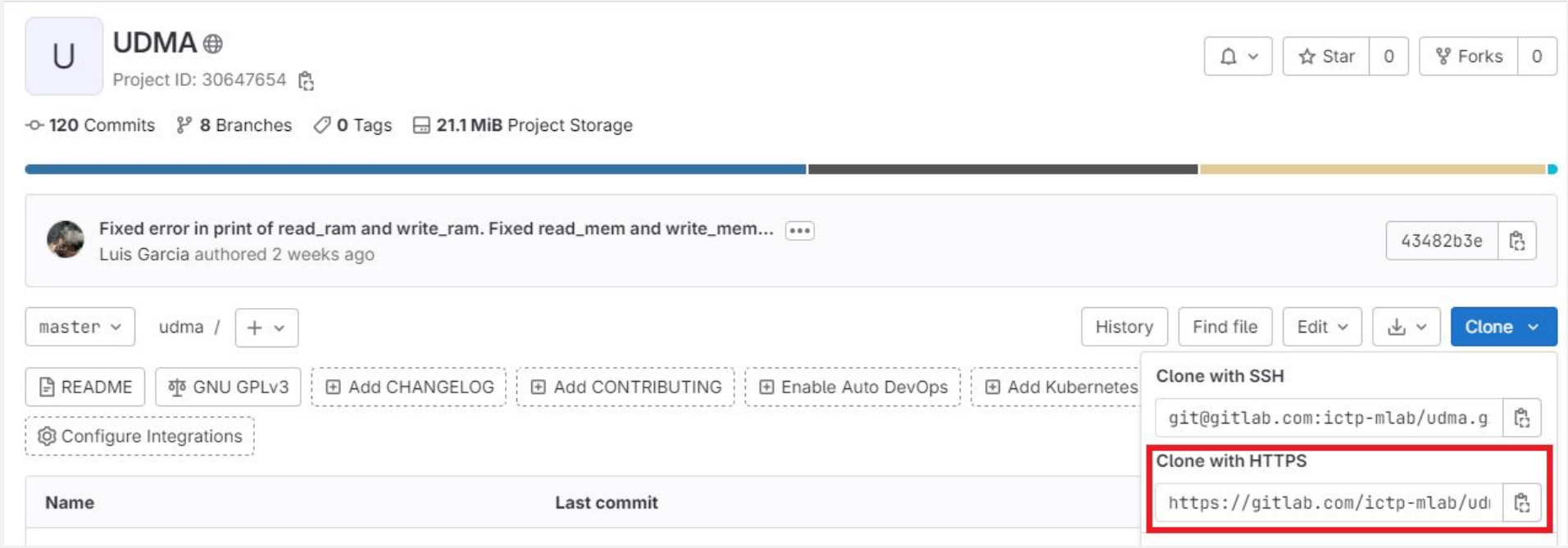
```
void network_thread(void *p)
    netif = &server_netif;

    xil_printf("\r\n\r\n");
    xil_printf("-----UDMA Server------\r\n");

#if LWIP_DHCP==0
    /* initialize IP addresses to be used */
    IP4_ADDR(&ipaddr,  192, 168, 1,  10);
    IP4_ADDR(&netmask, 255, 255, 255, 0);
    IP4_ADDR(&gw,      192, 168, 1,  1);
#endif

    /* print out IP settings of the board */

#if LWIP_DHCP==0
    print_ip_settings(&ipaddr, &netmask, &gw);
    /* print all application headers */
#endif

#if LWIP_DHCP==1 …
#endif

    /* 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)) {…

    netif_set_default(netif);

    /* specify that the network if is up */
    netif_set_up(netif);

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

#if LWIP_DHCP==1 …
#else
    xil_printf("\r\n");
    xil_printf("%20s %6s %s\r\n", "Server", "Port", "Connect With..");
    xil_printf("%20s %6s %s\r\n", "--------------------", "------", "--------------------");

    print_app_header();
    xil_printf("\r\n");
    sys_thread_new("udmad", udma_application_thread, 0,
        THREAD_STACKSIZE,
        DEFAULT_THREAD_PRIO);
    vTaskDelete(NULL);
#endif
```

And what is UDMA?

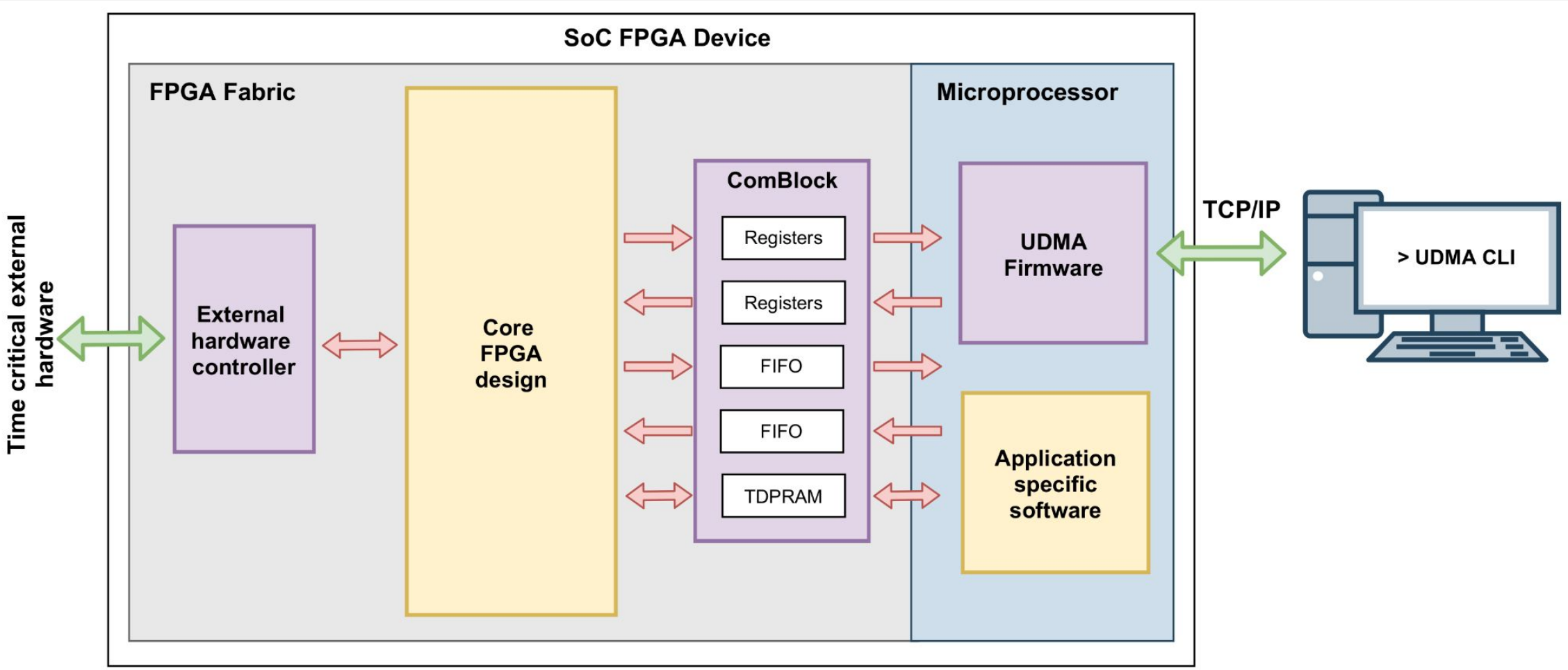
The Universal Direct Memory Access (UDMA) is a remote control suite developed at ICTP-MLAB for interfacing a PC with custom logic in a SoC-FPGA. It was tested inside FreeRTOS on top of lwIP.

The communication with the FPGA is done through the ComBlock.

https://gitlab.com/ictp-mlab/udma



SoC-FPGA of the experimental hardware platform.



Board communication

connect	Create the connect command to allow communication with the board via Ethernet
log	Starts serial logging to debug the transmission and processing of the messages
udma	Create the x_udma command to pass the UDMA instruction to the specified LRA

> Note: UDMA function is not completely implemented and must not be used unless specified in the release notes

Comblock Read

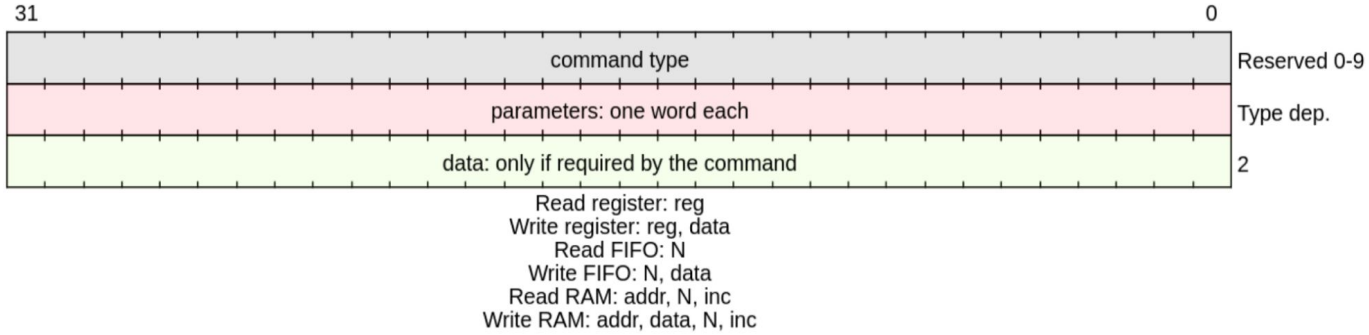
x_read_fifo	Create the x_read_fifo command to allow reading the FIFO of the Comblock
x_read_mem	Create the x_read_ram command to allow reading the RAM of the Comblock
x_read_ram	Create the x_read_ram command to allow reading the RAM of the Comblock
x_read_reg	Create the x_read_reg command to allow reading registers from the Comblock

Comblock Write

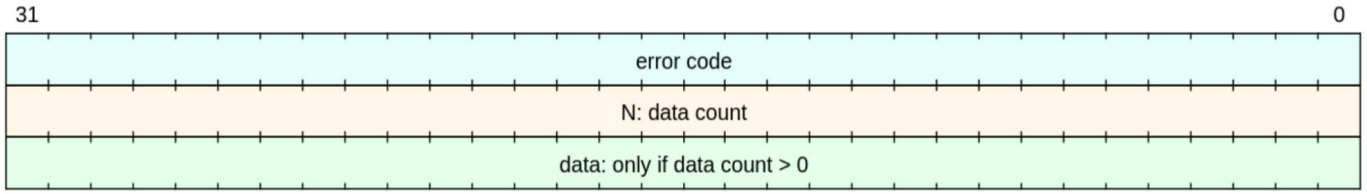
x_write_fifo	Create the x_write_fifo command to allow writing the FIFO of the Comblock
x_write_mem	Create the x_write_ram command to allow writing the RAM of the Comblock
x_write_ram	Create the x_write_ram command to allow writing the RAM of the Comblock
x_write_reg	Create the x_write_reg command to allow writing the registers of the Comblock

Communication protocol

The communication between the PC and the board relies over Ethernet TCP/IP. Using sockets you can connect to the board from any program. To send comands you must follow the following structure:



The board will always answer with a response with the error flag of the command, data count and data if required.



UDMA implementation on Jupyter Notebook

Lab3_NB.ipynb ●

smr-3983 > Labs > Lab3_UDMA > scripts > Lab3_NB.ipynb > Lab 3 - UDMA > Interfacing with hardware via UDMA > Connecting to ZedBoard > connectionStatus = 0

+ Code + Markdown | Run All Restart Clear All Outputs | Variables Outline ··· Python 3.8.12

```python
# MLab UDMA library
import udma

# All python libraries
from struct import pack, unpack
from time import sleep
```

[ ] Python

Interfacing with hardware via UDMA

Setting up UDMA and ZedBoard parameters

1. Set the IP address and port of your ZedBoard development board to match the settings specified in the `main.c` file of your Vitis project.

```python
IP_ADDRESS = '192.168.1.10'
IP_PORT = 7
```

[ ] Python

2. The **UDMA** class instance is being initialized with the provided IP settings. In this step, a **UDMA** object is created and assigned the name `zedBoard`. This name can be chosen arbitrarily.

```python
zedBoard = udma.UDMA_CLASS(IP_ADDRESS, IP_PORT)
```

[ ] Python

+ Code + Markdown

Questions?

1st Mesoamerican Workshop on Reconfigurable X-ray Scientific Instrumentation for Cultural Heritage

IAEA International Atomic Energy Agency
Atoms for Peace and Development

IUPAP Centenary YEARS 1922 - 2022

Gobierno de la República Guatemala
Secretaría Nacional de Ciencia y Tecnología

Academia Caroliana Orbis Conspicua Interamericana Guatemalensis

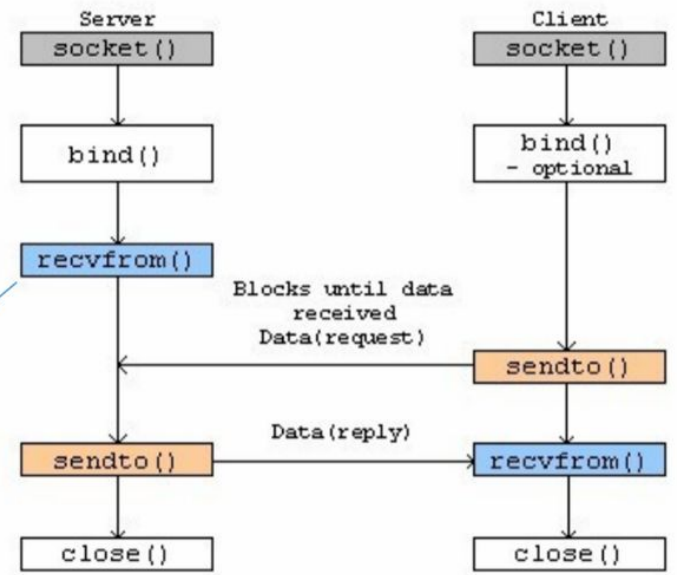
ICTP The Abdus Salam International Centre for Theoretical Physics

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



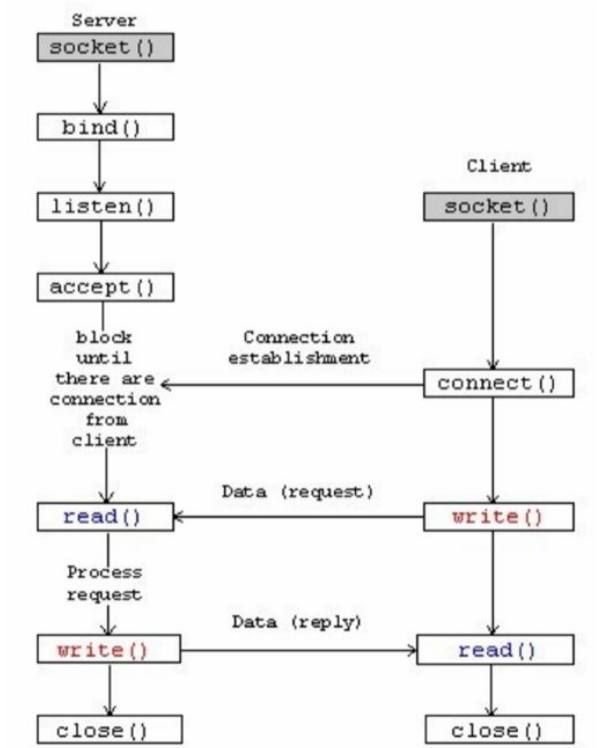

```
void echo_application_thread()
{
        int sock, new_sd;
        struct sockaddr_in address, remote;
        int size;

        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

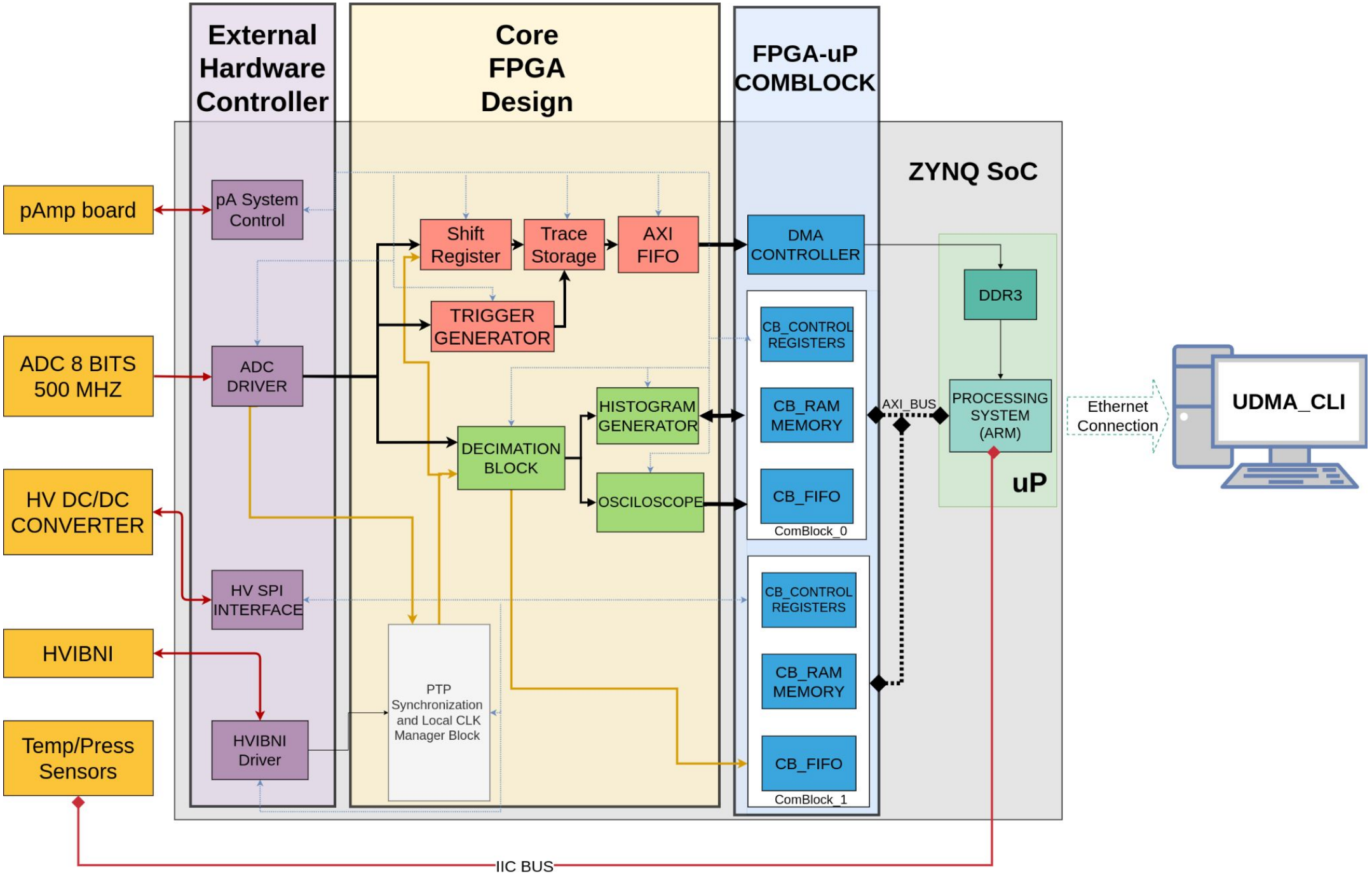
- Complex protocol with multiple phases
  - higher latency, lower throughput
  - Connection control

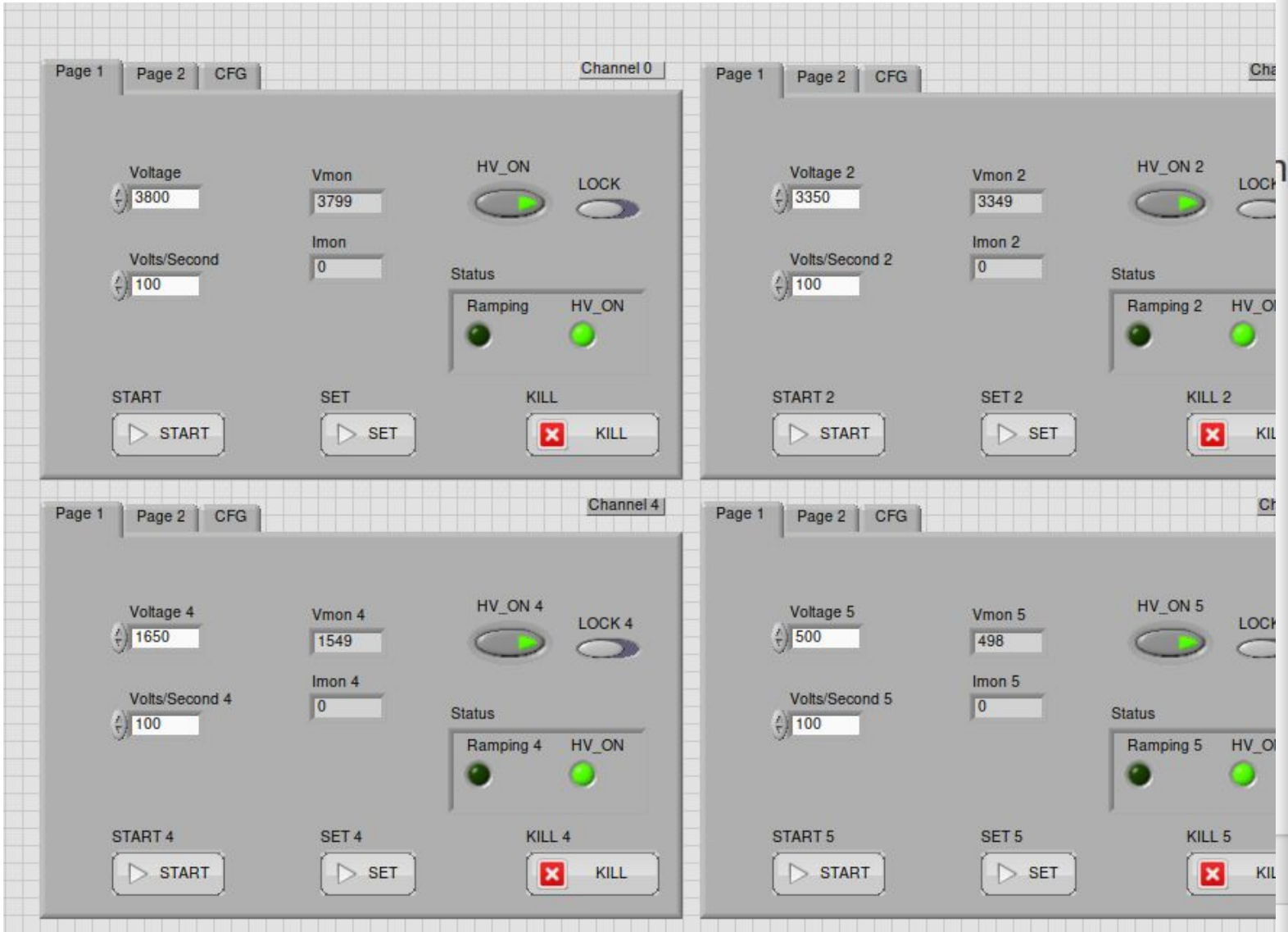


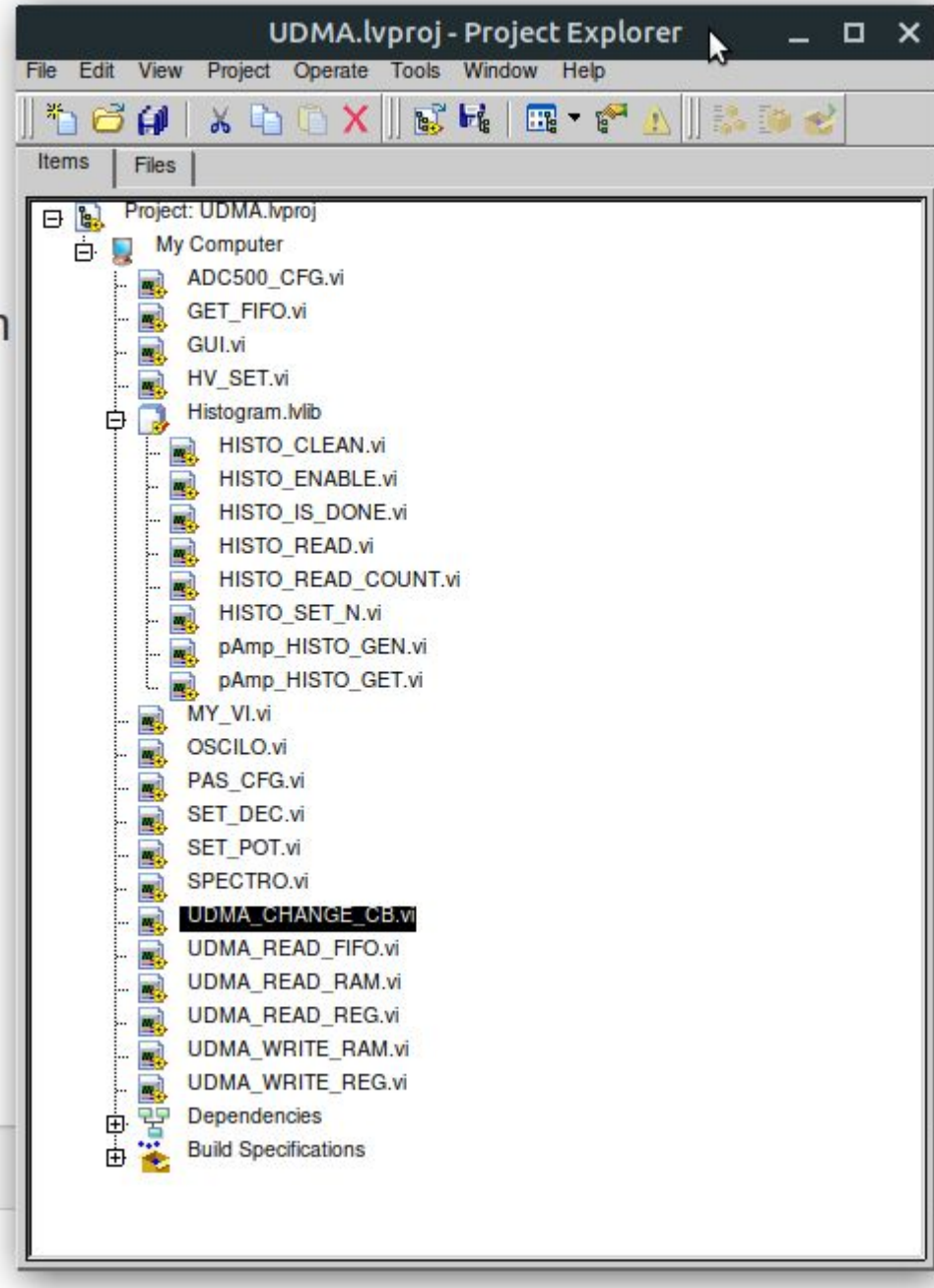
Used when loosing information can't be tolerated.
Example: HTTP, E-mail, binary Data, …

```
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);
}
```


External Hardware Controller
Core FPGA Design
FPGA-uP COMBLOCK
ZYNQ SoC
pAmp board
ADC 8 BITS 500 MHZ
HV DC/DC CONVERTER
HVIBNI
Temp/Press Sensors
pA System Control
ADC DRIVER
HV SPI INTERFACE
HVIBNI Driver
Shift Register
Trace Storage
AXI FIFO
TRIGGER GENERATOR
DECIMATION BLOCK
HISTOGRAM GENERATOR
OSCILOSCOPE
PTP Synchronization and Local CLK Manager Block
DMA CONTROLLER
CB_CONTROL REGISTERS
CB_RAM MEMORY
CB_FIFO
ComBlock_0
CB_CONTROL REGISTERS
CB_RAM MEMORY
CB_FIFO
ComBlock_1
AXI_BUS
DDR3
PROCESSING SYSTEM (ARM)
uP
Ethernet Connection
UDMA_CLI
IIC BUS


Page 1
Page 2
CFG
Channel 0
Voltage
3800
Vmon
3799
HV_ON
LOCK
Imon
0
Volts/Second
100
Status
Ramping
HV_ON
START
SET
KILL
Voltage 2
3350
Vmon 2
3349
HV_ON 2
Imon 2
0
Volts/Second 2
100
Ramping 2
START 2
SET 2
KILL 2
Channel 4
Voltage 4
1650
Vmon 4
1549
HV_ON 4
LOCK 4
Imon 4
0
Volts/Second 4
100
Ramping 4
START 4
SET 4
KILL 4
Voltage 5
500
Vmon 5
498
HV_ON 5
Imon 5
0
Volts/Second 5
100
Ramping 5
START 5
SET 5
KILL 5


UDMA.lvproj - Project Explorer
File
Edit
View
Project
Operate
Tools
Window
Help
Items
Files
Project: UDMA.lvproj
My Computer
ADC500_CFG.vi
GET_FIFO.vi
GUI.vi
HV_SET.vi
Histogram.lvlib
HISTO_CLEAN.vi
HISTO_ENABLE.vi
HISTO_IS_DONE.vi
HISTO_READ.vi
HISTO_READ_COUNT.vi
HISTO_SET_N.vi
pAmp_HISTO_GEN.vi
pAmp_HISTO_GET.vi
MY_VI.vi
OSCILO.vi
PAS_CFG.vi
SET_DEC.vi
SET_POT.vi
SPECTRO.vi
UDMA_CHANGE_CB.vi
UDMA_READ_FIFO.vi
UDMA_READ_RAM.vi
UDMA_READ_REG.vi
UDMA_WRITE_RAM.vi
UDMA_WRITE_REG.vi
Dependencies
Build Specifications