IP2022

Technical Introduction

August 1, 2006

Stream Engine by UBICOM

UBICOM

Agenda

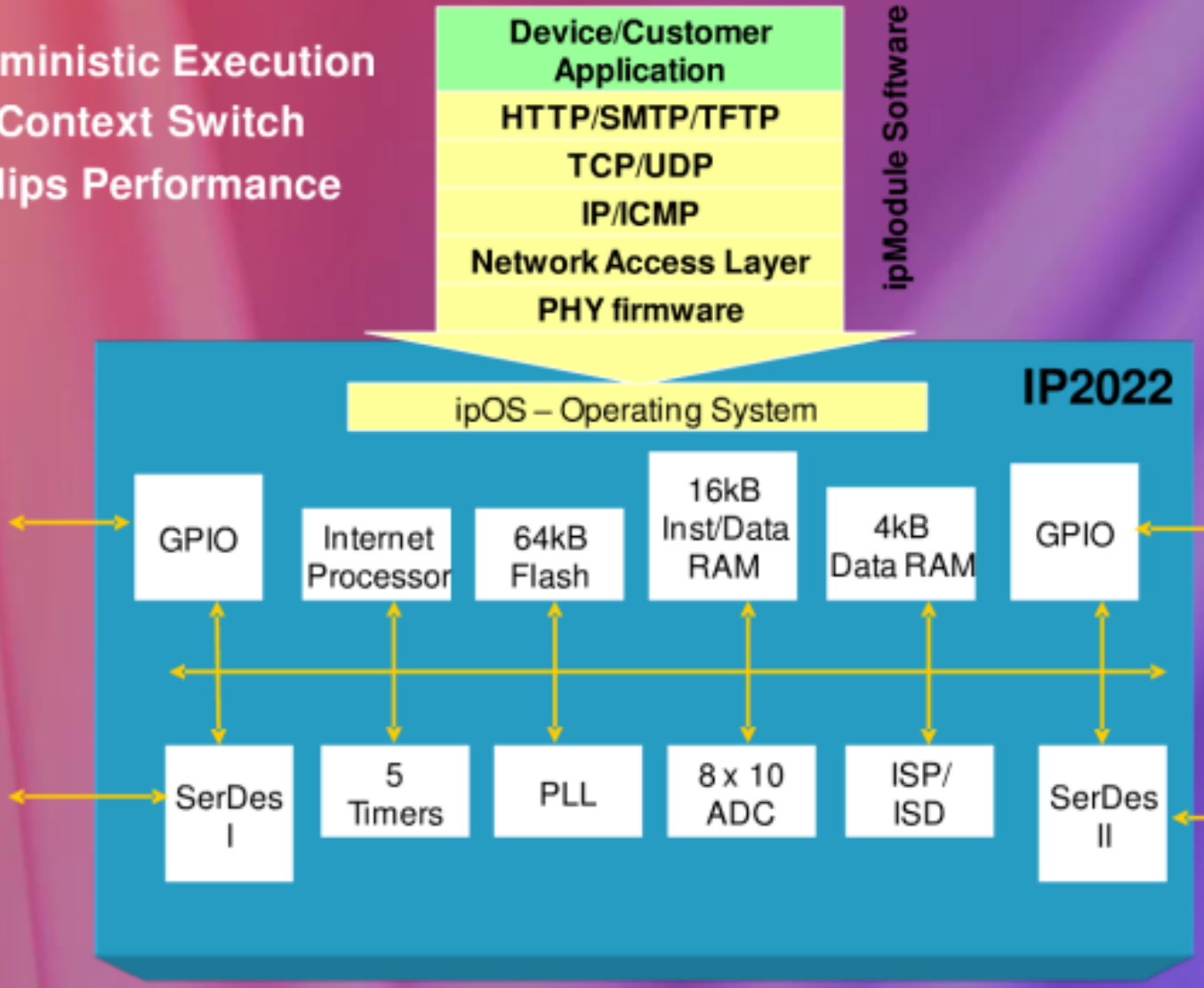
IP2k Update Architecture
- Using External Memory
- Revision Information

SDK Basics
- Structure
- ipModules
- TCP/IP Stack
- SDK status

Multiple Instances
- General Overview
- Step by Step Guide
- Hands on

IP2022 Internet Processor

- Deterministic Execution
- 3 clk Context Switch
- 120 Mips Performance



Stream Engine by UBICOM

UBICOM

IP2022 Features

120 MIPS performance with 4.8 MHz Xtal
- Guaranteed over industrial temp range

8-bit data path; 16-bit registers

16-bit instructions
- Single cycle instructions (except branches)

Deterministic architecture
- 3-cycle interrupt response
- Performs accurate real time functions

On Chip Memory
- 64kbyte Flash, 16Kbyte Program/Data RAM
- 4Kbyte Data RAM

Addresses up to 2Mbit external data memory

Extendable to 128KB program memory using ExtCode

Run time self programming capability

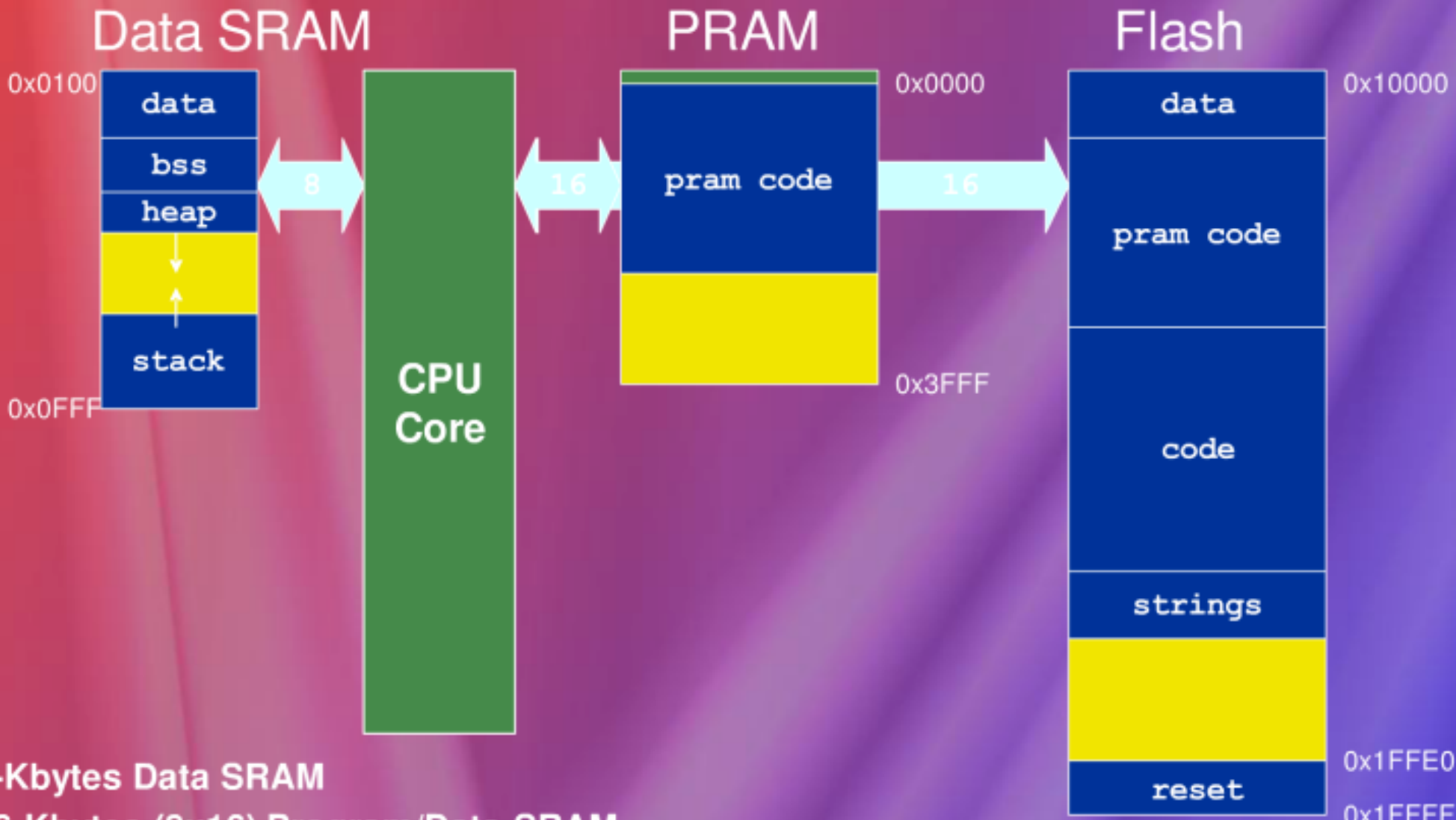
Stream Engine by UBICOM

UBICOM

IP2022 Features (Cont'd)

- **Configurable serialiser/de-serialiser blocks for multiple protocol support**
  - Ethernet, SPI, UART, GPSI
  - Linear feedback shift registers for error checking
- **Integrated PLL and clock management**
  - Speed changes under software control
- **Power management**
- **In-system debugging up to 120 MIPS with unlimited breakpoints**
- **Low EMI**
- **Pointer and stack operations optimized for C compiler**
- **Hardware peripherals**
  - ADC, Timers, POR, Watchdog, external memory interface

Memory Architecture (Internal Memory)



- 4-Kbytes Data SRAM
- 16-Kbytes (8x16) Program/Data SRAM
- 64-Kbytes (32kx16) Program Flash Memory

Stream Engine by UBICOM

UBICOM

Memory Considerations

Compiler only supports data stored in data RAM.

All memory assignment must be done with sections defined in the linker script.

- **.gpr**—general-purpose registers
- **.data**—pre-allocated, pre-initialized data memory
- **.text**—flash memory
- **.pram**—program space allocated in program RAM
- **.pram_data**—data space allocated in program RAM
- **.strings**—strings stored in flash memory
- **.reset**—reset vector
- **.config**—configuration block
- **.bss**—storage for globals. Initialized to zero

IP2K boot-up sequence

Linker script and Entry.S are responsible for boot loading the code and setting the memory locations

Main tasks done by Entry.S

- Clear general purpose registers. (**.gpr**)
- Pre-initialized variables in Data RAM. (**.data**)
- Clear the rest of Data RAM. (**.bss**)
- Load specified code into PRAM . (**.pram**)
- Clear the rest of PRAM.
- Reserve 16 registers for gcc
- Set reset vector to boot load sequence
- Initialize the stack
- Set the configuration fuses

IP2022 architectural advantages

Data memory

- Most instructions operate on a memory location with zero overhead (single cycle).
- No requirement for copying to and from registers – no load/store penalty.
- Copying from any memory location to any other is only two instructions/cycles

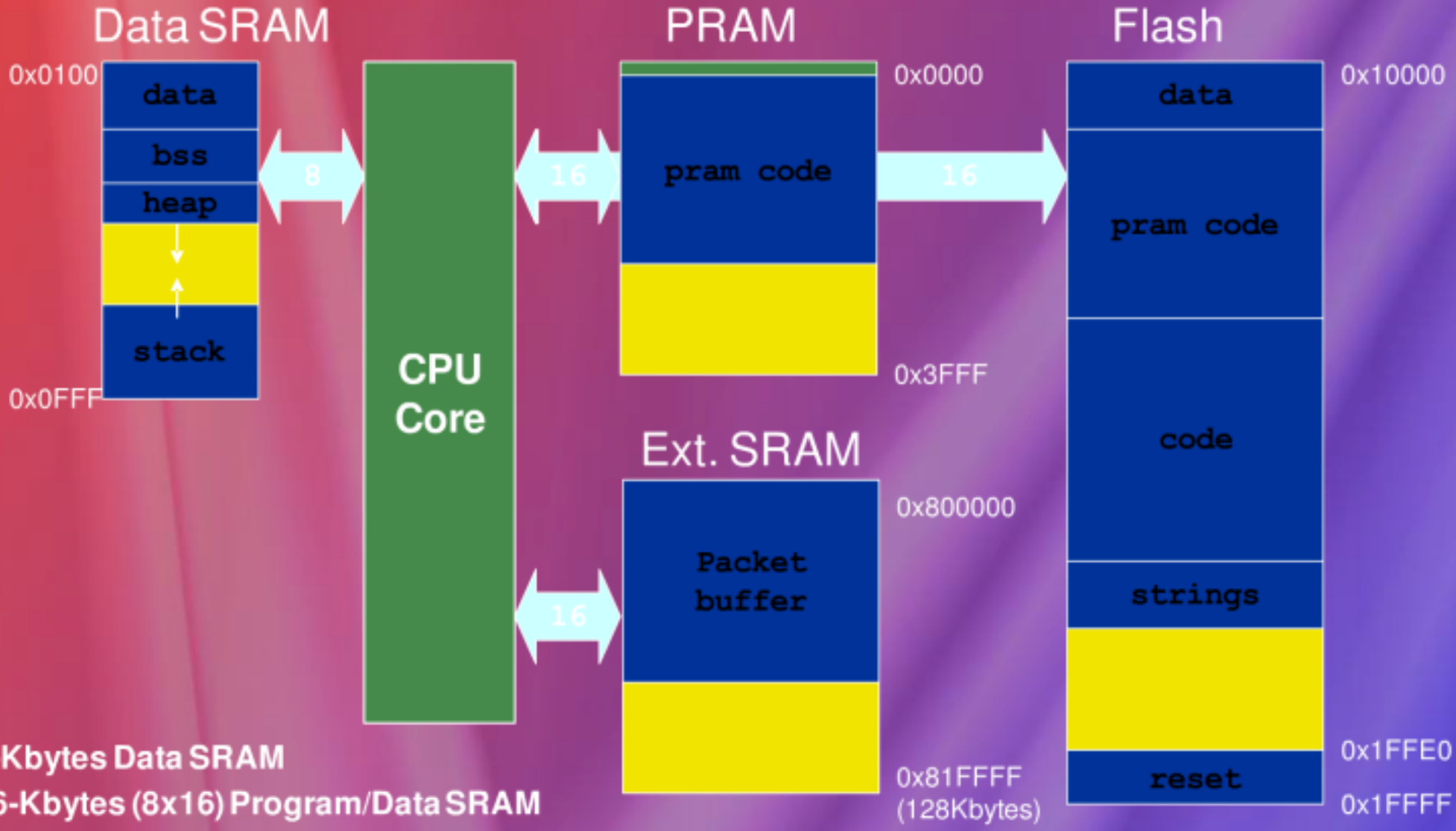
Stack Memory Access

- Stack is stored in data RAM and stack slots are addressable in the same way as other memory operands.
- Push/pop operations operate on two memory operands per instruction in a single cycle (top of stack is implied).
- Low cost of push/pop makes propagation of arguments and variables very low cost. Function call costs are low

External Memory access is non-blocking

- Takes 4 cycles, which can be used for other instructions

Memory Architecture (with external SRAM)



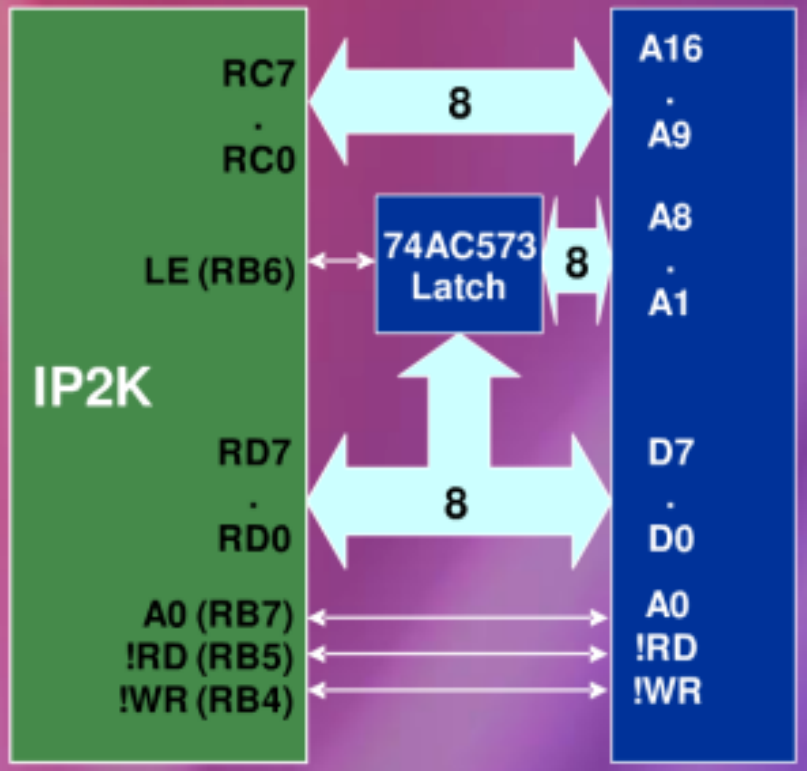
- 4-Kbytes Data SRAM
- 16-Kbytes (8x16) Program/Data SRAM
- 64-Kbytes (32kx16) Program Flash Memory
- External Data SRAM – 128Kbyte with internal addressing, up to 2Mbyte with additional I/O

Stream Engine by UBICOM

UBICOM

External SRAM Circuit – Linear Addressing

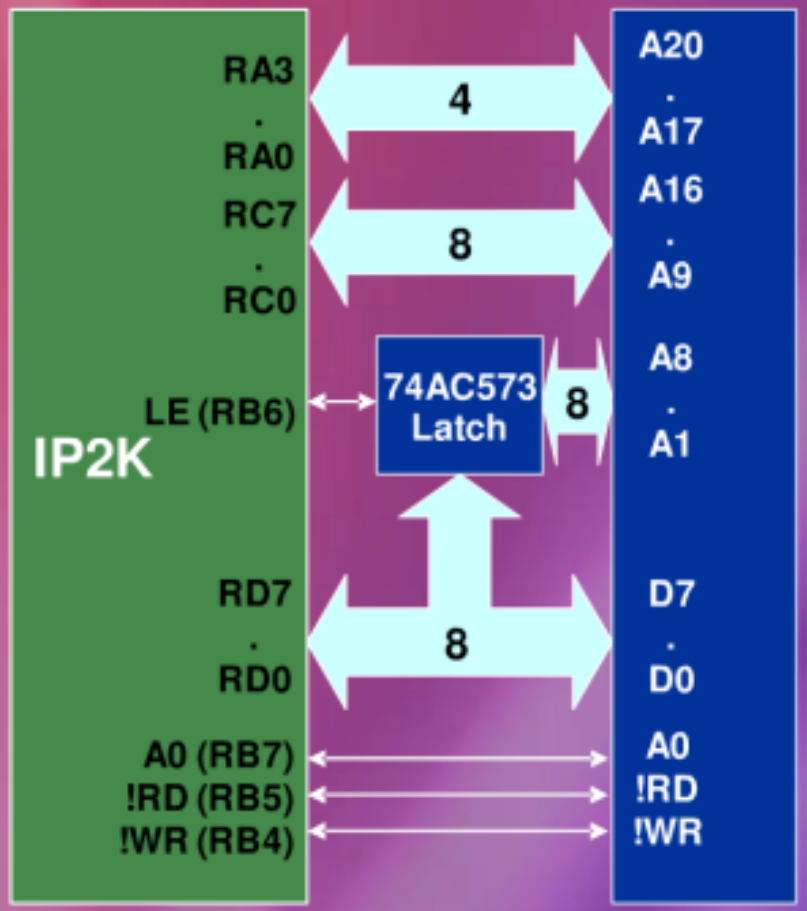
Up to 128Kbytes can be addressed linearly (no external decoding)



External SRAM Circuit – Extra Address I/O

Up to 2Mbytes can be addressed with additional external address I/O

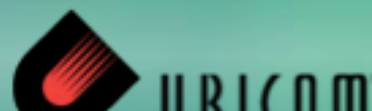
Additional address I/O handled in software



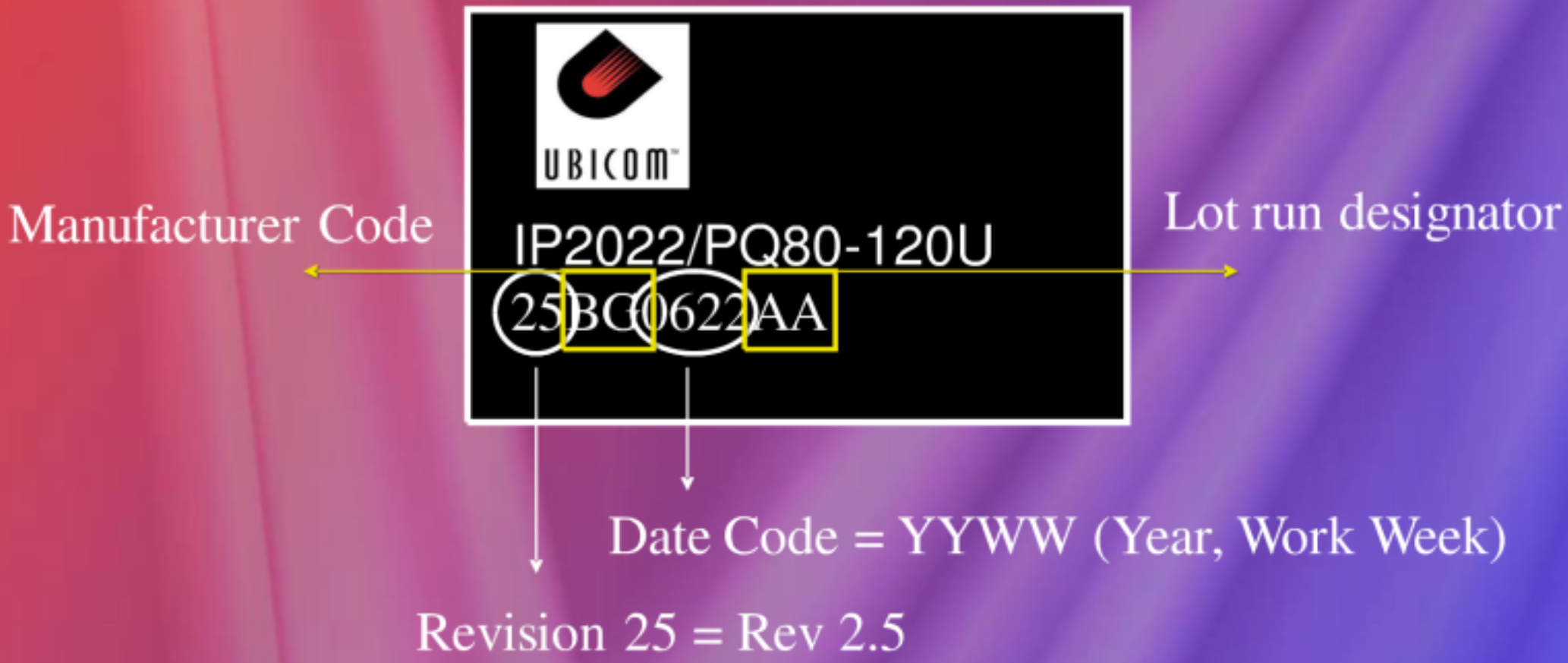
Stream Engine by UBICOM

UBICOM

Revision Information



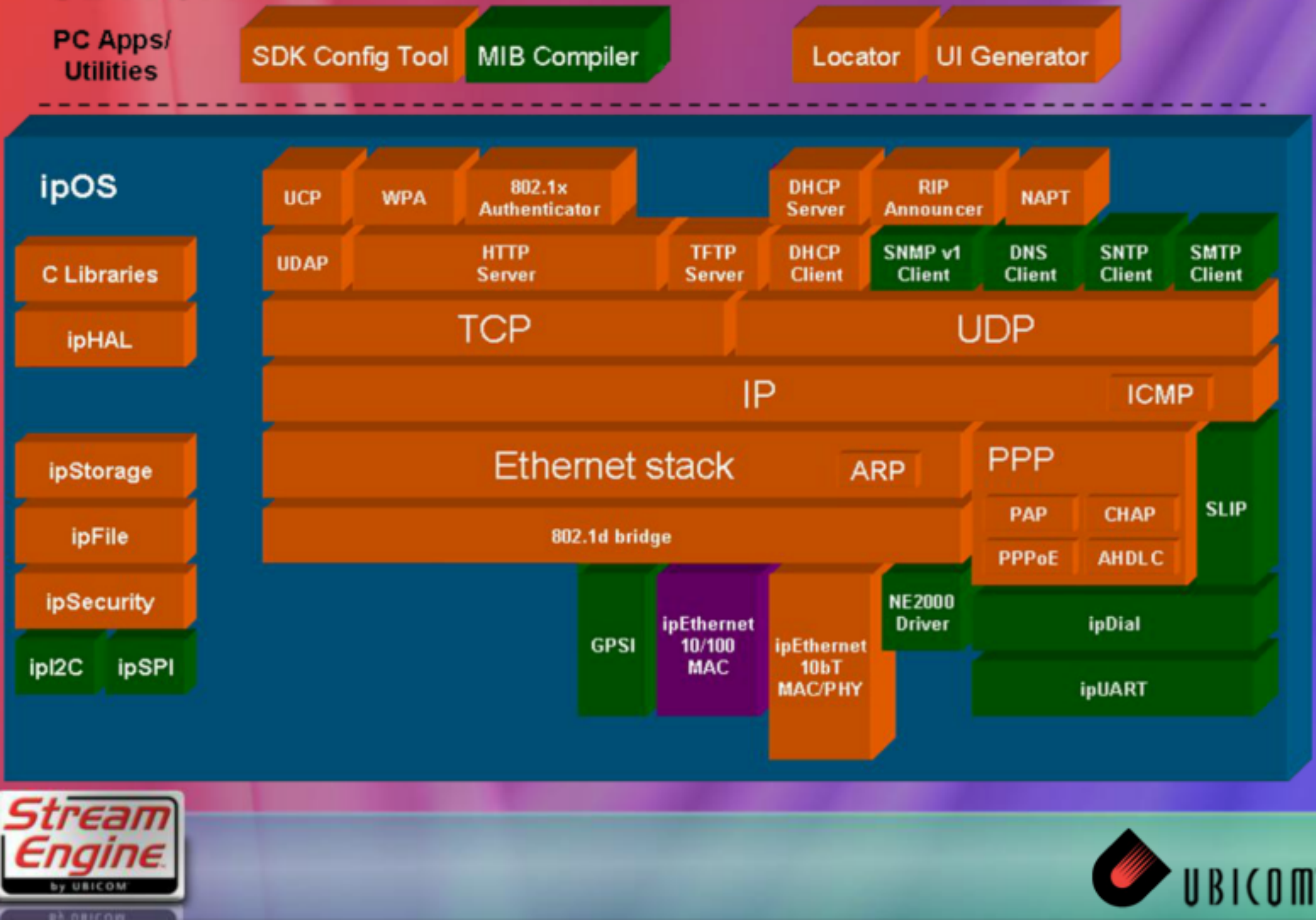
IP2022 Package Top Mark Indicator



SDK Basics

- **Overview of SDK**
  - Components
  - Structure
- **ipModules and packages**
- **Config Tool**
  - What is does and does not do..
- **Packet representation & Netbufs**
- **Using ipStack**
- **General user notes**

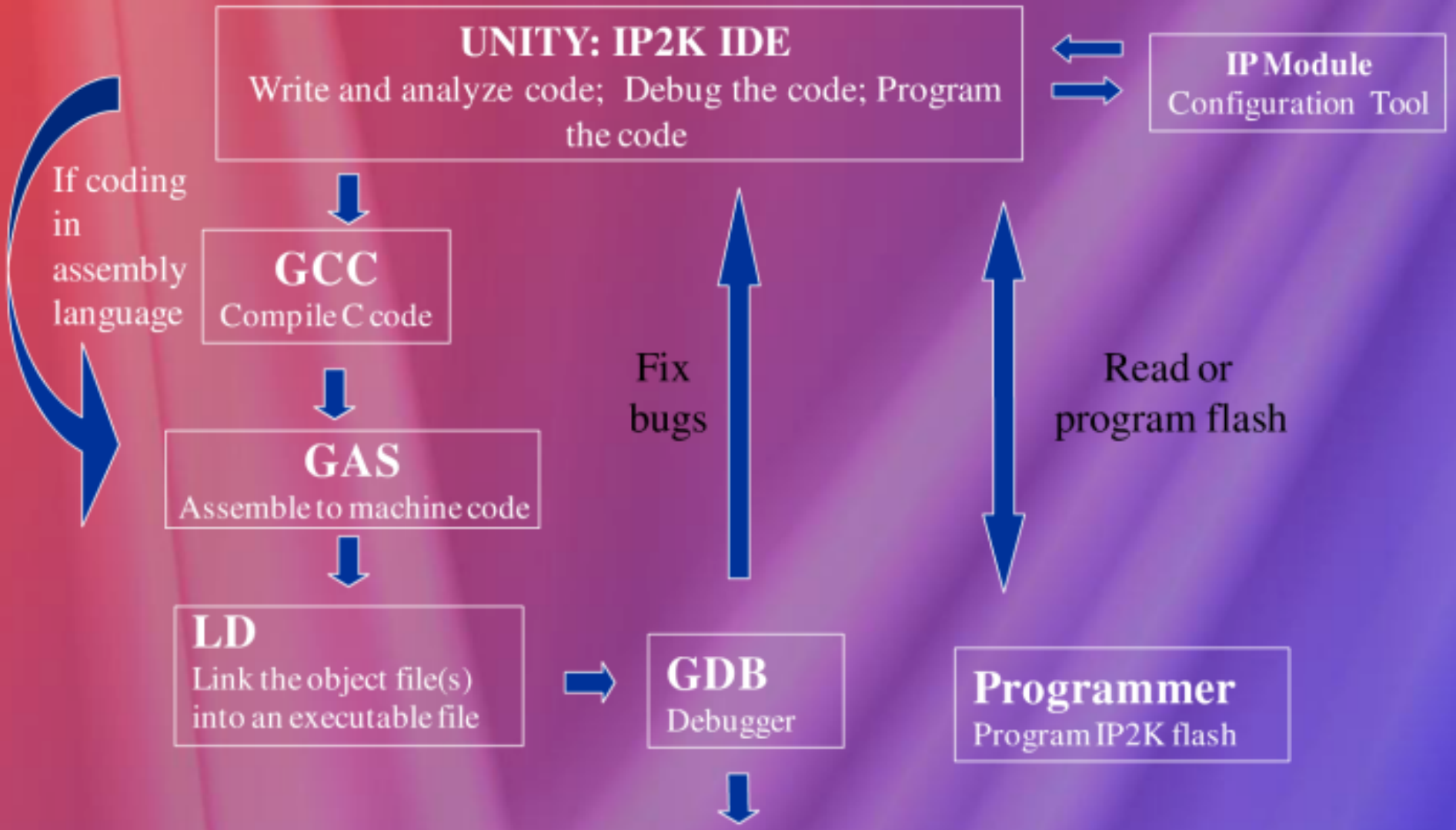
SDK 6.4



Stream Engine by UBICOM

UBICOM

IP2K Tool Flow



What are packages?

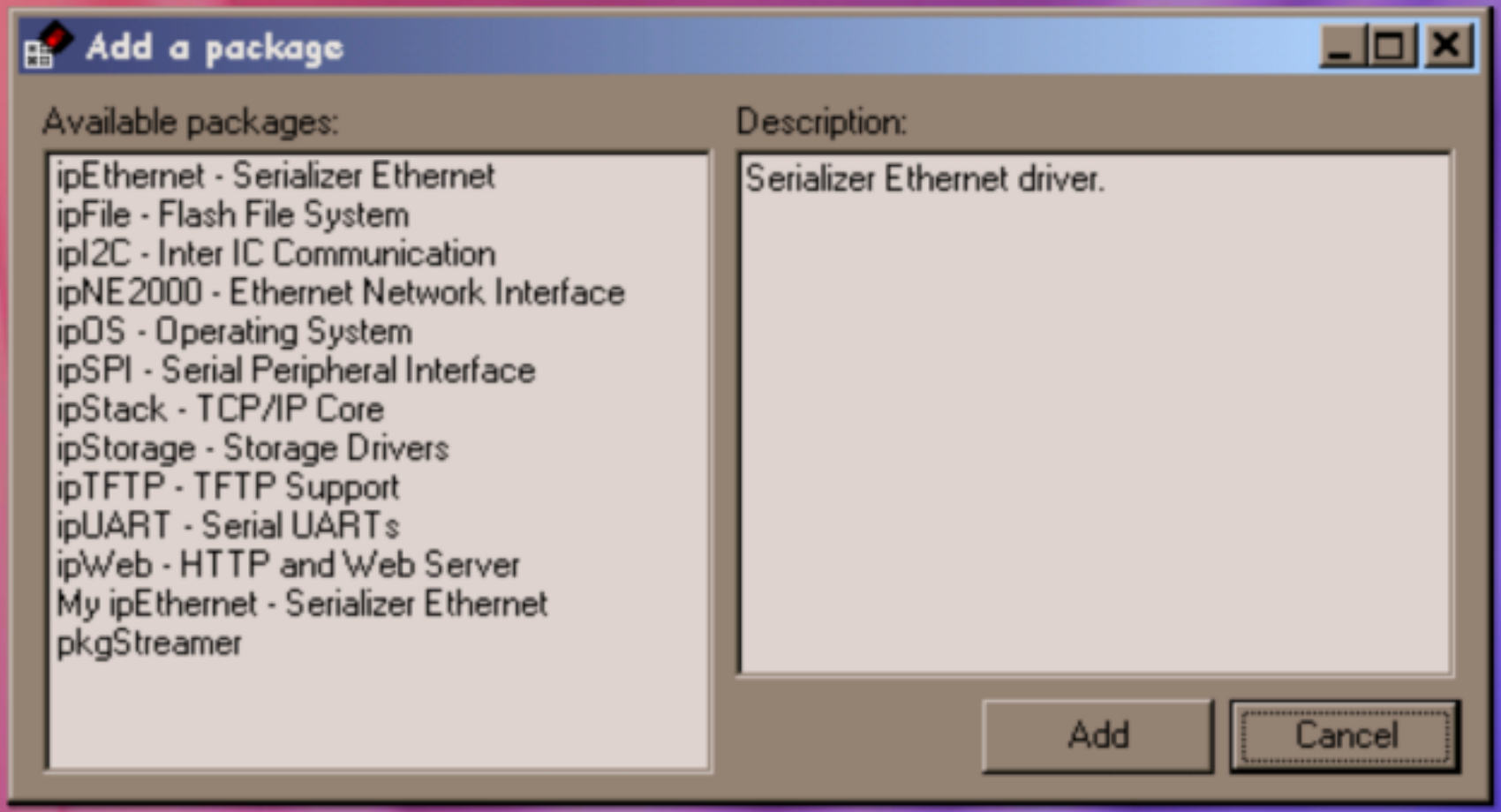
A package is a collection of related functions

When supplied and supported by Ubicom they are called ipModules

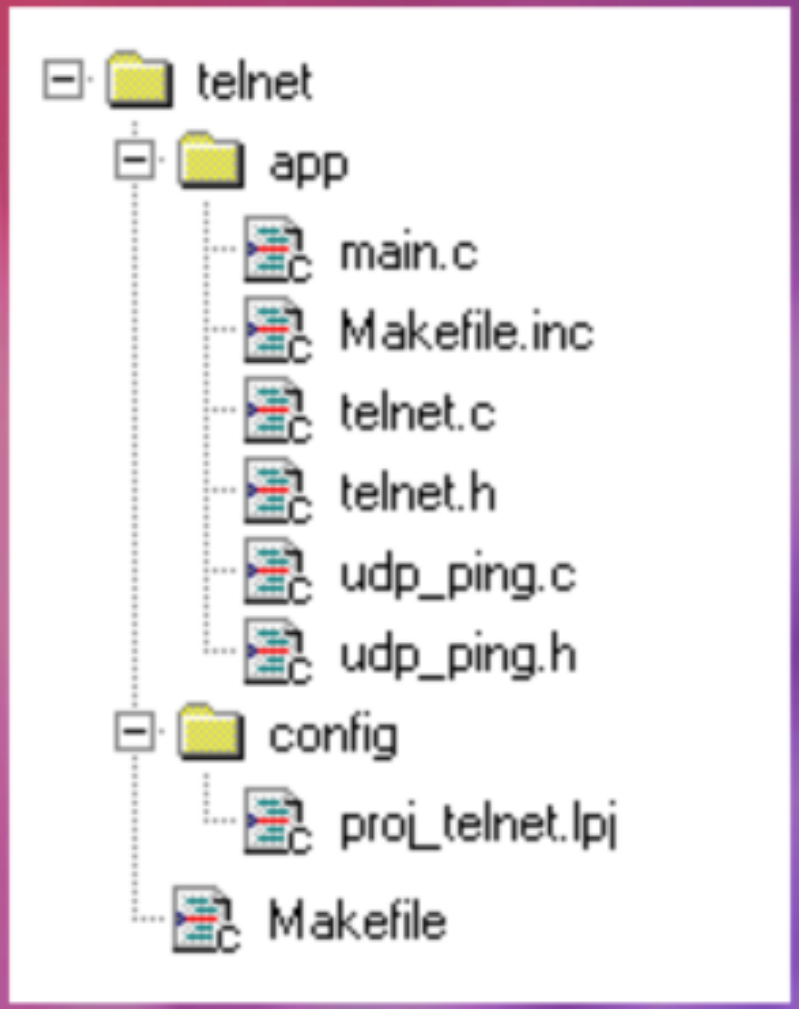
A user can make their own packages

- To improve maintainability of the project
- Manage a multi-user project by dividing functions into packages
- Make and share packages with other users at: http://portal.ubicom.com
  - DOWNLOADS – 3rd PARTY UNSUPPORTED – IP2K – SDK6.4

Available Packages



Project Structure



Application specific files

Configuration files

Makefile

Creating a project

- **Projects can be started from scratch or can use Ubicom supplied templates**
- **When you compile the project, all relevant SDK files are copied into the project sub-directory**
- **Use the configuration tool to set up the basic structure of the project**
  - Select ipModules to use
  - Configure of I/O's and ipModules
  - Generate config.h and config.mk files

Some notes on the configuration Tool

The Ubicom Configuration Tool

- Helps manage a large number of compile-time configuration options

by

- Generating include files for the build environment and the compiler

but does not

- Generate code
- Remove the requirement to read the documentation
- Guarantee than any particular configuration will compile or work

OS Concepts

OS manages any resources provided by the system

Arbitration and multiplexing

Resources are

- CPU,
- RAM,
- I/O

ipOS

ipOS is Ubicoms own operating system, optimised for embedded communications on multithreaded MCU's like the IP2K, IP3K and IP5K

It implements :-

- CPU arbitration
- Memory management (netbufs)
- Spinlocks
- Efficient program memory access routines
- Oneshot timers
- Utility functions

Oneshot Timers

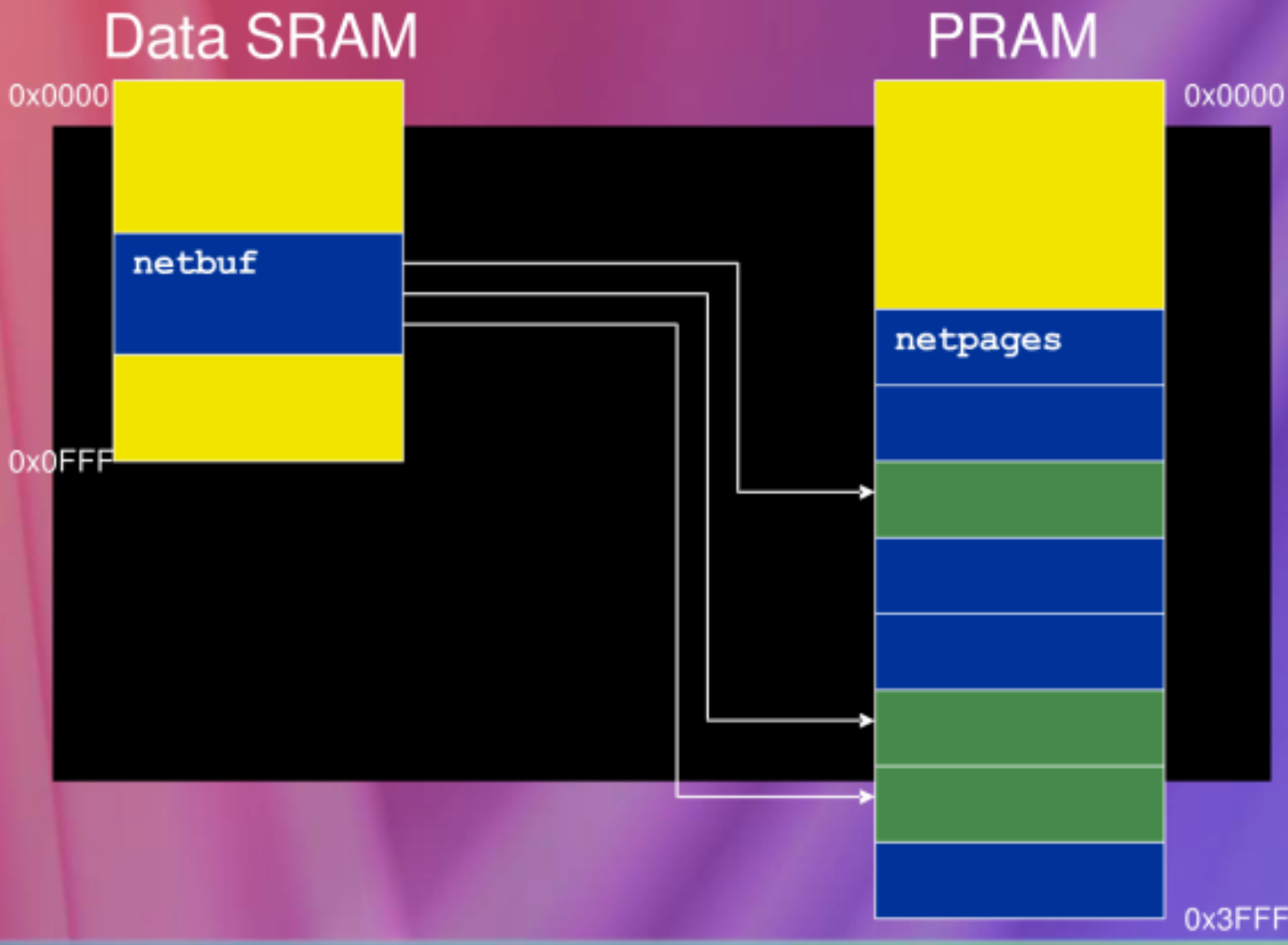
- Asynchronous timer mechanism
- Oneshot implies one-time use only (i.e. not periodic)
- Callback on expiry
- Typically used to implement timeouts (e.g. TCP, DHCP, sleep/wakeup)

Library Functions

ipOS contains a library of utility functions

- Controlling ports and I/O multiplexing
- Initializing interrupts
- String handling
- Debugging

Netpage Structure

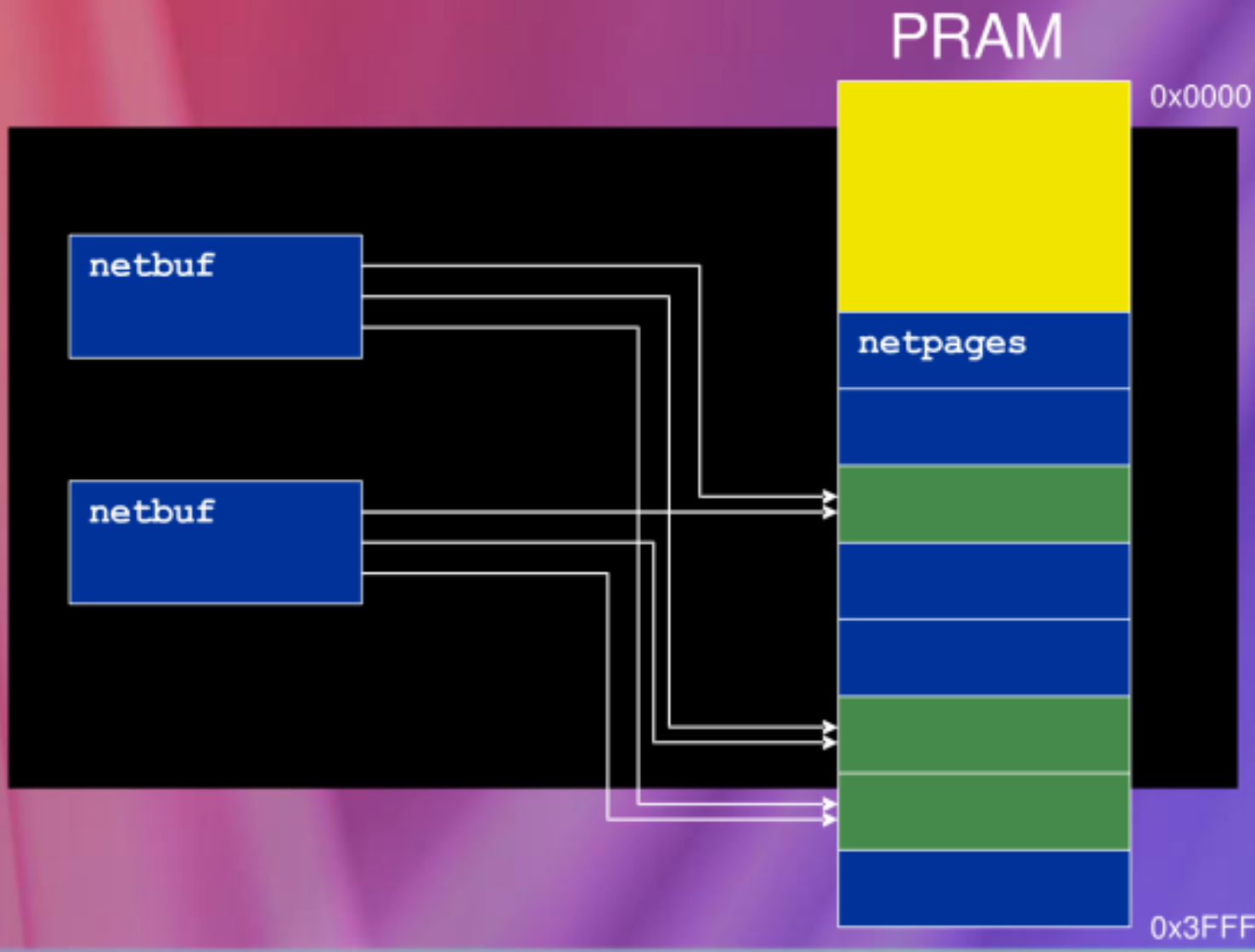


Netpage Structure

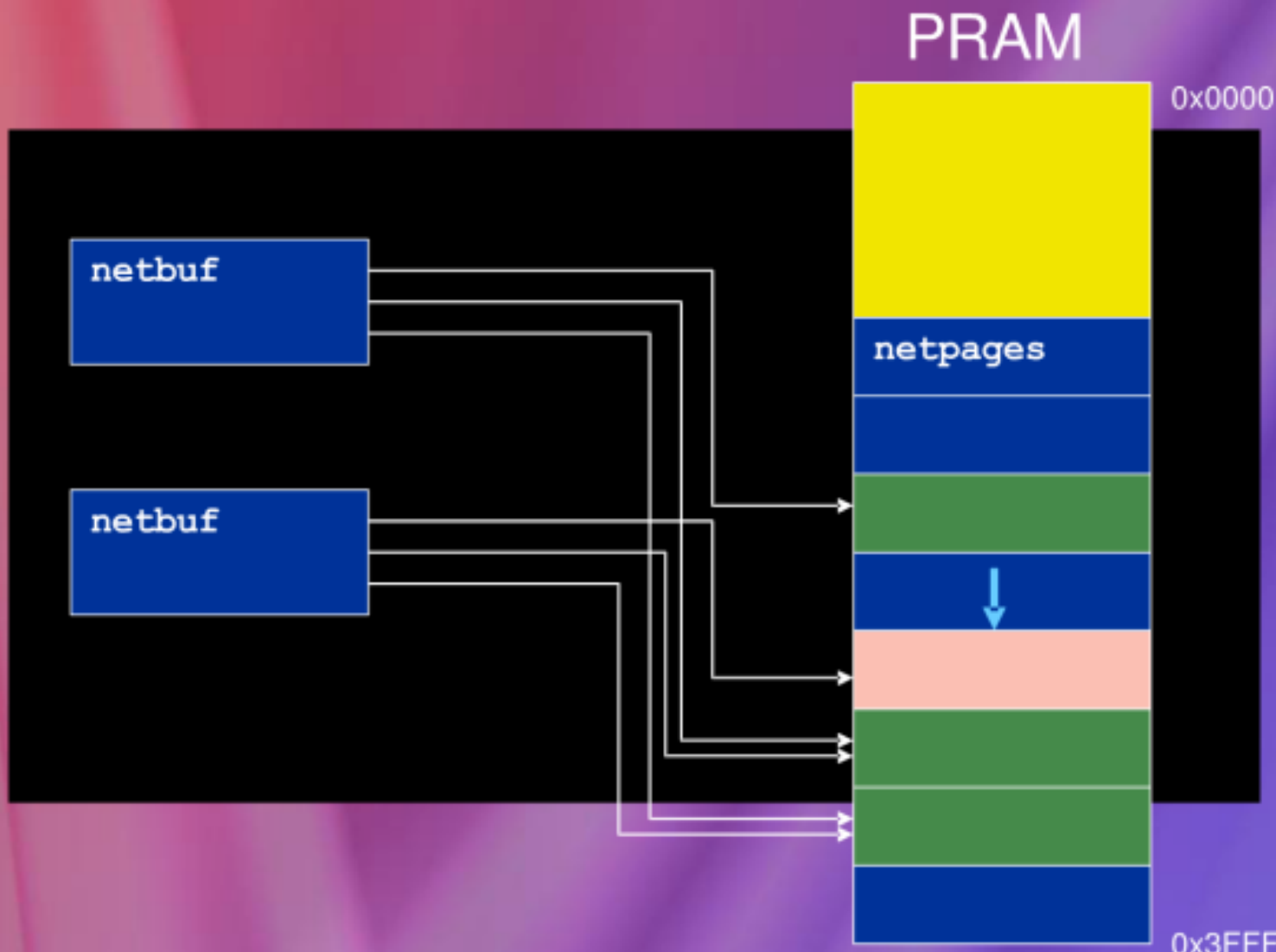
Each page is 256 bytes in size
Each netbuf points to 0-8 pages (=2kB packets)
Typically about 30 pages are available
Pages are allocated and reused randomly
Copy-on-write semantics
- Retransmitting or echoing packets is efficient

Can write forwards or backwards into a packet
- Writing backwards prepends pages to the start of the netbuf

Cloning a Netbuf



Cloning a Netbuf



Using Netbufs

Initialisation

- Initialise netpages
- netpage_init();

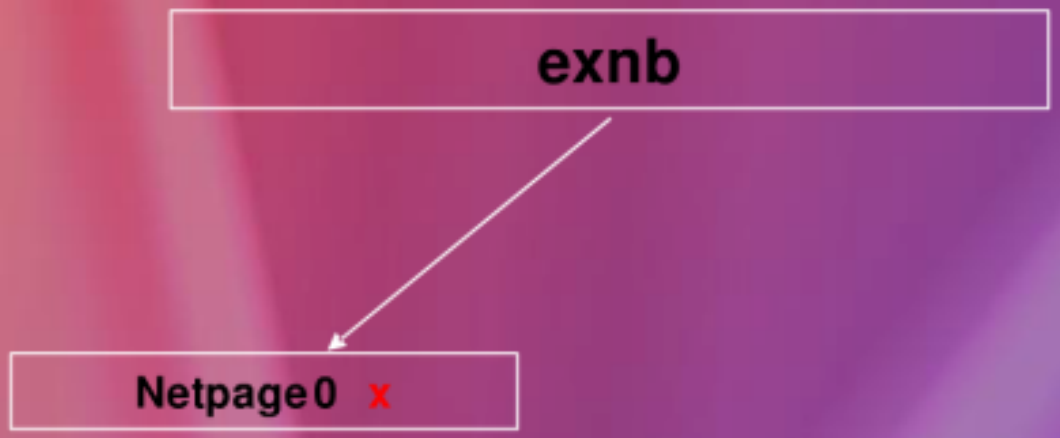
Create netbuf

- struct netbuf *exnb; // variable for referencing our netbuf
- exnb = netbuf_alloc(); // Allocate a netbuf
- No netpages have been allocated yet

Write to the netbuf

- See following example

Writing to the netbuf



- A write to a netbuf will cause a net page to be allocated to the netbuf.
- Given :

```
u8_t data = 'x';
netbuf_fwd_write_u8 (exnb,data)     // writes to the net page
```

Netbuf pointers

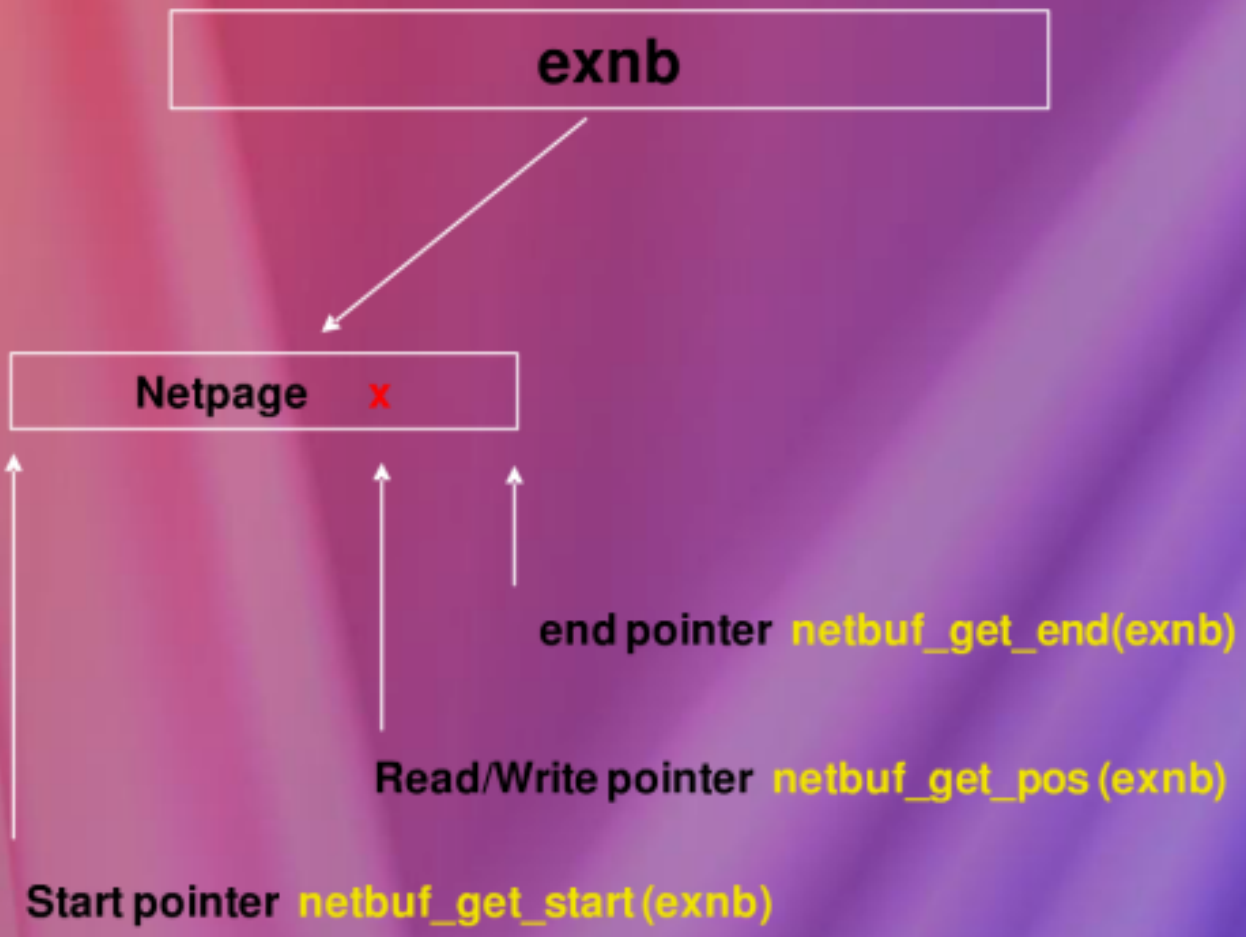
Three pointers are associated with netbufs

- Start of buffer
- End of buffer
- Current location of read/write pointer

The pointer values can be obtained with netbuf_get API functions

The pointer values can be altered using netbuf_set API functions.

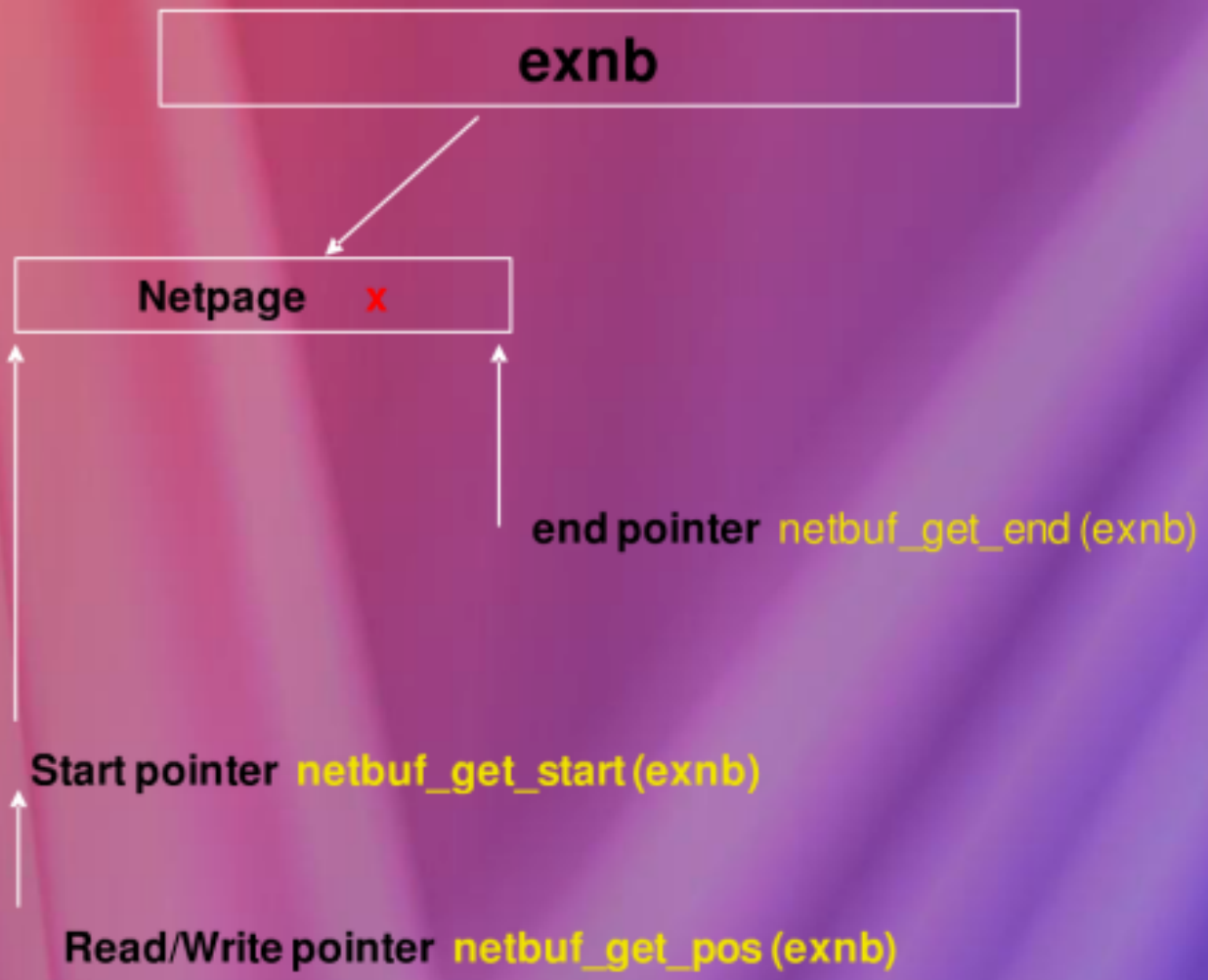
Netbuf pointers (cont)



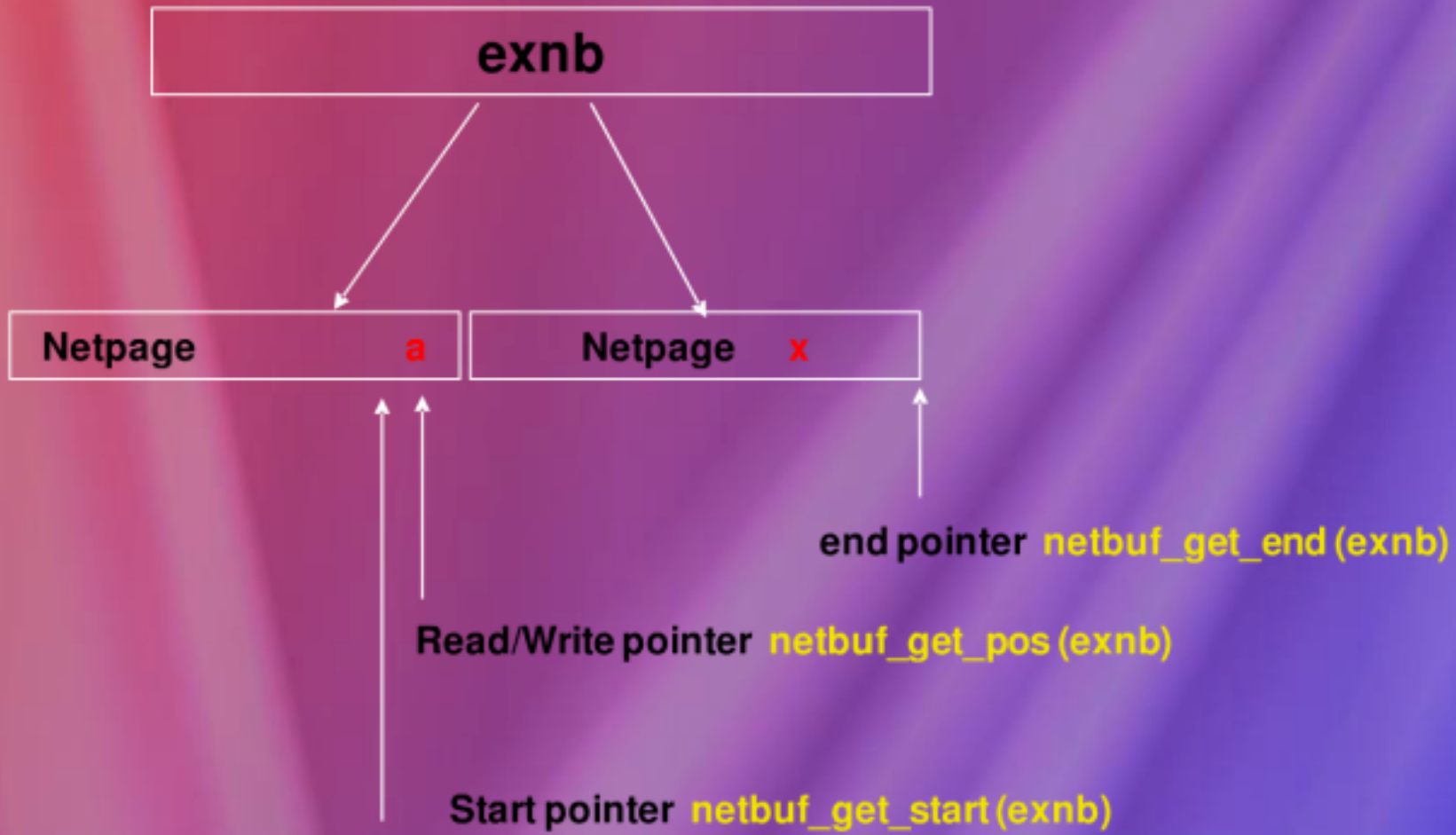
Dynamic growth of netbufs

- **Netbufs can grow in forward and reverse directions**
- **If the current read write pointer is at the start of a netbuf and you write in reverse**
  - netbuf_rev_write_u8(exnb,data) ;
  - The netbuf will be “grown” to comply with the operation.
- **This is ideal for stack processing as it allows blocks of data such as protocol layer headers to be written in front of the data fields**

Reverse write example –
before netbuf_rev_write(exnb,'a') instruction

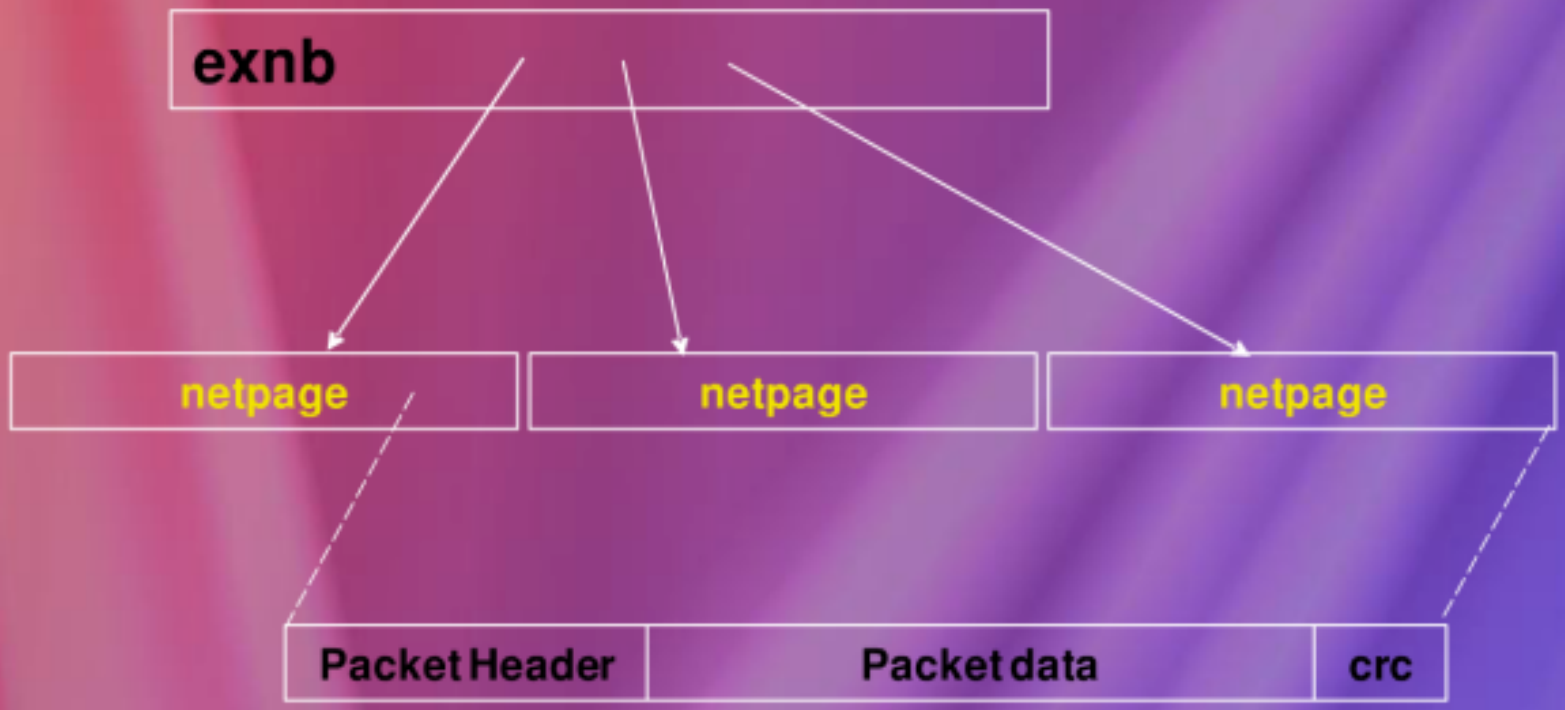


Reverse write example –
after netbuf_rev_write(exnb,'a') instruction



Same thing happens when you forward write beyond the end of the netpage boundary

Packet in Netbuf memory



Note :-

- It is not necessary for fields to meet on netpage boundaries
- The fields can easily span netpages.

Netbuf Summary

Generic data buffering mechanism

- Does not have to be packet data
- Low overhead and extremely efficient

There a rich set of API functions:

- Copy (forward and reverse)
- Write & Read (forward and reverse)
- Fill (forward and reverse)

There are specialized API functions for:

- CRC calculations
- Printf
- IP addressed.
- IP checksum

See the Unity <help><SDK><ipOS><Memory management > for details

Single task mode

- ipOS only offers single task mode
- Asynchonous processes (physical interfaces, timers, etc) must be polled
- Usually a polling loop resides in main.c
- Callback functions are used to control program flow
- Functions must return quickly

Callback Functions

Many parts of ipOS and ipStack use callback functions

A callback function is a function pointer which is assigned at runtime

Callbacks can be used to allow

- Dynamic creation of code
- Dynamically selecting between multiple interfaces (e.g. multiple applications using UDP)

To implementing a callback simply create a function with the required prototype

Pass the function name to the routine

Example Polling Loop

```
while (TRUE) {
   uart_vp_recv_poll(uarti);
   uart_vp_send_poll(uarti);
   timer_poll();
}
```

- Each physical layer exposes a polling interface
- Often transmit and receive are polled separately

Writing main.c

```
/*
 * main()
 */
int main(void)
{
        u8_t a = 0;

        /* Initialize the operating system */
        debug_init();
        heap_add((addr_t)(&_bss_end), (addr_t)RAMEND - (addr_t)(&_bss_end));

        /* Configure the ISR and start it running. */
        set_int_vector(isr);
        tmr0_isr_init();

        /* Loop forever, flashing the LEDs. */
        while (TRUE) {
                debug_set_lights(a++);
        }
}
```

Writing the ISR

```
void isr(void) __attribute__((naked));
void isr(void)
{
    asm("tmr0_freq = %b0" : : "P" ((int)((float)SYSTEM_FREQ/(float)
                                (1<<TMR0_PRESCALE)/(float)TMR0_INT_FREQ)));
    asm(" clrb      T0CFG.0   ");

    uart_vp_rx_isr();
    uart_vp_tx_isr();

    asm(" mov       w,#-tmr0_freq       ");
    asm(" reti      #%101             ");
}
```

- ISR is declared naked
- Inline assembly used for `reti`
- Configuration controls interrupts

Initializing the ISR

Helper functions are available to initialize the ISR

Interrupts must be disabled before changing the vector in mainline code

```
set_int_vector(isr);
tmr0_isr_init();
```

ipStack

Stream Engine by UBICOM

UBICOM

ipStack

ipStack a package containing the *core* TCP/IP functionality

Non-core protocols reside in other packages

- ipWeb = HTTP
- ipManage = SNMP

Protocols include

- ARP, SLIP and PPP
- IP, IP routing, IP loopback interface
- ICMP
- UDP, TCP
- DHCP client

Packet Representation

Each packet is represented by a netbuf structure

Packet data resides in pram

netbuf resides in data SRAM (18 bytes)

A netbuf can be considered an object in an OO sense

- The implementation is hidden by functions
- The memory used for packet storage might change (e.g. if external data memory is used)

Data Flow in the Stack

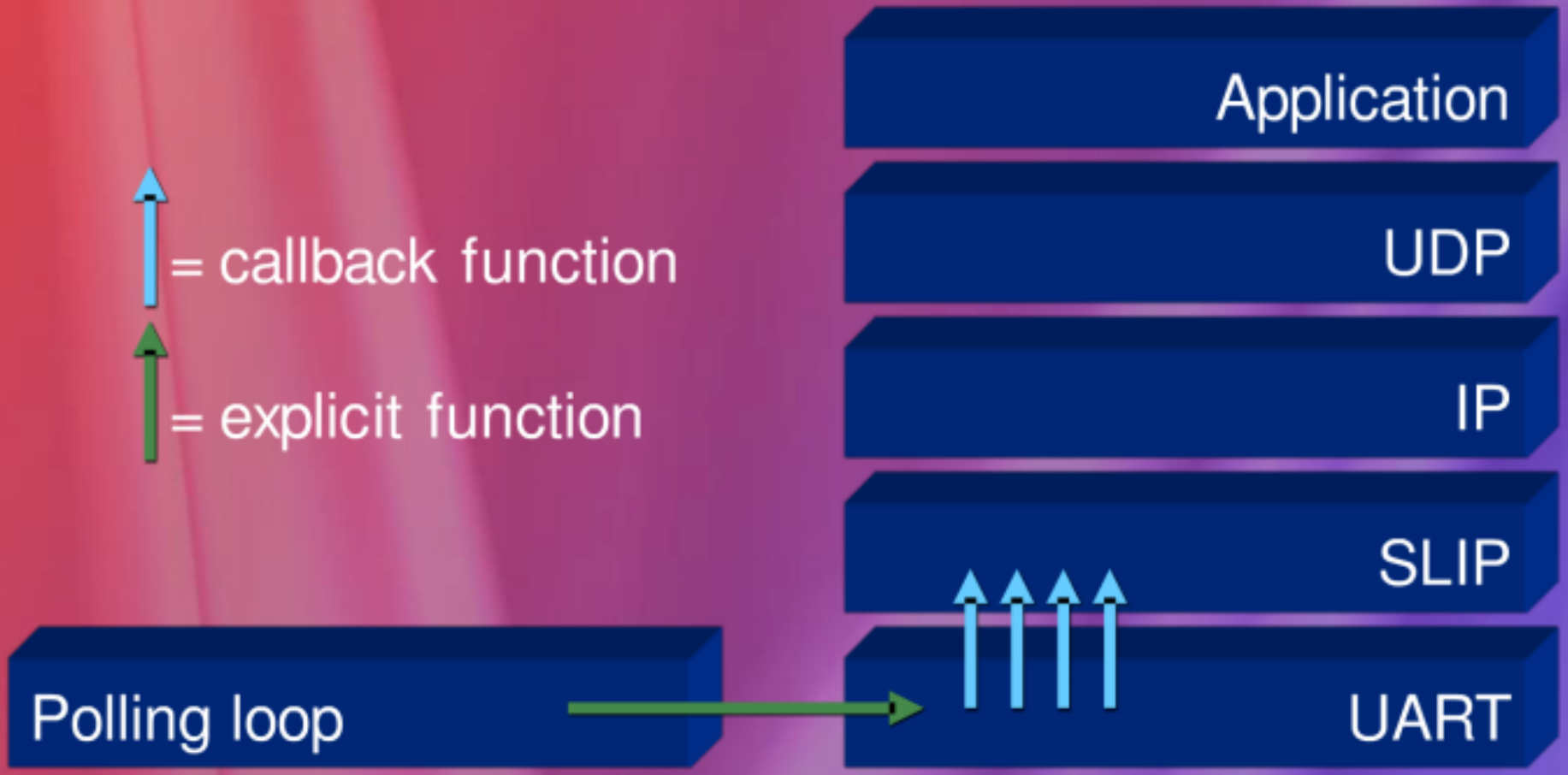
How is an incoming packet passed through the stack?

Imagine a packet arriving over a serial link and being processed by a UDP echo service

Data Flow in the Stack

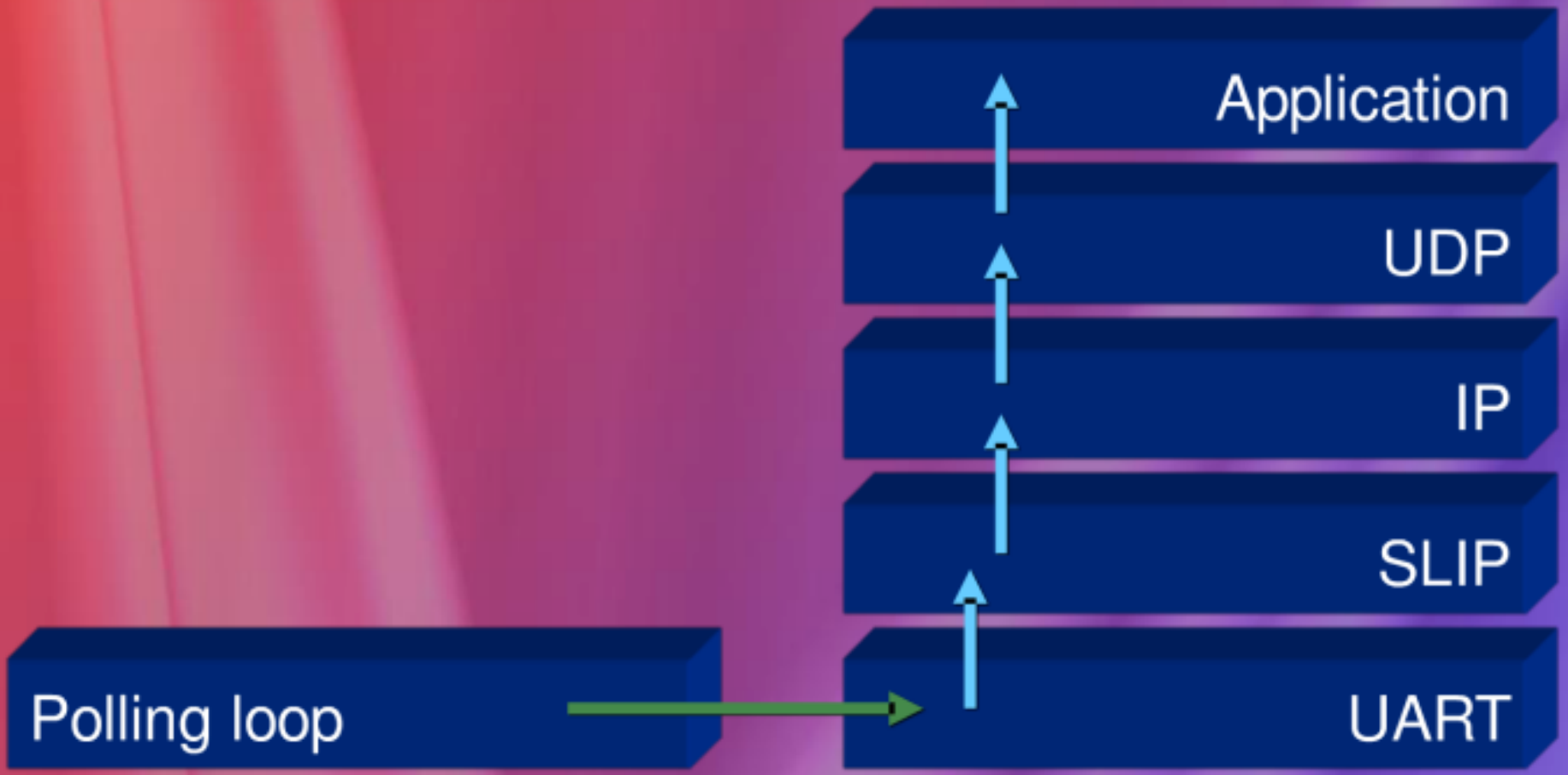
- The polling loop polls the UART receive
- If a byte is received it is passed to SLIP

Data Flow in the Stack



- SLIP aggregates the received bytes until a complete packet is recevied

Data Flow in the Stack

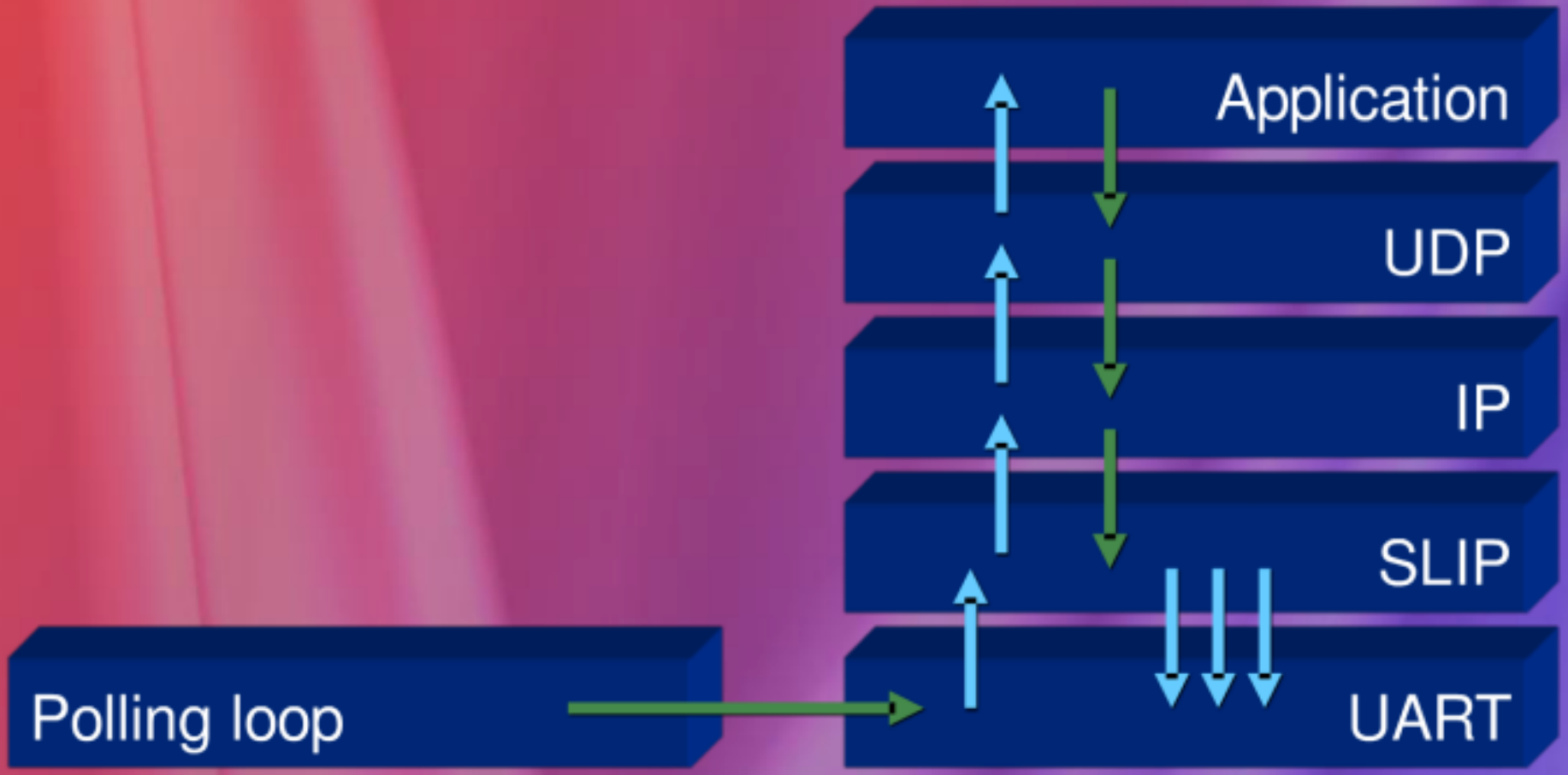


- When the end-of-packet marker is received the packet is passed up the stack
- Each layer processes the packet and passes it up to the next layer

Data Flow in the Stack

- If the application wishes to send a reply it allocates a new packet
- The new packet is passed back down the stack

Data Flow in the Stack



- As the UART is polled and the transmit buffer empties the packet is sent

Initializing ipStack

Before ipStack functions can be used portions of the stack must be initialized

At a minimum the netpage system must be initialized

If TCP is being used it needs initialization

```
netpage_init();
```

The datalink layer must be created and initialized

```
tcp_init();
```

Creating a Datalink Layer

Creating a datalink layer has two steps

1. Initialize the physical layer
2. Initialize the datalink layer

SLIP

```
uarti = uart_vp_instance_alloc();
serialipi = slip_instance_alloc(uarti, LOCAL_IP,
                                REMOTE_IP);
```

Ethernet

```
edi = ne2000_instance_alloc();
ethi = ethernet_instance_alloc(edi);
eii = ethernet_ip_instance_alloc(ethi, LOCAL_IP);
```

Example - The UDP API

```
#include <ipStack.h>

struct udp_socket *udp_socket_alloc(void *app_instance);

s8_t udp_listen(struct udp_socket *ts, u32_t addr, u16_t port,
    udp_dgram_recv recv);

typedef void (*udp_dgram_recv)(struct udp_socket *us, struct
    ip_datalink_instance *link, u32_t src_addr, u16_t src_port, u32_t
    dest_addr, u32_t spec_dest_addr, u16_t dest_port, u8_t ttl, u8_t tos,
    s16_t id, struct netbuf *nb);

void udp_send_netbuf(struct udp_socket *us, struct ip_datalink_instance
    *link, u32_t dest_addr, u16_t dest_port, u32_t src_addr, u16_t
    src_port, u8_t ttl, u8_t tos, s16_t id, u8_t df, struct netbuf *nb);
```

A Simple UDP Program

```
#include <ipStack.h>

void udp_ping_app_init(struct udp_instance *ui)
{
    struct udp_socket *us;

    /*
     * Allocate a socket.
     */
    us = udp_socket_alloc(NULL);

    /*
     * Start listening for incoming UDP packets on all
    * interfaces.
     */
    udp_listen(us, 0, UDP_PING_PORT, udp_ping_app_recv);
}
```

A Simple UDP Program

```
void udp_ping_app_recv(struct udp_socket *us,
                       struct ip_datalink_instance *idi,
                       u32_t src_addr, u16_t src_port,
                       u32_t dest_addr, u32_t spec_dest_addr,
                       u16_t dest_port, u8_t ttl, u8_t tos,
                       s16_t id, struct netbuf *nb)
{
    struct netbuf *nbrep;

    /*
     * Take the received message, swap the addresses and ports
     * and send it back!
     */
    nbrep = netbuf_clone(nb);
    udp_send_netbuf(us, NULL, src_addr, src_port, spec_dest_addr,
                    dest_port, UDP_TTL_DEFAULT, UDP_TOS_DEFAULT,
                    UDP_ID_DEFAULT, UDP_DF_DEFAULT, nbrep);

    netbuf_free(nbrep);
}
```

SDK Comments and Status

Very efficient

- ipOS, ipStack (TCP and UDP), ipEthernet ~ 27kB
- With Webserver and simple application < 35kB!
- This is class leading

Stack performance

- Throughput: 25Mbps on TCP
- Latency: <600 us TCP (UDP < 500 us)

Not a general micro

- Optimised for communications processing
- Need to use Ubicom libraries in the main
- Maintain code efficiency and processing speed
- Don't expect Scanf support!

Stack Performance figures

	SDK 6.4
TCP Throughput receive from IP2K	24.3 Mbps
TCP Throughput send to IP2K	23.3 Mbps
UDP Throughput receive from IP2K	33.7 Mbps
UDP Throughput send to IP2K	80.4 Mbps
TCP Latency (8 Byte)	510 us
UDP Latency (8 Byte)	456 us

Stream Engine by UBICOM

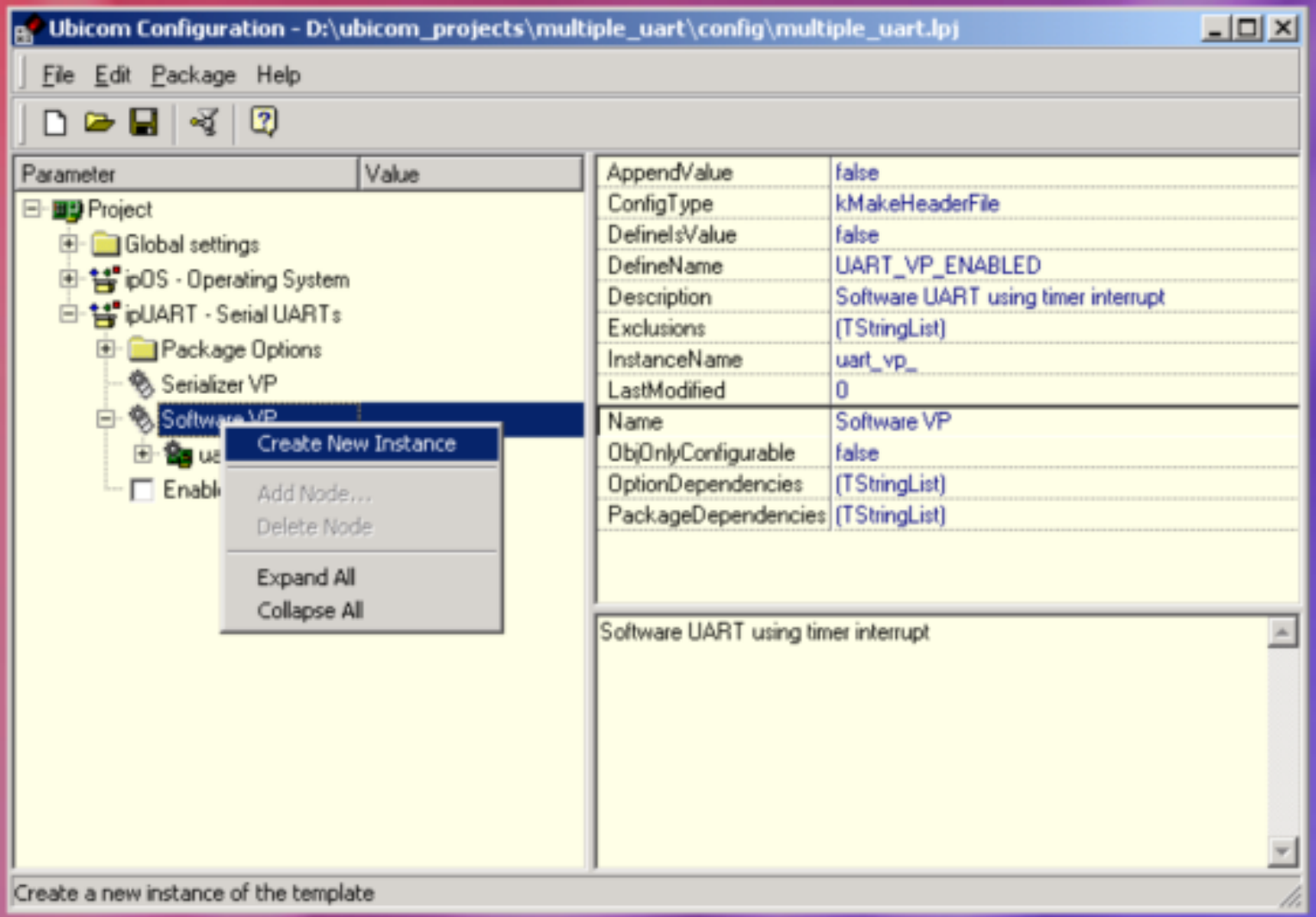
UBICOM

Multiple Instances

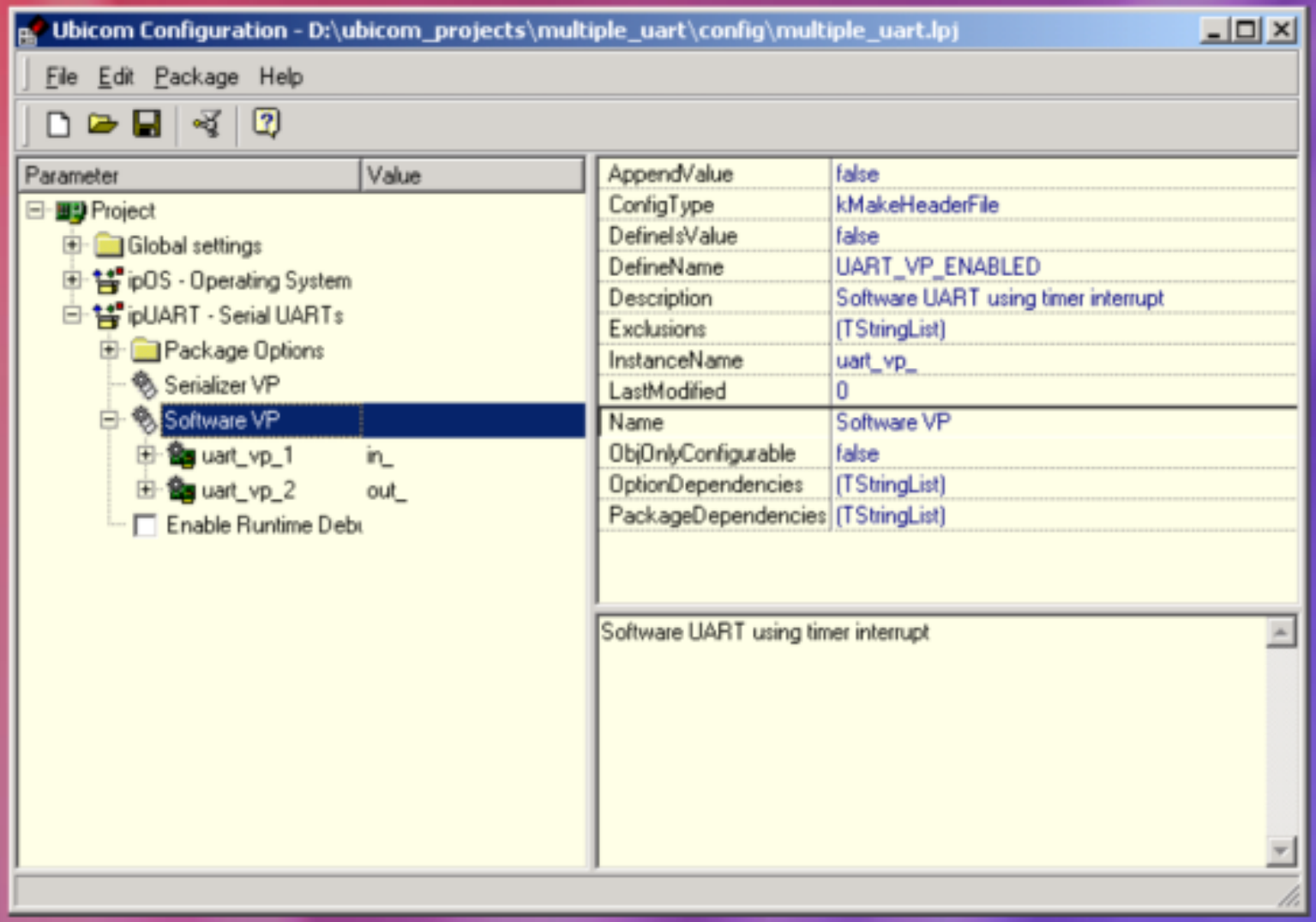
Goals

Learning to use the configuration tool to add multiple instances
Understand ISR/Task scheduling
How to use TMR0 timer slots
Know how to calculate correct timer values

Adding a Second UART



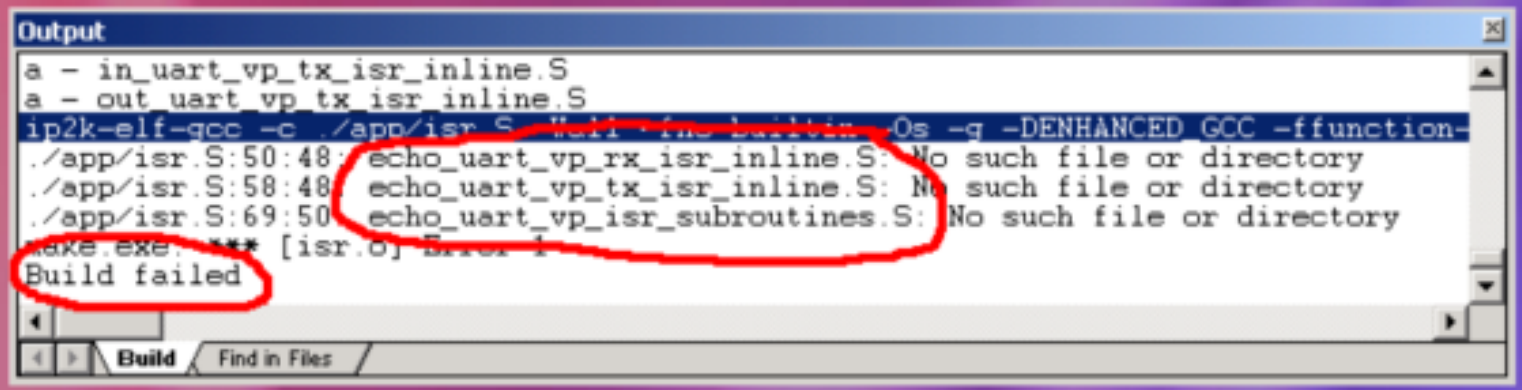
Change names



Stream Engine by UBICOM

UBICOM

Compiling



Stream Engine by UBICOM

UBICOM

Fix it

change all echo related routines from echo_* toS in_* in both main.c and isr.S

isr.s

Was:

```
#include "echo_uart_vp_rx_isr_inline.S"
#include "echo_uart_vp_tx_isr_inline.S"
#include "echo_uart_vp_isr_subroutines.S"
```

Has to be:

```
#include "in_uart_vp_rx_isr_inline.S"
#include "in_uart_vp_tx_isr_inline.S"
#include "in_uart_vp_isr_subroutines.S"
```

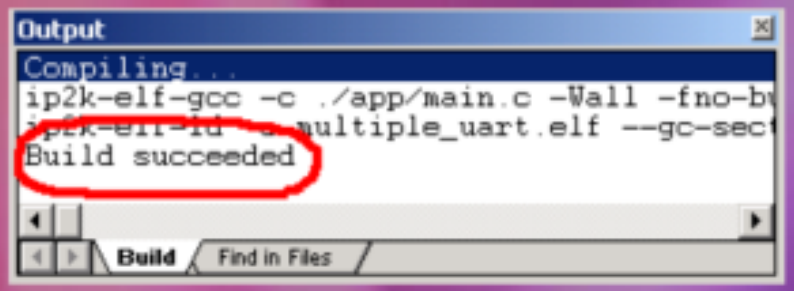
main.c

Replace:

```
uarti = echo_uart_vp_instance_alloc();
echo_uart_vp_recv_poll(uarti);
echo_uart_vp_send_poll(uarti);
```

By:

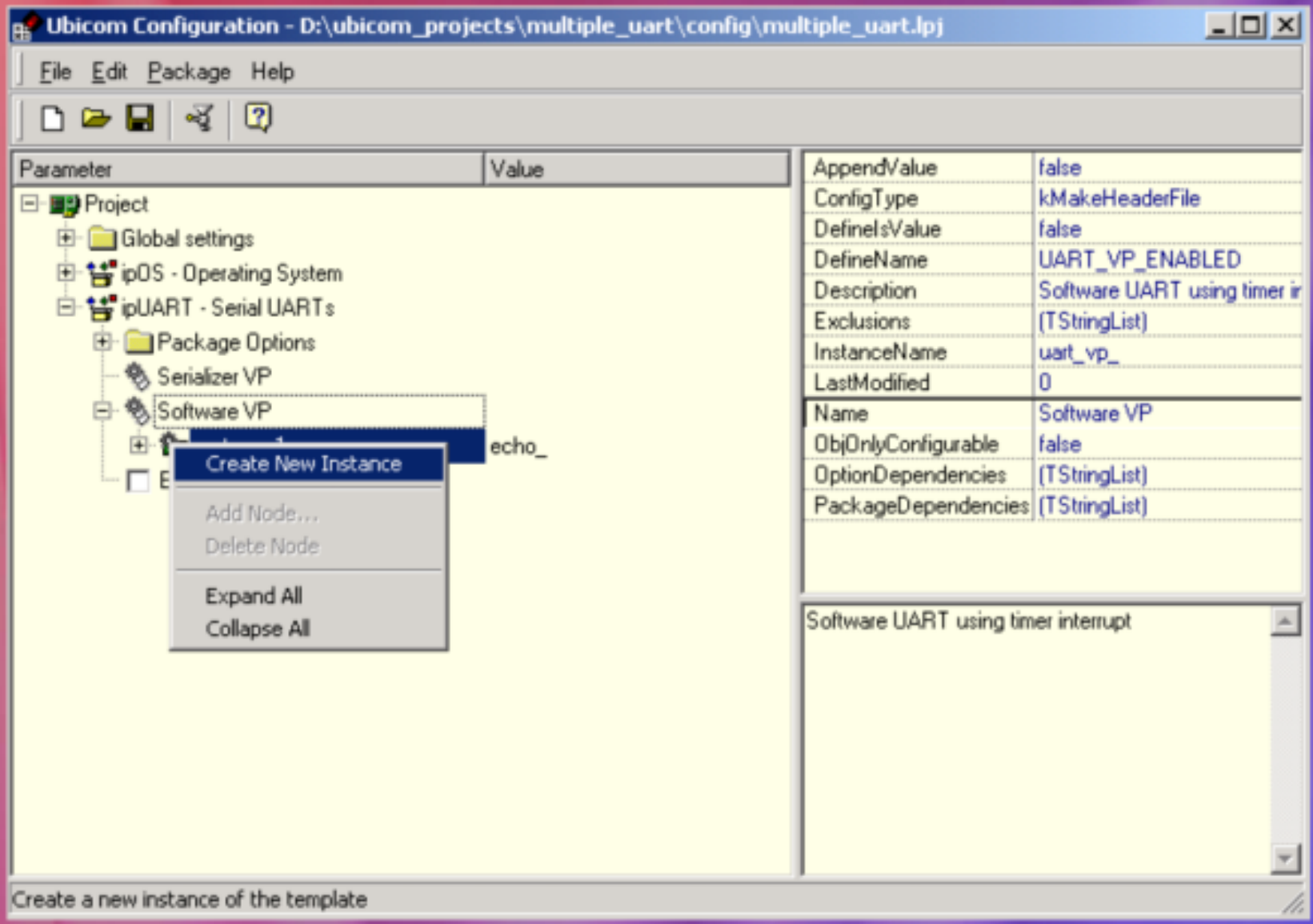
```
uarti = in_uart_vp_instance_alloc();
in_uart_vp_recv_poll(uarti);
in_uart_vp_send_poll(uarti);
```



Conclusion

Config tool does NOT write code

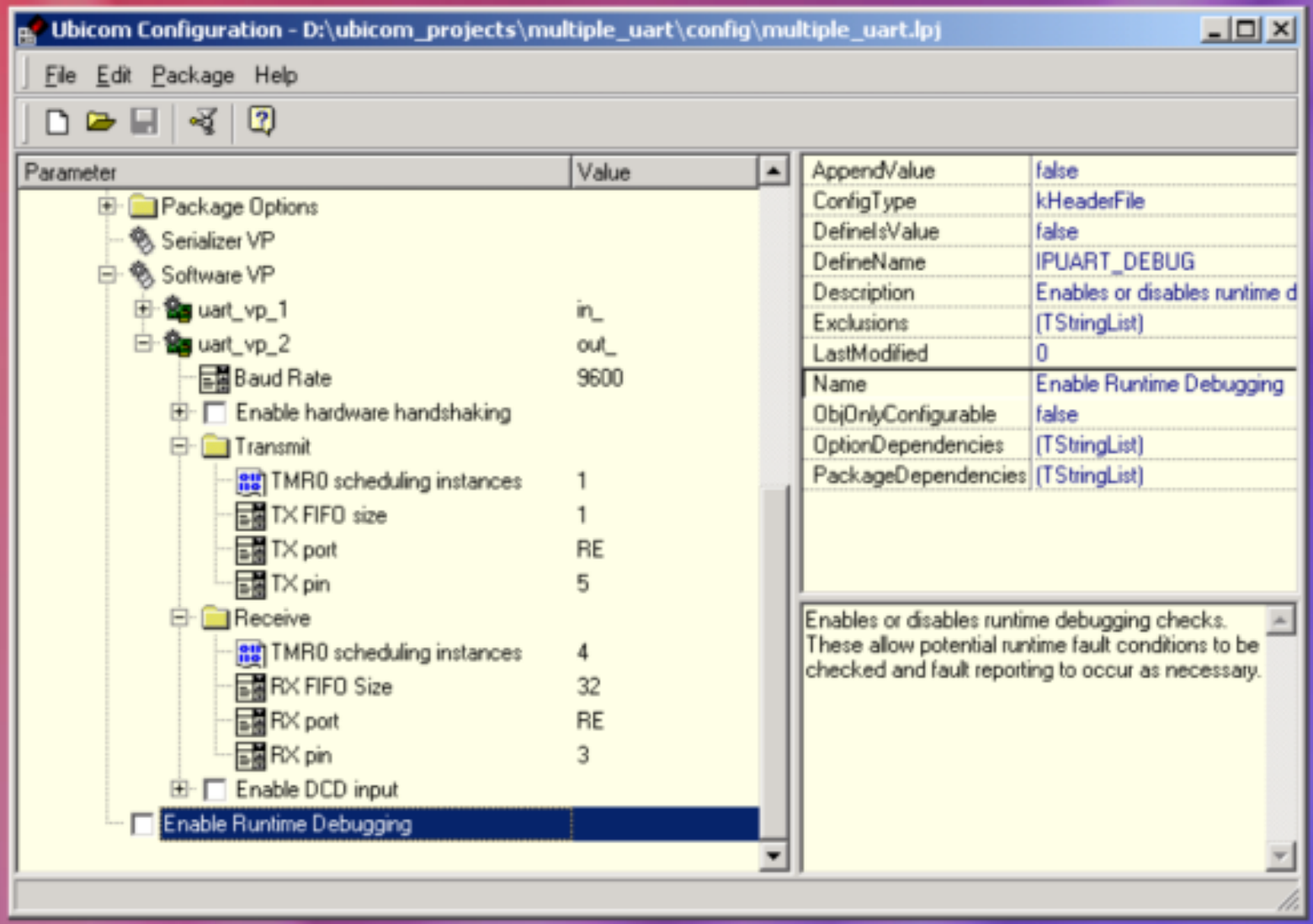
Creating a Second UART



Configuring/Activating Our Second UART

- Configuring second UART
- A closer look at the TMR0 and the ISR
- Recalculation of TMR0 and Prescale
- Adjusting isr to handle second UART
- Create the second UART instance

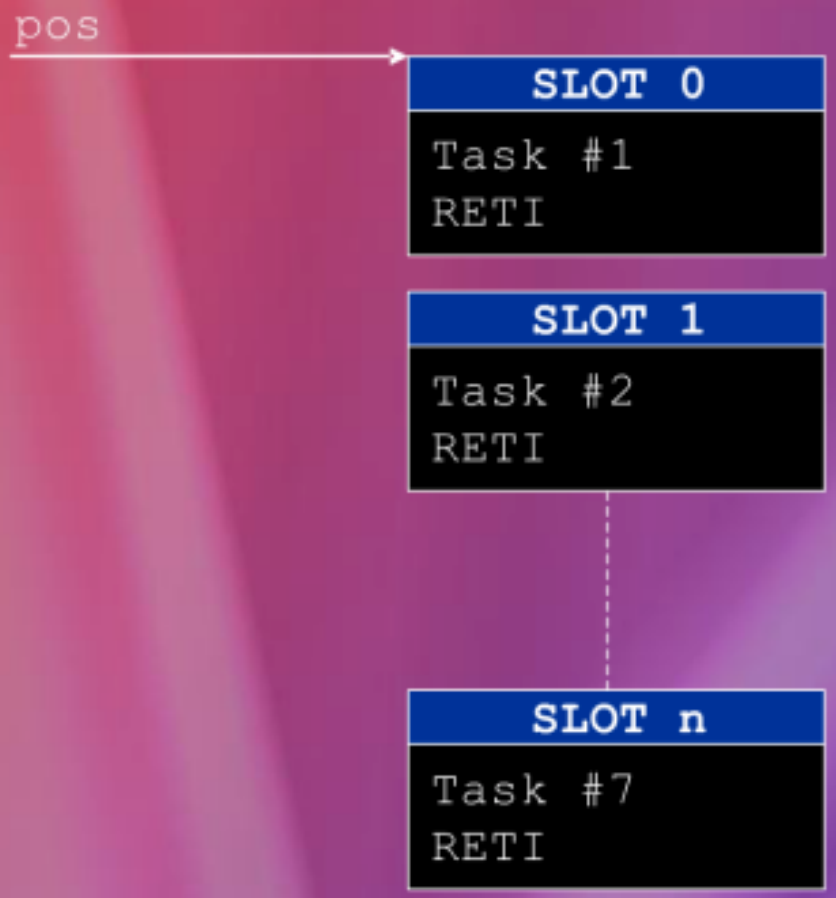
Configuring UART Pins



Stream Engine by UBICOM

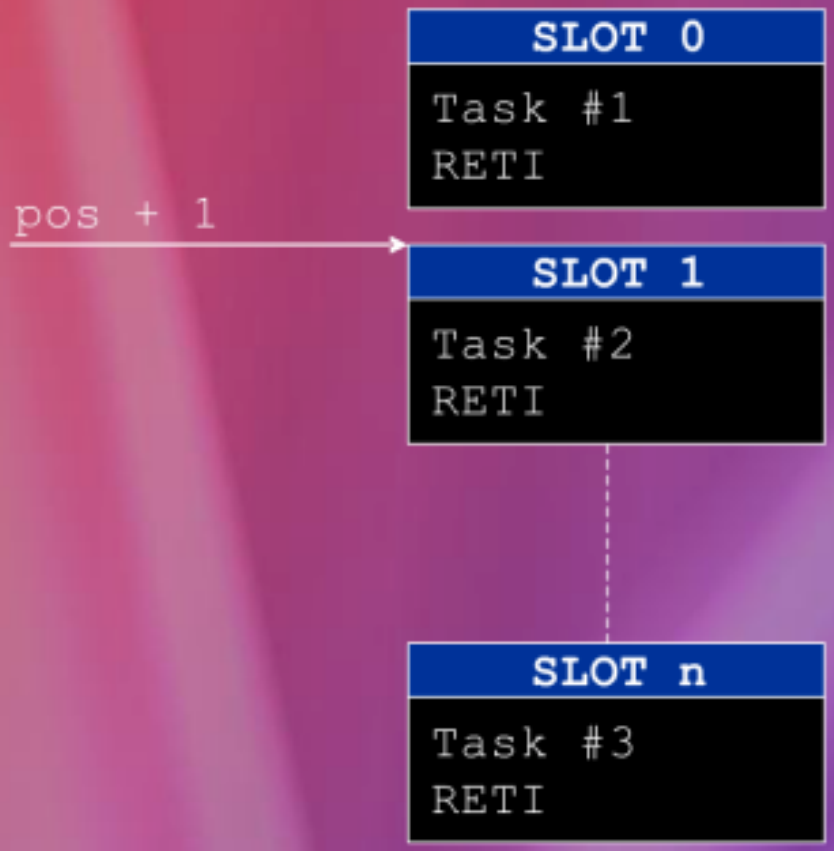
UBICOM

TMR0 Scheduling table

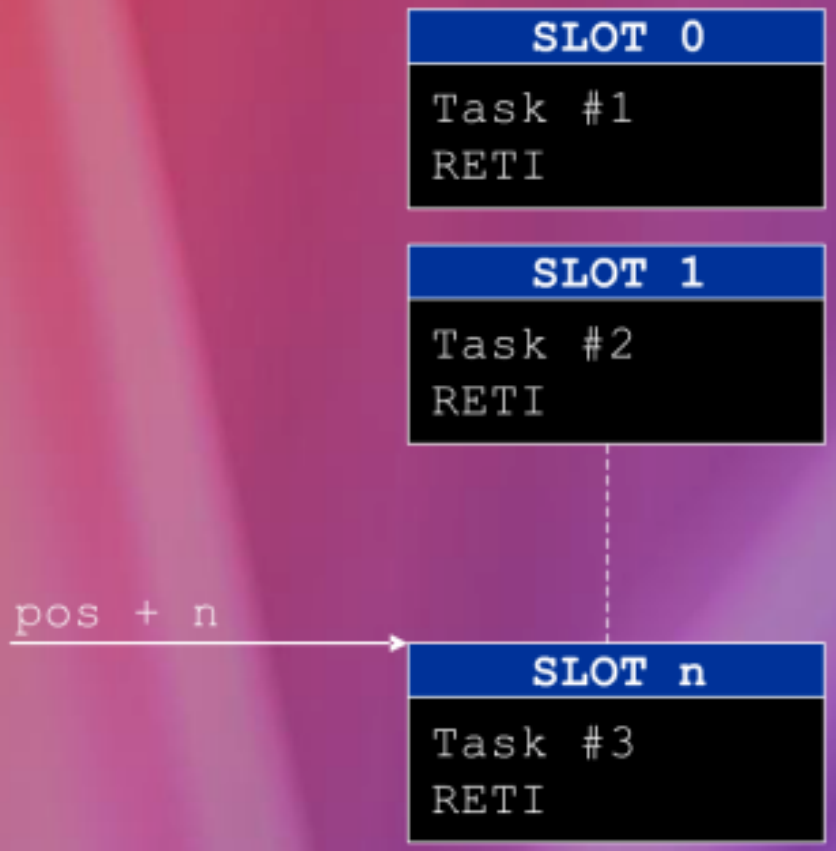


Slots are serviced at 1/n of the interrupt frequency

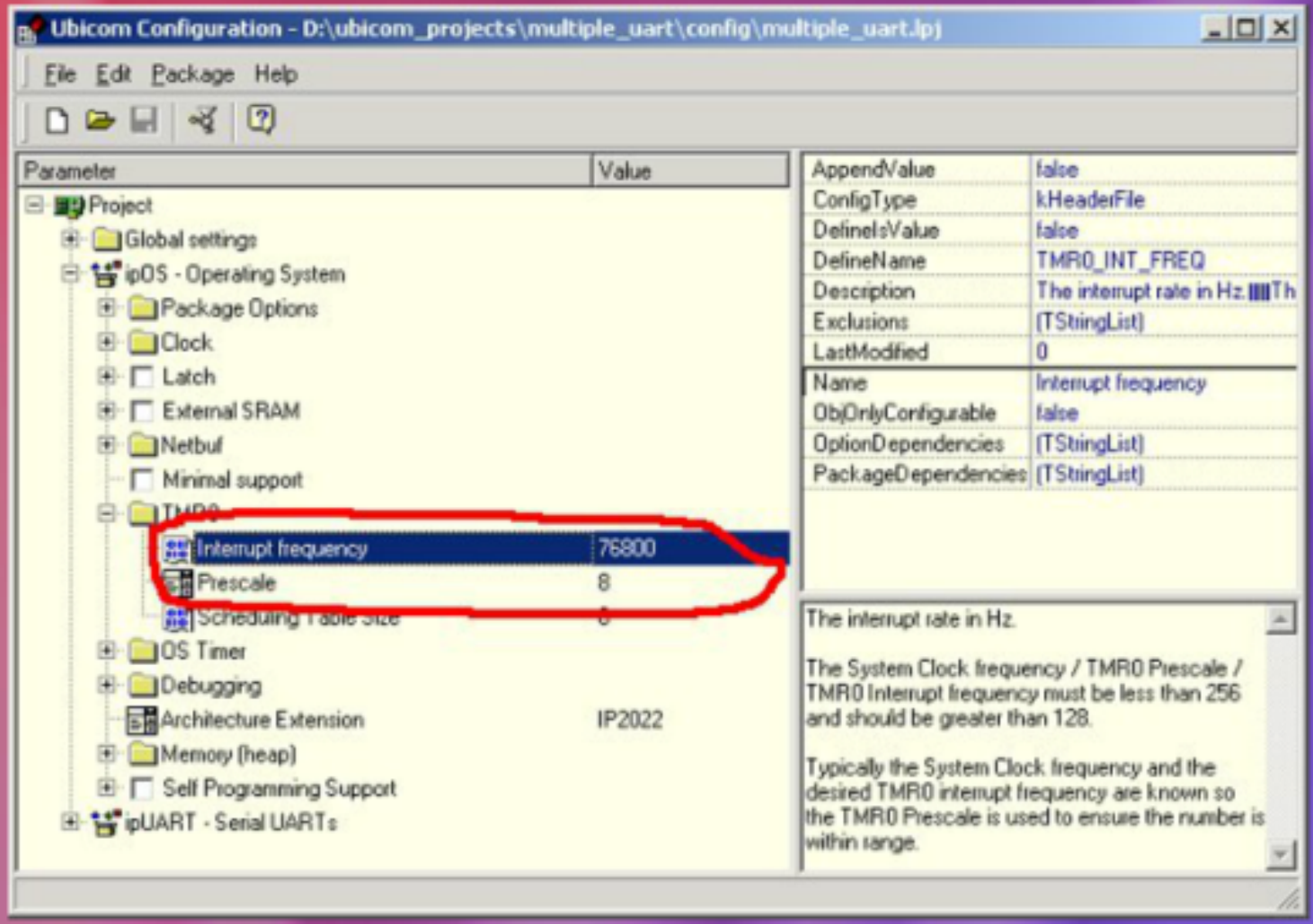
TMR0 Scheduling table



TMR0 Scheduling table



A closer look at TMR0



76800 / 8 = 9600 Baud

Stream Engine by UBICOM

UBICOM

```
;
; TMR0 slot code
;
; every task is handled at 1/8 of the interrupt frequency
;

_tmr0_slot0:
_tmr0_slot2:
_tmr0_slot4:
_tmr0_slot6:
    #include "echo_uart_vp_rx_isr_inline.S"
    #include "ip2k/tmr0_isr_end.S"

_tmr0_slot1:
_tmr0_slot5:
    #include "ip2k/tmr0_isr_end.S"

_tmr0_slot3:
    #include "echo_uart_vp_tx_isr_inline.S"
    #include "ip2k/tmr0_isr_end.S"

_tmr0_slot7:
;   #include "ip2k/ostimer_isr.S"
    #include "ip2k/tmr0_isr_end.S"

/*
 * isr subroutines
 */

    #include "echo_uart_vp_isr_subroutines.S"
```

A closer look at TMR0 and ISR

- tasks have to be short
- Handle one task per TMR0 slot
- Second UART means we need more TMR0 slots since there are more task service

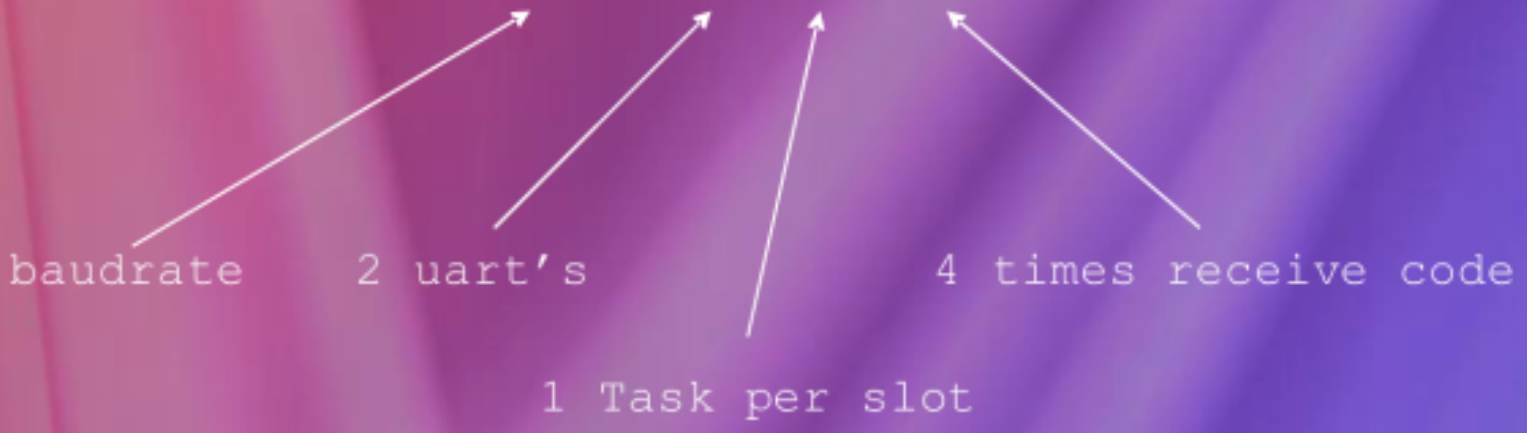
Recalculation of TMR0

2 UART's @ 9600 baud

4 times over sampling for receive

1 task per TMR0 slot

```
TMR0 Interrupt Frequency =
                    9600 x 2 x 2 x 4 = 153600
```



Recalculation of prescaler

```
128 < The System Clock frequency / TMR0 Prescale / TMR0 Interrupt <
   256

120000000 / TMR0 Prescale / 153600 = 192

TMR0 Prescale = 120000000 / 153600 / 192

TMR0 Prescale ~ 4
```

Use Unity instantiate second UART and install the ISR tasks