Vladimir Vassilevsky

vlv@abvolt.com

HALOS – a home made RTOS for embedded applications

Why RTOS?

Unified hardware – software interface

Providing basic services such as multitasking, messaging and timekeeping

Code reuse

Portability

Convenience

Separation of developer duties and responsibilities - especially useful when many people are working on the project

Why yet another RTOS?

Commercial RTOS: expensive, not very easy to use.

Open Source: poor code quality, "grey" copyright issues, no trust in critical applications

Kernels like DSP BIOS or VDK: incompatible API, changes from version to version, tied to the particular development board hardware.

Both commercial and open source OSes: lots of obscure system calls on initialization of the OS internals

Escaping monstrosity or primitivism

Specific inconveniences and limitations of the particular OSes

Single stack multitasking: impossible to share objects.

Round robin multitasking: hard to do real time processing.

OS is fun to develop.

OS is not that hard to develop.

Jean Labrosse's book tells how to develop an OS.

With any OS, you still have to write your application. Why OS should be bigger then the application?

Our goals:

Lean memory footprint

Suitable way of hardware abstraction

Minimize system related technicalities, use general purpose tools

100% proprietary code developed at home

Learning by making

Target: BlackFin

Retargetable for other CPUs of 100+ MIPS class

Problems

Runtime initialization: Due to the dependencies, the OS objects have to be created dynamically.

Efficient TCP/IP, FAT, USB host in multi threaded environment are difficult to develop.

Result

Practical memory footprint (OS + application): 64K code + 64K data

Practical minimal CPU: 50-100 MIPS (so the OS overhead ~ 1% CPU).

Included: FAT, TCP/IP stack

General

HALOS is the preemptive multitasking RTOS based on the C++ language. HALOS includes the RTOS kernel, the BSP level, and the optional components and protocols (TCP/IP, file systems, USB host and client, math. functions and templates, etc.)

HALOS is monolithic OS.

HALOS was designed with the primary goals of convenience and portability. Most of the OS code is in C++, with a few machine dependent functions in the assembly.

The OS is monolithic, i.e. the OS kernel is statically linked to the application code. The OS kernel is implemented as the library module. I.e. there is generally no need to recompile the kernel when working on the application.

At the current revision, HALOS does not support for virtual memory and memory protection.

HALOS Task Switch

HALOS Task Switch is the dedicated interrupt with the lowest priority. The scheduling, timekeeping, message passing, callbacks from timers and many other functions are running from there.

The task switch is activated by any event that can require rescheduling of the tasks: a message was sent, a task was created, activated or put to sleep, a timer tick happened. The hardware interrupts do not activate the task switch implicitly; if rescheduling is required, it should be requested in the interrupt. This is done to improve the efficiency; there is generally no need to reschedule tasks after every hardware interrupt. The scheduler starts the task with the maximum priority which is in the active state.

HALOS interrupt API

HALOS abstracts the interrupts and the interrupt controller. The code which does that is located in the file interrupts.cpp. At the API level, the interrupt description is independent from the C++ compiler and the hardware platform. Every possible source of the interrupt is assigned the int_id number. The assignment of int_id is CPU specific; it is defined in the file *interrupt_hardware_id.h* .

HALOS supports for the nested interrupts. All interrupts are using one dedicated stack. On entry to the interrupt, the stack is switched. This is done to reduce the required stack usage of the tasks (Otherwise, we should have to provide the stack space for all nested interrupts at the stack of every task – very inefficient memory usage).

The interrupt stack allocation is up to the user. It is not done automatically, because it is generally preferred to have the interrupt stack in the fast memory area.

Initialization:

void Interrupt_System_Init(u32 *interrupt_stack_top, u32 interrupt_stack_size_bytes);

This function should be called before using any interrupt related functionality.

The interrupt can have the priority from 0 (lowest) to 6(highest) for BlackFin. An interrupt can be preempted by an interrupt of the higher priority.

Interrupt handler:

From the point of view of the developer, an interrupt handler is a common C/C++ function:

void Some_Interrupt_Handler(void *ptr);

The optional parameter (void *ptr) can be used to pass a data to the interrupt handler. This is handy when the same interrupt handler function is used to handle different interrupt sources; the instances of the function can be distinguished. For example, the same interrupt function can be used to handle several UARTs.

The interrupt handler function should always check if the corresponding peripheral device needs to be serviced, and if so, it should reset the appropriate flags of this peripheral device. This has to be done because the same interrupt vector can be assigned to several different peripheral devices.

How to install the function as the interrupt handler:

u8 Install_Interrupt_Handler(void (*handler)(void *), void *param, u16 priority, u16 hardware_int_id);

This installs the user function as the interrupt handler, returns 1 on success, 0 on failure.

void (*handler)(void *) – the pointer to the user function used as the handler of the interrupt
void *param – the optional parameter passed to the interrupt function
u16 priority – the priority of this interrupt handler (BlackFin: 0(lowest)…6(highest))
u16 hardware_int_id – to what source of the interrupts this handler will be attached (see the definitions of the interrupt sources in the *interrupt_hardware_id.h*).

HALOS Task API

A task is a user defined common C++ function:

void Some_Task(void *optional_param)

The parameter can be used to distinguish the multiple instances of the same task function.
Inside the task, there is an endless loop, so the task never exits.
From the point of view of the multitasking system, the parent main() function is no different from any other user defined tasks.

The task has the priority setting: 0(lowest)….254(highest). Also, the task has the status setting: TASK_ACTIVE, TASK_SLEEP, TASK_WAIT, TASK_SEMAPHORED, TASK_SLEEP_TIMER.
The task scheduler launches the task which has the highest priority and is in ACTIVE state.
If the two tasks are assigned to the same priority, and both are in the active state, then the first in the order of creation task will be executed.

Every program has two tasks at the least: main() and Idle_Task();
Idle task is required to avoid the deadlock when all tasks are sleeping. In the idle task, the system background functions like stack/heap monitoring and profiling are executed.

Initialization:

u16 Multitask_System_Init(u16 max_number_of_tasks);

This should be done after the interrupts are initialized and before any of the multitasking API is used. Returns zero if there is not enough heap space to allocate the necessary core structures.

Creation of a task:

u16 CreateTask(void (*task)(void *), void *param, u32 stack_size_bytes, u8 priority, u8 task_status, u8 *name);

This function returns task_id - a unique ID of the task. On failure, INVALID_TASK_ID is returned.

void (*task)(void *) – a pointer to the task.
void *param – an optional parameter passed to the task function
u32 stack_size – a stack allocated for the task. 512 bytes is a reasonable value for BlackFin.
u8 priority – a priority of this task (0(lowest)….254(highest))
u8 task_status – a status of the task by creation (TASK_ACTIVE, TASK_SLEEPING)
u8 name – optional name of the task (up to 8 characters).

Other Task related functions:

The functions without explicit task_id parameters are referred to the current task context.
u16 GetTaskId() – returns the ID of the current context.
u16 GetTaskPriority();
u16 SetTaskPriority();
u16 GetTaskPriority(u16 task_id);
u16 SetTaskPriority(u16 task_id);
void TaskSleep(void);
void Sleep(u32 msec);

Messaging API

The tasks and the interrupts are communicating to each other only by the use of the OS message passing mechanism. This is required to ensure that the task which received a message is put into the active state, and that the each message is unique within the system.

A message is a structure:

```
typedef struct {
        u16 msg_id;
        void *msg;
        } HALOS_MESSAGE;
```

u16 msg_id - the unique field which identifies this message
void *msg - an optional pointer to a data which is associated with this message

A message can be sent from a task or from an interrupt. A message can be received by a task only; the interrupts can't receive messages. The messages are being moved through the message queues by the OS. This happens inside the task switch.

Every message has the associated send queue for this message. This queue is required because this message can be sent several times before the scheduler moves it to the receiver queues. The send queue may be connected to several queues of the tasks which receive this message. One message can be sent to up to 4 different receivers by default (the number of receiver queues per message can be altered at OS compile time).

First, the message must be created:

// This creates a message and a send queue for it, returns the MSG_ID, which is unique within the system.

```
HALOS_MESSAGE hm;
hm.msg_id =  halos_Register_Message(u16 message_queue_length);
```

// Now any task or interrupt can send this message

u8 halos_Send_Message(HALOS_MESSAGE &hm); // This sends the message, starts the scheduler,
u8 halos_Place_Message(HALOS_MESSAGE &hm); // This sends the message but it doesn't start the scheduler

If we want to receive this message from a task:

1. Create the message queue for this task. This message queue can be common for all messages received by this task.

   HALOS_MESSAGE_QUEUE:: class is the task message queue to receive messages

   HALOS_MESSAGE_QUEUE(u16 messages, u16 task_id) – this creates a message queue for a given task with the queue length of a given number of messages.

2. Subscribe this message queue for the message:

   u8 halos_Register_Message_Queue(u16 msg_id, u8 param,
   HALOS_MESSAGE_QUEUE*);

The u8 param is MESSAGE_BUFFERED_EVERY_TIME or MESSAGE_BUFFERED_ONE_TIME.
The particular message can be buffered in the receiver queue only once, or it can be put in the queue every time the message is sent.
If the message is put in the queue only once, this message would not be put in the queue if it is already there. This is useful if the message is signaling about some repeating event. There is no reason to buffer it every time; it is only wasting the space in the queue. On the contrary, if the order of messages is important, the message should be put into the queue at every time.

Semaphores/Critical sections

Class SEMAPHORE is the critical section with the automatic priority elevation mechanism to prevent the priority inversion problem.

SEMAPHORE can be declared as static variable as well as the automatic or dynamic variable.

SEMAPHORE prevents other tasks from the entering of an atomic critical section. This is useful for the hardware drivers and large functions which require the atomic access.

The same critical section may be entered many times within the same task. However to release the critical section, it should be released the same number of times as it was entered.

SEMAPHORE::EnterCriticalSection() – try to allocate critical section. If the critical section is locked, the current task is suspended until it is unlocked. If several tasks are waiting for a semaphore, on release it will be taken by the task with the highest priority.

SEMAPHORE::TryEnterCriticalSection() – try to enter the critical section. If the critical section is not locked, then it is entered and locked to the current task. In this case, the function returns nonzero. If the section is locked by the other task, returns zero.

SEMAPHORE::TryEnterCriticalSection(u32 ms) – try to enter the critical section until the timeout of given amount of milliseconds. If the critical section is not locked, then it is entered and locked to the current task. In this case, the function returns nonzero. If the section is locked by some other task, and the timeout is expired, the function returns zero. The timeout of 0 ms is allowed; in this case the function tries to enter the critical section only once; and exits immediately after that.

SEMAPHORE::LeaveCriticalSection() – release the critical section.

Priority elevation mechanism is built into the semaphores.

Suppose the task with the priority = 1 have allocated the critical section. Now the task with the priority = 2 is active and it is trying to enter the same critical section. This is a classic problem of the priority inversion deadlock: task#1 can't release the critical section because a higher priority task#2 is currently running, however task#2 can't run because the critical section is locked.

To resolve the situation, the semaphore automatically increases the priority of the task which locked the critical section to the priority of the task which is trying to enter the section. On the exit from the critical section, the priority of the task#1 is restored to the original value.

Timers

Class TIMER_CLASS provides the timer functionality.

The timers should be declared as auto or dynamic variables, since they rely on the OS services at the time of the initialization.

The timer can be set to u32 milliseconds. The real accuracy of the timer is about one millisecond.

A timer can be AUTO_RELOAD or SINGLE_SHOT. The timer can be created as STARTED or STOPPED.

When the timer is expired:

The counter of how many times the timer was expired is incremented, so this counter can be polled.

The timer message is sent, so tasks can subscribe for this message.

The timer callback function (if any) is called: void Callback(void *param);

Note that the callback function is executed in the context of the OS Task Switch.

Other TIMER_CLASS:: functions:

```
void Start();
void Stop();
void Restart();
void Reset();
u32 Read();
u32 Period();
void Period(u32 ms);
u8 Mode();
void Mode(u8 mode);
u32 Expired();
u16 Msg_Id();
void Set_Callback(void (*callback)(void *) = NULL, void *callback_param = NULL);
```

HALOS debug services

- RS232 I/O using standard prinf() functions.
- Monitoring of the usage of stacks and heap
- Monitoring of the percentage of the CPU usage by the tasks and the system (profiler).

HALOS drivers

It is preferred that the access to the slow devices like I2C, SPI, etc. should be implemented as the driver mechanism. There are two ways to do that:

1. Create a driver as the separate task. This is the heavyweight mechanism which is good for the tasks like file system, TCP/IP and such.
2. Old style function like access: SPI_Action(), I2C_Action() …

In the case of function-like access, make sure that the driver can't be reentered by the different task. Protect it with the SEMAPHORE.

Other consideration: The driver should put the calling task into TASK_WAIT state till the operation is complete.

Memory management

HALOS does not support for the virtual memory and memory protection, as it is unsupported by BlackFin DSP. The standard heap management operators "new" and "delete" are redefined to support for the multitasking.

TCP/IP stack and FAT include memory manager of their own, operating on the fixed size blocks.

FAT 32 file system

- HALOS includes FAT/FAT32 file system module.
- Fully compatible to Microsoft spec.
- API of the file system complies to SUSv3 (POSIX).
- Designed for multi threaded operation
- The main goal of the FAT development was maximal compatibility.

TCP/IP stack

HALOS includes TCP/IP stack with socket interface.

- Multiple client/server sockets.
- Multithreaded access.
- DHCP, automatic private addressing supported

USB host/client interface

Not there yet. We are working on it.

■