NOYECUBE

home tag guest book

C++

Hardware modeling with QEMU

by noyecube 2020. 8. 15.

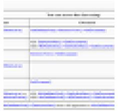
code_sample.zip
0.01MB

1. The need for QEMU

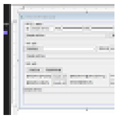
QEMU is open source software that provides virtualization and emulator capabilities. It features high performance using dynamic translators and can run the entire software stack on top of a virtual machine. It was created to emulate the x86 system, but now provides an emulation environment for various processors such as ARM, MIPS, and ALPHA.
Using an emulator has the advantage of being able to develop a system without actual physical hardware. QEMU provides emulation not only for the processor but also for the device unit, so you can get the same effect as running it on the actual target board.

Recent Posts
Popular Posts

[MFC] obtain a pointer t...
2022.10.12

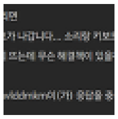


[MFC] Creating a...
2022.09.12

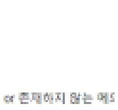


[MFC] Build Openssl...
2022.08.27

No monitor signal (tur...
2022.08.16



[MFC] C++ MiniDumpWr
2022.08.12



Software for embedded systems requires a lot of resources in the development process. In particular, debugging is difficult, and debugging equipment is also expensive. However, easy debugging is possible by using the monitor function provided by QEMU.
A typical example of QEMU being used is the Android emulator. The Android emulator is developed based on QEMU and provides a virtual target board named “Goldfish”. As a result, developers who had difficulty developing embedded systems due to complex development environments and expensive equipment can develop Android without any difficulties.

2. Hardware modeling

This article shows how to implement the UART function by configuring the memory layout of the target board based on the PowerPC architecture and implementing the interrupt processing routine using the API provided by QEMU. QEMU supports various architectures, CPUs, and reference boards, but if a separate external module is mounted on the target board to be actually used, it must be implemented directly through the hardware modeling process.
The target board is an MPC8560 based on the PowerPC architecture, and we aim to implement the UART function of this board. The basic target board based on the PowerPC architecture provided by QEMU is the MPC8544. The difference between the MPC8544 and the MPC8560 is the presence or absence of a communication processor module (CPM).

CATEGORIES (24)

Linux (1)

C++ (1)

Windows C++ Programming (8)

System Trading (8)

Programming (8)

Strategy (0)

Personal (5)

Other (1)

tag

GenerateDump,

simdjson c project,

dwm.exe, chart data,

xingapi t1857,

modeless dialog,

ebestsec, message map,

nvlddmkm, t8436,

t9945, simdjson library,

EXCEPTION_POINTERS,

libtcmalloc_minimal,

tcmalloc_minimal_large_unit

json parser, Log file,

mfc, XINGAPI, OpenAPI,

tcmalloc_minimal_unittest,

automatic trading,

crash dump,

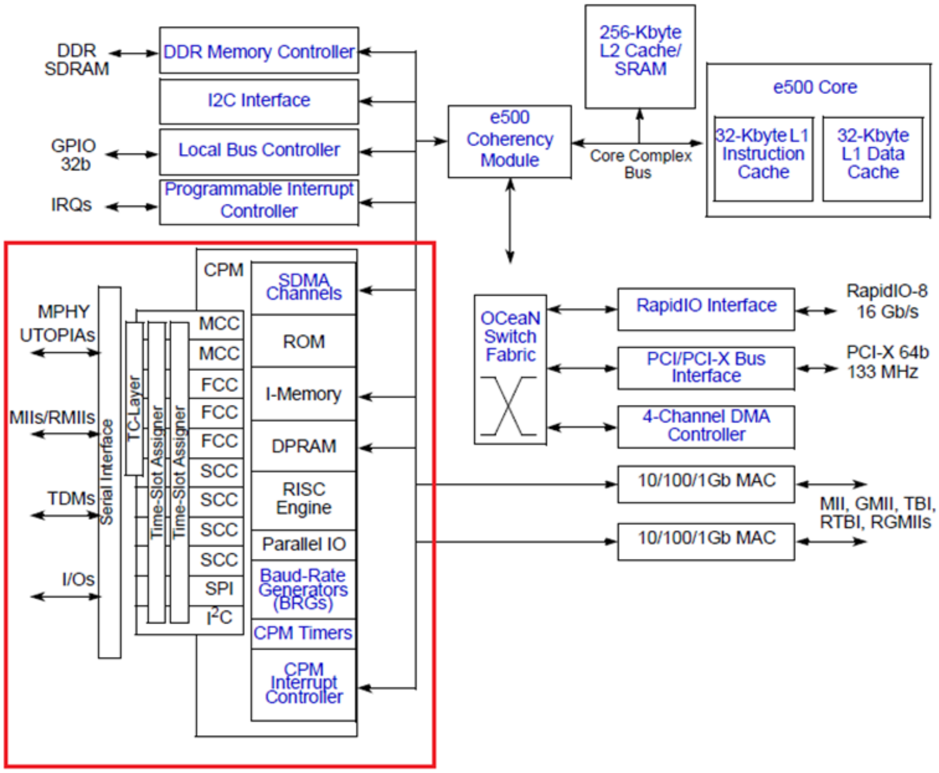
openssl windows,

Memory Dump,

openssl mfc,

fastest json parser,

t8430,



(MPC8560 block diagram)

The communication processor module (CPM) of the MPC8560 is a communication module for improving the performance of high bit-rate protocols such as UART and Fast Ethernet.

The communication processor (CP) can execute one instruction per clock and executes code in the internal ROM or I-Memory. It has an internal timer and generates an interrupt with this timer to monitor the buffer status. The shared memory area is used for data exchange between the CP and the e500 core processor. In PowerPC, the memory area is called Dual-port RAM, and in MPC8560, it has a size of 32-Kbytes. This area provides a flexible, high-speed communication medium.

eBEST Investment

Securities,

CArchive

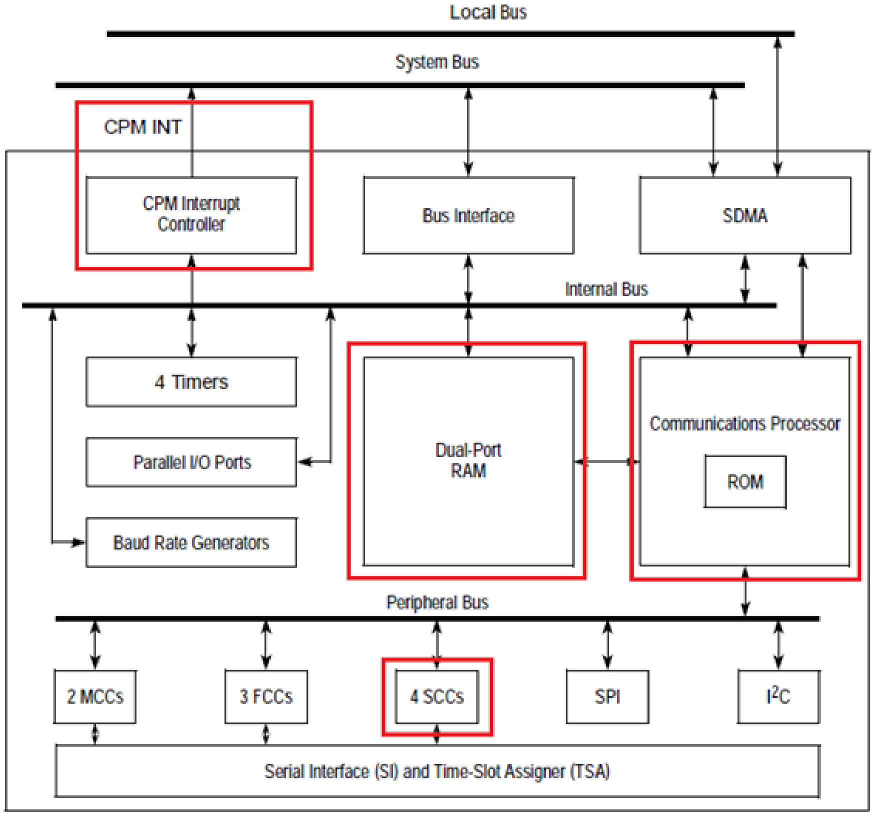
recent comments

Hi, how do I connect the...

different. CDlgXingMsg r...

IDD_DATACOLLECTOR_...

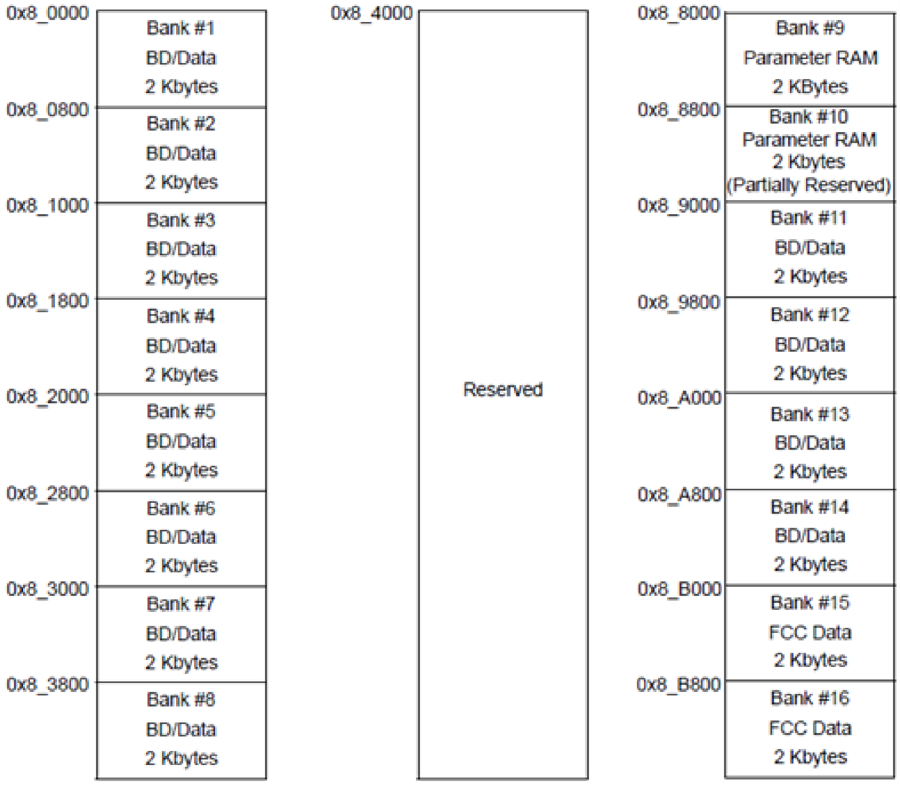
It seems that the authe...



(MPC8560 CPM block diagram)

In order to activate the UART function of the communication processor module (CPM), it is necessary to identify the modules required for functional operation in the **MPC8560 CPM block diagram .** In actual hardware, the communication processor (CP) executes the instructions loaded in the ROM or I-RAM inside the CPM independently of the e500 core and operates, but there is no CPM module code in QEMU. Therefore, it is necessary to create a CPM module and program it to operate the same as the actual hardware.

When the printf function is called in the e500 core processor where the kernel is running, the UART 1-byte output function is finally called, and the function writes the data to be output in the designated area of the dual-port RAM of the communication processor module (CPM). Also, the status register of the area where the data is recorded is monitored at a very fast cycle by the interrupt controller operated by the internal timer of the CPM, and when the status value is determined to be data input, it is output to the outside.

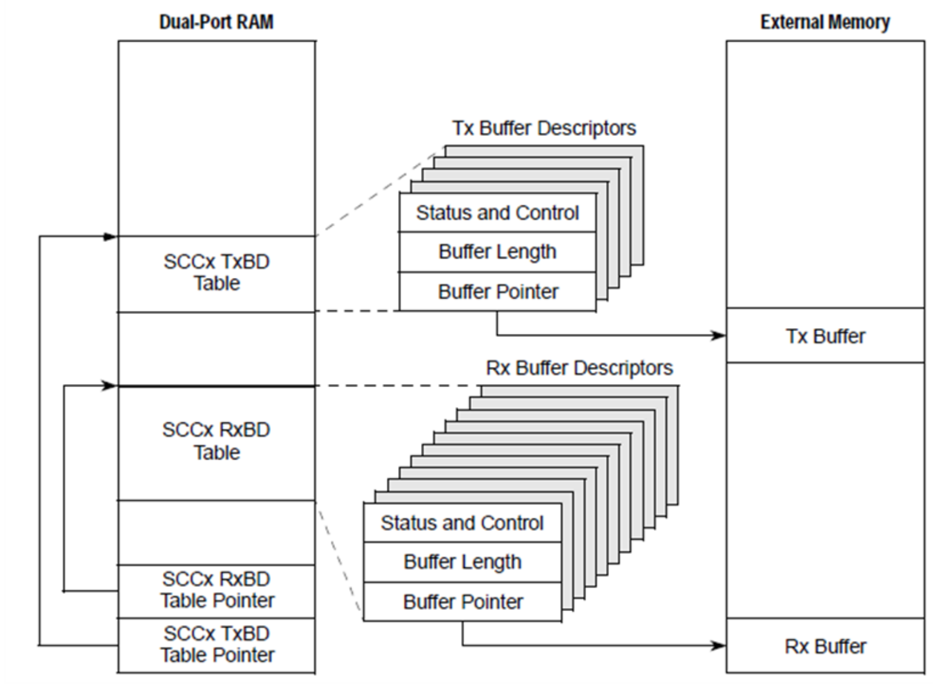


(Dual-Port RAM Memory Map)

In terms of software emulation, if the hardware polling method is implemented identically, CPU cycles are wasted. Instead, by using the memory-mapped I/O function supported by QEMU, I/O properties are given to the corresponding area of the dual-port RAM to be processed in an interrupt manner. In this way, the internal timer of the CPM does not have to be taken into account.

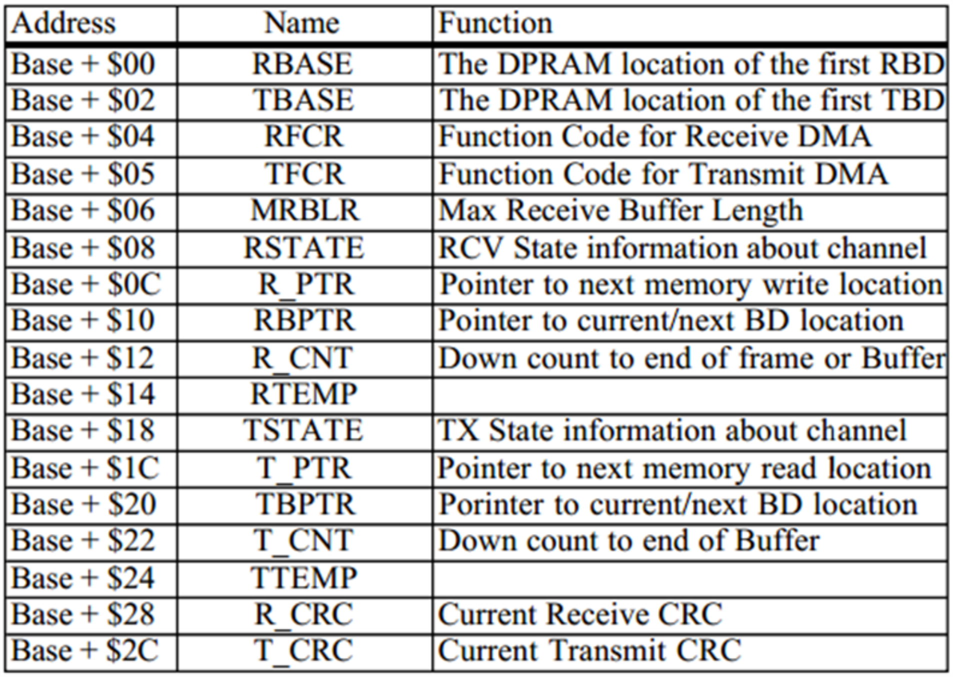
For the UART area of Dual-port RAM, you need to figure out the Dual-port RAM memory map and data storage format in the data sheet. The UART function is handled by the serial communication controller (SCC) inside the CPM, and it needs to be configured to operate in UART mode.

Data related to the SCC channel is stored in the Buffer Descriptor (BD) area of the Dual-port RAM. These values are shared by all serial controllers.



(SCC BD and Buffer Memory Structure)

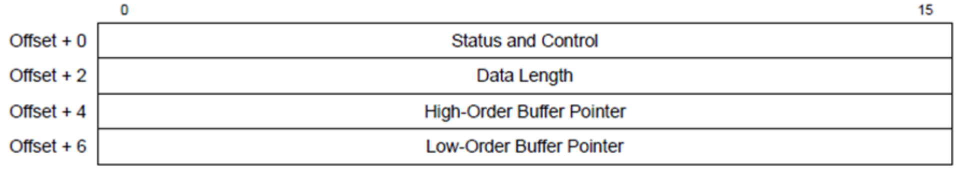
In order to activate the UART function with QEMU, you need to understand the hardware specifications in detail. In particular, it is necessary to accurately grasp the operating scenario, but since the scenario is not specifically described in the data sheet, the operating principle must be identified through the kernel source code or, if there is only a binary file, through analysis of the disassembly result.



Address	Name	Function
Base + $00	RBASE	The DPRAM location of the first RBD
Base + $02	TBASE	The DPRAM location of the first TBD
Base + $04	RFCR	Function Code for Receive DMA
Base + $05	TFCR	Function Code for Transmit DMA
Base + $06	MRBLR	Max Receive Buffer Length
Base + $08	RSTATE	RCV State information about channel
Base + $0C	R_PTR	Pointer to next memory write location
Base + $10	RBPTR	Pointer to current/next BD location
Base + $12	R_CNT	Down count to end of frame or Buffer
Base + $14	RTEMP	
Base + $18	TSTATE	TX State information about channel
Base + $1C	T_PTR	Pointer to next memory read location
Base + $20	TBPTR	Porinter to current/next BD location
Base + $22	T_CNT	Down count to end of Buffer
Base + $24	TTEMP	
Base + $28	R_CRC	Current Receive CRC
Base + $2C	T_CRC	Current Transmit CRC

(SCC Parameter RAM)

According to the figure above, you can see that there is a TBASE address value at an offset +2 from the memory bank base of the dual-port RAM to which SCC channel 1 is allocated. TBASE has a TX buffer descriptor table. The information in this table is as follows.



	0	15
Offset + 0	Status and Control	
Offset + 2	Data Length	
Offset + 4	High-Order Buffer Pointer	
Offset + 6	Low-Order Buffer Pointer	

(SCC Buffer Descriptors (BD))

The Status and Control field is used for exchanging status information between the e500 core processor and the CPM. Simultaneous access is not allowed. For this purpose, check the Status value and record the Control value. For example, when the e500 core processor outputs a character to UART, it checks the Status bit to see if the UART buffer is available before writing the data.

Mutual data exchange between the core processor and the communication processor module can be largely classified into four types.

- Command transmission of the core processor
  Changes the state of the SCC channel or initializes the SCC channel.
- Buffer descriptor manipulation
  Delivers the data to be transmitted and tells which memory address to store the received data. It also reports errors that occur during transmission and reception.
- Event Register
  An event occurs when the core processor writes data to a specific register. Conversely, it is used for the purpose of generating an interrupt to the core processor.
- Register property value setting
  Set the operation mode of SCC and set the clock. And it determines the type of physical interface.

  When each instruction is delivered from the core processor, the CPM performs the defined operation, and the types are as follows.
- INIT TX PARAMETERS
  Copy the TBASE value of SCC parameter RAM to TBPTR and initialize the TSTATE value to 0.

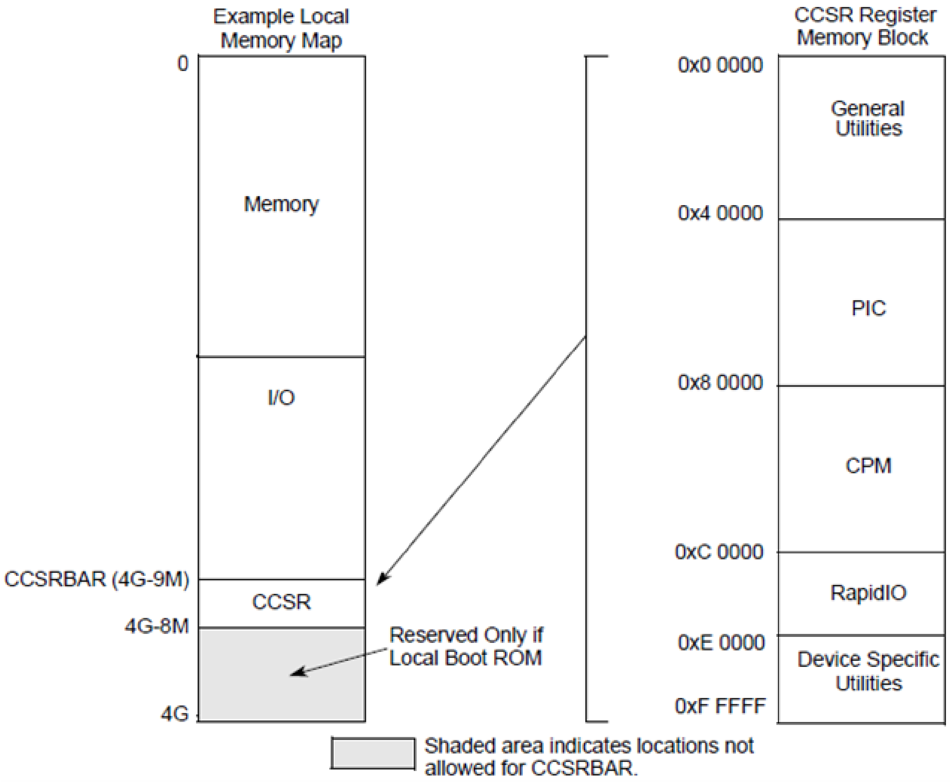
- INIT RX PARAMETERS
  Copy RBASE value of SCC parameter RAM to RBPTR and initialize RSTATE value to 0.
- ENTER HUNT MODE
  Sends a command to the channel to ignore all input data and wait until it returns to the IDLE state.
- STOP TX
  Stops waiting data transmission.
- GRACEFUL STOP
  Transmits the buffer of the TX channel to the end and then ends the transmission process.
- RESTART TX
  Resume transmission in a stopped state.

When the communication processor module (CPM) receives a data transmission command from the core processor, it operates in the following scenario. Access the first buffer descriptor by referring to the TxBD table pointer. At this time, if the “INIT TX PARAMETERS” command has not been executed in advance, TBPTR has an unknown value, so be careful. Detects whether the Ready bit value is set in the Status field of the buffer descriptor. As mentioned above, this Ready bit is polled by the internal clock of the CPM, and polled at 128 clock cycles in case of Ethernet and 64 clock cycles in case of UART. Since this is an internal operation of the CPM, the core processor is unaware of it. If the Ready bit is set, it signals that there is data to be transmitted, so the buffer length value is stored in the temporary counter area, T_CNT, and the starting address of the buffer is stored in the temporary pointer area, T_PTR. If this operation is not performed in the hardware, you need to look at the TX clock setting part again. When transmission starts, T_CNT decreases and T_PTR increases. At this time, if the value of TSTATE is set to 0, care must be taken as the core processor may intervene in the middle of transmission. Several buffers can be used depending on the length of the transmission data. Therefore, when the buffer is emptied, it is always necessary to check whether data exists in the next buffer.

When a low-level I/O function is called from the kernel, data is directly written to the CPM register and an event is generated. However, when a high-level I/O function such as printf is called, interrupts are exchanged between the core processor and the CPM as well as memory I/O operations. . If a character string exceeding bytes is input to printf, the loop runs internally so that it is stored in the TX buffer in units of bytes. At this time, in order for the loop to be called, the interrupt generated by the CPM must be processed by the core processor and the ACK must be delivered to the CPM. . The way to solve this is in the interrupt section below.

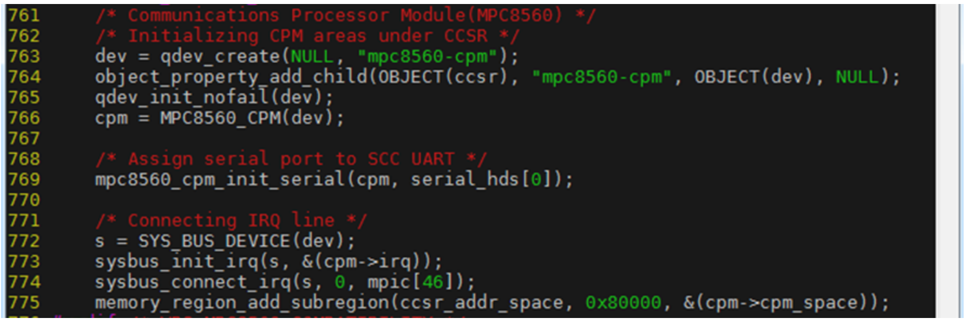
Memory

To add the dual-port RAM of the communication processor module (CPM) to the existing memory map, the structure of the upper memory map must be identified and the address assigned to the dual-port RAM area must be checked.


Example Local Memory Map
0
Memory
I/O
CCSRBAR (4G-9M)
CCSR
4G-8M
Reserved Only if Local Boot ROM
4G
Shaded area indicates locations not allowed for CCSRBAR.
CCSR Register Memory Block
0x0 0000
General Utilities
0x4 0000
PIC
0x8 0000
CPM
0xC 0000
RapidIO
0xE 0000
Device Specific Utilities
0xF FFFF

(Top-Level Register Map)

Looking at the figure above, you can see that the CPM memory area is lo cated in the lower tree of the Configuration, Control, and Status Register Map (CCSR) area, so you must allocate the CPM memory area under the C CSR area. Prior to this, device registration is required so that QEMU can r ecognize the CPM as a peripheral device.



```
    /* Communications Processor Module(MPC8560) */
    /* Initializing CPM areas under CCSR */
    dev = qdev_create(NULL, "mpc8560-cpm");
    object_property_add_child(OBJECT(ccsr), "mpc8560-cpm", OBJECT(dev), NULL);
    qdev_init_nofail(dev);
    cpm = MPC8560_CPM(dev);

    /* Assign serial port to SCC UART */
    mpc8560_cpm_init_serial(cpm, serial_hds[0]);

    /* Connecting IRQ line */
    s = SYS_BUS_DEVICE(dev);
    sysbus_init_irq(s, &(cpm->irq));
    sysbus_connect_irq(s, 0, mpic[46]);
    memory_region_add_subregion(ccsr_addr_space, 0x80000, &(cpm->cpm_space));
```

(CPM device initialization, hw/ppc/e500.c)

Since the CPM to be implemented is a peripheral device of the e500 processor-based board of the PowerPC architecture, insert the above initialization code into the “ppce500_init” function of the “hw/ppc/e500.c” file where the e500 processor is implemented. The role of each function is as follows.

- DeviceState qdev_create(BusState *bus, const char *name) This function creates a new device. Only objects are created, but the actual operation is performed after qdev_init-type functions are called.

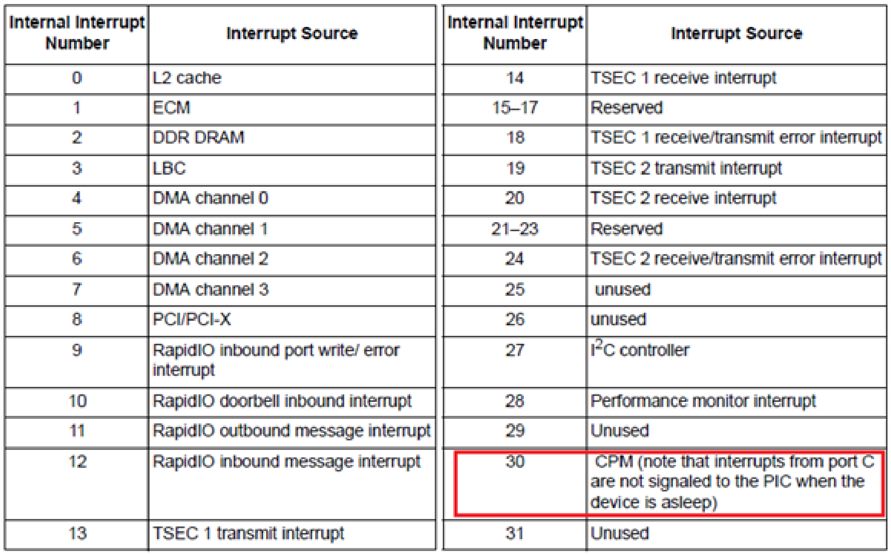
- void object_property_add_child(Object *obj, const char *name, Object *child, Error **errp)
  Since the CPM object was created with the qdev_create function, the object tree must be configured. Since we know that CPM is a sub-module of CCSR, we register it as a child object of CCSR. At this time, you need to get a CCSR object, and you can use the OBJECT macro.

- qdev_init_nofail(DeviceState *dev)
  Previously, the CPM object was created with the create function, and the object was registered as a sub-module of CCSR. Now, call the init function so that the actual operation can be performed. This function is similar to the qdev_init function, but if an error occurs, the program terminates instead of returning the error value.

- void mpc8560_cpm_init_serial(CPMState *s, CharDriverState *chr)
  This
  function registers the handler of the serial device, sets the size under fifo creation, sets the polling cycle, timeout, and baud rate. The inside of the function is covered in detail in the part describing the UART function setting.

- sysbus_connect_irq(SysBusDevice *dev, int n, qemu_irq irq)
  To transfer the IRQ generated by CPM to the core processor, this function must be used to set. To do this, the CPM must check the assigned internal interrupt number. If you look at the picture below,

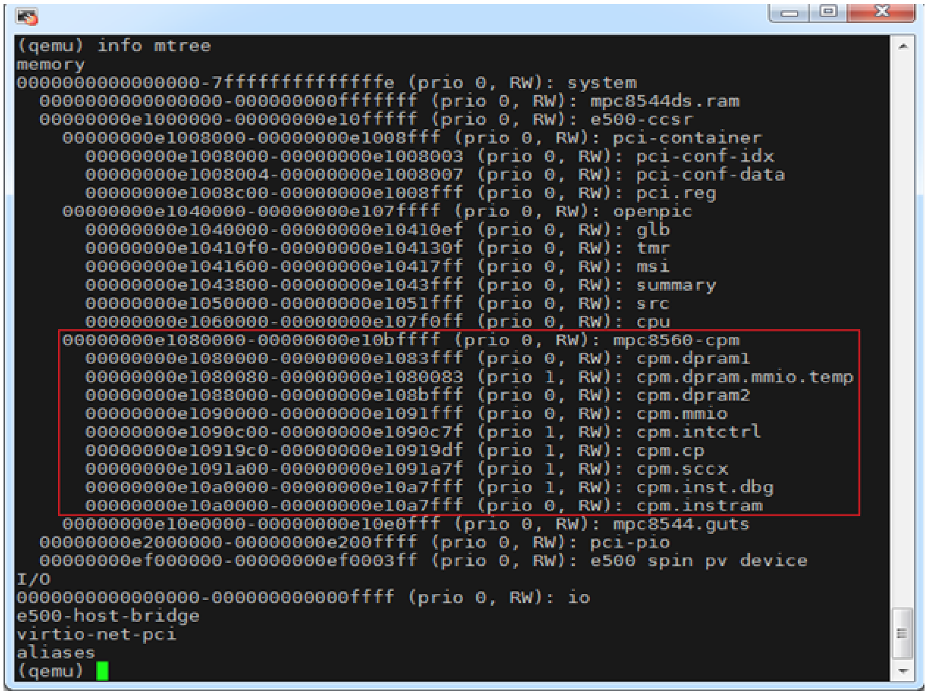
you can see that number 30 is allocated as an internal interrupt number. However, you can see that the previous QEMU code passes “mpic[46]” as an argument value. This is because QEMU manages external and internal interrupts in one array, so “mpic” array parameters from 0 to 15 are external interrupts. because it is assigned to



Internal Interrupt Number	Interrupt Source
0	L2 cache
1	ECM
2	DDR DRAM
3	LBC
4	DMA channel 0
5	DMA channel 1
6	DMA channel 2
7	DMA channel 3
8	PCI/PCI-X
9	RapidIO inbound port write/ error interrupt
10	RapidIO doorbell inbound interrupt
11	RapidIO outbound message interrupt
12	RapidIO inbound message interrupt
13	TSEC 1 transmit interrupt
14	TSEC 1 receive interrupt
15–17	Reserved
18	TSEC 1 receive/transmit error interrupt
19	TSEC 2 transmit interrupt
20	TSEC 2 receive interrupt
21–23	Reserved
24	TSEC 2 receive/transmit error interrupt
25	unused
26	unused
27	I^2C controller
28	Performance monitor interrupt
29	Unused
30	CPM (note that interrupts from port C are not signaled to the PIC when the device is asleep)
31	Unused

(internal interrupt assignment)

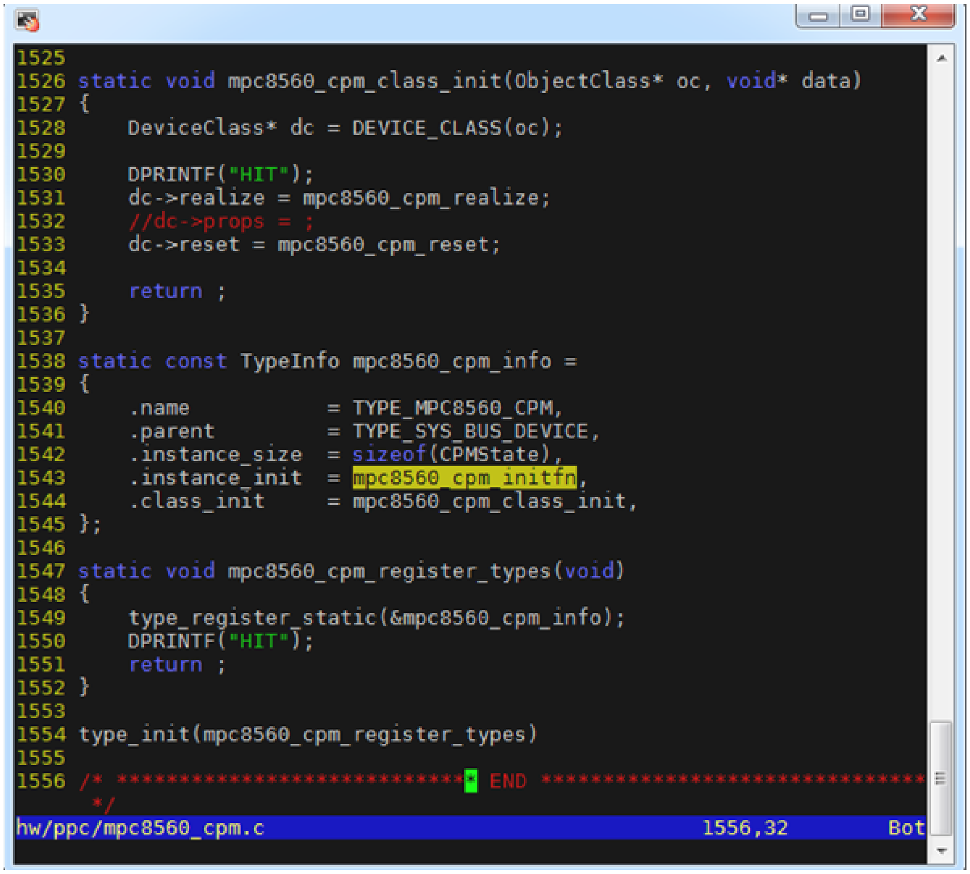
- void memory_region_add_subregion()
  This function adds a subregion to the existing memory region in the form of a tree. Since the CPM memory area is located at offset 0x80000 from the start of the CCSR memory area, 0x80000 is passed as an offset factor. The result can be seen in the figure below. You can enter QEMU monitor mode by entering “ctrl+a+c” in QEMU, and in this state, you can check all information of the internal system. Memory tree information can be checked with the “info mtree” command. It can be seen that the memory of the entire system is located at the top and the memory area of “mpc8560-cpm” is located at the bottom of “e500-ccsr”. The lower memory area of “mpc8560-cpm” was additionally registered in the CPM device.



```
(qemu) info mtree
memory
0000000000000000-7ffffffffffffffe (prio 0, RW): system
  0000000000000000-00000000ffffffff (prio 0, RW): mpc8544ds.ram
  00000000e1000000-00000000e10fffff (prio 0, RW): e500-ccsr
    00000000e1008000-00000000e1008fff (prio 0, RW): pci-container
      00000000e1008000-00000000e1008003 (prio 0, RW): pci-conf-idx
      00000000e1008004-00000000e1008007 (prio 0, RW): pci-conf-data
      00000000e1008c00-00000000e1008fff (prio 0, RW): pci.reg
    00000000e1040000-00000000e107ffff (prio 0, RW): openpic
      00000000e1040000-00000000e10410ef (prio 0, RW): glb
      00000000e10410f0-00000000e104130f (prio 0, RW): tmr
      00000000e1041600-00000000e10417ff (prio 0, RW): msi
      00000000e1043800-00000000e1043fff (prio 0, RW): summary
      00000000e1050000-00000000e1051fff (prio 0, RW): src
      00000000e1060000-00000000e107f0ff (prio 0, RW): cpu
    00000000e1080000-00000000e10bffff (prio 0, RW): mpc8560-cpm
      00000000e1080000-00000000e1083fff (prio 0, RW): cpm.dpram1
      00000000e1080080-00000000e1080083 (prio 1, RW): cpm.dpram.mmio.temp
      00000000e1088000-00000000e108bfff (prio 0, RW): cpm.dpram2
      00000000e1090000-00000000e1091fff (prio 0, RW): cpm.mmio
      00000000e1090c00-00000000e1090c7f (prio 1, RW): cpm.intctrl
      00000000e10919c0-00000000e10919df (prio 1, RW): cpm.cp
      00000000e1091a00-00000000e1091a7f (prio 1, RW): cpm.sccx
      00000000e10a0000-00000000e10a7fff (prio 1, RW): cpm.inst.dbg
      00000000e10a0000-00000000e10a7fff (prio 0, RW): cpm.instram
    00000000e10e0000-00000000e10e0fff (prio 0, RW): mpc8544.guts
  00000000e2000000-00000000e200ffff (prio 0, RW): pci-pio
  00000000ef000000-00000000ef0003ff (prio 0, RW): e500 spin pv device
I/O
0000000000000000-000000000000ffff (prio 0, RW): io
e500-host-bridge
virtio-net-pci
aliases
(qemu) 
```

(memory tree information)

Through the above process, CPM was created and activated as a peripheral device of the e500 core, and the memory area is also secured. Now, we need to implement the detailed operation through internal programming of the actual CPM. To do so, you need to know the device programming conventions of QEMU. This convention is very similar to the way Linux device drivers work. First, create a structure to contain the information of the peripheral device to be created, and call the initialization function at the device initialization stage.



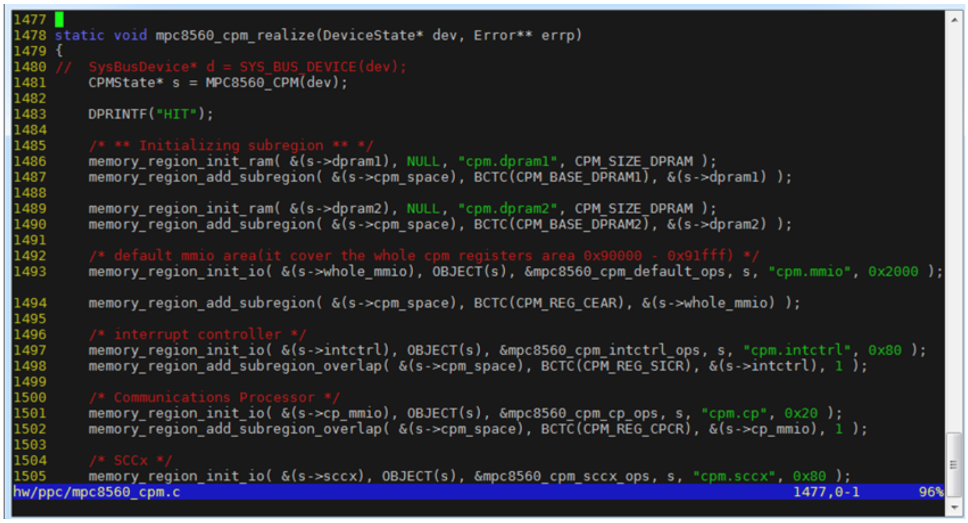
```
static void mpc8560_cpm_class_init(ObjectClass* oc, void* data)
{
    DeviceClass* dc = DEVICE_CLASS(oc);

    DPRINTF("HIT");
    dc->realize = mpc8560_cpm_realize;
    //dc->props = ;
    dc->reset = mpc8560_cpm_reset;

    return ;
}

static const TypeInfo mpc8560_cpm_info =
{
    .name           = TYPE_MPC8560_CPM,
    .parent         = TYPE_SYS_BUS_DEVICE,
    .instance_size  = sizeof(CPMState),
    .instance_init  = mpc8560_cpm_initfn,
    .class_init     = mpc8560_cpm_class_init,
};

static void mpc8560_cpm_register_types(void)
{
    type_register_static(&mpc8560_cpm_info);
    DPRINTF("HIT");
    return ;
}

type_init(mpc8560_cpm_register_types)

/* ***************************** END ******************************* */
hw/ppc/mpc8560_cpm.c                                        1556,32        Bot
```

(initialization part of CPM device file)

The type_init() function is used to call the initialization function of the newly added device. Before that, save the device name and the function pointer to be called during initialization in the TypeInfo type structure. The function that actually performs device initialization is a function with _realize appended to the suffix, and in CPM, the lower memory area of CPM is initialized.

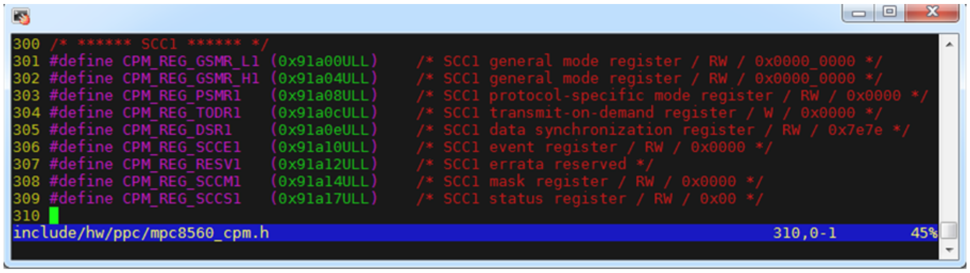
For memory area initialization, simple I/O properties and Memory-mapped I/O properties can be selected.

Since there are two dual-port RAM areas in CPM, they were named “cpm.dpram1” and “cpm.dpram2” respectively and configured as lower memory areas of CPM. Among them, the memory-mapped I/O property is given to the address area where the status value is changed when performing the UART function. The API that can give this property is the memory_region_init_io() function.



```
static void mpc8560_cpm_realize(DeviceState* dev, Error** errp)
{
//  SysBusDevice* d = SYS_BUS_DEVICE(dev);
    CPMState* s = MPC8560_CPM(dev);

    DPRINTF("HIT");

    /* ** Initializing subregion ** */
    memory_region_init_ram( &(s->dpram1), NULL, "cpm.dpram1", CPM_SIZE_DPRAM );
    memory_region_add_subregion( &(s->cpm_space), BCTC(CPM_BASE_DPRAM1), &(s->dpram1) );

    memory_region_init_ram( &(s->dpram2), NULL, "cpm.dpram2", CPM_SIZE_DPRAM );
    memory_region_add_subregion( &(s->cpm_space), BCTC(CPM_BASE_DPRAM2), &(s->dpram2) );

    /* default mmio area(it cover the whole cpm registers area 0x90000 - 0x91fff) */
    memory_region_init_io( &(s->whole_mmio), OBJECT(s), &mpc8560_cpm_default_ops, s, "cpm.mmio", 0x2000 );

    memory_region_add_subregion( &(s->cpm_space), BCTC(CPM_REG_CEAR), &(s->whole_mmio) );

    /* interrupt controller */
    memory_region_init_io( &(s->intctrl), OBJECT(s), &mpc8560_cpm_intctrl_ops, s, "cpm.intctrl", 0x80 );
    memory_region_add_subregion_overlap( &(s->cpm_space), BCTC(CPM_REG_SICR), &(s->intctrl), 1 );

    /* Communications Processor */
    memory_region_init_io( &(s->cp_mmio), OBJECT(s), &mpc8560_cpm_cp_ops, s, "cpm.cp", 0x20 );
    memory_region_add_subregion_overlap( &(s->cpm_space), BCTC(CPM_REG_CPCR), &(s->cp_mmio), 1 );

    /* SCCx */
    memory_region_init_io( &(s->sccx), OBJECT(s), &mpc8560_cpm_sccx_ops, s, "cpm.sccx", 0x80 );
hw/ppc/mpc8560_cpm.c                                                              1477,0-1       96%
```

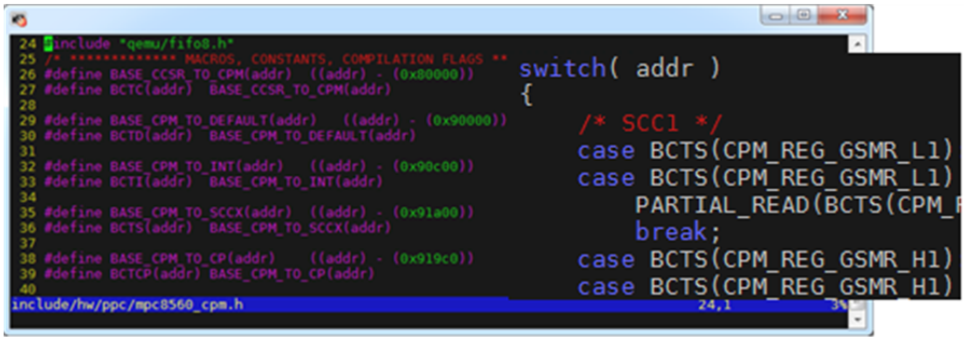
(CPM lower memory area initialization)



```
/* ****** SCC1 ****** */
#define CPM_REG_GSMR_L1 (0x91a00ULL)    /* SCC1 general mode register / RW / 0x0000_0000 */
#define CPM_REG_GSMR_H1 (0x91a04ULL)    /* SCC1 general mode register / RW / 0x0000_0000 */
#define CPM_REG_PSMR1   (0x91a08ULL)    /* SCC1 protocol-specific mode register / RW / 0x0000 */
#define CPM_REG_TODR1   (0x91a0cULL)    /* SCC1 transmit-on-demand register / W / 0x0000 */
#define CPM_REG_DSR1    (0x91a0eULL)    /* SCC1 data synchronization register / RW / 0x7e7e */
#define CPM_REG_SCCE1   (0x91a10ULL)    /* SCC1 event register / RW / 0x0000 */
#define CPM_REG_RESV1   (0x91a12ULL)    /* SCC1 errata reserved */
#define CPM_REG_SCCM1   (0x91a14ULL)    /* SCC1 mask register / RW / 0x0000 */
#define CPM_REG_SCCS1   (0x91a17ULL)    /* SCC1 status register / RW / 0x00 */

include/hw/ppc/mpc8560_cpm.h                                                   310,0-1        45%
```

(Filled with Chinese characters and Chinese characters after reading CPM's register map and data sheet)

When the memory-mapped I/O property is set from 0x100 to 0x110, the pre-set handler is called when a memory access is detected at 0x104. At this time, the memory address value passed to the handler is 0x4, not 0x104. However, since each register address in the register map has an absolute address, a separate calculation is required. To improve readability, it is desirable to use register names with absolute address values as they are, so the following macro is defined separately.



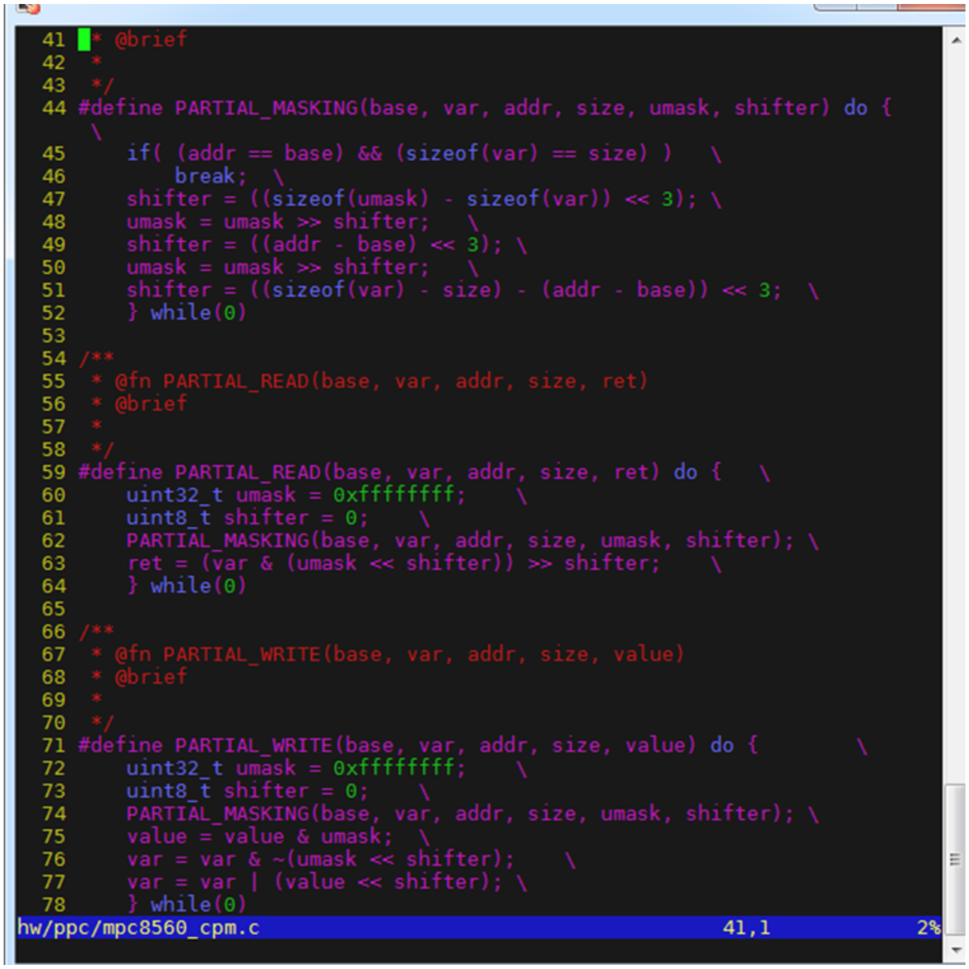
```
#include "qemu/fifo8.h"
/* ************** MACROS, CONSTANTS, COMPILATION FLAGS **
#define BASE_CCSR_TO_CPM(addr)  ((addr) - (0x80000))
#define BCTC(addr)  BASE_CCSR_TO_CPM(addr)

#define BASE_CPM_TO_DEFAULT(addr)   ((addr) - (0x90000))
#define BCTD(addr)  BASE_CPM_TO_DEFAULT(addr)

#define BASE_CPM_TO_INT(addr)   ((addr) - (0x90c00))
#define BCTI(addr)  BASE_CPM_TO_INT(addr)

#define BASE_CPM_TO_SCCX(addr)  ((addr) - (0x91a00))
#define BCTS(addr)  BASE_CPM_TO_SCCX(addr)

#define BASE_CPM_TO_CP(addr)    ((addr) - (0x919c0))
#define BCTCP(addr) BASE_CPM_TO_CP(addr)

include/hw/ppc/mpc8560_cpm.h                                     24,1           3%
```

```
switch( addr )
{
    /* SCC1 */
    case BCTS(CPM_REG_GSMR_L1)
    case BCTS(CPM_REG_GSMR_L1)
        PARTIAL_READ(BCTS(CPM_
        break;
    case BCTS(CPM_REG_GSMR_H1)
    case BCTS(CPM_REG_GSMR_H1)
```

(absolute address-relative address macro)

In the above figure, when the CPM_REG_SCCE1 register is accessed, the handler registered at SCC1 base address 0x91a00 is called and the offset of CPM_REG_SCCE1 is passed as a factor.

Much of the emulation process is dedicated to register manipulation. Registers use bit operators, and if macros are not used, code readability is reduced. Normally, when operating registers, read from bit X to bit size Y, manipulate the value, and then write it back to the same place. So, I defined and used the PARTIAL_XXX macro with a prefix meaning partial manipulation. (See the figure below)



```
 * @brief
 *
 */
#define PARTIAL_MASKING(base, var, addr, size, umask, shifter) do {
 \
    if( (addr == base) && (sizeof(var) == size) )   \
        break;  \
    shifter = ((sizeof(umask) - sizeof(var)) << 3); \
    umask = umask >> shifter;    \
    shifter = ((addr - base) << 3); \
    umask = umask >> shifter;    \
    shifter = ((sizeof(var) - size) - (addr - base)) << 3;  \
    } while(0)

/**
 * @fn PARTIAL_READ(base, var, addr, size, ret)
 * @brief
 *
 */
#define PARTIAL_READ(base, var, addr, size, ret) do {   \
    uint32_t umask = 0xffffffff;    \
    uint8_t shifter = 0;    \
    PARTIAL_MASKING(base, var, addr, size, umask, shifter); \
    ret = (var & (umask << shifter)) >> shifter;    \
    } while(0)

/**
 * @fn PARTIAL_WRITE(base, var, addr, size, value)
 * @brief
 *
 */
#define PARTIAL_WRITE(base, var, addr, size, value) do {        \
    uint32_t umask = 0xffffffff;    \
    uint8_t shifter = 0;    \
    PARTIAL_MASKING(base, var, addr, size, umask, shifter); \
    value = value & umask;  \
    var = var & ~(umask << shifter);    \
    var = var | (value << shifter); \
    } while(0)
hw/ppc/mpc8560_cpm.c                                    41,1          2%
```

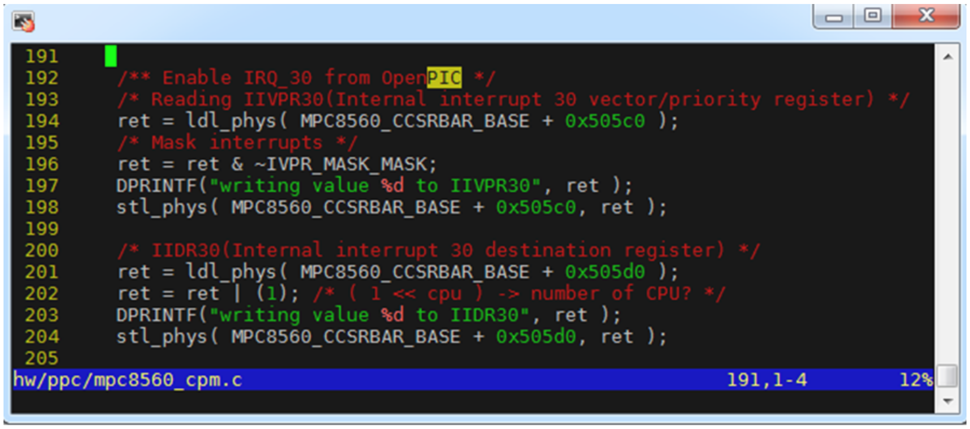
(macro for register bit masking)

interrupt

When an interrupt is normally transmitted from the peripheral device to the core processor, an ACK is transmitted to the core processor. When thi s ACK is received, the corresponding interrupt bit in the Pending register can be initialized. QEMU provides the “qemu_irq” function to generate an interrupt.

In the case of a kernel that operates on real hardware, most of the hardware initialization is performed in the bootloader. **In the case of a custom operating system other than Linux, the hardware initialized in the bootloader is often not initialized separately. As a result, code that works normally on real hardware may not work on QEMU.**

As a result , it is advantageous to include the hardware initialization code in the kernel to support the emulator when developing the kernel because separate debugging is required . Otherwise, you have to manipulate the hardware in QEMU, but you must be careful because the hardware itself will initialize itself and patch dependent on a specific kernel image.

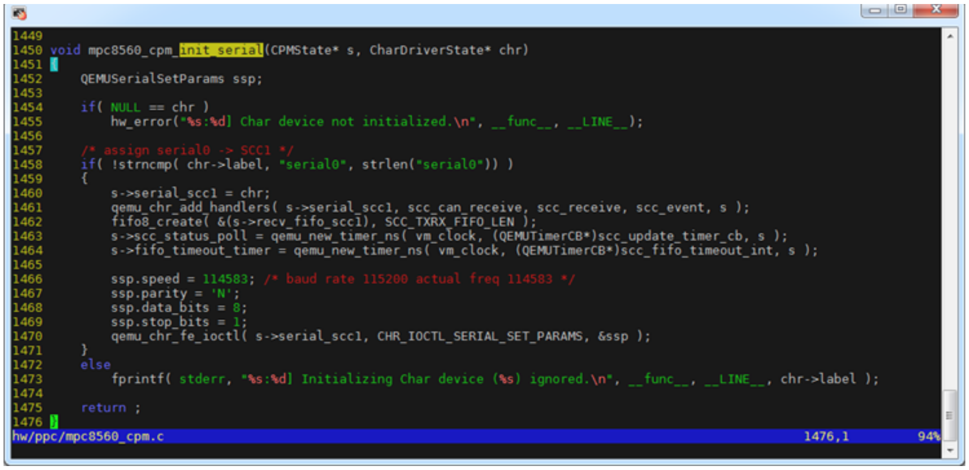


```

    /** Enable IRQ_30 from OpenPIC */
    /* Reading IIVPR30(Internal interrupt 30 vector/priority register) */
    ret = ldl_phys( MPC8560_CCSRBAR_BASE + 0x505c0 );
    /* Mask interrupts */
    ret = ret & ~IVPR_MASK_MASK;
    DPRINTF("writing value %d to IIVPR30", ret );
    stl_phys( MPC8560_CCSRBAR_BASE + 0x505c0, ret );

    /* IIDR30(Internal interrupt 30 destination register) */
    ret = ldl_phys( MPC8560_CCSRBAR_BASE + 0x505d0 );
    ret = ret | (1); /* ( 1 << cpu ) -> number of CPU? */
    DPRINTF("writing value %d to IIDR30", ret );
    stl_phys( MPC8560_CCSRBAR_BASE + 0x505d0, ret );

hw/ppc/mpc8560_cpm.c                                       191,1-4        12%
```

(Example of undesirable hardware interrupt initialization code)

UART function

QEMU provides character device emulation. UART function can be used by using this character device. Attributes to be set by the user include send/receive processing handlers, fifo size settings, poll mode, and transmission speed related settings. An example is shown in the figure below.



```c

void mpc8560_cpm_init_serial(CPMState* s, CharDriverState* chr)
{
    QEMUSerialSetParams ssp;

    if( NULL == chr )
        hw_error("%s:%d] Char device not initialized.\n", __func__, __LINE__);

    /* assign serial0 -> SCC1 */
    if( !strncmp( chr->label, "serial0", strlen("serial0")) )
    {
        s->serial_scc1 = chr;
        qemu_chr_add_handlers( s->serial_scc1, scc_can_receive, scc_receive, scc_event, s );
        fifo8_create( &(s->recv_fifo_scc1), SCC_TXRX_FIFO_LEN );
        s->scc_status_poll = qemu_new_timer_ns( vm_clock, (QEMUTimerCB*)scc_update_timer_cb, s );
        s->fifo_timeout_timer = qemu_new_timer_ns( vm_clock, (QEMUTimerCB*)scc_fifo_timeout_int, s );

        ssp.speed = 114583; /* baud rate 115200 actual freq 114583 */
        ssp.parity = 'N';
        ssp.data_bits = 8;
        ssp.stop_bits = 1;
        qemu_chr_fe_ioctl( s->serial_scc1, CHR_IOCTL_SERIAL_SET_PARAMS, &ssp );
    }
    else
        fprintf( stderr, "%s:%d] Initializing Char device (%s) ignored.\n", __func__, __LINE__, chr->label );

    return ;
}
hw/ppc/mpc8560_cpm.c                                                                 1476,1        94%
```

(Character device initialization code)

sympathy Subscribe

tag

cpm dual port ram EMULATOR mpc8560

PowerPC qemu UART emulator

0 comments

name	password

Please enter your valuable comments.

□ secret message registration

MAIL. noyecube.gmail.com