Unit IV – Case Study of RTOS

- ❖RT Linux
- ❖MicroC/OS-II
- ❖Embedded Linux
- ❖Tiny OS
- ❖Basic Concepts of Android OS.

Introduction

Kernel of an RTOS

- Used for real-time programming features to meet hard and soft real time constraints.
- Provides for preemption points at kernel, user controlled dynamic priority changes, fixed memory blocks, asynchronous IOs, user processes in kernel space and other functions for a system.

Complex multitasking embedded system design requirements are

_ Integrated Development Environment,
_ Multiple task functions in Embedded C or Embedded C++,
_ Real time clock─ hardware and software timers,
_ Scheduler,
_ Device drivers and device manager,
_ Functions for inter inter-process communications using the signals, event flag group, semaphore-handling functions, functions for the queues, mailboxes, pipe, and sockets.
_ Additional functions for example, TCP/IP or USB port, other networking functions.
_ Error handling functions and Exception handling functions, and
_ Testing and system debugging software for testing RTOS as well as developed embedded application

RTOS features in general have the following features:

_ Basic kernel functions and scheduling: Preemptive or Preemptive plus time slicing
_ Support to Limited Number of tasks and threads
_ Task priorities and Inter Service Threads priorities definitions
_ Priority Inheritance feature or option of priority ceiling feature
_ Task synchronization and IPC functions
_ Support to task and threads running in kernel space
_ IDE consisting of editor, platform builder, GUI and graphics software, compiler, debugging and host target support tools
_ Device Imaging tool and device drivers
_ Clock, time and timer functions,
_ Support to POSIX,
_ Asynchronous IOs,
_ Fixed memory blocks allocation and deallocation system,
_ Support to different file systems and flash memory systems
_ TCP/IP protocols, network and buses protocols,
_ Development environment with Java
_ Componentization (reusable modules for different functions), which leads to small footprint (small of size of RTOS codes placed in ROM image)
_ Support to number of processor architectures, such as INTEL, ARM, Philips …

Types of RTOSes

Following are the types of RTOSes

In-House Developed RTOSes: In-house RTOS has the codes written for the specific need based, and application or product and customizes the in-house design needs. Generally either a small level application developer uses the in-house RTOS or a big research and development company uses the codes built by in-house group of engineers and system integrators.

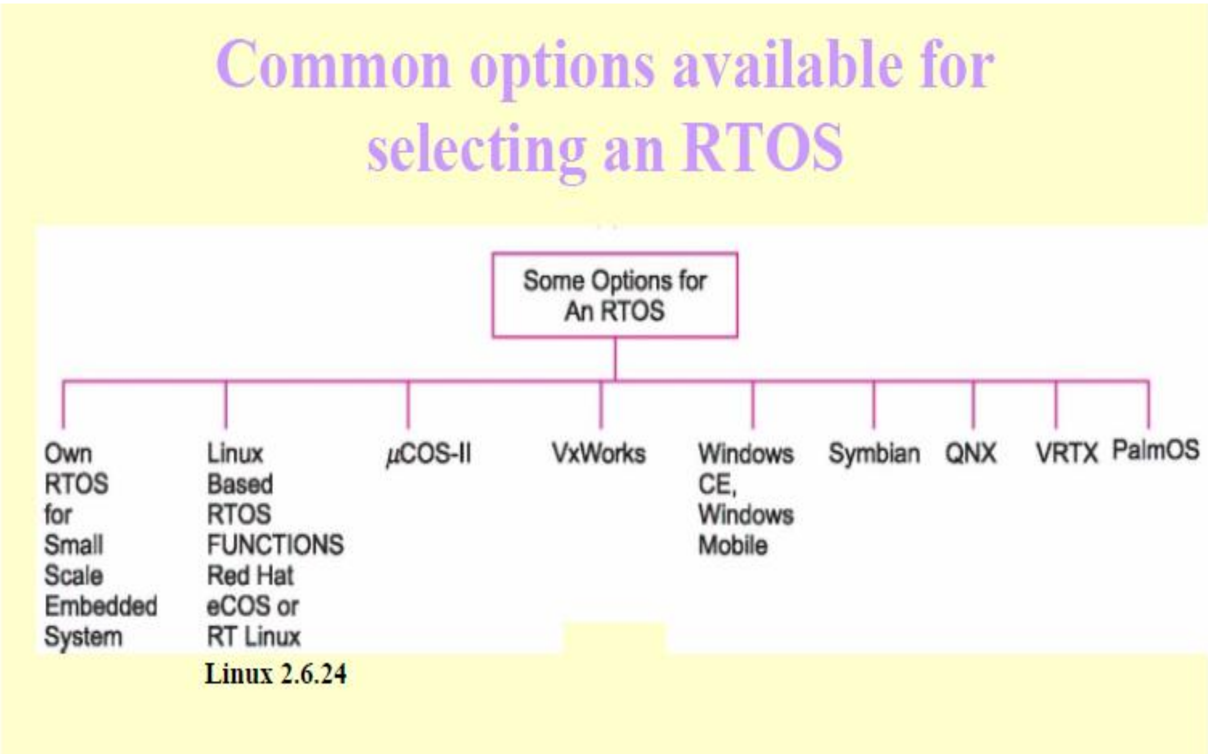
Broad based Commercial RTOSes:
A broad-based commercial RTOS package offers the following advantages.

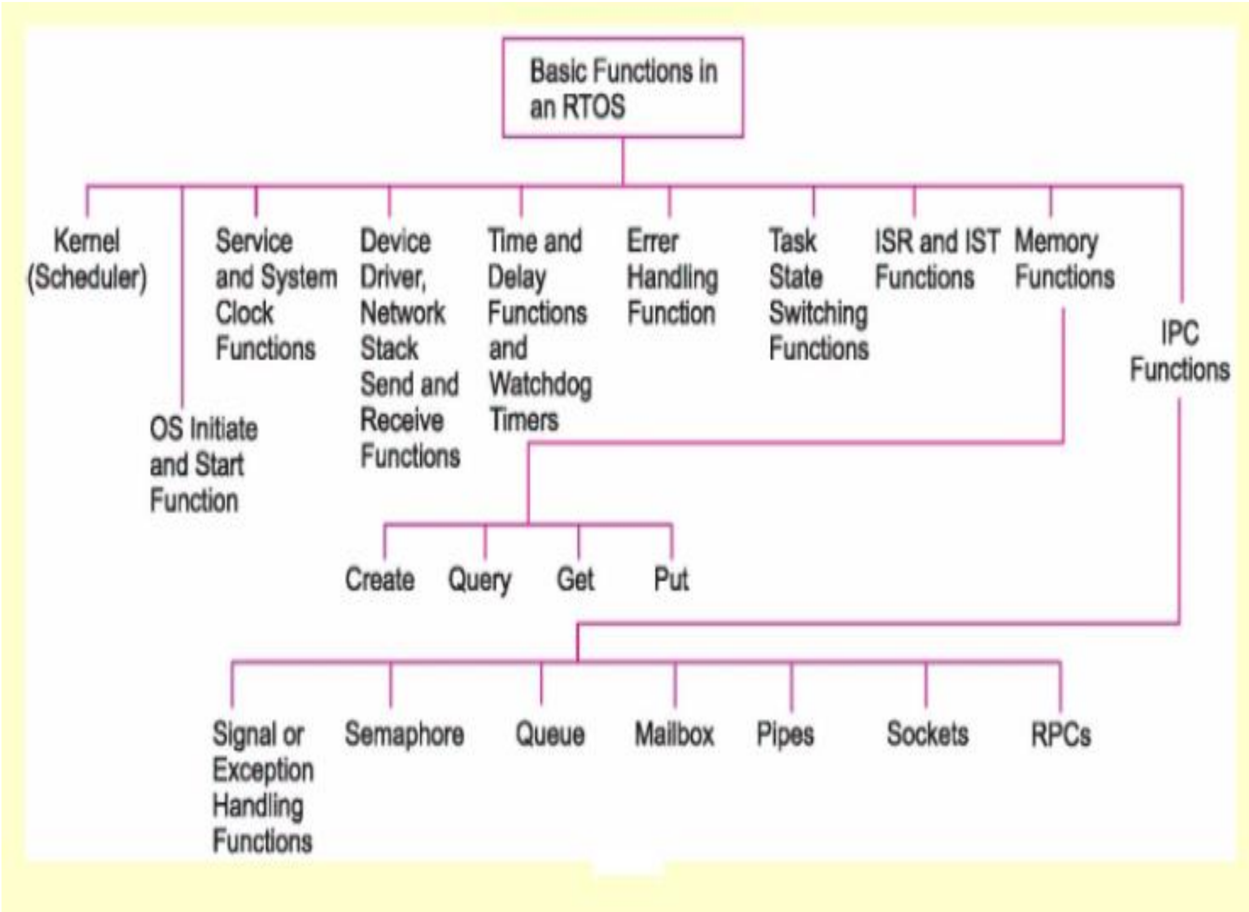
- Provides an advantage of availability of off the self thoroughly tested and debugged RTOS functions.
- Provides several development tools.
- Testing and debugging tools
- Support to many processor architectures like ARM as well as x86, MIPS and SuperH.
- Support to GUIs
- Support to many devices, graphics, network connectivity protocols and file systems
- Support to device software optimization (DSO)
- Provides error and exceptional handling functions can be ported directly as these are already well tested by thousands of users
- Simplifies the coding process greatly for a developer
- Helps in building a product fast
- Aids in building robust and bug-free software by thorough testing and simulation before locating the codes into the hardware
- Saves large amount of RTOS, tools and inhouse documentation development time.
- Saving of time results in little time to market an innovative and new product.
- Saves the maintenance costs.
- Saves the costs of keeping in-hose engineers.

General Purposes OSes with RTOS: Embedded Linux or Windows XP is general purpose OS. They are not componentized. Footprint (the code that goes as ROM image) is not reducible. The tasks are not assignable priorities. However, they offer powerful GUIs, rich multimedia interfaces and have low cost.

The general purpose OSes can be used in combination with the RTOS functions. For example, Linux 2.6.24 and RTLinux are real time kernels over the Linux kernel. Other example is 'Windows XP Embedded' for x86 architecture

Special Focus RTOSes: Used with specific processors like ARM or 8051 or DSP, for example**,** OSEK for automotives or Symbian OS for Mobile phones.





μC/OS-II Features

Jean J. Labrosse designed it in 1992.μC/OS-II name derives from Micro-Controller Operating System. Also known as MUCOS, UCOS, ..

Basic Feature

_ Scalable OS – only needed OS functions become part of the application codes
_ Configuration file includes the user definitions for needed functions
_ Multitasking preemptive scheduler
_ Elegant code
_ Is said to offer best high quality/performance ratio
_ It has a pre-certifiable software component for safety critical systems, including avionics system clock A DO-178B and EUROCAE ED-12B, medical FDA 510(k), and IEC 61058 standard for transportation and nuclear systems, respectively
_ Source code has been certified by Department of Defense, USA for use in Avionics and in medical applications.
-**Applications:** Automotive, avionics, consumer electronics, medical devices, military, aerospace, networking, and systems-on-a-chip.

MUCOS real time kernel additional support

_ μC/BuildingBlocks [an embedded system building blocks (software components) for hardware peripherals, for example clock (μC/Clk) and LCD (μC/LCD)]
_ μC/FL (an embedded flesh memory loader)

_ μC/FS (an embedded memory file system)
_ μC/GUI (an embedded GUI platform),
_ μC/Probe (a real time monitoring tool),
_ μC/TCP-IP (an embedded TCP/IP stack),
_ μC/CAN (an embedded Controller Area Network bus)
_ μC/MOD (an embedded Modbus) and
_ μC/USB device and μC/USB host (an embedded USB devices framework).

Source Files:

1. *Processer-dependent source files:* Two header files at master are (i) os_cpu.h is processor definition header file (ii) the kernel building configuration file is os_cfg.h
2. *Processor-independent source files:* Two files, MUCOS header and C files are ucos_ii.h and ucos_ii.c.

MUCOS Naming Basics

_ OS or OS_ prefix denotes that the function or variable is a MUCOS operating system function or variable
_ For examples, OSTaskCreate ( ) ─ aMUCOS function that creates a task,
_ OS_NO_ERR─ a MUCOS macro that returns true in case no error is reported from a function
_ OS_MAX_TASKS─ user definable constant for specifying maximum number of tasks in user application

MUCOS Basic Functions

- System Level – OS initiate, start, system timer set, ISR enter and exit
- Task Service Functions – create, run, suspend, resume, ..
- Task delay
- Memory allocation, partitioning ..
- IPCs – Semaphore, Queue and Mailbox
- Same Semaphore function usable as event flag, resource key and counting semaphore
- Mailbox one message pointer per mailbox
- Queue permit array of message-pointers

Summary of Commonly Used μC/OS-II Functions and Data Structures

Data Structures

OS_EVENT:

This structure is used in the following functions: OSQCreate(), OSQPend(), OSQPost(), OSSemCreate(), OSSemPend(),OSSemPost(). All of these functions make use of queues to either protect resources or pass messages between tasks. A pointer to the created structure is returned by the two create functions. Once these queues have been created treat the returned pointers as the head of the queue and always pass the pointer to this structure to the Pend and Post functions as the pevent argument. Traversal is not necessary.

See below for further information regarding the individual functions. Data structure located in file

```
src/uC/ucos_ii.h
typedef struct {
    INT8U    OSEventType;                    /* Type of event control block (see
OS_EVENT_TYPE_???)      */
    INT8U    OSEventGrp;                                     /* Group corresponding to
tasks waiting for event to occur */
```

```
    INT16U   OSEventCnt;                                  /* Semaphore Count (not
used if other EVENT type)          */
    void    *OSEventPtr;                                  /* Pointer to message or
queue structure                  */
    INT8U    OSEventTbl[OS_EVENT_TBL_SIZE];               /* List of tasks waiting
for event to occur               */
    char     OSEventName[OS_EVENT_NAME_SIZE];             /* Compile time directive
currently has OS_EVENT_NAME_SIZE = 32 */
} OS_EVENT;


#define  OS_EVENT_TYPE_UNUSED      0      /* All possible values for OSEventType in
OS_EVENT struct */
#define  OS_EVENT_TYPE_MBOX        1
#define  OS_EVENT_TYPE_Q           2
#define  OS_EVENT_TYPE_SEM         3
#define  OS_EVENT_TYPE_MUTEX       4
#define  OS_EVENT_TYPE_FLAG        5
```

OS_STK:

Each task has stack entries of this type. The data structure is located in file src/uC/os_cpu.h

```
typedef unsigned short OS_STK;                  /* Each stack entry is 16-bit wide
*/
```

OS_STK_DATA:

A variable of type OS_STK_DATA is filled in when calling OSTaskStkChk() to get the statistics about the stack of each task. Further information regarding the OSTaskChkTask function is located below. The OS_STK_DATA data structure is located in file src/uC/ucos_ii.h

```
typedef struct {
    INT32U  OSFree;                    /* Number of free bytes on the stack for a
specific task */
    INT32U  OSUsed;                    /* Number of bytes used on the stack for a
specific task*/
} OS_STK_DATA;
```

OS_TCB:

A variable of type OS_TCB is filled in when calling OSTaskQuery() to get information about a task. Further information regarding OSTaskQuery is located below. The OS_TCB data structure is located in file src/uC/ucos_ii.h

```
typedef struct os_tcb{
        OS_STK          *OSTCBStkPtr;          //Stack Pointer
        void            *OSTCBExtPtr;          //TCB extension pointer
        OS_STK          *OSTCBStkBottom;       //Ptr to bottom of stack
        INT32U          OSTCBStkSize;          //Size of task stack (#elements)
        INT16U          OSTCPOpt;              //Task options
        struct os_tcb   *OSTCBNext;            //Pointer to next TCB
        struct os_tcb   *OSTCBPrev;            //Pointer to previous TCB
        OS_EVENT        *OSTCBEventPtr;        //Pointer to ECB
        void            *OSTCBMsg;             //Message received
        OS_FLAG_NODE    *OSTCBFlagNode;        //Pointer to event flag node
        OS_FLAGS        OSTCBFlagsRdy          //Event flags that made task ready
        INT16U          OSTCBDly;              //Nbr ticks to delay task or, timeout
        INT8U           OSTCBStat;             //Task Status
        INT8U           OSTCBPrio;             //Task Priority (0 = highest)
```

```
        INT8U           OSTCBX;
        INT8U           OSTCBY;
        INT8U           OSTCBBitX;
        INT8U           OSTCBBitY;
        BOOLEAN         OSTCBDelReq;           //Flag to tell task to delete itself
}OS_TCB;
```

Global Variables

Most of the global variables for uC/OS-II are located in file src/uC/ucos_ii.h. Do not modify these globals. Reading from them is necessary for some exercises in the labs.

```
OS_EXT  INT8S               OSCPUUsage;            /* Percentage of CPU used
*/
```

Vector Table Entries

Initialize vector table entry #0 with the pointer of the context switching function. This function will be called on every context switch. Initialize the table before doing anything else. Without the context switching vector (TRAP # 0) pointing to the correct function uC/OS-II will not function correctly. The context switching function OSCtxSw() is located in file src/uC/os_cpu_a.s This is given to you. DO NOT modify the code that moves the Trap location into the vector table.

```
*((int *)0x80) = (int)OSCtxSw; /* set up vector to Context Switch (TRAP #0) */
```

Initialization Functions

OSInit()

Function Prototype	Arguments	Returns	Notes
void OSInit(void); Location: src/uC/os_core.c	none	nothing	Call this function first inside your begin() function. Initializes uC/OS-II and must be called before calling OSStart()

Example:

```
begin (void)                          /* SYSTEM ENTRY POINT */
{
    *((int *)0x80) = (int)OSCtxSw;    /* set up vector to Context Switch (TRAP #0) */
    OSInit();
    .
    .
    OSStart();                          /* away we go ! */
}
```

OSStart()

Function Prototype	Arguments	Returns	Notes
void OSStart(void); Location: src/uC/os_core.c	none	nothing	OSStart begins the multitasking. Call this as a part of your initialization function but make sure that you have called OSInit() first.

Example:

```
begin (void)                              /* SYSTEM ENTRY POINT */
{
    *((int *)0x80) = (int)OSCtxSw;      /* set up vector to Context Switch (TRAP #0) */
    OSInit();
    .
    .
    OSStart();                            /* away we go ! */
}
```

OSStatInit()

Function Prototype	Arguments	Returns	Notes
void OSStatInit(void); Location: src/uC/os_core.c	none	nothing	If CPU stats are required, this function must be called. It must be called from the first and only task created. This first and only task may,in turn, create other tasks once OSStatInit has been called.

Example:

```
begin (void)                              /* SYSTEM ENTRY POINT */
{
    *((int *)0x80) = (int)OSCtxSw;      /* set up vector to Context Switch (TRAP #0) */
    OSInit();
    TaskCreate("StartTask",StartTask,StartTask_ID);
    OSStart();                            /* away we go ! */
}



void StartTask(void *data)
{
    ScopeInit();                /* Initialize oscilloscope triggering routine */
    TickInit();         /* Start OS ticker,  see os/os_cpu_a.s */
    OSStatInit();               /* Initialize statistics task */
    OSTaskCreateExt(...);       /* all clear to call tasks now */
    OSTaskCreateExt(...);
    .
    .
    .
    OSTaskDel(OS_PRIO_SELF);   /* This task only runs once */
}
```

Task Functions

OSTaskCreateExt()

Function Prototype	Arguments	Returns	Notes
INT8U OSTaskCreateExt (void(*task)(void*pd), void *pdata, OS_STK *ptos, INT8U prio, INT16U id, OS_STK *pbos, INT32U stk_size, void *pext, INT16U opt) ; Location: src/uC/os_task.c	**task** Pointer to task's code that must be declared as void Task (void *) **pdata** Pointer to data that is passed to task when it is created **ptos** Pointer to the top of the task's stack. For stacks that grow down in memory ptos needs to point to the highest valid memory location on the stack. **prio** Unique priority to assign to this task. The lower the number the higher the priority. **id** Task's ID number which is not currently used. Set this to the priority of the task. **pbos** Pointer to the bottom of the task's stack. For stacks that grow downward in memory pbos must point to the lowest valid stack location. **stk_size** Number of 16 bit entries available on the stack. See typedef of OS_STK in src/uC/os_cpu.h and above. **pext** Pointer to a user supplied memory location used as TCP extension. User defined location or data structure. **opt** Options for the task created. Lower 8 bits are reserved for	One of the following error codes: **OS_NO_ERR** Function was successful in creating the task **OS_PRIO_EXIST** A task already exists with that priority. In uC each task must have a unique priority **OS_PRIO_INVALID** prio is higher than OS_LOWEST_PRIO, currently set to 63 **OS_NO_MORE_TCB** uC has run out of OS_TCBs to assign	Stack must be declared as type OS_STK At some point during the execution of the task one of the services offered by uC/OS-II must be called to wait for time to expire, suspend the task or wait for an event like a mailbox or semaphore. Otherwise the task may never cede the processeor and other tasks with lower priorities may never get a time slice. uC/OS-II is not a round robin OS. Consequently, all task must eventually cede for all task to get servicing. Don't assign user tasks priorities 0, 1, 2, 3, OS_LOWEST_PRIO-3, OS_LOWEST_PRIO-2, OS_LOWEST_PRIO-1, or OS_LOWEST_PRIO. These are reserved by uC/OS-II. The other 56 application tasks are therefore available.

	uC/OS-II but applications may use the upper 8 bits for application specific options. Possible uC/OS-II predefined options are: **OS_TASK_OPT_STK_CHK** Specifies whether stack checking is allowed for the task **OS_TASK_OPT_STK_CLR** Specifies whether the stack need to be cleared **OS_TASK_OPT_SAVE_FP** N/A we have no floating point registers on our CPU32		

Example:

```
#define TASK_OPT        OS_TASK_OPT_STK_CHK + OS_TASK_OPT_STK_CLR

void TaskCreate(char *TaskDesc, void *TaskFunc, INT8U TaskID)
{
        INT8U err;
        err=OSTaskCreateExt(TaskFunc,
                 (void *)0,
                 &TaskStack[TaskID][TASK_STACK_SIZE-1],
                 TaskID+5,
                 TaskID,
                 &TaskStack[TaskID][0],
                 TASK_STACK_SIZE,
                 &TaskData[TaskID],
                 TASK_OPT);
        if (!err)
                strcpy(TaskData[TaskID].TaskName, TaskDesc);
        else
                disp_err(err);
}
```

OSTaskDel()

Function Prototype	Arguments	Returns	Notes
INT8U OSTaskDel (INT8U prio) ; Location: src/uC/os_task.c	**prio** priority number of the task to be deleted.	**OS_NO_ERR** Call was successful **OS_TASK_DEL_IDLE** This value is returned if you attempt to delete the idle task, this is not permitted **OS_TASK_DEL_ERR** Task to be deleted does not exist. **OS_PRIO_INVALID** prio is higher than OS_LOWEST_PRIO.	Specify the priority of the task to be deleted or pass in OS_PRIO_SELF if priority of task is unknown. This task's code is not actually removed but the task is placed in the dormant state and can be recreated and made active by calling OSTaskCreate or OSTaskCreateExt. Be careful when deleting a task that owns associated resources. If a task owns resources like mailboxes, semaphores etc. call

		OS_TASK_DEL_ISR This value is returned if you attempt to delete a task from an ISR.	OSTaskDelReq() instead to deal with those issues safely.

Example:

```
        void StartTask(void *data)
        {
                ScopeInit();            /* Initialize oscilloscope triggering routine */
                TickInit();             /* Start OS ticker,  see src/uC/os_cpu_a.s */
                OSStatInit();           /* Initialize statistics task */
                LEDQueue = OSQCreate (LEDQueueTbl, 10);
                .
                .
                .
                OSTaskDel(OS_PRIO_SELF);        /* We don't want the Start ask to run
more than once */
        }
```

OSTaskStkChk()

Function Prototype	Arguments	Returns	Notes
INT8U OSTaskStkChk (INT8U prio, OS_STK_DATA *pdata); Location: src/uC/os_task.c	**prio** priority number of the task about which you want stack information. If the value OS_PRIO_SELF is passed then the stack of the calling task is checked. **pdata** pointer to a variable of type OS_STK_DATA that is used by the function	**OS_NO_ERR** Call was successful **OS_PRIO_INVALID** prio is higher than OS_LOWEST_PRIO or not equal to OS_PRIO_SELF **OS_TASK_NOT_EXIST** Specified task does not exist **OS_TASK_OPT_ERR** this value is returned if the task was created by OSTaskCreate or if OS_TASK_OPT_STK_CHK was not specified when the OSTaskCreateExt call was used.	Execution time for this task depends on the size of the stack for each task and is, therefore, nondeterministic. To calculate the total stack size used add .OSFree and .OSUsed together. Currently, all of the stacks in the labs are the same size (= 512 bytes). Don't call this function inside an ISR due to its nondeterministic nature and possible length of time for completion.

Example:

```
        OS_STK_DATA   StackData;
        INT8U         err;
        err = OSTaskStkChk(OS_PRIO_SELF,&StackData);
```

OSTaskQuery()

Function Prototype	Arguments	Returns	Notes
INT8U OSTaskQuery (INT8U prio, OS_TCB *pdata); Location: src/uC/os_q.c	**prio** priority number of the task about which you want task information. If the value OS_PRIO_SELF is passed then the stack of the calling task is checked. **pdata** pointer to a structure of type OS_TCB, which contains a copy of the task's control block	**OS_NO_ERR** Call was successful **OS_PRIO_ERR** Attempt to obtain information from an invalid task **OS_PRIO_INVALID** prio is higher than OS_LOWEST_PRIO or not equal to OS_PRIO_SELF	You must allocate an OS_TCB structure before calling this function, and passing the pointer in as a parameter. Your copy obtains a snapshot of the desired task's control block. DO NOT modify any of the fields in the OS_TCB control blocks. Reading them is sufficient for the labs in CMPE401.

Example:

```
OS_TCB    task_data;
INT8U     err;
err = OSTaskQuery(OS_PRIO_SELF,&task_data);
```

Queue Functions

OSQCreate()

Function Prototype	Arguments	Returns	Notes
OS_EVENT * OSQCreate (void **start, INT16U size); Location: src/uC/os_q.c	**start** base address of storage area **size** number of elements in the storage area	A pointer to the event control block allocated to the queue is returned if the call succeeds. If it fails a NULL pointer is returned.	Always create queues before using them. Generally, queues are created for intertask communication. One task posts a message and another task retrieves it. Otherwise race conditions could result and cause many potential problems if tasks attempt to simultaneously access common resources.

Example:

```
        begin (void)
        {
                *((int *)0x80) = (int)OSCtxSw; /* set up vector to Context Switch (TRAP
#0) */
                OSInit();
                TaskCreate("StartTask",StartTask,StartTask_ID);
                OSStart();
        }

        void StartTask(void *data)
        {
```

```
            ScopeInit();
            TickInit();
            .
            .
            TxQueueA = OSQCreate (TxQueueTblA, FIFO_SIZE);
            TxQueueB = OSQCreate (TxQueueTblB, FIFO_SIZE);
            .
            .
            OSTaskDel(OS_PRIO_SELF);      /* This task only runs once */
        }
```

OSQPend()

Function Prototype	Arguments	Returns	Notes
void * OSQPend (OS_EVENT *pevent, INT16U timeout, INT8U * err) ; Location: src/uC/os_q.c	**pevent** Pointer to queue from which the message is to be recieved. This is the same pointer that was returned when the queue was created using OSQCreate() **timeout** Pass in 0 if you want to wait forever for a message. Pass a value in ticks (0 - 65535) to give up on receiving the message after the period has lapsed. The function will return and the task will resume once the number of ticks has expired. **err** **OS_NO_ERR** Message was received **OS_TIMEOUT** Message was not received withing the specified timeout. **OS_ERR_EVENT_TYPE** pevent is not pointing to a message queue **OS_ERR_PEVENT_NULL** pevent is a NULL pointer **OS_ERR_PEND_ISR** This function was called from an ISR and uC/OS-II must suspend the task. To avoid this don't call this function from an ISR.	If successful OSQPend returns a message sent by a task and *err contains OS_NO_ERR. If unsuccessful a NULL pointer is returned and *err contains one of the error codes as specified in the arguments field.	Always create queues before using them and don't call this function from inside an ISR Messages are placed in the queue by one task and retreived by another. Call this function to retrieve possible messages. If multiple tasks are waiting for a message the highest priority task is resumed.

Example:

```
        LedQueueData = *(LED_DATA *)OSQPend(LEDQueue, 0, &err);  /* this will block
until a queue entry is available */
        if (err)
                disp_err(err);
```

OSQPost()

Function Prototype	Arguments	Returns	Notes
INT8U OSQPost (OS_EVENT *pevent, void *msg) ; Location: src/uC/os_q.c	**pev**ent Pointer to the queue into which the message is deposited. Use the pointer that was returned when the queue was created using OSQCreate() **msg** Pointer- sized variable that is user defined. Don't post a NULL pointer.	**OS_NO_ERR** Message was deposited in the queue. **OS_Q_FULL** No room in the queue. **OS_ERR_EVENT_TYPE** pevent is not pointing to a message queue. **OS_ERR_PEVENT_NULL** pevent is a NULL pointer. **OS_ERR_POST_NULL_PTR** msg is a NULL pointer.	Always create Queues before using them and never pass in NULL pointers as arguments. Use this function to send a message to another task via a previously created queue. If multiple tasks are waiting for a message the highest priority task is resumed.

Example:

```
void LedBufferPost(char num, char state)
{
        INT8U err;
        if (LedBufferIndex >=10)
               LedBufferIndex=0;
        LedBuffer[LedBufferIndex].LedNum = num;
        LedBuffer[LedBufferIndex].State = state;
        err = OSQPost(LEDQueue, (void *)&LedBuffer[LedBufferIndex++]);
}
```

Semaphore Functions

OSSemCreate()

Function Prototype	Arguments	Returns	Notes
OS_EVENT* OSSemCreate (INT16U value); Location: src/uC/os_sem.c	**value** Initial value of the semaphore that can be from 0 - 65536. Pass in 0 to indicate that the resource is not available.	OSSemCreate returns the created event control block if the function succeeds. If it fails OSSemCreate returns a NULL pointer	Always create semaphores before using them. When creating semaphores the value indicates how many tasks can obtain the semaphore concurrently. Pass in 1 if the semaphore is protecting a single resource (ie. a memory location). If there is more than one resource available (ie. a block of structures that can be assigned) then pass in that number.

Example:

```
OS_EVENT *TxSemA, *TxSemB, *TxBufferLock;

void StartTask(void *data)
{
        ScopeInit();
        TickInit();
        TxSemA = OSSemCreate(0);
        TxSemB = OSSemCreate(0);
        TxBufferLock =  OSSemCreate(1);
        .
        .
        DuartInit();
        OSTaskDel(OS_PRIO_SELF);       /* This task only runs once */
}
```

OSSemPend()

Function Prototype	Arguments	Returns	Notes
void OSSemPend (OS_EVENT *pevent, INT16U timeout, INT8U *err) Location: src/uC/os_sem.c	**pevent** Pointer to the semaphore. This was returned when the semaphore was created with OSSemCreate() **timeout** Pass in 0 if you want to wait forever for a message. Pass a value in ticks (0 - 65535) to give up on receiving the message after the period has lapsed. The function will return and the task will resume once the number of ticks has expired. **err** **OS_NO_ERR** Semaphore is available **OS_TIMEOUT** Semaphore was not obtained withing the specified timeout. **OS_ERR_EVENT_TYPE** pevent is not pointing to a semaphore **OS_ERR_PEVENT_NULL** pevent is a NULL pointer **OS_ERR_PEND_ISR** This function was called from an ISR and uC/OS-II must suspend the task. To avoid this don't call this function from an ISR.	nothing	Always create semaphores before using them. Call this function when a task needs to use a shared resource safely, is waiting for an event,or needs to synchronize its activities with an ISR or a task. Note that this call blocks while waiting for the semaphore to be free. Don't use this function from an ISR use the non-blocking OSSemAccept instead()

Example:

```
void TxTask1 (void *data)      /* send the uppercase alphabet to TxA */
{
        char byte;
        char err;
        while(1)
        {
                byte=0x41;
                OSSemPend(TxBufferLock, 0, &err);
                while (byte <= 0x5A)
                        TxBufferPost(0, byte++);
                TxBufferPost(0, CR);
                TxBufferPost(0, LF);
                OSSemPost(TxBufferLock);
                OSTimeDly(64);
        }
}
```

OSSemPost()

Function Prototype	Arguments	Returns	Notes
INT8U OSSemPost (OS_EVENT *pevent); Location: src/uC/os_sem.c	**pevent** Pointer to the semaphore. This was returned when the semaphore was created with OSSemCreate()	**OS_NO_ERR** Semaphore has been signalled or released. **OS_SEM_OVF** Semaphore counts has overflowed (> 65535). **OS_ERR_EVENT_TYPE** pevent is not pointing to a semaphore **OS_ERR_PEVENT_NULL** pevent is a NULL pointer	Always create semaphores before using them. Calling this function will signal, or release the semaphore so that another task can obtain it. If the value is 0 or greater the value is incremented and the function returns to the caller.

Example:

```
void TxTask1 (void *data)      /* send the uppercase alphabet to TxA */
{
        char byte;
        char err;
        while(1)
        {
                byte=0x41;
                OSSemPend(TxBufferLock, 0, &err);
                while (byte <= 0x5A)
                        TxBufferPost(0, byte++);
                TxBufferPost(0, CR);
                TxBufferPost(0, LF);
                OSSemPost(TxBufferLock);
                OSTimeDly(64);
        }
}
```

Time Functions

OSTimeDly()

Function Prototype	Arguments	Returns	Notes
void OSTimeDly (INT16U ticks) ; Location: src/uC/os_time.c	**ticks** Number of clock ticks to delay the current task. (0 - 65,535) (see OS_TICKS_PER_SEC in configuration file src/uC/os_cfg.h)	nothing	Use this function to reschedule a task for a later time. Calling the function with all parameters set to zero simply returns to the caller immediately.

Example:

```
while(1)
{
        /* One second = 64 ticks*/
        OSTimeDly(64);
}
```

OSTimeDlyHMSM()

Function Prototype	Arguments	Returns	Notes
void OSTimeDlyHMSM (INT8U hours, INT8U minutes, INT8U seconds, INT8U milli) ; Location: src/uC/os_time.c	**hours** Number of hours of delay (0 - 255) **minutes** Number of minutes of delay (0 - 59) **seconds** Number of seconds of delay (0 - 59) **milli** Number of milliseconds of delay (0 - 999). milliseconds are rounded to nearest number of ticks so be careful. Be aware of tick rate when only using milliseconds. If the number of milliseconds passed in is smaller that one tick the delay may not occur at all.	**OS_NO_ERR** Call was successful **OS_TIME_INVALID_MINUTES** Minutes argument is greater than 59 **OS_TIME_INVALID_SECONDS** Seconds argument is greater than 59 **OS_TIME_INVALID_MILLI** Milliseconds argument is greater than 999 **OS_TIME_ZERO_DLY** All arguments are given as zero	Use this function to reschedule the task for a later time. Calling the function with all parameters set to zero simply returns to the caller immediately.

Example:

```
while(1)
{
        LEDValue = *(char *)OSQPend(LEDQueue, 0, &err);
        if (err)
                disp_err(err);
        led_current |= LEDValue;
        *LED = led_current;
        OSTimeDlyHMSM(0,0,0,50);
        led_current &= ~LEDValue;
        *LED = led_current;
}
```

Embedded Linux

What is the difference between RTLinux and Linux

Desktop linux - a linux distribution installed on your desktop machine and intended for day-to day use, like running an application at work, browsing, reading e-mail, writing documents, watching movies etc

Embedded linux - a linux distribution for embedded devices that control for example home appliances such as DVDs, microwave ovens, washing machines - this linux has a very small footprint, since it has to run inside a microcontroller with low resources (such as ARM, typically speeds of MHz or tens of MHz, memory 4-8-16 MB) an it is reduced to the minimum components needed to do the job.

RTLinux - Real Time linux - as above, but with real-time constraints - the designer guarantees maximum response times for any operation. RT linux is used to control industrial machinery, in automotive applications etc., where you must guarantee time responses and stability in order not to break expensive machinery and potentially endanger human lives.

Embedded Linux system:

- An embedded system running the Linux kernel
- Userspace tools & configuration likely to be very different from desktop (uClibc instead of glibc, BusyBox instead of coreutils, etc.)
- Embedded Linux development distribution:
- Includes all the tools and packages required for developing software for embedded Linux systems.
- Embedded Linux target distribution: Includes binaries and related packages to be used directly in embedded Linux system.
- **Support for many embedded applications**: Database (SQL Lite, Metalite), webserver (Boa, thttpd) Graphics (PEG, Nano)

Types of embedded Linux systems

Embedded Linux systems are generally classified by criteria that would provide information about the structure of the system. Thus it may be classified primarily on the basis of size and timing constraints.

Size:

Linux features a micro-kernel architecture which actually consumes very little memory of about 100 KB which combined with the networking stack and a few basic utilities can fit in quite nicely in 500 K of memory and can be adapted to work with very little RAM and ROM (as low as 256KB ROM and 512KB RAM). A few examples of small footprint Embedded Linux are ETLinux, LEM, uClinux, uLinux, ThinLinux etc. The physical size of an embedded system determines the capabilities offered by the hardware. There are three broad categories of the embedded LINUX on the basis of size: small, medium and large.

Small Systems:

- Low powered CPU
- > 4 MB of ROM (Normally NOR Flash-based)
- 8 to 16 MB RAM

Medium-Size Systems:

- Medium powered CPU
- > 32 MB RAM (NOR Flash-based mainly, sometimes NAND Flash based)
- 64-128 MB RAM
- (optional) NAND Flash -based secondary memory removable memory cards

Large-Size systems:

- powerful CPU/multiple CPUs
- permanent storage
- large RAM

Different types of Embedded Linux versions

There are already many examples of Embedded Linux systems; it's safe to say that some form of Linux can run on just about any computer that executes code. The ELKS (Embeddable Linux Kernel Subset) project, for example, plans to put Linux onto a Palm Pilot. Here are a couple of the more well-known small footprint Embedded Linux versions:

ETLinux -- a complete Linux distribution designed to run on small industrial computers, especially PC/104 modules.

LEM -- a small (<8 MB) multi-user, networked Linux version that runs on 386s.

LOAF -- "Linux On A Floppy" distribution that runs on 386s.

uClinux -- Linux for systems without MMUs. Currently supports Motorola 68K, MCF5206, and MCF5207 ColdFire microprocessors.

uLinux -- tiny Linux distribution that runs on 386s.

ThinLinux -- a minimized Linux distribution for dedicated camera servers, X-10 controllers, MP3 players, and other such embedded applications.

Software and hardware requirements

Several user-interface tools and programs enhance the versatility of the Linux basic kernel. It's helpful to look at Linux as a continuum in this context, ranging from a stripped-down micro-kernel with memory management, task switching and timer services to a full-blown server supporting a complete range of file system and network services.

A minimal Embedded Linux system needs just three essential elements:

- A boot utility
- The Linux micro-kernel, composed of memory management, process management and timing services
- An initialization process

To doing anything useful while remaining minimal, you also need to add:

- Drivers for hardware
- One or more application processes to provide the needed functionality

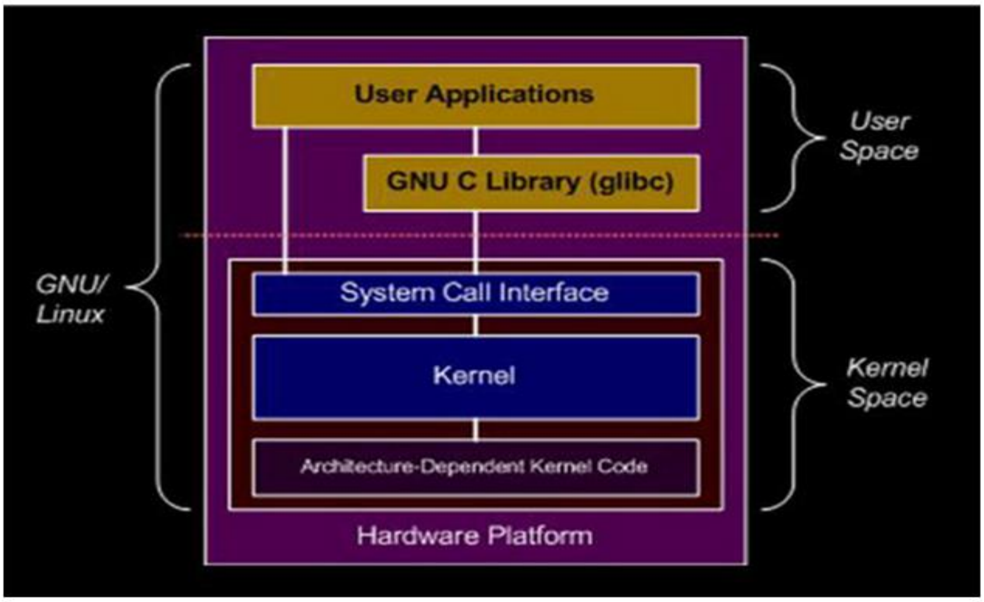
As additional requirements become necessary, you might also want:

- A file system (perhaps in ROM or RAM)
- TCP/IP network stack
- A disk for storing semi-transient data and swap capability
- A 32-bit internal CPU (required by all complete Linux systems)

Few applications areas :

Embedded Systems running through Linux

- Smartphones, Tablets
- Development boards (PCBs)
- Routers
- Robotics



Kernel Subsystems:

1) ***Process management*** – Schedule all the processes and control multitasking

2) ***Memory Management*** – Manages the physical memory and provides memory mapping, shared virtual memory, swapping etc.

3) ***File system Management*** – Manages the file system including device files

4) ***Inter-Process Communication*** – Manages and control the communication between various processes

5) ***Network Interface*** – Provides network access to the Linux machine via protocols like TCP, UDP, IPV4, IPV6 etc

6) ***Device driver*** – It forms a medium for the communication between various processes with the actual hardware.

Linux Devices

Char device

_ Character and block devices.
_ char device─ parallel port, LCD matrix display, or serial port or keypad or mouse.
_ Character access ─ byte-by-byte and analogous to the access from and to a printer device.

Block device

• block device─ a file system (disk).
• Linux permits a block device to read and write byte-by-byte like a char device or read and write block-wise like a block device. A part of the block can be accessed

Net device

_ A net device is a device that handles network interface device (card or adapter) using a line protocols, for example tty or PPP or SLIP. A *network interface* receives or sends packets using a protocol and sockets, and the kernel uses the modules related to packet transmission.

Registering and De-registering and related functions of Linux Modules

Function	Action
module initialization	• module initialization, handling the errors, prevention of unauthorized port accesses, usage-counts, root level security and clean up. • A module creates by compiling without main ( ). • A module is an object file. • For example, object module1.o creates from module1.c file by command $ gcc –c {flags} module1.c
init_module()	• Called before the module is inserted into the kernel. • The function returns 0 if initialization succeeds and –ve value if does not. • The function registers a handler for something with the kernel. • Alternatively it replaces one of the kernel functions by overloading.

insmod	• Inserts module into the Linux kernel. The object file module1.o, inserts by command $ insmod module1.o.
rmmod	• A module file module1.o is deleted from the kernel by command $ rmmod module1
cleanup	• A kernel level void function, which performs the action on an rmmod call from the execution of the module. The cleanup_module() is called just before the module is removed. The cleanup_module() function negates whatever init_module() did and the module unloads safely
Registering modules	
register_capability	A kernel level function for registering
unregister_capability	A kernel level function for deregistering
register_symtab	A symbol table function support, which exists as an alternative to declaring functions and variables static

```
#include <linux/module.h>
#include <linux/config.h>
#include <linux/init.h>

MODULE_LICENSE("GPL");
static int __init minimal_init(void)
{
  return 0;
}
static void __exit minimal_cleanup(void)
{
}
module_init(minimal_init);
module_exit(minimal_cleanup);
```

IPC functions

• Linux/ipc.h included to support IPCs
• signals on an event─ Linux header file Linux/signal.h, included to support
• multithreading ─ Linux/pthread.h included
• mutex and semaphores ─ Linux/sem.h included
• Message queues ─ Linux/msg.h

Creating a POSIX thread.

1) Pthreads are created using `pthread_create()`.

```
#include <pthread.h>

int pthread_create (pthread_t *thread_id, const pthread_attr_t *attributes,
                void *(*thread_function)(void *), void *arguments);
```

This function creates a new thread. pthread_t is an opaque type which acts as a handle for the new thread. attributes is another opaque data type which allows you to fine tune various parameters, to use the defaults pass NULL. thread_function is the function the new thread is executing, the thread will terminate when this function terminates, or it is explicitly killed. Arguments is a void * pointer which is passed as the only argument to the thread_function.

2) Pthreads terminate when the function returns, or the thread can call `pthread_exit()` which terminates the calling thread explicitly.

```
Int pthread_exit (void *status);
```

status is the return value of the thread. (note a thread_function returns a void *, so calling return(void *) is the equivalent of this function.

3) One Thread can wait on the termination of another by using `pthread_join()`

```
Int pthread_join (pthread_t thread, void **status_ptr);
```

The exit status is returned in status_ptr.

4) A thread can get its own thread id, by calling `pthread_self()`

```
pthread_t pthread_self ();
```

5) Two thread id's can be compared using `pthread_equal()`

```
Int pthread (pthread_t t1, pthread_t t2);
```

Returns zero if the threads are different threads, non-zero otherwise.

Semaphores

sem_open( )function creates a new named semaphore or opens an existing named semaphore. After the semaphore has been opened, it can be operated on using *sem_post( )* and *sem_wait*( ). When a process has finished using the semaphore, it can use *sem_close*( ) to close the semaphore. When all processes have finished using the semaphore, it can be removed from the system using *sem_unlink*( ).

Mutexes

Mutexes have two basic operations, lock and unlock. If a mutex is unlocked and a thread calls lock, the mutex locks and the thread continues. If however the mutex is locked, the thread blocks until the thread 'holding' the lock calls unlock.

There are 5 basic functions dealing with mutexes.

1) Note that you pass a pointer to the mutex, and that to use the default attributes just pass `NULL` for the second parameter.

```
Int pthread_mutex_init (pthread_mutex_t *mut, const pthread_mutexattr_t *attr);
```

2) Locks the mutex :

```
Int pthread_mutex_lock (pthread_mutex_t *mut);
```

3)Unlocks the mutex :

```
Int pthread_mutex_unlock (pthread_mutex_t *mut);
```

4) Either acquires the lock if it is available, or returns EBUSY.

```
Int pthread_mutex_trylock (pthread_mutex_t *mut);
```

5)Deallocates any memory or other resources associated with the mutex.

```
Int pthread_mutex_destroy (pthread_mutex_t *mut);
```

A short example

Consider the problem we had before, now lets use mutexes:

```
THREAD 1                         THREAD 2
pthread_mutex_lock (&mut);
                                 pthread_mutex_lock (&mut);
a = data;                        /* blocked */
a++;                             /* blocked */
data = a;                        /* blocked */
pthread_mutex_unlock (&mut);     /* blocked */
                                 b = data;
                                 b--;
                                 data = b;
                                 pthread_mutex_unlock (&mut);
[data is fine.  The data race is gone.]
```

Message Queue

The POSIX message queue API is as follows:

Function	**Summary**
mq_open()	Initialize a named queue
mq_close()	close a message queue
mq_getattr()	get the current attributes of a message queue
mq_notify()	notify the calling process when the queue becomes nonempty
mq_open()	open or create a message queue
mq_receive()	receive a message from a queue
mq_send()	put a message into a message queue
mq_setattr()	set the flags for a message queue
mq_unlink()	unlink (i.e. delete) a message queue

RT Linux

• For real time tasks and predictable hard real time behaviour, an extension of Linux is a POSIX hard real-time environment using a real time core.
• The core is called RTLinuxFree and RTLinuxPro , freeware and commercial software respectively. V. Yodaiken developed RTLinux, later FSM Labs commercialized RTLinuxPro and now Wind River has acquired it.
• Relatively simple modifications, which converts the existing Linux kernel into a hard real-time environment.
• Deterministic interrupt-latency ISRs execute at RTLinux core and other in-deterministic processing tasks are transferred to Linux.
• The forwarded Linux functions are placed in FIFO with sharing of memory between RTLinux threads as highest priority and Linux functions running as low priority threads.

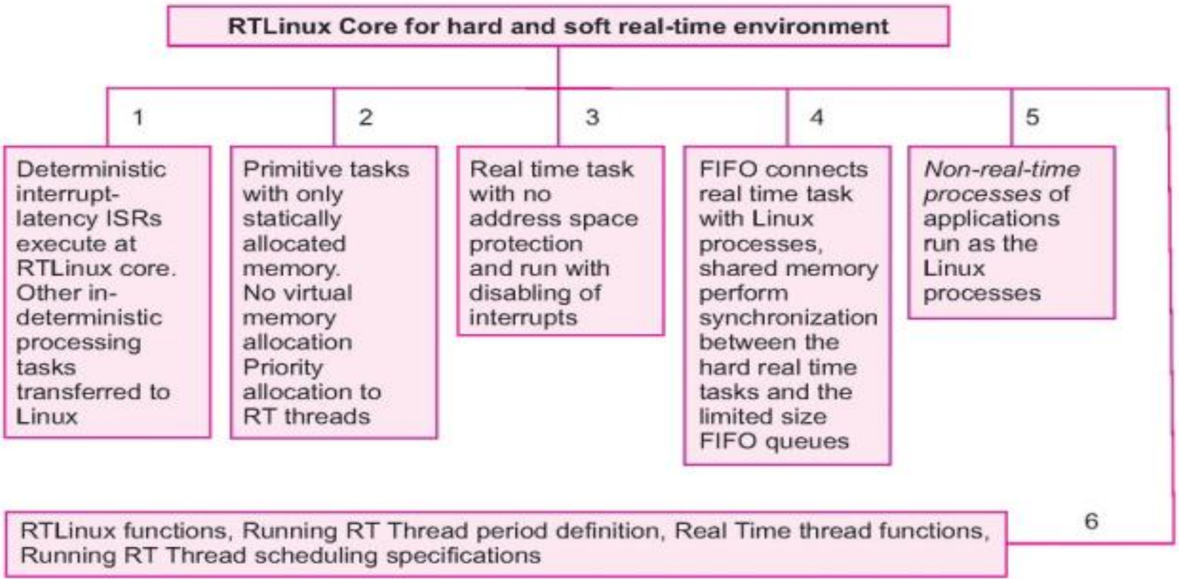


Figure1: RT Linux basic features

Running the task for hard real time performance has the following configuration:

_ Run the primitive tasks with only statically allocated memory.
_ The dynamic memory allocation or virtual memory allocation introduces unpredictable allocation and load timings
_ Run the real time task with no address space protection.
_ The memory address protection involves additional checks, which also introduce the unpredictable allocation and load timings
_ Run with disabling of interrupts so that other interrupts don’t introduce the unpredictability.
_ Run a simple fixed priority scheduler.
_ Run with disabling of interrupts so that other interrupts don’t introduce the unpredictability.
_ Run a simple fixed priority scheduler.
_ Applications can be configured to run differently.
_ RTLinux allows flexibility in defining realtime task behaviour, synchronization and communication
_ RTLinux kernel designed with modules, which can be replaced to make behavior flexible wherever possible
_ Applications run as the Linux processes.

Programming with RTLinux

_ include rtl.mk /* Include RTLinux make file. The rtl.mk file is an include file which contains all the flags needed to compile the code. */
_ all: module1.o /* Object file at module1.o */
_ clean: rm -f .o **/*** Remove using function rm object files inserted before this file */
_ module1.0: module1.c **/*** module1.0 is object file of source file module1.c */
_ $(cc) ${include} ${cflags} -c module1.c /* Compile, include, Cflags C module module1.c */

Functions in RTLinux

- The *init_module()* which is called when the module is inserted into the kernel. It should return 0 on success and a negative value on failure.
- The *cleanup_module()* which is called just before the module is removed.
- This command creates a module file named module.o, which can be inserted into the kernel by using the '*insmod*' command :

```
$ insmod module.o
```

- Similarly, for removing the module, use the *'rmmod*' command :

```
$ rmmod module
```

Creating RTLinux Threads

A realtime application is usually composed of several ``threads" of execution. Threads are light-weight processes which share a common address space. In RTLinux, all threads share the Linux kernel address space. The advantage of using threads is that switching between threads is quite inexpensive when compared with context switch.

RT thread functions

1) The init_module() invokes pthread_create(). This is for creating a new thread that executes concurrently with the calling thread. *This function must only be called from the Linux kernel thread (i.e., using init_module()).*

```
int  pthread_create(pthread_t  * thread,
                    pthread_attr_t * attr,
                    void * (*thread_code)(void *),
                    void * arg);
```

The new thread created is of type pthread_t, defined in the header pthread.h. This thread executes the function thread_code(), passing it *arg* as its argument. The *attr* argument specifies thread attributes to be applied to the new thread. If *attr* is NULL, default attributes are used.

So here, thread_code() is invoked with no argument. thread_code has three components - initialization, run-time and termination.

2) In the initialization phase, is the call to pthread_make_periodic_np().

```
int pthread_make_periodic_np(pthread_t thread,
                             hrtime_t start_time,
```

```
                              hrtime_t period);
```

pthread_make_periodic_np marks the *thread* as ready for execution. The thread will start its execution at *start_time* and will run at intervals specified by *period* given in nanoseconds.

3) *gethrtime* returns the time in nanoseconds since the system bootup.

```
hrtime_t gethrtime(void);
```

This time is never reset or adjusted. gethrtime always gives monotonically increasing values. hrtime_t is a 64-bit signed integer.

4) The only way to stop the program is by removing it from the kernel with the *rmmod* command. This invokes the *cleanup_module()*, which calls *pthread_delete_np*() to cancel the thread and deallocate its resources.

An example program

The best way to understand the working of a thread is to trace a real-time program. For example, the program shown below will execute once every second, and during each iteration it will print 'Hello World'.

The Program code (file - hello.c) :

```
#include <rtl.h>
#include <time.h>
#include <pthread.h>

pthread_t thread;

void * thread_code(void)
{
        pthread_make_periodic_np(pthread_self(), gethrtime(), 1000000000);

        while (1)
        {
                pthread_wait_np ();
                rtl_printf("Hello World\n");
        }

        return 0;
}

int init_module(void)
{
   return pthread_create(&thread, NULL, thread_code, NULL);
}

void cleanup_module(void)
{
   pthread_delete_np(thread);
}
```

Real-time FIFO Functions

Realtime FIFOs are First-In-First-Out queues that can be read from and written to by Linux processes and RTLinux threads. FIFOs are uni-directional – you can use a pair of FIFOs for bi-directional data exchange.

To use the FIFOs, the system/rtl posixio.o and fifos/rtl fifo.o Linux modules must be loaded in the kernel. RT-FIFOs are Linux character devices with the major number of 150. Device entries in /dev are created during system installation. The device file names are /dev/rtf0, /dev/rtf1, etc., through /dev/rtf63 (the maximum number of RT-FIFOs in the system is configurable during system compilation).

Before a realtime FIFO can be used, it must be initialized: #include int rtf_create(unsigned int fifo, int size); int rtf_destroy(unsigned int fifo); rtf create allocates the buffer of the specified size for the fifo buffer. The fifo argument corresponds to the minor number of the device. rtf destroy deallocates the FIFO.

After the FIFO is created, the following calls can be used to access it from RTLinux threads: open(2) , read(2) , write(2) and close(2) .

Function	**Description**
rtf_create	create a real-time fifo
rtf_create_handler	install a handler for real-time fifo data
rtf_create_rt_handler	install a handler for real-time fifo data
rtf_destroy	remove a real-time fifo created with rtf create
rtf_flush	empty a real-time FIFO
rtf_get	read data from a real-time fifo
rtf_link_user_ioctl	install an ioctl handler for a real-time FIFO
rtf_put	write data to a real-time fifo
rtf_make_user_pair	make a pair of RT-FIFOs act like a bidirectional FIFO
rtl_allow_interrupts	control the CPU interrupt state
rtl_free_irq	install and remove real-time interrupt handlers
rtl_free_soft_irq	install and remove software interrupt handlers
rtl_get_soft_irq	install and remove software interrupt handlers
rtl_request_irq	install and remove real-time interrupt handlers

Tiny OS

Introduction

- Most widely used operating system for sensor networks
- Sensor-actuator networks
- Embedded robotics
- Developed at UC, Berkeley
- It is written in the programming language nesC, as a set of cooperating tasks and processes.

Why TinyOS?

- Traditional OSes are not suitable for networked sensors
- Characteristics of networked sensors
- Small physical size & low power consumption
- Software must make efficient use of processor & memory, enable low power communication
- Concurrency intensive
- Simultaneous sensor readings, incoming data from other nodes
- Many low-level events, interleaved / high-level processing
- Limited physical parallelism (few controllers, limited capability)
- Diversity in design & usage
- Software modularity – application specific

TinyOS

TinyOS is a lightweight operating system specifically designed for low-power wireless sensors. TinyOS differs from most other operating systems in that its design focuses on ultra low-power operation. Rather than a full-fledged processor, TinyOS is designed for the small, low-power microcontrollers motes have. Furthermore, TinyOS has very aggressive systems and mechanisms for saving power.

TinyOS makes building sensor network applications easier. It provides a set of important services and abstractions, such as sensing, communication, storage, and timers. It defines a concurrent execution model, bso developers can build applications out of reusable services and components without having to worry about unforeseen interactions. TinyOS runs on over a dozen generic platforms, most of which easily support adding new sensors. Furthermore, TinyOS's structure makes it reasonably easy to port to new platforms.

TinyOS applications and systems, as well as the OS itself, are written in the nesC language. nesC is a C dialect with features to reduce RAM and code size, enable significant optimizations, and help prevent low-level bugs like race conditions.

What TinyOS provides

At a high level, TinyOS provides three things to make writing systems and applications easier:

- a component model, which defines how you write small, reusable pieces of code and compose them into larger abstractions,
- a concurrent execution model, which defines how components interleave their computations as well as how interrupt and non-interrupt code interact,
- application programming interfaces (APIs), services, component libraries and an overall component structure that simplify writing new applications and services.

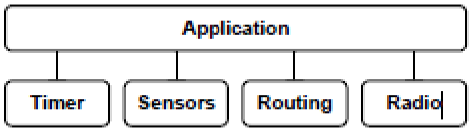
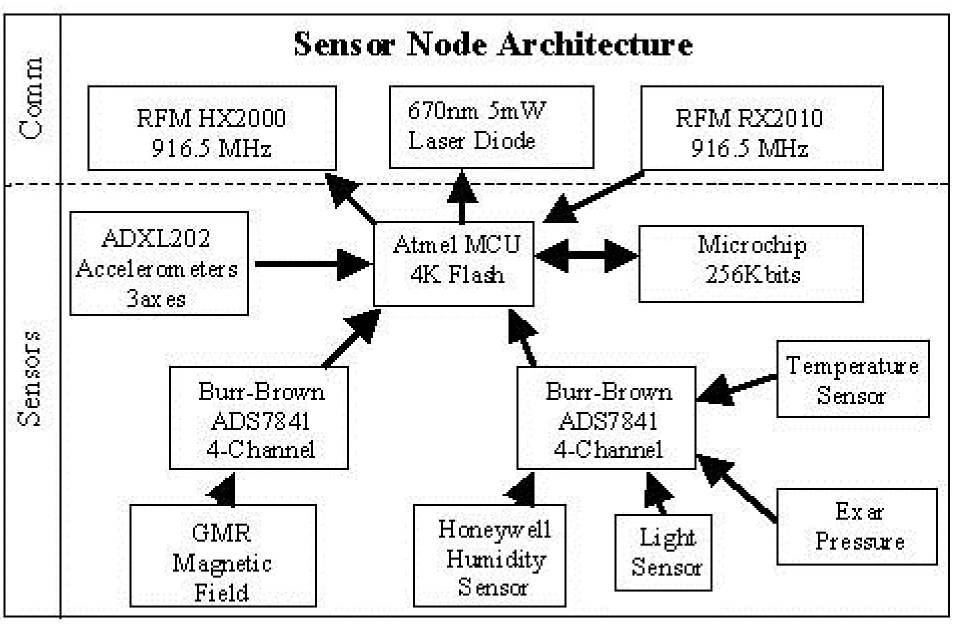


Figure 1: Example application architecture. Application code uses a timer to act periodically, sensors to collect data, and a routing layer to deliver data to a sink.



The component model is grounded in nesC. It allows you to write pieces of reusable code which explicitly declare their dependencies. For example, a generic user button component that tells you when a button is pressed sits on top of an interrupt handler. The component model allows the button implementation to be independent of which interrupt that is – e.g., so it can be used on many different hardware platforms – without requiring complex callbacks or magic function naming conventions.

The concurrent execution model enables TinyOS to support many components needing to act at the same time while requiring little RAM. First, every I/O call in TinyOS is split-phase: rather than block until completion, a request returns immediately and the caller gets a callback when the I/O completes. Since the stack isn't tied up waiting for I/O calls to complete, TinyOS only needs one stack, and doesn't have threads. Any component can post a task, which TinyOS will run at some later time. Because low-power devices must spend most of their time asleep, they have low CPU utilization and so in practice tasks tend to run very soon after they are posted (within a few milliseconds). Furthermore, because tasks can't preempt each other, task code doesn't need to worry about data races

Finally, TinyOS itself has a set of APIs for common functionality, such as sending packets, reading sensors, and responding to events. TinyOS's Hardware Abstraction Architecture (HAA), which defines how to build up from low-level hardware (e.g. a radio chip) to a hardware-independent abstraction (e.g. sending packets).

TinyOS itself is continually evolving. Within the TinyOS community, "Working Groups" form to tackle engineering and design issues within the OS, improving existing services and adding new ones.

Networked, Embedded Sensors

TinyOS is designed to run on small, wireless sensors. Networks of these sensors have the potential to revolutionize a wide range of disciplines, fields, and technologies. Recent example uses of these devices include: Golden Gate Bridge Safety. High-speed accelerometers collect synchonized data on the movement of and oscillations within the structure of San Francisco's Golden Gate Bridge. This data allows the maintainers of the bridge to easily observe the structural health of the bridge in response to events such as high winds or traffic, as well as quickly assess possible damage after an earthquake .Being wireless avoids the need for installing and maintaining miles of wires.

Volcanic Monitoring. Accelerometers and microphones observe seismic events on the Reventador and Tungurahua volcanoes in Ecuador. Nodes locally compare when they observe events to determine their location, and report aggregate data to a camp several kilometers away using a long-range wirelesss link. Small, wireless nodes allow geologists and geophysicsts to install dense, remote scientific instruments , obtaining data that answers otherwise questions about unapproachable environments.

Datacenter Provisioning. Data centers and enterprise computing systems require huge amounts of energy, to the point at which they are placed in regions that have low power costs. Approximately 50% of the energy in these systems goes into cooling, in part due to highly conservative cooling systems. By installing wireless sensors across machine racks, the data center can automatically sense what areas need cooling and can adjust which computers do work and generate heat [12]. Dynamically adapting these factors can greatly reduce power consumption, making the IT infrastructure more efficient and reducing environmental impact.

While these three application domains are only a small slice of where networks of sensors are used, they show the key differences between these networks and most other computing systems. First, these "sensor networks" need to operate unattended for long periods of time. Second, they gather data from and respond to an unpredictable environment. Finally, for reasons of cost, deployment simplicity, and robustness, they are wireless. Together, these three issues – longevity, embedment, and wireless communication – cause sensor networks to use different approaches than traditional, wired, and human-centric or machine-centric systems.

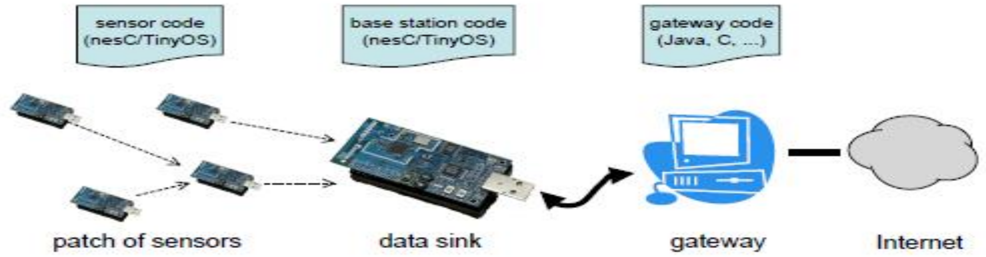


Figure 2: A typical sensor network architecture. Patches of ultra-low power sensors, running nesC/TinyOS, communicate to gateway nodes through data sinks. These gateways connect to the larger Internet.

The sheer diversity of sensor network applications means that there are many network architectures, but a dominant portion of deployments tend to follow a common one, shown in Figure 2. Of ultra-low power sensors self-organize to form an ad-hoc routing network to one or more data sink nodes.

These sensor sinks are attached to gateways, which are typically a few orders of magnitude more powerful than the sensors: gateways run an embedded form of Linux, Windows, or other multitasking operating system. Gateways have an Internet connection, either through a cell phone network, long-distance wireless, or even just wired Ethernet.

Energy concerns dominate sensor hardware and software design. These nodes need to be wireless, small, low-cost, and operate unattended for long periods. While it is often possible to provide large power resources, such as large solar panels, periodic battery replacement, or wall power, to small number of gateways, doing so to every one of hundreds of sensors is infeasible.

Anatomy of a Sensor Node (Mote)

Since energy consumption determines sensor node lifetime, sensor nodes, commonly referred to as motes, tend to have very limited computational and communication resources. Instead of a full-fledged 32-bit or 64-bit CPU with megabytes or gigabytes of RAM, they have 8-bit or 16-bit microcontrollers with a few kilobytes of RAM. Rather than gigahertz, these microcontrollers run at 1-10 megahertz. Their low-power radios can send tens to hundreds of kilobits per second, rather than 802.11’s tens of megabits. As a result, software needs to be very efficient, both in terms of CPU cycles and in terms of memory use.

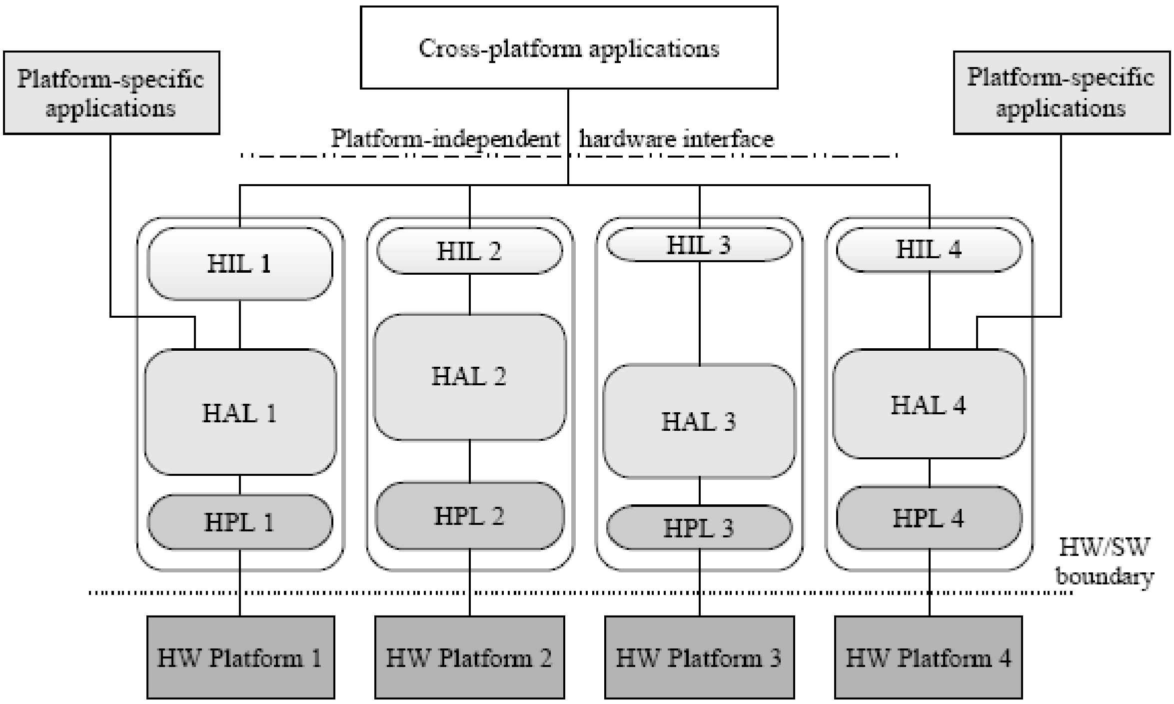


Fig. TinyOS Hardware Abstraction Architecture (HAA)

Names and Program Structure

Program structure is the most essential and obvious difference between C and nesC. C programs are composed of variables, types and functions defined in files that are compiled separately and then linked together. nesC programs are built out of components that are connected (“wired”) by explicit program statements; the nesC compiler connects and compiles these components as a single unit. To illustrate and explain these differences in how programs are built, we compare and contrast C and nesC implementations of two very simple “hello world”-like mote applications, Powerup (boot and turn on a LED) and Blink (boot and repeatedly blink a LED).

HelloWorld!

The closest mote equivalent to the classic “HelloWorld!” program is the “Powerup” application that simply turn on one of the motes LEDs at boot, then goes to sleep.
A C implementation of Powerup is fairly simple:

```
#include "mote.h"
int main()
{
mote_init();
led0_on();
sleep();
   }
```

Listing .1: Powerup in C

The Powerup application is compiled and linked with a “mote” library which provides functions to perform hardware initialization (mote init), LED control (led0 on) and put the mote in to a low-power sleep mode (sleep). The “mote.h” header file simply provides declarations of these and other basic functions. The usual C main function is called automatically when the mote boots.

The nesC implementation of Powerup is split into two parts. The first, the PowerupC module, contains the executable logic of Powerup (what there is of it. . .):

```
module PowerupC {
uses interface Boot;
uses interface Leds;
}
   implementation {

event void Boot.booted() {
call Leds.led0On();
}
   }
```

Listing .2: PowerupC module in nesC

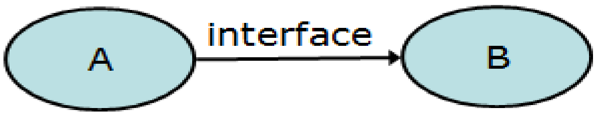
This code says that PowerupC interacts with the rest of the system via two interfaces, Boot and Leds, and provides an implementation for the booted event of the Boot interface that calls the led0On2 command of the Leds interface. Comparing with the C code, we can see that the booted event implementation takes the place of the main function, and the call to the led0On command the place of the call to the led0 on library function.

This code shows two of the major differences between nesC and C: where C programs are composed of functions, nesC programs are built out of *components* that implement a particular service (in the case of PowerupC, turning a LED on at boot-time). Furthermore, C functions typically interact by calling each other directly, while the interactions between components are specified by interfaces: the interface’s user makes requests (*calls commands*) on the interface’s *provider*, the provider makes callbacks (*signals events*) to the interface’s user. Commands and events themselves are like regular functions (they can contain arbitrary C code); calling a command or signaling an event is just a function call. PowerupC is a user of both Boot and Leds; the booted event is a callback signaled when the system boots, while the led0On is a command requesting that LED 0 be turned on.

nesC interfaces are similar to Java interfaces, with the addition of a *command* or *event* keyword to distinguish requests from callbacks:

```
interface Boot {
event void booted();
}
interface Leds {
command void led0On();
command void led0Off();
command void led0Toggle();
...
   }
```

Listing 3: Simple nesC interfaces



The second part of Powerup, the PowerupAppC *configuration*, specifies how PowerupC is connected to TinyOS's services:

```
configuration PowerupAppC { }
implementation {
components MainC, LedsC, PowerupC;
MainC.Boot -> PowerupC.Boot;
PowerupC.Leds -> LedsC.Leds;
}
```

Listing .4: PowerupAppC configuration in nesC

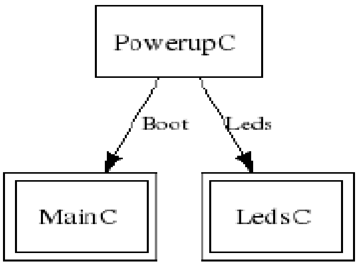


Figure 2: Wiring Diagram for Powerup application

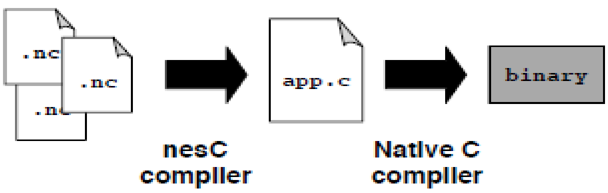


Figure 3: The nesC compilation model. The nesC compiler loads and reads in nesC components, which it compiles to a C file. This C file is passed to a native C compiler, which generates a mote binary.

This says that the PowerupAppC application is built out of three *components* (modules or configurations), MainC (system boot), LedsC (LED control), and PowerupC (our powerup module). PowerupAppC explicitly specifies the connections (or *wiring*) between the interfaces provided and used by these components. When MainC has finished booting the system it signals the booted event of its Boot interface, which is connected by the wiring in PowerupAppC to the booted event in PowerupC. This event then calls the led0On command of its Leds interface, which is again connected (wired) by PowerupAppC to the Leds interface provided by LedsC. Thus the call turns on LED 0. The resulting component diagram is shown in Figure 2 — this diagram was generated automatically from PowerupAppC by nesdoc, nesC's documentation generation tool.

Essential Differences: Components, Interfaces and Wiring

The three essential differences between C and nesC — components, interfaces and wiring — all relate to naming and organizing a program's elements (variables, functions, types, etc). In C, programs are broken into separate files which are connected via a *global namespace*: a symbolX declared in one file is connected by the linker to a symbol X *defined* in another file. For instance, if `file1.c` contains:

```
extern void g(void); /* declaration of g */
int main() /* definition of main */
{
   g(); g();
}
```

and `file2.c` contains:

```
void g(void)
{
   printf("hello world!");
}
```

then compiling and linking `file1.c` and `file2.c` connects the calls to `g()` in main to the definition of g in `file2.c`. The resulting program prints "hello world!" twice.

nesC's components provide a more systematic approach for organizing a program's elements. A component (module or configuration) groups related functionality (a timer, a sensor, system boot) into a single unit, in a way that is very similar to a class in an object-oriented language. For instance, TinyOS represents its system services as separate components such as LedsC (LED control, seen above), ActiveMessageC (sending and receiving radio messages), etc.

Interfaces bring further structure to components: components are normally specified in terms of the set of interfaces (Leds, Boot, SplitControl, AMSend) that they provide and use, rather than directly in terms of the actual operations. Interfaces simplify and clarify code because, in practice, interactions between components follow standard patterns: many components want to control LEDs or send radio messages, many services need to be started or stopped, etc.

Rather than connect declarations to definitions with the same name, nesC programs use wiring to specify how components interact: PowerupAppC wired PowerupC's Leds interface to that provided by the LedsC component, but a two-line change could switch that wiring to the `NoLedsC` component (which just does nothing):

```
components PowerupC, NoLedsC;
PowerupC.LedsC -> NoLedsC.Leds;
```

without affecting any other parts of the program that wish to use LedsC.

```
#include "mote.h"
timer_t mytimer;
void blink_timer_fired(void)
{
leds0_toggle();
}
int main()
{
mote_init();
timer_start_periodic(&mytimer, 250, blink_timer_fired);
sleep();
}
```

Listing .5: Powerup with blinking LED in C

In this example, the Blink application declares a global mytimer variable to hold timer state, and calls timer start periodic to set up a periodic 250ms timer. Every time the timer fires, the timer implementation performs a callback to the blink timer fired function specified when the timer was set up. This function simply calls a library function that toggles LED 0 on or off.

The nesC version of Blink is similar to the C version, but uses interfaces and wiring to specify the connection between the timer and the application:

```
module BlinkC {
 uses interface Boot;
 uses interface Timer;
 uses interface Leds;
 }
 implementation {

   event void Boot.booted() {
   call Timer.startPeriodic(250);
 }
 event void Timer.fired() {
  call Leds.led0Toggle();
 }
}
```

Listing .6: Powerup with blinking LED in nesC (slightly simplified)

The BlinkC module starts the periodic 250ms timer when it boots. The connection between the startPeriodic command that starts the timer and the fired event which blinks the LED is implicitly specified by having the command and event in the same interface:

```
interface Timer {
command void startPeriodic(uint32_t interval);
event void fired();
...
}
```

Finally, this Timer must be connected to a component that provides an actual timer. BlinkAppC wires BlinkC.Timer to a newly allocated timer MyTimer:

```
configuration BlinkAppC { }
implementation {
components MainC, LedsC, new TimerC() as MyTimer, BlinkC;
  BlinkC.Boot -> MainC.Boot;
  BlinkC.Leds -> LedsC.Leds;
  BlinkC.Timer -> MyTimer.Timer;
}
```

Listing .7: Powerup with blinking LED configuration (slightly simplified)

The -> and <- operators

The -> operators connect providers and users, binding callers and callees. Let's return to the PowerupToggle application and step through how its wiring works. The module PowerupToggleC uses the Leds interface. The configuration PowerupToggleAppC wires PowerupToggleC.Leds to LedsC.Leds:

```
configuration PowerupToggleAppC {}
implementation {
components MainC, LedsC, PowerupToggleC;
  PowerupToggleC.Boot -> MainC.Boot;
  PowerupToggleC.Leds -> LedsC.Leds;
}
```

Listing .8: The PowerupToggleAppC configuration revisited

Leds

LedsC provides an abstraction of 3 LEDs. While some platforms have more or fewer than 3, the Leds interface has 3 for historical reasons. Also, breaking up the LEDs into 3 instances of the same interface would be a lot of extra wiring. In addition to LedsC, there is also a NoLedsC, which can be dropped in as a null replacement: calls to NoLedsC do nothing.

```
configuration LedsC {
provides interface Leds;
}
   configuration NoLedsC {
    provides interface Leds;
   }
```

Printf

Sometimes, when debugging, it can very useful to have a mote send simple text messages. TinyOS has a printf – like the C standard library function – library for this purpose. You can use printf in your components, and the printf library will send appropriate packets over the serial port. You must start the printf library via PrintfC's SplitControl.start.

```
configuration PrintfC {
provides {
interface SplitControl as PrintfControl;
interface PrintfFlush;
}
   }
```

Example: Blink Configuration

```
configuration Blink {
}
implementation {
  components Main, BlinkM, SingleTimer, LedsC;
  Main.StdControl -> SingleTimer.StdControl;
  Main.StdControl -> BlinkM.StdControl;
```

```
  BlinkM.Timer -> SingleTimer.Timer;
  BlinkM.Leds -> LedsC;
}
```

Example: Blink Module

```
module BlinkM {
  provides {
    interface StdControl;
  }
  uses {
    interface Timer;
    interface Leds;
  }
}
implementation {
  command result_t StdControl.init() {
    call Leds.init();
    return SUCCESS;
  }
command result_t StdControl.start() {
    // Start a repeating timer that fires every 1000ms
    return call Timer.start(TIMER_REPEAT, 1000);
  }
command result_t StdControl.stop() {
    return call Timer.stop();
  }
event result_t Timer.fired()
  {
    call Leds.yellowToggle();
    return SUCCESS;
  }
}
```

Difference in how programs are structured in C, C++ and nesC

In C, the typical high-level programming unit is the file, with an associated header file that specified and documents the file's behavior. The linker builds applications out of files by matching global names; where this is not sufficient to express program structure (e.g. for callbacks), the programmer can use function pointers to delay the decision of which function is called at what point.

C++ provides explicit language mechanisms for structuring programs: classes are typically used to group related functionality, and programs are built out of interacting objects (class instances). An abstract class can be used to define common class specification patterns (like sending a message); classes that wish to follow this pattern then inherit from the abstract class and implement its methods—Java's interfaces providesimilar functionality. Like in C, the linker builds applications by matching class and function names. Finally, virtual methods provide a more convenient and more structured way than function pointers for delaying beyond link-time decisions about what code to execute.

In nesC, programs are built out of a set of cooperating components. Each component uses interfaces to specify the services it provides and uses; the programmer uses wiring to build an application out of components by writing wiring statements, each of which connects an interface used by one component to an interface provided by another. Making these wiring statements explicit instead of relying on implicit name matching eliminates the requirement to use dynamic mechanisms (function pointers, virtual methods) to express concepts such as callbacks from a service to a client.

structural element	C	C++	nesC
program unit	file	class	component
unit specification	header file	class declaration	component specification
specification pattern	—	abstract class	interface
unit composition	name matching	name matching	wiring
delayed composition	function pointer	virtual method	wiring

Table 1: Program Structure in C, C++ and nesC

Android OS

Introduction:

Android is a Linux based operating system it is designed primarily for touch screen mobile devices such as smart phones and tablet computers. The operating system has developed a lot in last 15 years starting from black and white phones to recent smart phones or mini computers. One of the most widely used mobile OS these days is android. The android is software that was founded in Palo Alto of California in 2003.

The android is a powerful operating system and it supports large number of applications in Smart phones. These applications are more comfortable and advanced for the users. The hardware that supports android software is based on ARM architecture platform. The android is an open source operating system means that it’s free and any one can use it. The android has got millions of apps available that can help you managing your life one or other way and it is available low cost in market at that reasons android is very popular.

The android development supports with the full java programming language. Even other packages that are API and JSE are not supported. The first version 1.0 of android development kit (SDK) was released in 2008 and latest updated version is jelly bean.

Android Versions

All the versions of the Android are appears in **Alphabetical Order.** The version history is given below :

A : Alpha (1.1)

B: Beta (1.2)

History of Android

- Android mobile operating system began with the release of the Android beta in November 2007. The first commercial version, Android 1.0, was released in September 2008. Since then it has worked on Alphabetically literally!

Alpha

Beta

Cupcake (1.5)

Donut (1.6)

Eclair (2.0–2.1)

Froyo (2.2–2.2.3)

Gingerbread (2.3–2.3.7)

Honeycomb (3.0–3.2.6)

Ice Cream Sandwich (4.0–4.0.4)

Jelly Bean (4.1–4.3.1)

KitKat (4.4–4.4.4)

Lollipop (5.0–5.1.1)

Marshmallow (6.0–6.0.1)

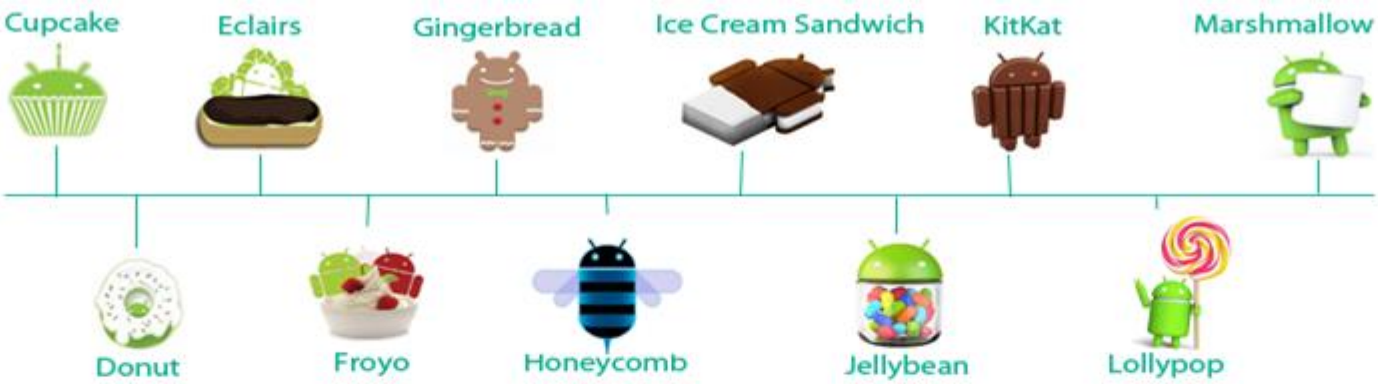


Fig. : Flow Chart Showing Various Updates In Original Version of Android

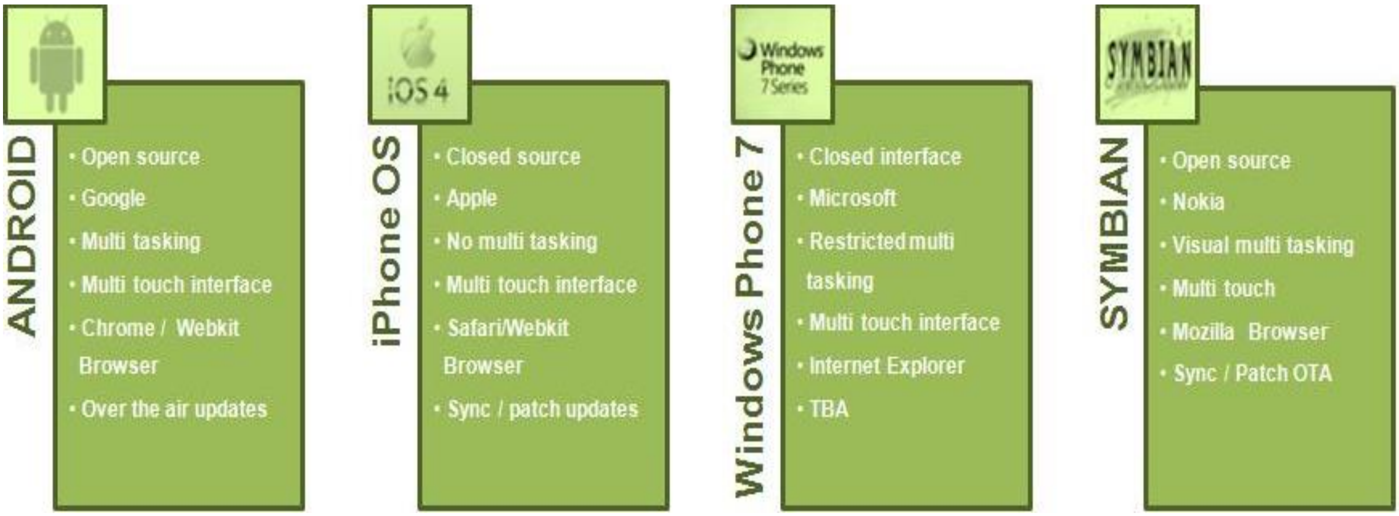


Fig.: Various Mobile Operating System Available In Markets

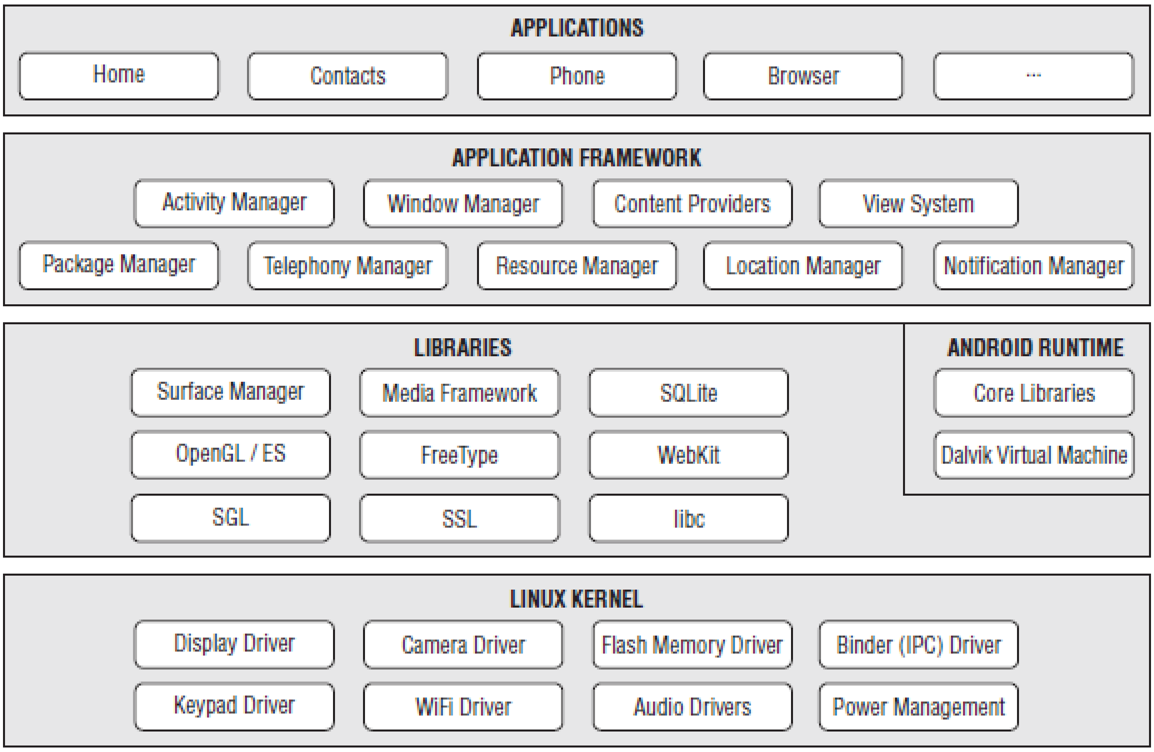
Features of Android

As Android is open source and freely available to manufacturers for customization, there are no fixed hardware and software configurations. However, Android itself supports the following features:

➤ **Storage** — Uses SQLite, a lightweight relational database, for data storage.
➤ **Connectivity** — Supports GSM/EDGE, IDEN, CDMA, EV-DO, UMTS, Bluetooth (includes A2DP and AVRCP), WiFi, LTE, and WiMAX.
➤ **Messaging** — Supports both SMS and MMS.
➤ **Web browser** — Based on the open-source WebKit, together with Chrome's V8 JavaScript engine
➤ **Media support** — Includes support for the following media: H.263, H.264 (in 3GP or MP4 container), MPEG-4 SP, AMR, AMR-WB (in 3GP container), AAC, HE-AAC (in MP4 or 3GP container), MP3, MIDI, Ogg Vorbis, WAV, JPEG, PNG, GIF, and BMP
➤ **Hardware support** — Accelerometer Sensor, Camera, Digital Compass, Proximity Sensor, and GPS
➤ **Multi-touch** — Supports multi-touch screens
➤ **Multi-tasking** — Supports multi-tasking applications
➤ **Flash support** — Android 2.3 supports Flash 10.1.
➤ **Tethering** — Supports sharing of Internet connections as a wired/wireless hotspot

Architecture of Android

In order to understand how Android works, take a look at Figure 1-1, which shows the various layers that makes up the Android operating system (OS).



The Android OS is roughly divided into five sections in four main layers:

➤ **Linux kernel** — This is the kernel on which Android is based. This layer contains all the low level device drivers for the various hardware components of an Android device.

➤ **Libraries** — These contain all the code that provides the main features of an Android OS. For example, the SQLite library provides database support so that an application can use it for data storage. The WebKit library provides functionalities for web browsing.

➤ **Android runtime** — At the same layer as the libraries, the Android runtime provides a set of core libraries that enable developers to write Android apps using the Java programming language. The Android runtime also includes the Dalvik virtual machine, which enables every Android application to run in its own process, with its own instance of the Dalvik virtual machine (Android applications are compiled into the Dalvik executables). Dalvik is a specialized virtual machine designed specifically for Android and optimized for battery-powered mobile devices with limited memory and CPU.

➤ **Application framework** — Exposes the various capabilities of the Android OS to application developers so that they can make use of them in their applications.

➤ **Applications** — At this top layer, you will find applications that ship with the Android device (such as Phone, Contacts, Browser, etc.), as well as applications that you download and install from the Android Market. Any applications that you write are located at this layer.

The Required Tools

For Android development, we can use a Mac, a Windows PC, or a Linux machine. All the tools needed are free and can be downloaded from the Web.

Eclipse

The first step towards developing any applications is obtaining the integrated development environment (IDE). In the case of Android, the recommended IDE is Eclipse, a multi-language software development environment featuring an extensible plug-in system. It can be used to develop various types of applications, using languages such as Java, Ada, C, C++, COBOL, Python, etc.

For Android development, you should download the Eclipse IDE for Java EE Developers (www.eclipse .org/downloads/packages/eclipse-ide-java-eedevelopers/heliossr1). Six editions are available: Windows (32 and 64-bit), Mac OS X (Cocoa 32 and 64), and Linux (32 and 64-bit). Simply select the relevant one for your operating system

Once the Eclipse IDE is downloaded, unzip its content (the eclipse folder) into a folder, say C:\Android\.

Android SDK

The next important piece of software you need to download is, of course, the Android SDK. The Android SDK contains a debugger, libraries, an emulator, documentation, sample code, and tutorials.

We can download the Android SDK from http://developer.android.com/sdk/ index.html.

Once the SDK is downloaded, unzip its content (the android-sdk-windows folder) into the C:\Android\ folder, or whatever name you have given to the folder you just created.

Android Development Tools (ADT)

The Android Development Tools (ADT) plug-in for Eclipse is an extension to the Eclipse IDE that supports the creation and debugging of Android applications. Using the ADT, you will be able to do the following in Eclipse:

➤ Create new Android application projects.

➤ Access the tools for accessing your Android emulators and devices.
➤ Compile and debug Android applications.
➤ Export Android applications into Android Packages (APK).
➤ Create digital certificates for code-signing your APK.

To install the ADT, first launch Eclipse by double-clicking on the *eclipse.exe* file located in the *eclipse* folder.

Anatomy of an Android Application

First, note the various files that make up an Android project in the Package Explorer in Eclipse .The various folders and their files are as follows:

➤ *src* — Contains the .java source files for your project.
➤ *Android 2.3 library* — This item contains one file, android.jar, which contains all the class libraries needed for an Android application.
➤ *gen* — Contains the R.java file, a compiler-generated file that references all the resources found in your project.
➤ *assets* — This folder contains all the assets used by your application, such as HTML, text files, databases, etc.
➤ *res* — This folder contains all the resources used in your application. It also contains a few other subfolders: drawable-<*resolution*>, layout, and values.
➤ *AndroidManifest.xml* — This is the manifest file for your Android application. Here you specify the permissions needed by your application, as well as other features (such as intent-filters, receivers, etc.).

The main.xml file defines the user interface for your activity. Observe the following in bold:

```
<TextView
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:text="@string/hello" />
```

The @string in this case refers to the strings.xml file located in the res/values folder. Hence, @string/hello refers to the hello string defined in the strings.xml file, which is "Hello World, MainActivity!":

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
<string name="hello">Hello World, MainActivity!</string>
<string name="app_name">HelloWorld</string>
</resources>
```

It is recommended that you store all the string constants in your application in this strings.xml file and reference these strings using the @string identifier. That way, if you ever need to localize your application to another language, all you need to do is replace the strings stored in the strings.xml file with the targeted language and recompile your application.

Observe the content of the AndroidManifest.xml file:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
package="net.learn2develop.HelloWorld"
```

```
android:versionCode="1"
android:versionName="1.0">
<application android:icon="@drawable/icon" android:label="@string/app_name">
<activity android:name=".MainActivity"
android:label="@string/app_name">
<intent-filter>
<action android:name="android.intent.action.MAIN" />
<category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
</activity>
</application>
<uses-sdk android:minSdkVersion="9" />
</manifest>
```

The `AndroidManifest.xml` file contains detailed information about the application:

➤ It defines the package name of the application as `net.learn2develop.HelloWorld`.

➤ The version code of the application is 1. This value is used to identify the version number of your application. It can be used to programmatically determine whether an application needs to be upgraded.

➤ The version name of the application is 1.0. This string value is mainly used for display to the user. You should use the format: <*major*>.<*minor*>.<*point*> for this value.

➤ The application uses the image named `icon.png` located in the drawable folder.

➤ The name of this application is the string named `app_name` defined in the `strings.xml` file.

➤ There is one activity in the application represented by the `MainActivity.java` file. The label displayed for this activity is the same as the application name.

➤ Within the definition for this activity, there is an element named `<intent-filter>`:

- The action for the intent filter is named `android.intent.action.MAIN` to indicate that this activity serves as the entry point for the application.

- The category for the intent-filter is named `android.intent.category.LAUNCHER` to indicate that the application can be launched from the device's Launcher icon.

➤ Finally, the `android:minSdkVersion` attribute of the `<uses-sdk>` element specifies the minimum version of the OS on which the application will run.

As you add more fi les and folders to your project, Eclipse will automatically generate the content of R.java, which at the moment contains the following:

```
package net.learn2develop.HelloWorld;
public final class R {
  public static final class attr {
}
   public static final class drawable {
      public static final int icon=0x7f020000;
}
    public static final class layout {
       public static final int main=0x7f030000;
}
```

```
  public static final class string {
      public static final int app_name=0x7f040001;
      public static final int hello=0x7f040000;
     }
  }
```

Finally, the code that connects the activity to the UI (`main.xml`) is the `setContentView()` method, which is in the `MainActivity.java` file:

```
package net.learn2develop.HelloWorld;

import android.app.Activity;
import android.os.Bundle;

public class MainActivity extends Activity {
    /** Called when the activity is first created. */
   @Override
    public void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
}
```

Here, `R.layout.main` refers to the main.xml file located in the `res/layout` folder. As you add additional XML files to the `res/layout` folder, the filenames will automatically be generated in the `R.java file`. The `onCreate()` method is one of many methods that are fired when an activity is loaded.

Activity Class

The Activity base class defines a series of events that governs the life cycle of an activity. The Activity class defines the following events:

- `onCreate()` — Called when the activity is first created
- `onStart()` — Called when the activity becomes visible to the user
- `onResume()` — Called when the activity starts interacting with the user
- `onPause()` — Called when the current activity is being paused and the previous activity is being resumed
- `onStop()` — Called when the activity is no longer visible to the user
- `onDestroy()` — Called before the activity is destroyed by the system (either manually or by the system to conserve memory)
- `onRestart()` — Called when the activity has been stopped and is restarting again.

Application:

Displaying Notifications on the Status Bar

For messages that are important, you should use a more persistent method. In this case, you should use the `NotificationManager` to display a persistent message at the top of the device, commonly known as the *status bar* (sometimes also referred to as the *notification bar*).

1. Using Eclipse, create a new Android project and name it **`Notifications`**.

2. Add a new class file named `NotificationView.java` to the `src` folder of the project . In addition, add a new `notification.xml` file to the `res/layout` folder as well.

3. Populate the `notification.xml` file as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent" >
<TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="Here are the details for the notification..." />
</LinearLayout>
```

4.Populate the `NotificationView.java` file as follows:

```
package net.learn2develop.Notifications;

import android.app.Activity;
import android.app.NotificationManager;
import android.os.Bundle;

public class NotificationView extends Activity
{
   @Override
     public void onCreate(Bundle savedInstanceState)
     {

        super.onCreate(savedInstanceState);
        setContentView (R.layout.notification);

        //---look up the notification manager service---
        NotificationManager nm = (NotificationManager)
            getSystemService (NOTIFICATION_SERVICE);

        //---cancel the notification that we started
        nm.cancel (getIntent ().getExtras ().getInt ("notificationID"));
     }
}
```

5. Add the following statements in bold to the `AndroidManifest.xml` file:

```
<? xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="net.learn2develop.Notifications"
    android: versionCode="1"
    android: versionName="1.0">
  <application android: icon="@drawable/icon" android:label="@string/app_name">
     <activity android: name=".MainActivity"
               android:label="@string/app_name">
         <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
         </intent-filter>
      </activity>
       <activity android:name=".NotificationView"
```

```
                 android:label="Details of notification">
          <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.DEFAULT" />
       </intent-filter>
    </activity>
</application>
<uses-sdk android:minSdkVersion="9" />
<uses-permission android:name="android.permission.VIBRATE" />
</manifest>
```

6. Add the following statements in bold to the `main.xml` file:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
     android:orientation="vertical"
     android:layout_width="fill_parent"
     android:layout_height="fill_parent" >
<Button
     android:id="@+id/btn_displaynotif"
     android:layout_width="fill_parent"
     android:layout_height="wrap_content"
     android:text="Display Notification" />
</LinearLayout>
```

7. Finally, add the following statements in bold to the `MainActivity.java` file:

```
package net.learn2develop.Notifications;

import android.app.Activity;
import android.os.Bundle;
import android.app.Notification;
import android.app.NotificationManager;
import android.app.PendingIntent;
import android.content.Intent;
import android.view.View;
import android.widget.Button;

public class MainActivity extends Activity {
     int notificationID = 1;
     /** Called when the activity is first created. */
     @Override
     public void onCreate(Bundle savedInstanceState) {
          super.onCreate(savedInstanceState);
          setContentView(R.layout.main);

          Button button = (Button) findViewById(R.id.btn_displaynotif);
          button.setOnClickListener(new Button.OnClickListener() {
             public void onClick(View v) {
                displayNotification();
         }
    });
}

protected void displayNotification()
{
      //---PendingIntent to launch activity if the user selects
     // this notification---
     Intent i = new Intent(this, NotificationView.class);
     i.putExtra("notificationID", notificationID);

     PendingIntent pendingIntent =
        PendingIntent.getActivity(this, 0, i, 0);
```

```
        NotificationManager nm = (NotificationManager)
             getSystemService(NOTIFICATION_SERVICE);

        Notification notif = new Notification(
              R.drawable.icon,
              "Reminder: Meeting starts in 5 minutes",
               System.currentTimeMillis());

       CharSequence from = "System Alarm";
       CharSequence message = "Meeting with customer at 3pm...";

       notif.setLatestEventInfo(this, from, message, pendingIntent);

       //---100ms delay, vibrate for 250ms, pause for 100 ms and
       // then vibrate for 500ms---
       notif.vibrate = new long[] { 100, 250, 100, 500};
       nm.notify(notificationID, notif);
    }
}
```

8. Press F11 to debug the application on the Android Emulator.

9. Click the Display Notification button (see the top left of Figure 2) and a notification will appear on the status bar.

10. Clicking and dragging the status bar down will reveal the notification (see the right of Figure 2).

11. Clicking on the notification will reveal the `NotificationView` activity. This also causes the notification to be dismissed from the status bar.

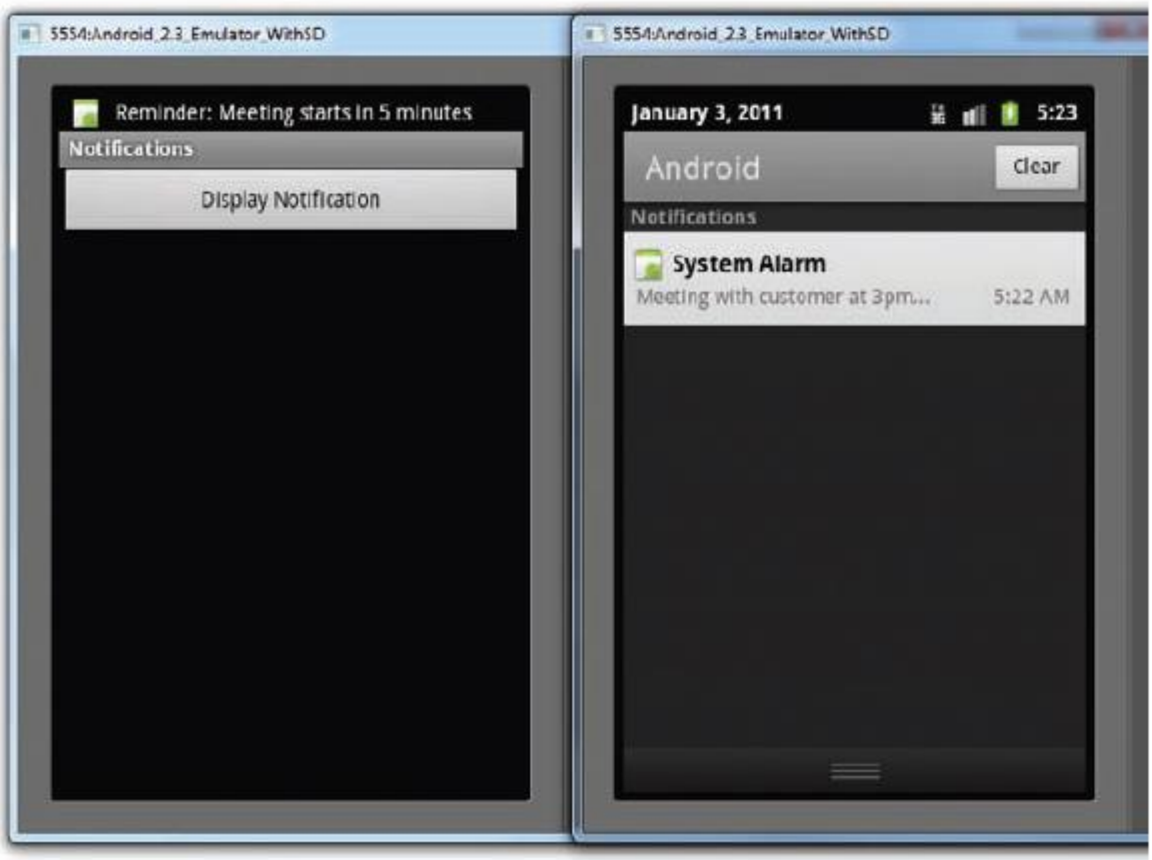


FIGURE 2

How It Works

To display a notification, you first created an `Intent` object to point to the `NotificationView` class:

```
//---PendingIntent to launch activity if the user selects
// this notification---
Intent i = new Intent(this, NotificationView.class);
i.putExtra("notificationID", notificationID);
```

This intent will be used to launch another activity when the user selects a notification from the list of notifications. In this example, you added a key/value pair to the `Intent` object so that you can tag the notification ID, identifying the notification to the target activity. This ID will be used to dismiss the notifications later.

You would also need to create a `PendingIntent` object. A `PendingIntent` object helps you to perform an action on your application's behalf, often at a later time, regardless of whether your application is running or not. In this case, you initialized it as follows:

```
PendingIntent pendingIntent =
PendingIntent.getActivity(this, 0, i, 0);
```

The `getActivity()` method retrieves a `PendingIntent` object and you set it using the following arguments:

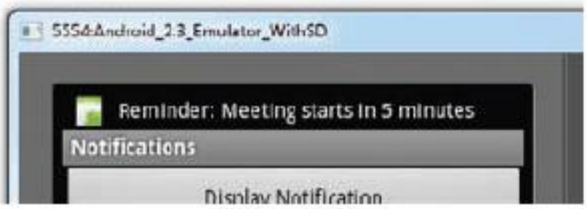
➤ context — Application context
➤request code — Request code for the intent
➤ intent — The intent for launching the target activity
➤ flags — The flags in which the activity is to be launched

You then obtain an instance of the `NotificationManager` class and create an instance of the `Notification` class:

```
NotificationManager nm = (NotificationManager)
     getSystemService(NOTIFICATION_SERVICE);

Notification notif = new Notification(
      R.drawable.icon,
     "Reminder: Meeting starts in 5 minutes",
      System.currentTimeMillis());
```

The `Notification` class enables you to specify the notification's main information when the notification first appears on the status bar. The second argument to the `Notification` constructor sets the "ticker text" on the status bar.



Next, you set the details of the notification using the `setLatestEventInfo()` method:

```
CharSequence from = "System Alarm";
CharSequence message = "Meeting with customer at 3pm...";
notif.setLatestEventInfo(this, from, message, pendingIntent);
//---100ms delay, vibrate for 250ms, pause for 100 ms and
// then vibrate for 500ms---
notif.vibrate = new long[] { 100, 250, 100, 500};
```

The preceding also sets the notification to vibrate the phone. Finally, to display the notification you use the `notify()` method:

```
nm.notify(notificationID, notif);
```

When the user clicks on the notification, the `NotificationView` activity is launched. Here, you dismiss the notification by using the `cancel()` method of the `NotificationManager` object and passing it the ID of the notification (passed in via the `Intent` object):

```
//---look up the notification manager service---
NotificationManager nm = (NotificationManager)
   getSystemService(NOTIFICATION_SERVICE);

//---cancel the notification that we started
nm.cancel(getIntent().getExtras().getInt("notificationID"));
```