Design and development of microkernel for ARM1176JZF-S

[1]A.Sneha, [2]P.Shakira, [3]N. Neelima

[12] Department of Electronics and communications Engineering, MRITS, HYDERABAD.

[1]sneha.anantharam@gmail.com, [2] shakira15210@gmail.com, [3]nagalla1981@gmail.com

Abstract- **In this project a 32-task Real Time Microkernel is designed using which multi tasking can be done on the targeted processor ARM1176JZFS from ARM limited. The Micro kernel includes a preemptive priority scheduler and context switching modules for carrying out multi-tasking. Routines to create and manage tasks will be developed. Once created, the tasks will be scheduled by our own scheduler automatically. Subsequently, inter task communication mechanism is added to this scheduler, to make it a small real-time kernel. Tool we are using RVDS(Real view development suit)**

Keywords— **I.P.C, thread creation, scheduling, context switching.**

I Introduction

Real-time systems are those systems whose response is deterministic in time. A real-time microkernel is the near-minimum amount of software that can provide the mechanisms needed to implement a real-time operating system. These mechanisms include low-level address space management, thread management, and inter-process communication (I.P.C). As an operating system design approach, microkernels permit typical operating system services, such as device drivers, protocol stacks, file systems and user interface code, to run in user space. Throughout this document OS and Microkernel are used for same meaning though OS includes device drivers and file system management, which is not present in microkernel.

Two sets of functions are developed in this project. First one is Kernel functions and second is application functions. Kernel functions are mainly for carrying out task creation, multi-tasking and Inter task communication. The number of application functions can be from 1 to 32. Each of these application functions is created as a task by the microkernel and scheduled by the pre-emptive priority scheduler. Multi tasking of these application tasks is demonstrated in this project. Following section explains the responsibilities of the various functions implemented in the Kernel along with the function prototypes of the Kernel functions.

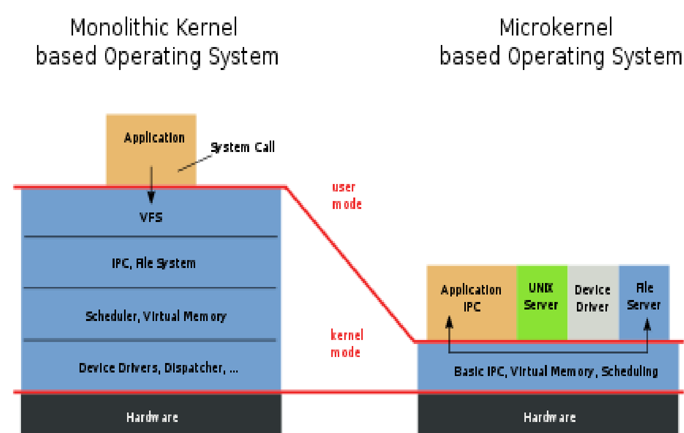

Fig 1 Difference between monolithic kernel and microkernel

In computing, the kernel is the central component of most computer operating systems; it is a bridge between applications and the actual data processing done at the hardware level. The kernel's responsibilities include managing the system's resources (the communication between hardware and software components).[1] Usually as a basic component of an operating system, a kernel can provide the lowest-level abstraction layer for the resources (especially processors and I/O devices) that application software must control to perform its function. It typically makes these facilities available to application processes through inter-process communication mechanisms and system calls.

Operating system tasks are done differently by different kernels, depending on their design and implementation. While monolithic kernels execute all the operating system code in the same address space to increase the performance of the system, microkernels run most of the operating system services in user space as servers, aiming to improve maintainability and modularity of the operating system.[2] A range of possibilities exists between these two extremes.

© 2012 JCT JOURNALS. ALL RIGHTS RESERVED

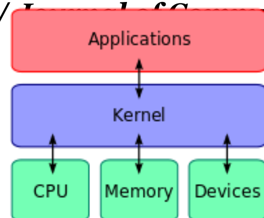

Fig 2 Kernel connecting the application software to the hardware of a computer

In most cases, the boot loader starts executing the kernel in supervisor mode. The kernel then initializes itself and starts the first process. After this, the kernel does not typically execute directly, only in response to external events (*e.g.*, via system calls used by applications to request services from the kernel, or via interrupts used by the hardware to notify the kernel of events). Additionally, the kernel typically provides a loop that is executed whenever no processes are available to run; this is often called the *idle process*.

Kernel development is considered one of the most complex and difficult tasks in programming. Its central position in an operating system implies the necessity for good performance, which defines the kernel as a critical piece of software and makes its correct design and implementation difficult. For various reasons, a kernel might not even be able to use the abstraction mechanisms it provides to other software. Such reasons include memory management concerns (for example, a user-mode function might rely on memory being subject to demand paging, but as the kernel itself provides that facility it cannot use it, because then it might not remain in memory to provide that facility) and lack of reentrancy, thus making its development even more difficult for software engineers.

A kernel will usually provide features for low-level scheduling of processes (dispatching), inter-process communication, process synchronization, context switching, manipulation of process control blocks, interrupt handling, process creation and destruction, and process suspension and resumption.

II. ARM1176JZF-S PROCESSOR

The ARM1176JZF-S processor incorporates an integer core that implements the ARM11 ARM architecture v6. It supports the ARM and Thumb™ instruction sets, Jazelle technology to enable direct execution of Java byte codes, and a range of SIMDDSP instructions that operate on 16-bit or 8-bit data values in 32-bit registers.

The ARM1176JZF-S processor features

• TrustZone™ security extensions

• Provision for *Intelligent Energy Management* (IEM™)

• High-speed *Advanced Microprocessor Bus Architecture* (AMBA) *Advanced Extensible Interface* (AXI) level two interfaces supporting prioritized

multiprocessor implementations.

• An integer core with integral Embedded ICE-RT logic

• An eight-stage pipeline

• Branch prediction with return stack

• Low interrupt latency configuration

• Internal coprocessors CP14 and CP15

• *Vector Floating-Point* (VFP) coprocessor support

• External coprocessor interface

• Instruction and Data *Memory Management Units* (MMUs), managed using

MicroTLB structures backed by a unified Main TLB

• Instruction and data caches, including a non-blocking data cache with

Hit-Under-Miss (HUM)

• Virtually indexed and physically addressed caches

• 64-bit interface to both caches

• Level one *Tightly-Coupled Memory* (TCM) that you can use as a local RAM with

DMA

• Trace support

• JTAG-based debug.

TrustZone security extensions

The ARM1176JZF-S processor supports TrustZone security extensions to provide a secure environment for software. This section summarizes processor elements that TrustZone uses. The TrustZone approach to integrated system security depends on an established trusted code base. The trusted code is a relatively small block that runs in the Secure world in the processor and provides the foundation for security throughout the system. This security applies from system boot and enforces a level of trust at each stage of a transaction.
The processor has:

• seven operating modes that can be either Secure or Non-secure

• Secure Monitor mode, that is always Secure.

© 2012 JCT JOURNALS. ALL RIGHTS RESERVED

Except when the processor is in Secure Monitor mode, the NS bit in the Secure

Configuration Register determines whether the processor runs code in the Secure or Non-secure worlds. The Secure Configuration Register is in CP15 register c1.

Secure Monitor mode is used to switch operation between the Secure and Non-secure worlds.

Secure Monitor mode uses these banked registers:

R13_mon Stack Pointer

R14_mon Link Register

SPSR_mon Saved Program Status Register

The processor implements this instruction to enter Secure Monitor mode:

SMC Secure Monitor Call, switches from one of the privileged modes to the

Secure Monitor mode.

Operating modes

In all states there are eight modes of operation:

• User mode is the usual ARM program execution state, and is used for executing

most application programs

• *Fast interrupt* (FIQ) mode is used for handling fast interrupts

• *Interrupt* (IRQ) mode is used for general-purpose interrupt handling

• Supervisor mode is a protected mode for the OS

• Abort mode is entered after a data abort or prefetch abort

• System mode is a privileged user mode for the OS

• Undefined mode is entered when an undefined instruction exception occurs.

• Secure Monitor mode is a Secure mode for the TrustZone Secure Monitor code.

Modes	Mode type	State of core	
		NS bit = 1	NS bit = 0
User	User	Non-secure	Secure
FIQ	privileged	Non-secure	Secure
IRQ	privileged	Non-secure	Secure
Supervisor	privileged	Non-secure	Secure
Abort	privileged	Non-secure	Secure
Undefined	privileged	Non-secure	Secure
System	privileged	Non-secure	Secure
Secure Monitor	privileged	Secure	Secure

Fig.3 lists the mode structure for the processor

Pipeline stages

• the two Fetch stages

• a Decode stage

• an Issue stage

• the four stages of the ARM1176JZF-S integer execution pipeline.

These eight stages make up the processor pipeline.

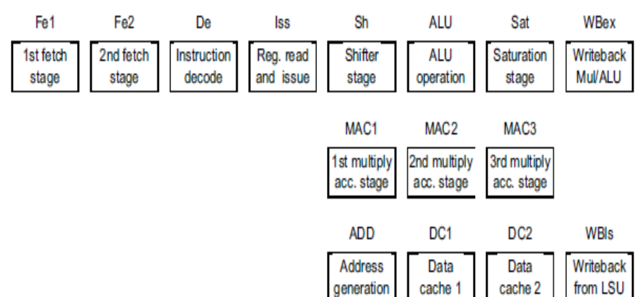

Fig.4 Pipeline stages

The pipeline operations are

Fe1: First stage of instruction fetch where address is issued to memory and data returns from memory

Fe2: Second stage of instruction fetch and branch prediction.

© 2012 JCT JOURNALS. ALL RIGHTS RESERVED

De: Instruction decode.

Iss:Register read and instruction issue.

Sh:Shifter stage.

ALU: Main integer operation calculation.

Sat: Pipeline stage to enable saturation of integer results.

WBex: Write back of data from the multiply or main execution pipelines.

MAC1: First stage of the multiply-accumulate pipeline.

MAC2:Second stage of the multiply-accumulate pipeline.

MAC3: Third stage of the multiply-accumulate pipeline.

ADD: Address generation stage.

DC1: First stage of data cache access.

DC2 :Second stage of data cache access.

WBls: Write back of data from the Load Store Unit.

By overlapping the various stages of operation, the ARM1176JZF-S processor maximizes the clock rate achievable to execute each instruction. It delivers a throughput approaching one instruction for each cycle.

The Fetch stages can hold up to four instructions, where branch prediction is performed on instructions ahead of execution of earlier instructions.

The Issue and Decode stages can contain any instruction in parallel with a predicted branch.

The Execute, Memory, and Write stages can contain a predicted branch, an ALU or multiply instruction, a load/store multiple instruction, and a coprocessor instruction in parallel execution.

PB1176JZF-S The Platform Baseboard for ARM1176JZF-S is a software and hardware development board based on ARM architecture v6:

• If the baseboard is used on its own, it is a fast software development platform with an ARM1176JZF processor and a memory system running at ASIC speed. The basic system provides a good platform for developing systems supporting ARM11 processors that feature TrustZone® Technology, CoreSight™, and *Intelligent Energy Management* (IEM™). The ARM1176JZF-S development chip is much faster than a software simulator or a core implemented in RealView Logic Tiles.

• If FPGA-based RealView Logic Tiles are stacked on the Platform Baseboard, custom AMBA v3 peripherals, processors and DSPs can be added to the ARM subsystem provided by the baseboard. The expanded system can be used to develop *Advanced Microprocessor Bus*

Architecture (AMBA®) compatible peripherals and to test ASIC designs. The fast processor core and the peripherals present in the ARM1176JZF-S development chip, PB1176JZF-S FPGA, and RealView Logic Tile FPGA enable you to develop and test complex systems operating at, or near, their target operating frequency.

The major components in the PB1176JZF-S are:

• The ARM1176JZF-S development chip

• An FPGA containing additional peripherals and controllers

• 128MB of 32-bit wide Mobile DDR RAM

• 8MB of 32-bit wide static PSRAM

• 2 x 64MB of 32-bit wide NOR flash

• Up to 320MB of static memory in an optional PISMO™static memory expansion board

• Programmable clock generators

• *Time-Of-Year* (TOY) clock with backup battery

• Connectors for VGA, color LCD display(CLCD) interface board, PCI, UART, GPIO, keyboard/mouse, Smart Card, USB, audio, MMC, SSP, and Ethernet

• Real View Logic Tile connector for one or more optional RealView Logic Tiles to develop custom IP

• Debug and test connectors for JTAG, ChipScope, and CoreSight Trace port

• DIP switches and LEDs

• 2 row by 16 character LCD display

• Power conversion and voltage control circuitry

The UART is an *Advanced Microcontroller Bus Architecture* (AMBA) compliant

System-on-Chip (SoC) peripheral that is developed, tested, and licensed by ARM.

The UART is an AMBA slave module that connects to the *Advanced Peripheral Bus*

(APB). The UART includes an *Infrared Data Association* (IrDA) *Serial InfraRed* (SIR) protocol *ENcoder/DECoder* (ENDEC).

© 2012 JCT JOURNALS. ALL RIGHTS RESERVED

The UART provides

• Compliance to the *AMBA Specification (Rev 2.0)* onwards for easy integration Into SoC implementation.

• Programmable use of UART or IrDA SIR input/output.

• Separate 32×8 transmit and 32×12 receive *First-In, First-Out* (FIFO) memory buffers to reduce CPU interrupts.

• Programmable FIFO disabling for 1-byte depth.

• Programmable baud rate generator. This enables division of the reference clock by (1×16) to (65535×16) and generates an internal ×16 clock. The divisor can be a fractional number enabling you to use any clock with a frequency >3.6864MHz as the reference clock.

• Standard asynchronous communication bits (start, stop and parity). These are added prior to transmission and removed on reception.

• Independent masking of transmit FIFO, receive FIFO, receive timeout, modem status, and error condition interrupts.

• Support for *Direct Memory Access* (DMA).

• False start bit detection.

• Line break generation and detection.

• Support of the modem control functions CTS, DCD, DSR, RTS, DTR, and RI.

• Programmable hardware flow control.

• Fully-programmable serial interface characteristics:

— data can be 5, 6, 7, or 8 bits

— even, odd, stick, or no-parity bit generation and detection

— 1 or 2 stop bit generation

— baud rate generation, dc up to UARTCLK/16

• IrDA SIR ENDEC block providing:

— programmable use of IrDA SIR or UART input/output

— support of IrDA SIR ENDEC functions for data rates up to 115200 bps half-duplex

— support of normal 3/16 and low-power (1.41-2.23µs) bit durations

— programmable division of the UARTCLK reference clock to generate the appropriate bit duration for low-power IrDA mode.

• Identification registers that uniquely identify the UART. These can be used by an operating system to automatically configure itself.

UART operation Control data is written to the UART Line Control Register, UARTLCR. This register is 30-bits wide internally, but is externally accessed through the APB interface by writes to the following registers:

UARTLCR_H Defines the:

• transmission parameters

• word length

• buffer mode

• number of transmitted stop bits

• parity mode

• break generation.

UARTIBRD Defines the integer baud rate divider

UARTFBRD Defines the fractional baud rate divider

III. Micro Kernel design and code development:

A microkernel is the near-minimum amount of software that can provide the mechanisms needed to implement an operating system. These mechanisms include low-level address space management, thread management, and inter-process communication (IPC). In terms of source code size, microkernels (as a rule of thumb) tend to be under 10,000 lines of code. MINIX3 for example has around 4,000 lines of code. Kernels larger than 20,000 lines are generally not considered microkernels. As an operating system design approach, microkernels permit typical operating system services, such as device drivers, protocol stacks, file systems code, to run in user space. If the hardware provides multiple rings or CPU modes, the microkernel is the only software executing at the most privileged level (generally referred to as supervisor or kernel mode). Microkernels are closely related to exokernels. They also have much in common with hypervisors, but the latter make no claim to minimality and are specialized to supporting virtual machines; indeed, the L4 microkernel frequently finds use in a hypervisor capacity.

The historical term nanokernel has been used to distinguish modern, high-performance microkernels from earlier implementations which still contained many system services. However, nanokernels have all but replaced their microkernel progenitors, and the term has fallen into disuse.

Microkernel includes the following concepts

1. Task related functions for task creation, priority changing, deletion etc.

© 2012 JCT JOURNALS. ALL RIGHTS RESERVED

2. Scheduler for scheduling using priority pre-emptive feature
3. Context switch to swap two tasks
4. Semaphore related functions for intertask communication

The real-time microkernel has two major responsibilities. First one is Task management and scheduling and second one is Inter task communication and synchronization as explained below.

Task management and scheduling

Task (or "process", or "thread") management is a primary job of the operating system: tasks must be created and deleted while the system is running; tasks can change their priority levels, their timing constraints, their memory needs; etcetera. Task management for an RT Kernel is a bit more critical than for a general purpose OS. If a real-time task is created, it has to get the memory it needs without delay, and that memory has to be locked in main memory in order to avoid access latencies due to swapping; changing run-time priorities influences the run-time behavior of the whole system and hence also the predictability which is so important for an RTOS. So, dynamic process management is a potential problem for an RTOS, therefore not recommended, though it is implemented in this project. In general, multiple tasks will be active at the same time, and the OS is responsible for sharing the available resources (CPU time, memory, etc.) over the tasks. The CPU is one of the important resources, and deciding how to share the CPU over the tasks is called "scheduling". The general trade-off made in scheduling algorithms is between, on the one hand, the simplicity (and hence efficiency) of the algorithm, and, on the other hand, its optimality. Real-time operating systems favour simple scheduling algorithms, because these take a small and deterministic amount of computing time, and require little memory footprint for their code.

General purpose and real-time operating systems differ considerably in their scheduling algorithms. They use the same basic principles, but apply them differently because they have to satisfy different performance criterions. general purpose OS aims at maximum average throughput, a real-time OS aims at deterministic behaviour and an embedded OS wants to keep memory footprint and power consumption low. A large variety of "real-time" scheduling algorithms exists, but some are standard in most real-time operating systems: static priority scheduling, earliest deadline first and rate monotic scheduling.. In this project a dynamic priority pre- emptive scheduling algorithm is implemented. Shown Below are the prototypes of the Task Management and scheduling related functions implemented in the project.

osrc_t thread_create (int 32*ptid, int32 pri, void (*pfunc)(void))

Creates a thread with the entry point pointed to by pfunc, and makes it ready to run. If the priority is higher than the current thread's priority, it schedules the created thread. Returns OK if successful, or ERR on error.

void thread_suspend (int32 tid)
Suspends the specified task (puts it in the wait state).
void thread_resume (int32 tid)

Resumes the specified task (puts it in the ready/running state depending on the priority).

int32 thread self (void)

Returns the id of the current task.

int32 thread_getpri (int32 tid)

Returns the priority of the specified task.

myrc_t thread_setpri (int32 tid, int32 pri)

Sets the priority of the specified task. Additionally, performs scheduling if required due to the change in the task priority.

void os_init (void)

initializes the common OS data structures. Calls thread_init for thread specific initializations. Later, this function will call other object initialization functions.

void thread_init (void)

Initializes thread specific data structures.
void thread_schedule (void)

Runs the thread scheduler.

void thread_switch_context (int32 *pcurcxt, int32 *pnxtcxt) (Assembly code)

Internal function to do the context switch. Takes in the pointers to the arrays containing contexts of the current and new task. Swaps these contexts and returns to new task.

© 2012 JCT JOURNALS. ALL RIGHTS RESERVED

Communication and synchronization

Second major responsibility of an OS is commonly known under the name of Inter- Process Communication (IPC). ("Process" is, in this context, just another name for "task".) The general name IPC collects a set of programming primitives that the operating system makes available to tasks that need to exchange information with other tasks, or synchronize their actions. Again, an RTOS has to make sure that this communication and synchronization take place in a deterministic way. Besides communication and synchronization with other tasks that run on the same computer, some tasks also need to talk to other computers, or to peripheral hardware (such as analog input or output cards). This involves some peripheral hardware, such as a serial line or a network, and special purpose device drivers. In this project Inter task communication is carried out using semaphores. Semaphore is an object used for intertask communication in operating system. It is for informing the kernel about the status of the task for its waiting for a resource or releasing a resource. Every semaphore must be created before it is used. Below are the prototypes of the Inter Task Communication Synchronization related functions implemented in the project.

osrc_t ossem_create (int32 count, int32 *psid)

Creates a semaphore with the specified initial count and returns the id in *psid.

Returns OK if successful, or ERR on error.

void sem_post (int32 semid)

Posts the specified semaphore. Reschedules the tasks if posting the semaphore could cause a task switch.

void sem_wait (int32 sid)

Waits for a semaphore. If the semaphore is unavailable, puts the current task in waiting state and schedules another task.

Context switching is done when a processor switches from one task to another task. Context is mainly the Register snapshot of the processor when a task is under execution. This function is called after the scheduler if task switching is required. In this project is this function is implemented using ARM assembly language programming. This routine saves all the registers of the first task on the stack of the task or context area and restores all the registers from the stack or context area of the next task. At the end of this task the control switches to second task.

void thread_switch_context (int32 *pcurcxt, int32 *pnxtcxt) (Assembly code)

```
; //save all the registers of current task
    STR R2,[R0,#0X0]
    STR R3,[R0,#0X4]
    STR R4,[R0,#0X8]
    STR R5,[R0,#0XC]
    STR R6,[R0,#0X10]
    STR R7,[R0,#0X14]
    STR R8,[R0,#0X18]
    STR R9,[R0,#0X1C]
    STR R10,[R0,#0X20]
    STR R11,[R0,#0X24]
    STR R12,[R0,#0X28]
    STR R12,[R0,#0X2C]
    STR R14,[R13]
    SUB R13,R13,#04
    STR R13,[R0,#0X4C]

Restore all the second task register
  values from the TCB to Processor
    LDR R2,[R1,#0X0]
    LDR R3,[R1,#0X4]
    LDR R4,[R1,#0X8]
    LDR R5,[R1,#0XC]
    LDR R6,[R1,#0X10]
    LDR R7,[R1,#0X14]
    LDR R8,[R1,#0X18]
    LDR R9,[R1,#0X1C]
    LDR R10,[R1,#0X20]
    LDR R11,[R1,#0X24]
    LDR R12,[R1,#0X2C]
    LDR R12,[R1,#0X28]
    LDR R0,[R1,#0X4C]
    ADD R0,R0,#04
    MOV R13,R0
    LDR R14,[R13]
    BX R14
    END
```

© 2012 JCT JOURNALS. ALL RIGHTS RESERVED

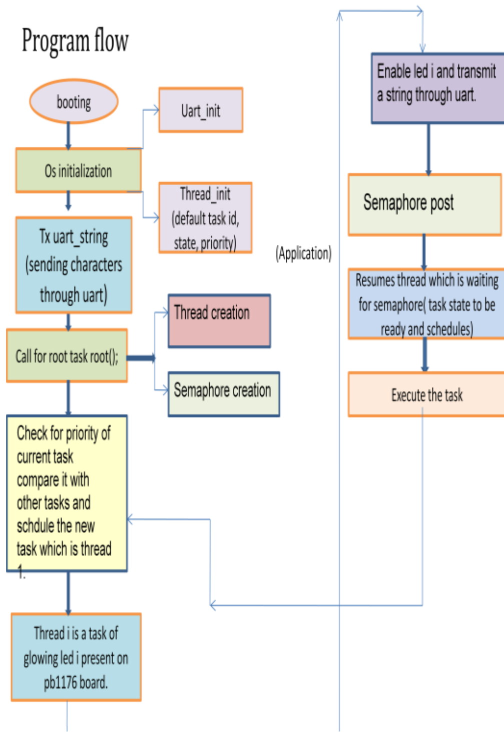

Fig.5 microkernel design flow

Conclusion: A real time microkernel for ARM1176JZF-S is successfully developed and demonstrated in this project. The tools used are Real View Development Suite(RVDS) for Software development and debugging. Microkernel developed in this project can be used for multi-tasking. The application areas are avionics on board computers, industry process automation, and auto mobile electronics. Microkernel developed in this project has a footprint of 28kb. This can be further reduced and interrupts can also be used efficiently by using advanced features of ARM processors.

Applications: Multi tasking embedded systems applications like onboard computers for Avionics Embedded systems like printers, digital storage oscilloscope. Which occupies less memory.

References:

*[1] www.csie.ntu.edu.tw/~ktw/rts/**uCOSII**- prn.pdf*

[2] www.ucos-ii.com,

[3]http://infocenter.arm.com/help/topic/com.arm.doc.ddi0183g/ DDI0183G_uart_pl011_r1p5_trm.pdf

[4]http://infocenter.arm.com/help/topic/com.arm.doc.ddi0301h/DDI 0301H_arm1176jzfs_r0p7_trm.pdf

[5]http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi 0190b/I1002697.html

[6]http://infocenter.arm.com/help/topic/com.arm. doc.ddi0190b/DDI0190.pdf

[7]http://infocenter.arm.com/help/topic/com.arm.doc.DDI0 190B_gpio_PL061_trm

[8]http://infocenter.arm.com/help/topic/com.arm.doc.DUI0 425F_realview_platform_baseboard_for_arm1176jzf_ug

Real-time systems by Jane W.S.Liu

© 2012 JCT JOURNALS. ALL RIGHTS RESERVED