

T. H. E. SOLUTION LLC

Product Development Consulting

THE Executive
Version 1.0

8/11/2004

Copyright © 2004 by T. H. E. Solution LLC

Table of Contents

1	INTRODUCTION	1
1.1	Tasks	1
1.2	Application interrupt handler	2
1.3	Events	3
2	APPLICATION PROGRAM INTERFACE	4
2.1	System	4
2.2	Tasks	4
2.3	Interrupt Control/Application Interrupt Handler	5
2.4	Events	6
2.5	Drivers	8
2.6	Configuration	8
3	DESIGN	10
3.1	Task Control Block	10
3.2	Kernel	10
3.3	Save Context	11
3.4	Base Interrupt Handler	12
3.5	Restore Context	13
	APPENDIX A, FILE SET	14
	APPENDIX B, PORTING CONSIDERATIONS	16
	APPENDIX C, GENERIC TEST SET	18
	APPENDIX D, KERNEL ERRORS	20

1 INTRODUCTION

For the 32 bit embedded world there are a number of fine real-time operating systems to use. This choice becomes a lot smaller when we enter the "deeply" embedded environment of 8 and 16 bit Microprocessors/controllers. This environment is still often implemented using an interrupt driven background with a poll loop controlled foreground. THE Executive is intended to provide a bare bones real-time executive that fits well in this environment. Its more important characteristics are:

1. It has a small footprint (the core code takes approximately 4 Kbytes).
2. It is provided open source.
3. It is cheap (and has no royalty fees).
4. It is primarily implemented in C so it can be easily moved to a vast majority of the 8/16 bit world.
5. It provides all the basic services expected of a real-time executive with little complexity.

This document provides a technical description of THE Executive and how to use it. Before we get into the details we need to understand three basic things tasks, interrupt handlers and events.

1.1 Tasks

There are many different ways to model application software. One way often used in the development of embedded applications is to use the execution flow of the software. With THE Executive there are two distinctly different types of software execution entities, tasks and interrupt handlers. The primary work of an application is done by a cooperating set of tasks. A task is a schedulable sequence of code. Every application has at least one task and normally it has many. A task typically performs some specific function (it has high functional cohesion) that requires a relatively "loose" coupling with other tasks. Because a task's "context" can be saved and restored, a task can be stopped and restarted at any time. THE Executive is a preemptive executive (it starts and stops tasks at its pleasure) that assures that the highest priority task that is ready to execute will execute. The life of a task is summarized in Figure 1.

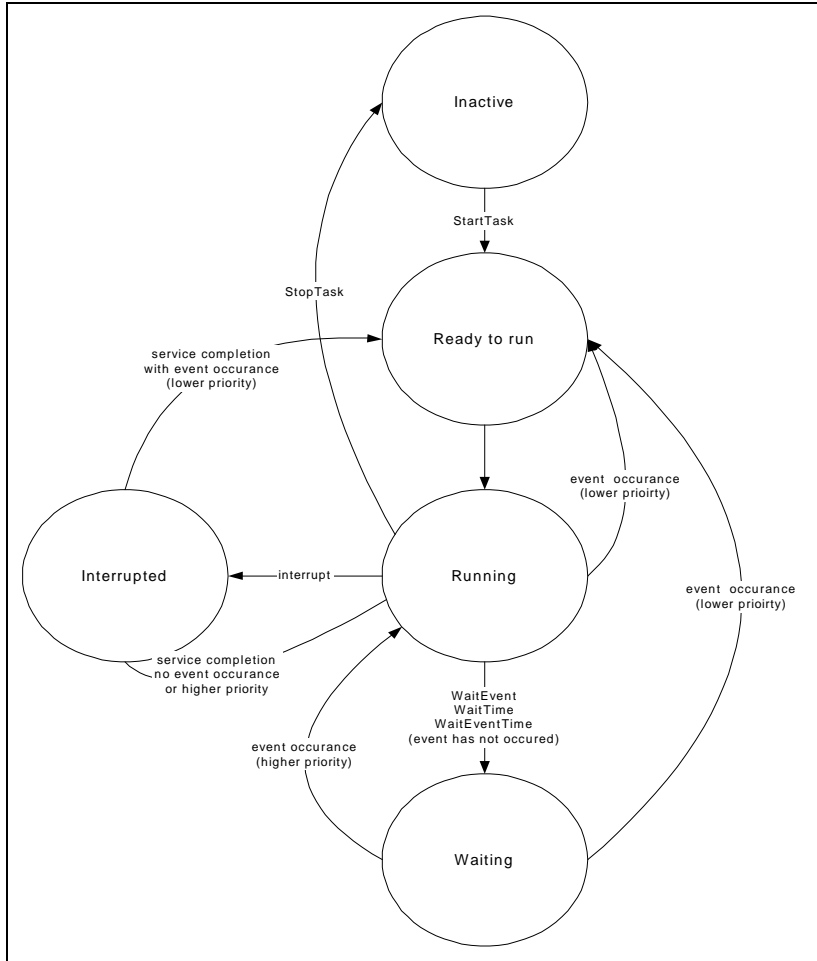


Figure 1, Task state diagram

1.2 Application interrupt handler

The other type of software execution entity is the interrupt handler. THE Executive provides the base level interrupt handling (saving context, talking to interrupt hardware etc.). This in turn executes the appropriate application level interrupt handler. Unlike tasks, interrupts are not scheduled. When they physically happen and the interrupt is enabled, the interrupt execution chain is started. This chain will execute to completion without other interruption unless the processor and interrupt control hardware supports nesting interrupts. In that case a higher priority interrupt can cause a another interrupt chain to execute (Note: The reference implementation supports nested interrupts). In either case once an interrupt occurs it will execute to completion before any task are executed.

1.3 Events

The binder that allows task to synchronize and interoperate with each other or with application interrupt handlers is the event. An event is any logical occurrence or physical resource that requires the cooperative behavior of multiple tasks or application interrupt handlers and tasks. It is implemented as a data structure that allows one task to "wait" on another task or an application interrupt handler. THE Executive supports three styles of events, binary, counting and gate. The binary and counting styles work in the same fashion as traditional semaphores. The gate style works in a gate fashion. Either it is closed and no one can "enter" or it is open and everyone can. Typical uses are:

Style	Use
Binary	Mutual exclusion Cooperative use of a single resource
Counting	Cooperative use of multiple resources
Gate	Use of a resource is either enabled or disabled

2 APPLICATION PROGRAM INTERFACE

The application uses a couple of data structures and a number of functions to interact with THE Executive. These are defined in the file exec.h. An explanation of the principal data structures and functions is provided in the following subsections.

2.1 System

These functions initialize and shutdown the real-time kernel and its application set:

KERNEL INITIALIZATION

Description: Initializes the kernel for use. Should be called only one time.

Passed parameter: none

Returned value: none

```
void Kinit(void)
```

SHUTDOWN

Description: Shuts down the kernel and its application system.

Passed parameters: none

Returned value: none

```
void Shutdown(void)
```

2.2 Tasks

The application defines the set of tasks that make up the application in the public "tasks" table:

```
struct process_def
{
    void (*entry) (void);
    unsigned char priority;
    unsigned stack_size;
};

struct process_def tasks[NO_APPL_TASK] =
{
    ApplInit,200,100,
    Monitor,1,100,
    SerialLink,2,128,
    MachineControl,3,128,
    ComplexMoveTask,2,128
};
```

THE executive provides functions to start and stop task and provide some useful task information:

```
START A TASK
```

Description: Activates a new task if it is not already active. New tasks are always placed in the ready queue.
Passed parameter: task - task's ID
Returned value: task was started (TRUE) or not (FALSE)

BOOL StartTask(unsigned char task)

DEACTIVATE THE RUNNING TASK

Description: Places the caller in the inactive state. The highest priority task ready to run is given run.
Passed parameter: none
Returned value: none

void StopTask(void)

TASK STATUS

Description: Returns the indicated task's status.
Passed parameter: taskid - task's ID
Returned value: task's status

unsigned char TaskStatus(unsigned char taskid)

TASK ID

Description: Returns the task's id.
Passed parameter: none
Returned value: task's ID

unsigned char TaskId(void)

2.3 Interrupt Control/Application Interrupt Handler

These functions provide the means to enable and disable interrupt(s) (all or specific ones) and to "attach" an application interrupt handler:

DISABLE INTERRUPTS AT CPU

Description: Disables interrupts at CPU and returns state prior to lock
Passed parameters: none
Returned value: prior CPU state

unsigned int Lock(void)

ENABLE INTERRUPTS AT CPU

Description: Enables interrupts at CPU
Passed parameter: none
Returned value: none

void Unlock(void)

RESTORE INTERRUPT AT CPU TO PRIOR STATE

Description: Restores interrupts to indicate state
 Passed parameter: old_state - CPU prior state from Lock
 Returned value: none

```
void Restore(unsigned int old_sr)
```

ENABLE INTERRUPTS

Description: Enables the indicated interrupt(s) at the interrupt controller.
 Passed parameters: mask - interrupt enable bit map (1 = enable)
 Returned value: none

```
void EnableInterrupts(unsigned char mask)
```

DISABLE INTERRUPTS

Description: Disables the indicated interrupt(s) at the interrupt controller.
 Passed parameters: mask - interrupt disable bit map (1 = disable)

```
void DisableInterrupts(unsigned char mask)
```

SET UP AIH

Description: Allows user to set up an application interrupt handler.
 Passed parameters: intrp_no - interrupt number
 aih - application interrupt handler
 Returned value: none

```
void SetAih(unsigned char intrp_no, void (*aih)(void))
```

2.4 Events

An event is a logical conception implemented with a very real data structure:

```
enum event_type {COUNTING,BINARY,GATE};
```

```
struct event
{
  struct tcb *fwd_link;
  struct tcb *back_link;
  int count;
  enum event_type type;
};
```

The forward and backward links provide the means for keeping a double linked list in priority order. The count keeps track of the event's current state:

Count	Binary/counting type event	Gate type event
0	Event has not occurred; no task waiting	Gate is closed

N	Event has occurred N time (limited to 1 for binary)	Gate is open (N must be 1)
-N	N task are waiting	NA

The event specific functions provide the ability to initialize and manipulate events:

INITIALIZE AND EVENT

Description: Initializes and event for subsequent use.

Passed parameters: evnt - pointer to event
 count - initial event count
 type - event type

Returned value: none

void EventInit(**struct** event *evnt,**int** count,**enum** event_type type)

WAIT FOR EVENT OCCURANCE

Description: Allows a task to wait for an event.

Passed parameters: evnt - pointer to event

Returned value: none

void WaitEvent(**struct** event *evnt)

WAIT FOR TIME

Description: Allows a task to wait for time.

Passed parameters: wtime - number of ticks to wait

Returned value: none

void WaitTime(**unsigned** wtime)

WAIT FOR AN EVENT WITH TIMEOUT

Description: Provides the ability for a task to wait on an event and time.

Passed parameters: evnt - pointer to event
 wtime - time to wait

Returned value: event occured (true) or not (false)

BOOL WaitEventTime(**struct** event *evnt,**unsigned** wtime)

SIGNAL AN EVENT OCCURANCE

Description: Allows a task or AIH to indicate that an event has occured.

Passed parameters: evnt - pointer to event

Returned value: none

void SignalEvent(**struct** event *evnt)

EVENT'S STATUS

Description: Returns the indicated event's count.

Passed parameters: evnt - pointer to event

Returned value: event's count

```
int EventStatus(struct event *evnt)
```

CLEAR AN EVENT

Description: Sets the event to not having occurred.

Passed parameters: evnt - pointer to event

Returned value: none

```
void ClearEvent(struct event *evnt)
```

GET SYSTEM TIME

Description: Returns the current system time.

Passed parameters: none

Returned value: number ticks since system start

```
unsigned long SysTime()
```

2.5 Drivers

A "driver" is the combination of interrupt handlers and interface functions that provide access to some underlying hardware or other resource that the user desires to encapsulate. THE Executive provides access to some hardware, interrupt control and the system timer, but in general the application software engineer needs to provide most of the access. An example serial port driver is provided in the reference implementation. It makes use of an application interrupt handler and application interface functions (that use events and character queues provided in another module) to provide the other parts of the application with a character oriented interface to the serial port. In general the construction of a driver is not formally specified. However, if you want the driver to be stated as part of kernel initialization it must be part of the public table "drivers":

```
struct driver_table_entry
{
    void (*InitRoutine)(void);
};

struct driver_table_entry drivers[] =
{
    SerialInit,
    0
};
```

2.6 Configuration

Beyond the tasks and drivers tables, the application developer must set a number of constants used to size the underlying

kernel data structure. These are summarized in the following table:

File	Constant
User.h	<p>NO_APPL_TASK - the total number of application tasks</p> <p>INIT_TASK - the task ID (the index in the tasks table) for the applications initialization task that is started after kernel initialization</p>
Kernel.h	<p>KERNEL_SPACE - the number of words (two bytes) reserved for the dynamic construction of task control blocks and the kernel's own stack. This must be greater than: $((\text{NO_APPL_TASK} \times \text{size of TCB}) + \text{sum of task's stack sizes} + \text{IDLE_STACK_SIZE} + \text{MIN_KERNEL_STACK})/2$</p>

3 DESIGN

The design of the THE Executive is based around a couple of key data structures and the code that operate on them. The header file kernel.h provides the definition of the data structures and internal functions. Each of the principal items will be examined in the following sub-sections.

3.1 Task Control Block

The task control block is the data structure that allows the executive to start and stop tasks. The structure itself is:

```
struct tcb
{
    struct tcb *fwd_link;
    struct tcb *back_link;
    struct tcb *tfwd_link;
    struct tcb *tback_link;
    unsigned char status;
    unsigned char priority;
    unsigned char taskid;
    unsigned char wake_source;
    unsigned long wake_time;
    unsigned int *ssp;
};
```

The first set of forward and backward links provide the means to insert a TCB in a priority ordered double linked list. The second set is used in time ordered double linked list. The status byte reflects the task's current state except for "ready". Limited knowledge of the prior state (whether the task was interrupted or not) is required to restore the task's context properly so when a task is inserted in the ready queue its prior status, interrupted, waiting etc., is preserved. The taskid byte hold's the task's ID (the task's index in the tasks table). The wake_source byte indicated whether a task was woken because of an event occurrence or because of a timeout (used with WaitEventTime). The wake_time is used when waiting for time (with WaitTime of WaitEventTime). It is set for the system tick count when the task should be woken. The ssp is the saved stack pointer. It preserves the task's stack pointer when the task is suspended.

3.2 Kernel

The kernel data structure, k, is nothing more then some pointers/linked list and the space to build task control blocks (including the space for the task's stack). It is defined as:

```
struct node
```

```

    {
    struct tcb *fwd_link;
    struct tcb *back_link;
    };

struct kernq
{
    struct tcb *run;
    struct node ready;
    struct tcb *delayq;
};

struct kernel
{
    struct kernq q;
    struct tcb *tb[NO_APPL_TASK + 1];
    unsigned short int kstack[KERNEL_SPACE];
};

extern struct kernel k;

```

The run pointer points to the TCB that is currently executing. The ready node is the head for the priority ordered double linked list used to hold task's that are ready to run but are not the highest priority. The delayq pointer is the forward link for the time ordered double linked list used to hold task's waiting for time. The kstack area is used to build the TCB for each task, including the task's header and actual stack area.

The executive is nothing more than a state machine that keeps track of tasks using its pointers/lists or the list associated with a event. When a task is suspended it's TCB is placed in the appropriate list and it's context is saved. When the current running task is suspended the highest priority task in the ready list is removed from the list and allowed to execute by restoring its context.

3.3 Save Context

A task's context is composed of the CPU general registers and local storage (i.e. its stack). In block structured languages such as C, the stack includes most of the information that defines "scope" and "local variables" visible in that scope. In addition it includes return addresses used to return from function calls etc. THE Executive only saves a context if it has to (i.e. on interruption, when a task must wait etc.). It always occurs (except with interrupts covered latter) with a call to Block() in suspend.c. The generic operation is diagramed in Figure 2. This preserves the task in a manner which causes it to resume execution at the return address from the call to Block() when its context is restored. Actual code

execution is continued in the Block routine by switching to the kernel's stack area. The last statement in Block is a restoration of the context of the task currently pointed to by run.

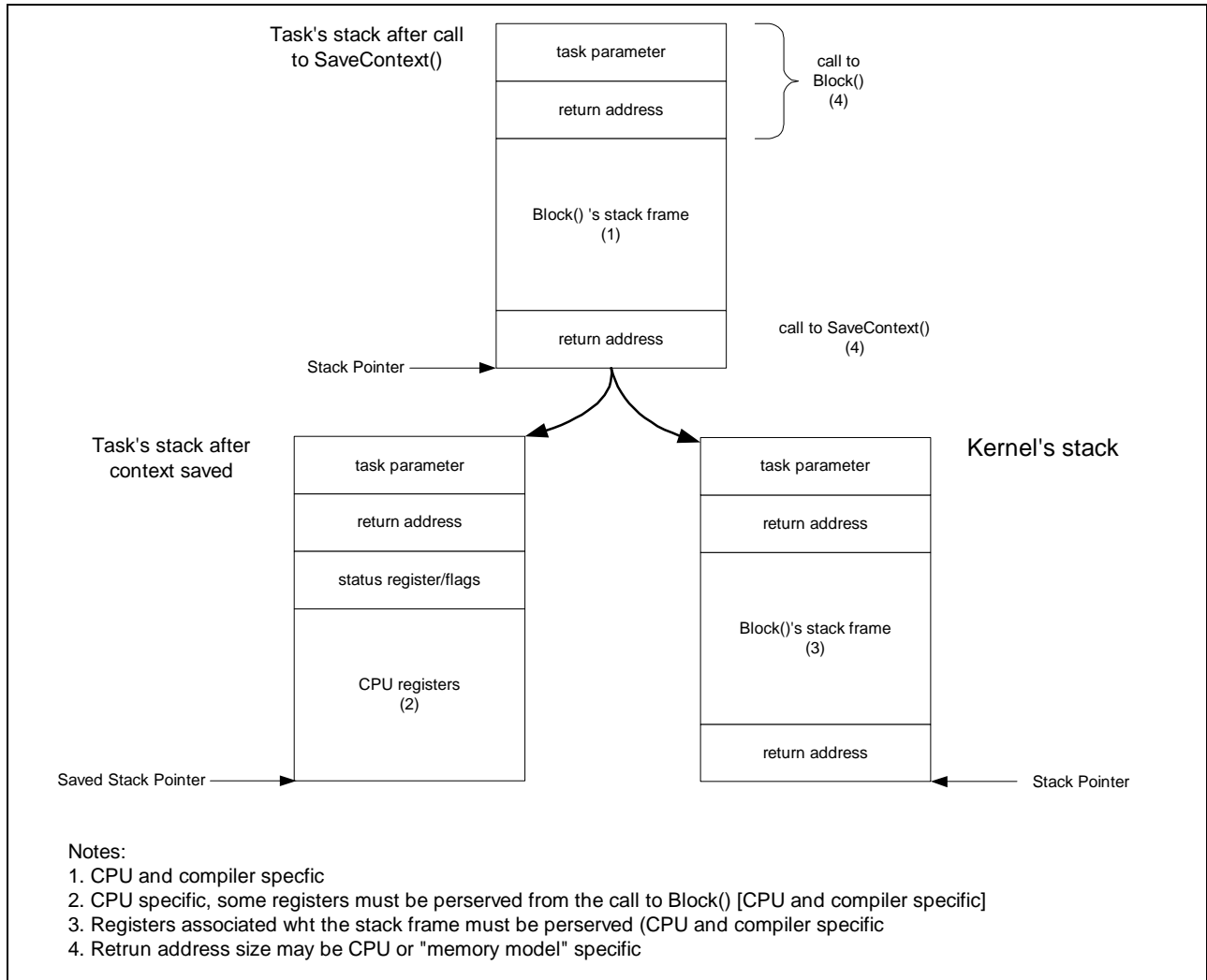


Figure 2, Save context

3.4 Base Interrupt Handler

THE Executive provides the lowest level interrupt handler. This guarantees that all interrupts are handled correctly. The execution in pseduo code is:

```

Save CPU registers
IF (interrupt depth is 0) THEN
    Check task for stack overflow
    IF (stack overflow) THEN
        KernelError

```

```

        ELSE
            Set task status to interrupted
            Switch to kernel stack
        ENDIF
    ENDIF
Increment interrupt depth
IntrpHandle()
Decrement interrupt depth
IF (interrupt depth is 0) THEN
    RestoreContext()
ELSE
    Restore from interrupt
ENDIF

```

The function `IntrpHandle` uses a table of application interrupt handlers to call the correct handler. The table is defined as:

```

struct intrp_entry
{
    void (*aih) (void);
};

extern struct intrp_entry intrp_table[NO_INTRP];

```

If no handler is available then `KernelError` is invoked.

3.5 Restore Context

Restoring a task's context is straight forward. The saved stack pointer is restored, the CPU registers are popped, and then either the status register/flags is popped and a return executed or a return from interrupt is executed, depending on the task's prior status (interrupted or not).

APPENDIX A, FILE SET

The reference implementation of THE Executive includes core files, support files and application user files. The files that make up the core portion of the executive are listed in the following table:

File	Description	External dependency
Bool.h	Boolean definitions	
Kernel.h	Non-dependent internal definitions	
Hdwe.h	Dependent internal definitions	Board hardware
Aih.c	Application interrupt handling functions	
Event.c	Event functions	
Intrp.c	Interrupt control functions	CPU & board hardware
Kernel.c	Space for kernel data	
Queue.c	Double linked list queue functions	
Support.asm	Executive functions that have to be implemented in assembly	CPU & compiler
Suspend.c	Task suspension functions	
Sys.c	Non-dependent system functions	
Syssupport.c	Dependent system functions	CPU & board hardware
Task.c	Task functions	
Timer.c	System timer driver	Board hardware

The files that provide support for the reference implementation are listed in the following table:

File	Description	External dependency
Cqueue.h	Definitions to use character FIFO queues	
Portio.h	Port based I/O function definitions	CPU
serial.h	Serial driver public definitions	
Cqueue.c	FIFO character queue implementation	
Portio.c	Port based I/O	CPU

	implementation	
Serial.c	Serial port driver	Board hardware

The application user required files are:

File	Description	External dependency
Exec.h	Definition of public executive data structures and functions	
User.h	Definition of application set constants	

APPENDIX B, PORTING CONSIDERATIONS

As indicated in Appendix A, a number of the files that make up THE Executive are CPU, compiler or board hardware specific. Most of these are relatively straight forward to port. Support.asm includes these routines that must be implemented in assembly. This is also relatively straight forward except for the function SaveContext. The "splitting" of a task's stack during the saving of a context is both CPU and compiler specific. The reference implementation used:

1. CPU - x86, real mode
2. Memory model - small
3. Assembler - Microsoft 7.10 (Visual Studio .NET 2003)
4. Compiler - Microsoft C/C++ version 8.0C (Microsoft Visual C++ version 1.52c) restricted to 8086 instructions

In this case the specific split is diagram in Figure 3. After making a port the generic test files provide a relatively quick means to check that the port is "stable" (see Appendix C).

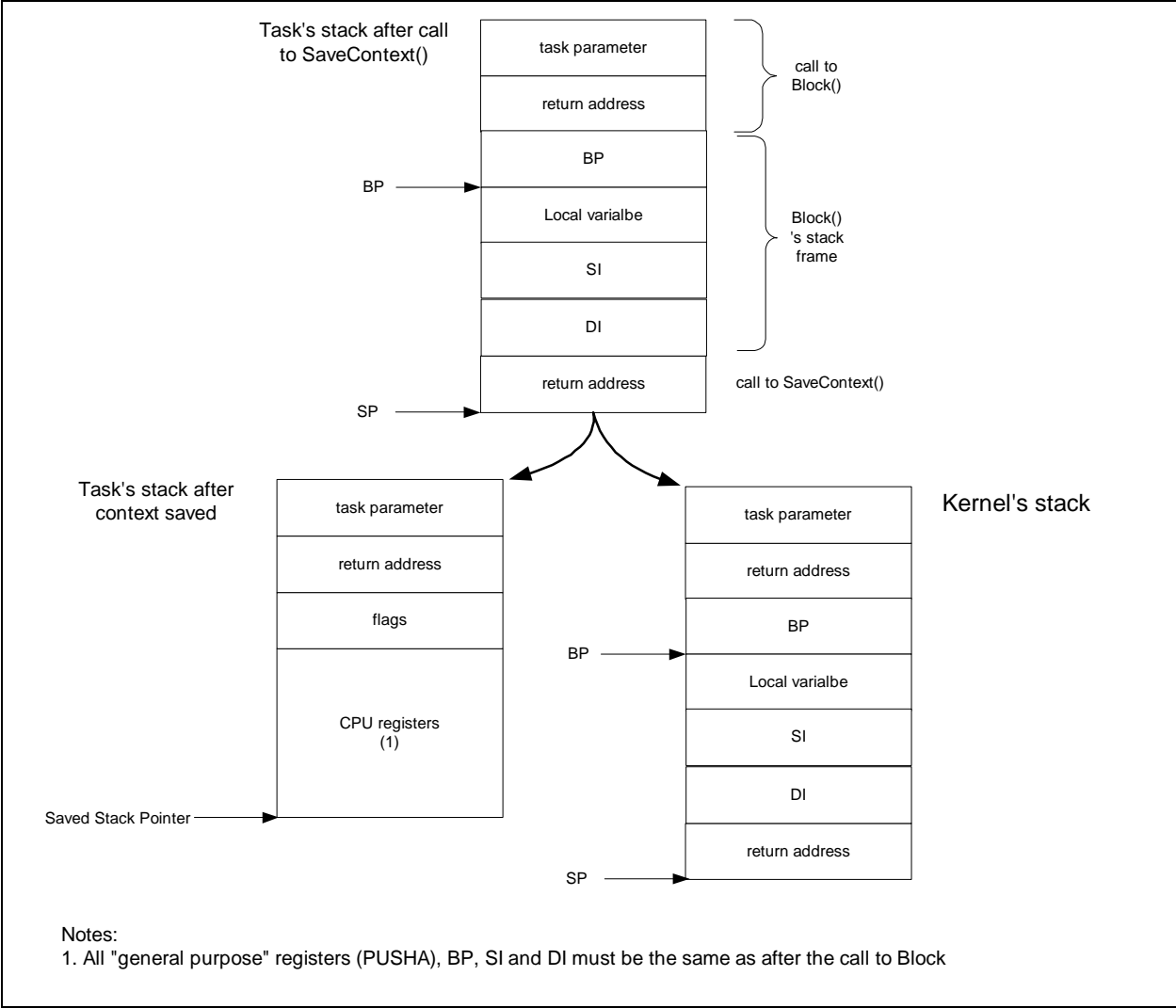


Figure 3, Reference implementation save context stack split

APPENDIX C, GENERIC TEST SET

The reference implementation includes five test files used to check for the correct generic operation of THE Executive (NOTE: In addition usage profile testing and error insertion has been used). The functions tested by these files is summarized below:

Functionality Used	1	2	3	4	5
Kinit	X	X	X	X	X
StartTask	X	X	X	X	X
StopTask	X	X	X	X	X
StartTask, StopTask, ...		X			X
TaskStatus	X				
TaskId		X			X
WaitTime	X	X	X	X	X
WaitEvent/SignalEvent, binary, task signal	X				
WaitEventTime/SignalEvent, binary, task signal			X		
WaitEvent/SignalEvent, counting, AIH signal	X	X	X	X	X
WaitEventTime/SignalEvent, counting, task signal				X	
WaitEvent/SignalEvent/ClearEvent, gate, task signal/clear		X			
WaitEventTime/SignalEvent/ClearEvent, gate, task signal/clear					X
WaitEventTime, binary, timeout			X		
WaitEventTime, counting, timeout				X	
WaitEventTime, gate, timeout					X
EvenStatus	X	X		X	
SysTime	X				
Lock	X	X	X	X	X
Restore	X	X	X	X	X
EnableInterrupts	X	X	X	X	X
DisableInterrupts	X	X	X	X	X
SetAIH	X	X	X	X	X

The tests use a serial port, COM1 configured at 57.6 kbps, no parity, 8 bits/character, 1 stop bit and no flow control, to provide basic user I/O. The implementation of the serial "driver" provides a good example of concurrent programming using THE Executive. The key points are:

1. All the key data structures and code are encapsulated in a single module (What was known in the concurrent programming world of the late 50's as a monitor, now commonly called an object).
2. The task side interface functions use Lock and Restore for mutual exclusion (from other tasks and the serial AIH) and counting style events for resource (characters) access.
3. The serial AIH disables the serial interrupt during processing for mutual exclusion from another serial interrupt (NOTE: The interrupt controller is signaled with the end of interrupt and interrupts are enabled prior to calling an AIH so it is possible depending on the UART). It only signal's events if never waits on them. Interrupt handlers can not be stopped and restarted by the executive.

APPENDIX D, KERNEL ERRORS

THE Executive traps errors that are unrecoverable and calls the function `KernelError`. In the reference implementation this disables interrupts and attempts to enter a debugger (if active). Upon return from the debugger it executes `Shutdown`. The errors are:

Error	Description
Kernel dump	Used for debugging
Bad run pointer	The task pointer set to be run is illegal (i.e. does not point to a task). The pointer is saved in the global variable <code>error_task_ptr</code> .
Unhandled interrupt	A physically received interrupt does not have an associated application interrupt handler.
Kernel stack too small	The size of the kernel stack after dynamically building the kernel data structure is too small.
Task stack overflow	During a context save the running task's stack area was found to have overflowed.