Akalon RTOS

Manual

(Revision 2014-1013)

Table of Contents

1 Introduction

1.1 Features

Akalon is an Open Source, Embedded Real-Time Operating System (RTOS) consisting of...

- A Micro Kernel that supports...
    - Pre-Emptive Multitasking based on Task Priorities or Round Robin scheduling
    - IPC Facilities (Messages, Semaphores)
    - Memory Management (Akalon supports a flat memory space)
    - Timers
    - Interrupt Handling
- Device Drivers
- Support Modules
- Board Support Packages (BSPs) that "glue" the Micro-Kernel, Device Drivers and Support Modules for specific Boards/Systems.
- Command Line Interface that provides the ability to call almost any C function.

Furthermore, the Akalon RTOS provides...

- Real-Time Performance.
- Small Footprint.
- Modularized design for easy porting into new Boards/Systems and Microprocessors.
- License based on the non-restrictive BSD License.
- Well commented source code written in ANSI C with a bit of assembler.

The Akalon RTOS also adheres to the follows set of *Guiding Principles*:

- To be Open Sourced (always).
- To be built with Open Source Tools.
- To be Simple and Flexible in it's Design and Implementation.

The Akalon RTOS can be installed on a standard PC thus converting a PC into a Real-Time Embedded System. This avoids the need for special embedded system hardware.

1.2 Nomenclature

Embedded System:	A Computing System built for a specific Task.
RTOS:	Real-Time Operating System. An Operating System that can respond to events in a timely and predictive manner.
Kernel:	Heart of an Operating System.
Device Driver:	Device Specific Software.
Module:	In Akalon, this is a Component that is not part of the Kernel nor is a Device Driver.
Application:	Task specific Software written by the User.
BSP:	Board Support Package. Support software that integrates an OS, Drivers, Modules, Applications, etc on a ***Target.***
Host:	System that contains the Source code, Tools, etc. used in developing an ***Embedded System.***
Target:	System that executes the RTOS, Device Drivers, Modules, Applications, etc. This is also the ***Embedded System.***
SBC:	Single Board Computer.
Cross Development Tools:	Microprocessor specific Tools that resides on the ***Host.***

1.3 Documentation Organization

The intention is to contain all Akalon related information in one document. Hence, the following guides are included in this Manual:

Application Developer Guide:	For developing Akalon Applications
Prerequisites:	Good understanding of Embedded Systems
BSP Developer Guide:	For porting Akalon to different Systems/Boards.
Prerequisites:	Application Developer Guide and a good understanding of the overall System hardware.
Driver Developer Guide:	For developing Akalon Device Drivers
Prerequisites:	Application Developer Guide BSP Developer Guide
Processor Porting Guide:	For porting Akalon to a different processor
Prerequisites:	Application Developer Guide BSP Developer Guide Driver Developer Guide Good understanding of the Microprocessor.

However, due to the dynamic nature of the following Guides, they are separated from this Manual and resides in the `docs` sub-directory.

BSP-xxx.txt:	Information on BSP where xxx is the specific BSP.
Modules.txt:	Information on all Modules.
Devices.txt:	Information on all Device Drivers.
Processors.txt:	Information on supported Microprocessors.

1.4 Getting Started

1.4.1 Prerequisites

- A Host Development Platform (Host PC) running a Unix variant (ex. Linux, BSD).
- The GCC Cross Compiler and the binutils tool set for the Target Microprocessor.
- A Target System. Akalon can use a Standard PC as a Target System.

1.4.2 Download and Unpacking Source

- The source code for Akalon can be downloaded from http://sourceforge.net/projects/akalon

- The tar file will have the format `akalon-YYYY-MMDD.tar`

- When the above file is un-tar'd (by `tar -xvf akalon-YYYY-MMDD.tar`), this will create a directory `<>/akalon/YYYY-MMDD`. When future versions of the source code is unpacked, a different version directory `YYYY-MMDD` is created.

- In this document, Akalon sub-directories are prefixed with `<base>` which implies a generic directory `<>/akalon/VYYYY-MMDD`

- Akalon's version number is of the format `YYYY-MMDD` and the associated string can be found in file `<>/akalon/<version>/inc/akalon.h`

```
#define AKALON_VERSION "YYYY-MMDD"
```

1.4.3 Source Tree

Note: In this document, Akalon sub-directories are prefixed with `<base>` which implies a generic directory `<>/akalon/YYYY-MMDD`

`<base>/bsp`	:	BSP specific source code
`<base>/bsp/XXX/bld`	:	Default BSP Build Location (contains binaries).
`<base>/cpu`	:	CPU specific source code
`<base>/devices`	:	Device Driver source code
`<base>/docs`	:	Documentation
`<base>/inc`	:	Includes files
`<base>/kernel`	:	Kernel source code
`<base>/modules`	:	Module source code
`<base>/pre_built`	:	Pre-built BSP images for an Intel PC
`<base>/tools`	:	Tools source code

1.4.4 Compiling and Installing a BSP

- Most BSPs are expected to follow the build method below, but since each BSP is unique, please refer to the BSP specific documentation located in the `<base>/docs` subdirectory for more information. The following example assumes the Akalon BSP for a standard Intel PC.

    - Read the BSP specific information file in `<base>/docs/BSP-pc.txt`

    - If needed, fine-tune the `<base>/bsp/XXX/Makefile` to match the host toolset (i.e. Compiler, Assembler, Linker, Archiver, etc.). Again, refer to the `<base>/docs/BSP-pc.txt` documentation for details.

    - Change to the BSP directory...

      ```
      $ cd <base>/bsp/pc
      ```

      By default, the BSP is built in the directory `<base>/bsp/XXX/tmp.` But this could be changed in the `<base>/bsp/XXX/Makefile`

    - Build the BSP...

      ```
      $ make clean all
      ```

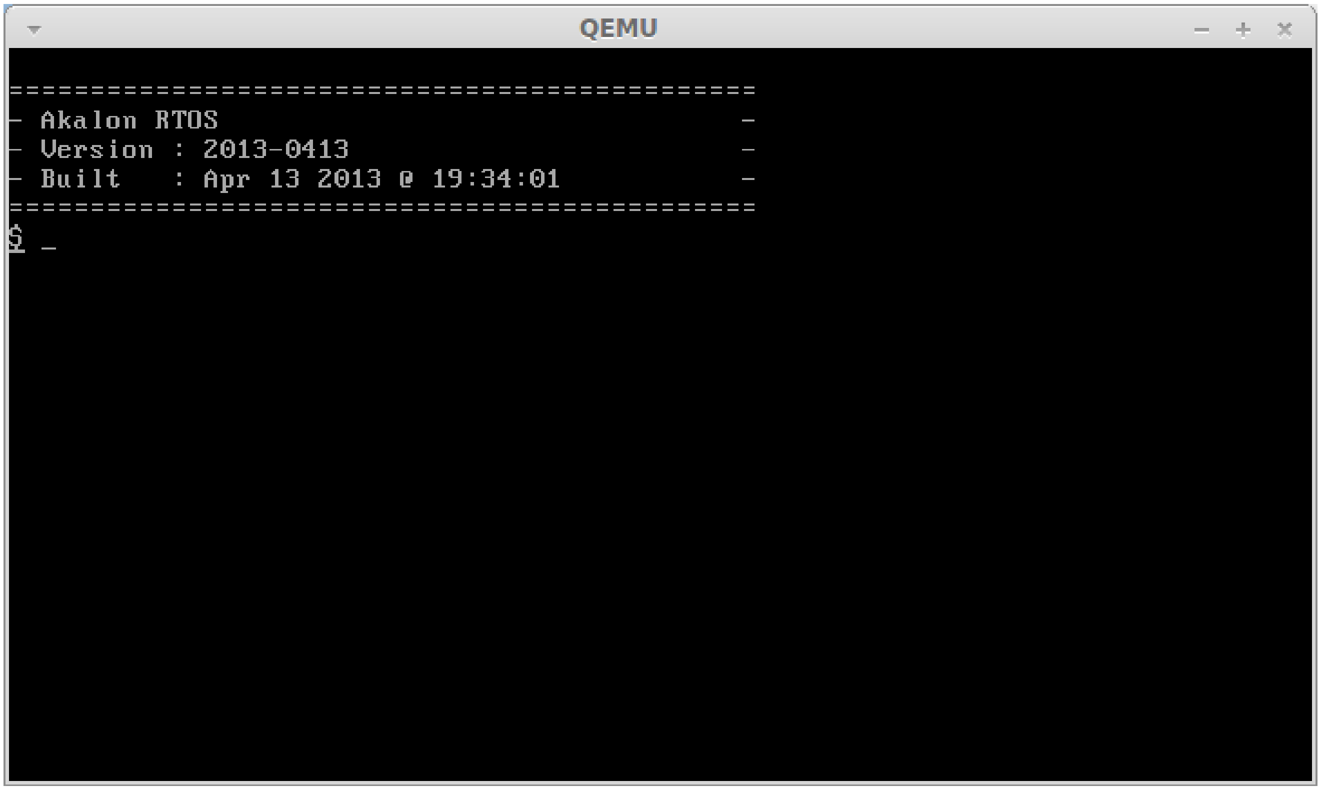
    - When the build is completed, the library, objects and executables will be located in the `<base>/bsp/XXX/bld` sub-directory. The contents of this sub-directory differ from one BSP to another, but at a minimum, it should contain the following files...

        - `akalon-<BSP>-<CPU>.bin` : Akalon RTOS executable.
        - `akalon-<BSP>-<CPU>.o` : Akalon RTOS object file.
        - `libakalon-<ARCH>-<CPU>.a` : Akalon Library.

- Loading the BSP into the Target System is the responsibility of the Boot-Loader. Refer to the BSP specific info file `<base>/docs/BSP-pc.txt` on how this can be accomplished.

- If a BSP doesn't exist for a particular Target System, modify a similar existing BSP or write a BSP from scratch (Refer to Section *4. BSP Developer's Guide* on how to write a BSP from scratch).

1.4.5 Command Line Interface (CLI)



- The Akalon RTOS provides a Command Line Interface (CLI) module which can be used as an interface.

- One of the main benefits of Akalon's CLI is the ability to call almost any C function that is compiled into the system. This is analogous to the Shell functionality provided by some commercial RTOSs.

- The CLI is very flexible in it's design and gives the ability to configure it's Input and Output sources which could be 2 different entities. For example, the Input can be from a Keyboard while the Output can be to a Monitor or both Input and Outputs can be via the Serial Port.

- The CLI is normally configured by the BSP. The source code for the CLI is in module `cli`

2 Application Developer Guide

2.1 Overview

- An Akalon RTOS Application Developer interfaces to 3 main Akalon components: *The Kernel*, *Device Drivers* and *Modules*.

- The interface to the Kernel is through standard C function calls, but interfacing to Modules and Device Drivers requires the Stack Link Interface described later.

2.1.1 Coding Standard

Comments:

Rule: Use a `/* */` sequence on every line

Ex:
```
/* This is a Comment. */
/* So is this line.   */
```

Header File Starting/Terminating:

Rule: Use `#ifndef - #define - #endif` to avoid circular dependencies.

Ex:
```
#ifndef    SYSTEM_H  /* Name of the .h file */
#define    SYSTEM_H

 .... File Contents

#endif /* ! SYSTEM_H */
```

Basic Types:
```
u8    /* Unsigned 8 Bits  */
s8    /* Signed 8 Bits    */

u16   /* Unsigned 16 Bits */
s16   /* Signed 16 Bits   */

u32   /* Unsigned 32 Bits */
s32   /* Signed 32 Bits   */

u64   /* Unsigned 64 Bits */
s64   /* Signed 64 Bits   */

usys  /* Processor's unsigned native type */
ssys  /* Processor's signed native tpe    */
```

Note: Basic types are all defined in the processors specific include file (ex. `ia32.h`, `arm.h`)
For portability, use types that are `usys` and `ssys`

Name Format:

Rule: Use "_" to separate words.
Ex: `a_variable`

Rule: General naming format *<module><noun><verb>*
Ex: `pci_info_get`

Rule: For structures, end name with a "`_t`"
Ex: `sem_t`

Function Calls:

Rule: Always returns status. If data needs to be returned, this is done with a caller instantiated input parameter.

Ex: `usys   get_data (usys port, usys size, u8 *data) ;`

Function Return Codes:

```
#define  GOOD          0   /* Return Was Good                  */
#define  BAD           1   /* Error Generic Error              */
#define  E_VALUE       2   /* Error due to Value (ex. 1,2,3)    */
#define  E_ITEM        3   /* Error due to Item (ex. DONT_WAIT) */
#define  E_TIME        4   /* Error due to a Time Out           */
#define  E_INIT        5   /* Error due to Not Initialized      */
#define  E_ID          6   /* Error due to Invalid ID           */
#define  E_MEM         7   /* Error due to lack of Memory       */
#define  E_RES         8   /* Error due to the lack of Resource */
```

2.2 Kernel Interface

- Akalon RTOS contains a Micro kernel that provides the basic building blocks needed to build any Embedded System (the kernel interface consists of less that 2 dozen function calls).

- These Building Blocks include Tasks, Semaphores, Message Boxes, Timers, Device Management and Interrupt Handling facilities.

2.2.1 Tasks

- In Akalon, Tasks are the basic unit of execution.

- Each Task can be in 3 states: *Executing*, *Ready* and *Waiting*. All Tasks start in the *Ready* State

- All Tasks are pre-emptible except the interrupt task (see below).

- Tasks have a priority assigned at Task creation. The highest priority is 1, while the lowest priority is the max natural value of the processor (ex. On a 32-bit system, it's `0xffffffff`)

- Two Tasks can have the same priority. In this case, Round-Robin scheduling will me used in scheduling.

- The Kernel creates two tasks: The *Kernel* Task and the *Interrupt* Task. The Kernel Task runs when no Tasks are in a Ready State and is pre-empted when ever another task becomes Read. The Interrupt Task is executed when servicing and ISR. Thus, all ISR's are executed under the Interrupt Task's context. The Interrupt Task cannot be preempted.

- To Create a Task, call `task_new()`
- To Delete a Task, call `task_del()`
- To Delay the currently running Task, call `task_delay()`
- To put a Task to Sleep indefinitely, call `task_sleep()`
- To Wake a Task, call `task_wake()`
- To get information on a task, call `task_info()`

2.2.2 Semaphores

- Fastest method for communicating between Tasks.

- Akalon doesn't differentiate between *Binary* and *Counting* Semaphores since a Binary Semaphore can easily be created by setting the maximum count to 1.
-
- Create a Semaphore by calling `sem_new()`.
- Give/Post a Semaphore by calling `sem_give()`
- Get a Semaphore by calling `sem_get()`

2.2.3 Messages

- Messages are sent to a Message Box associated with a Task. Thus, when a Task is created, a Message Box too is created.

- When sending a Message, the Message Box address is the associated Task ID.

- Akalon requires the maximum number of messages and the maximum size of a message at Task creation time. If the Task is never to receive a message, these values can be set to zero.

- Message Box's are created when a task is created by `task_new()`
- Messages are received by calling `msg_get()`
- Messages are sent by calling `msg_send()`

2.2.4 Timers

- Timers are of two Types: *One-shot* and *Repeat* timers. The Difference is that when a Timer expires, a *One-shot* Timer needs to get restarted while the *Repeat* timer doesn't.

- When a Timer expires, the corresponding notify function will be called from an interrupt context.

- Create a Timer by calling `timer_new()`
- Delete a Timer by calling `timer_delete()`
- Start a Timer by calling `timer_start()`
- Stop a Timer by calling `timer_stop()`
- Get a Timer's information by calling `timer_info()`

2.2.5 Interrupts

- Most Microprocessors start with interrupts disabled. Akalon enables interrupts after initializing the kernel and just before starting multitasking.

- When an interrupt occurs, the kernel saves the currently running task and switches to the interrupt task. If an ISR is registered, it will execute this routine under the interrupt task.

- Calls that do not block (ex. `sem_give()`, `msg_send()`) can be called from ISRs
- Calls that do block because of a wait for a resource can also be called from an ISR if the resource is available. However, doing this is just DUMB !!!

- An ISR can be connected to an interrupt with `int_config()`

- Enabling/Disabling the interrupt is done by the user.

2.2.6 Memory

- Akalon supports the `malloc()` call, but currently doesn't support the `free()` call. To overcome this limitation, allocate memory statically.

2.2.7 General

- The Akalon OS is initialized with the `os_init()` call. This is generally called from the BSP.

- To pause the Akalon OS (needed when executing atomic/protected code segments), use the function `os_pause()` which will disable multitasking and interrupts. However, make sure to restart the Akalon OS.

- The Akalon OS can be re-started with the `os_restart()` call. Make sure not to call the function from an interrupt context.

2.3 Stack-Links

- Akalon Modules and Device Drivers are completely autonomous entities and need to interface to other entities, besides the kernel. This is achieved via the Stack-Link Interface.

- The Stack-Link interface defines a standard interface for a Module to link to other Modules above it or to Modules or Device Drivers below it. This same interface provides Device Drivers the ability to interface to Modules above it.

- The interface consists of Data Transmit, Receive and Configuration Functions for upper and lower entities and are connected via the `os_link()` function call.

- Normally, the BSP will call the `os_link()` function to correctly connect existing Modules and Drivers since it (the BSP) has the responsibility for providing the Glue between the Kernel, Device Drivers and Modules. But an application too can also use this function to connect entities that conform the the Slack-Link Interface.

- The `os_link()` function is flexible enough to “stack” a different entity for inputs and outputs.

2.4 Application Integration

- The user written application needs to is compiled against the Akalon OS by interface files located in directory `<base>/inc` and all Akalon's interface files are located in this directory.
- The entire kernel interface is accessed by including file `<base>/inc/akalon.h`

- After compilation, the user written application needs to get linked in with the target BSP which already includes the Akalon Kernel and specific Drivers/Modules for the Target. This requires rebuilding the BSP to include the User Application.

- The user application Entry Point function can be included by...

    1. Calling the User Application Entry Point in function `bsp_post_init()` or
    2. Calling it from the Command Line.

- The Akalon Kernel, Devices, Modules are all included in the library file `<base>/tmp/libakalon-<ARCH>-<CPU>.a` except the BSP specific code.

3 BSP Developer Guide

3.1 Introduction

- A Board Support Package (BSP) is what “glues” the Kernel, Device Drivers, Modules, etc for a particular Board/System.

- Information on existing BSPs can be found in the `<base>/docs` sub-directory. Files in this sub-directory has the format `BSP-<bsp_name>.txt`

- Source code for existing BSPs can be found in sub-director `<base>/bsp/<bsp_name>`

- The purpose of the BSP Developer Guide is to provides information on how to create a Akalon BSP for a specific Board/System.

3.2 Prerequisites

- Good understanding of Target Hardware
- Akalon support for the Microprocessor
- GNU Build Tools
- Name of new BSP
- Good Understanding of Akalon's Initialization Time Line (next section)

3.3 Initialization Time-line

- After a system reset (including a power-on reset), a microprocessor will fetch its first instruction from a fixed location. Normally a ROM device resides at this location.

- The ROM device contain code to initializes the processor and set system components into a quiescent state. It then initializes the device that contains the Operating System and then loads it into RAM. Collectively, this component is known as the Boot ROM. To improve performance, some Boot ROMs move part of the Boot ROM code into RAM and execute it.

- An Akalon RTOS based system is no different from the above. First, the Boot ROM code initialize the processor and then proceeds to put system components into a quiescent state. Finally, it moves Akalon and the associated application from the Boot ROM into RAM and then calls the Akalon initializing function **`os_init()`**. It is important that all interrupts are disabled before calling **`os_init()`**.

- **`os_init()`** then initializes the kernel data structures and spawns the Idle Task (**`task_idle`**)with the highest priority.

- When the Idle Task starts executing, it will call the BSP's post initialization function **`bsp_post_init()`**. The purpose of this function is to initialize BSP components that require

Akalon resources while interrupts are disabled. It's in `bsp_post_init()` that an application can be created and initialized.

- The Idle Task will then lower it's priority to the lowest priority, get the highest priority task and kick off multitasking with interrupts enabled.

- When there is no tasks that are ready to be run, the Idle Task will run on a loop until another Task gets into a Ready State.

- All Akalon OS initialization routines are located in file `<base>/kernel/kernel.c`

3.4 Implementation

- Select a debug output port and implement a polling based output function that is non-interrupt driven and non-blocking. This function needs to have the same syntax as the standard character output function **`putchar()`** and should be named **`dbg_putchar()`**. Implementing this function will considerably ease the development of a Akalon based BSP since this gives the ability to call the standard output function **`printf()`** (Under-the-hood, **`printf()`** calls **`putchar()`** which in-turn calls **`dbg_putchar()`** until the kernel and the standard IO module is initialized).

- Create the new BSP sub-directory in the Akalon distribution

```
$ cd <base>/akalon/<version>
$ mkdir ./bsp/<bsp_name>
```

- Create file `<base>/bsp/<name>/asm.s`

  This assembly language file is to contain all assembly language routines for the BSP including the **`start()`** function which is the first code executed by the processor. Because **`start()`** is the first executed routine, it needs to get placed (by the linker) at the processor's start/reset address. Initially this function puts the processor into a quiescent state, then sets up the stack and transfers control to function **`bsp_pre_init()`** which is the first C routine to execute. If a boot-loader is used, **`start()`** can be linked to any address but the boot-loader has to load this code at that particular address.

- Create file `<base>/bsp/<name>/bsp.c`

  This file is to include the function **`bsp_pre_init`**() which is the first C code to get executed. Eventually, this function will call Akalon's kernel initialization routine `os_init()`

  Also included in this file is function **`bsp_post_init()`** which is called by the kernel's *Kernel Task.* The purpose of this function is to...

  - Initialize needed Device Drivers and Modules by calling their **`xxx_init()`** routines. This information is found in files `<>/docs/devices.txt, <>/docs/modules.txt,`

<>/inc/devices.h, and <>/inc/modules.h and

- And or information and initialization functions for all available Modules and Device Drivers). Also, Components relevant to the BSP. This include the Stack-Links used in connecting Modules and Device Driver combinations. The User Application Entry Function can also be initialized from this function. It's important to note that all interrupts are still disabled when this function is called and care should be taken not to enable interrupts in this routine.

- If the Command Line module is called in function **bsp_post_init()**, then the **func_tbl[]** needs to be implemented. This is done with <>/tools/func_gen.

- Create files <base>/bsp/<name>/Makefile and <base>/bsp/<name>/src_files

  The makefile is called from the build script <base>/build and needs to implement rules **all:** and **clean:** which builds and cleans the entire Akalon BSP (eventually, even the User Application needs to be linked into the BSP). The output of **all:** needs to create the following files...

  - akalon-<bsp>-<cpu>.bin : Fully linked Akalon RTOS executable
  - akalon-<bsp>-<cpu>.o : Fully linked Akalon RTOS object.

3.5 Completion

- Create the BSP information file <>/docs/BSP-<name>.txt and include the following information...

  - Description
  - Version
  - Author
  - Details
  - How to Build (including Tool-chain and version)
  - How to Load

- If you want Humanity to benefit from your work, please send the BSP so it can be incorporated into Akalon's next release.

4 Driver Developer Guide

4.1 Introduction

- Device Drivers are initialized by calling their `xxx_init()` functions which are all located in file `<base>/inc/devices.h` . This is normally done in the BSP in function **`bsp_post_init()`**
- The internals implementation of an Akalon Device Driver is mainly up to the Driver Developer.
- However, Device Drivers need to interface to upper Modules (eg. An Ethernet Device Driver needs to interface to the Network Stack) and this is accomplished with the Device Driver conforming to the Stack-Link Interface.

4.2 Stack-Link Interface

- A Stack-Link Interface is between an Upper and Lower Modules
- To Stack-Link, the Upper Module needs to implement a Lower Interface while a Lower Module needs to implement an Upper Interface.
- An Upper Module can have a Upper interface too if it's to be further linked as a Lower Module and the Lower Module too can have Lower interface if it's to be linked as an Upper Module. This provides the flexibility of "stacking" a module in-between two modules.
- An Module can be a Transmitter, a Receiver or both but if the Upper Module is a Transmitter, then the Lower Module needs to be a Receiver and vise-versa.
- The Upper Module communicates with the Lower Module over it's Lower Interfaces while the Lower Module interfaces to the Upper Module over it's Upper Interfaces.
- If a Module is to be a Transmitter, it can call an external function in another Module to send the data or it can provide an internal function where another Module can call it to take the data.
- Likewise, if a Module is a Receiver, it can call an external function in another module to get the data or it can provide an internal function where another Module can call to give the data.
- If the Module is a Transmitter, the Stack-Link variable (i.e. function pointer) it calls to send data to another Module has the format `xe_tx_func` (where `x = u` or `l` depending if the function is the upper or lower interface) and it's internal function that another Module can call to get the data has the format `xi_tx_func` (where `x = u` or `l` depending if the function is the upper or lower interface)
- If the Module is a Receiver, the Stack-Link variable (i.e. function pointer) it calls to get data from another Module has the format `xe_rx_func` (where `x = u` or `l` depending if the function is the

upper or lower interface) and it's internal function that another Module can call to send data has the format `xi_rx_func` (where `x = u` or `l` depending if the function is the upper or lower interface)

- An Internal or External Transmit or Receive functions (Upper and Lower Interfaces) can operate in Blocking or Polling Modes and the Mode is specified when the function call is made. Modules and Device Drivers need to handle both these modes.

- If a Module or a Device Driver is to conform to a Stack-Link interface, it needs to declare , allocate and initialize a `link_t` structure with it's upper/lower internal transmit/receive functions set. The `link_t` structure which has the following format.

```
typedef  struct    link_t
{
    /* Upper Layer - Internal */
    usys (*tx_ui)   (usys type, usys buf_size, u8 *buf, usys *ret_size) ;
    usys (*rx_ui)   (usys type, usys buf_size, u8 *buf) ;

    /* Upper Layer - External */
    usys (*tx_ue)   (usys type, usys buf_size, u8 *buf) ;
    usys (*rx_ue)   (usys type, usys buf_size, u8 *buf, usys *ret_size) ;

    /* Lower Layer - Internal */
    usys (*tx_li)   (usys type, usys buf_size, u8 *buf, usys *ret_size) ;
    usys (*rx_li)   (usys type, usys buf_size, u8 *buf) ;

    /* Lower Layer - External */
    usys (*tx_le)   (usys type, usys buf_ize, u8 *buf) ;
    usys (*rx_le)   (usys type, usys buf_size, u8 *buf, usys *ret_size) ;

} link_t ;
```

NOTE: Only the Internal function pointers needs to be initialized.

- The Stack-Link interface between modules get initialized with function call **`os_link()`**. When this function call returns, applicable external function calls are initialized and Modules or Device Drivers can call external functions to Transmit and/or Receive data. This function is normally called in the BSP after the Kernel is initialized and in function **`bsp_port_init()`**.

- A Device Driver should never “migrate” an interrupt to a Module (i.e. Call a external Transmit or Receive from an interrupt context)

- All Supported Module interfaces are documented in file `<base>/docs/Modules.txt` and is defined in file `<base>/docs/modules.h`

- Likewise, all supported Device Driver interfaces are documented in file `<base>/docs/Devices.txt` and is defined in file `<base>/docs/devices.h`

- Although the Stack-Link interface is initially difficult to understand, it's design is simple and obvious and is much easier to integrate Modules and Devices that other methods.

5 Processor Porting Guide

* Task Switching
* Task switching with interrupts
*

6 API

6.1 Kernel Interface

int_config

Syntax	:	`usys  int_config (usys int_num, usys status, void (*isr)()) ;`	
Description	:	Connect an ISR to an Interrupt.	
Input/s	:	`int_num`	Interrupt Number. This number is Processor/BSP dependent.
		`status`	Currently not implemented.
		`isr`	ISR entry point.
Return/s	:	Possible return codes are...	
		`GOOD`	`isr` was associated with the interrupt `int_num`
		`E_VALUE`	`int_num` is incorrect.
Definition File	:	`akalon.h`	
Notes	:	This Function can be called from any context and is non-blocking.	

msg_get

Syntax :

```
usys  msg_get (usys *src_id, usys *size, void *loc,
               usys time_out) ;
```

Description : Get a Message.

Input/s :

`src_id` Caller instantiated location to store Message sender's ID

`size` Caller instantiated location to store the size of the Message

`loc` Caller instantiated (i.e. allocated) area to store a message. Make sure this area is equal to (in Bytes) the biggest message the task can receive.

`time_out` Wait time for obtaining a message. Possible values are...

(1) Timeout in System Tics

(2) `DONT_WAIT` Returns immediately if no messages are available

(3) `WAIT_FOREVER` Wait until a message is available

Return/s : Possible return codes are...

`GOOD` Call was successful and a message was received. The following information is returned...

- `src_id` ID of message sender. If an interrupt context (ISR or a Timer) sent the message, the `src_id` will be zero.
- `size` Size of the message is bytes.
- `loc` Message

`E_RES` No Message. Called returned because `time_out` was set to `DONT_WAIT`

`E_TIME` Timeout occurred before a message was received.

Definition File : `akalon.h`

Notes : **WARNING:** This function might Block. Therefore, do not call it from an interrupt context.

msg_send

Syntax : `usys  msg_send (usys dst_id, usys size, void *loc) ;`

Description : Sends a Message.

Input/s :

`dst_id`	Task ID of the recipient
`size`	Size of the message in bytes. Make sure this value is equal or less than the message size the recipient's maximum message size.
`loc`	Pointer to the message.

Return/s : Possible return codes are...

`GOOD`	Call was successful and a message was sent. The message location `loc` can be reused since the message was copied to the receiver's message box.
`E_ID`	Invalid Receiver `id`
`E_VALUE`	Invalid message `size`. Maximum message size of recipient is smaller than `size`
`E_RES`	Recipient's message box is full.

Definition File : `akalon.h`

Notes : This function can be called from an interrupt context.

If the message destination is to a Message Box associated with a higher priority task, the calling task is suspended and the higher priority task is executed.

After the call returns, the caller is free to reuse the message location (i.e. `data.loc`) since the message was copied to the recipient's message box.

os_init

Syntax	:	`usys  os_init (usys mem_start, usys mem_size) ;`
Description	:	Initialize the Akalon OS.
Input/s	:	`mem_start` Start of Memory available to the Akalon OS (aka Akalon's Heap).
		`mem_size` Size of Akalon's Heap
Return/s	:	This function will never return unless there is an Error.
Definition File	:	`akalon.h`
Notes	:	This function will spawn the Kernel Task, Interrupt Tasks and will then call the BSP initialization routine `bsp_post_init()` and will then proceed to initialize interrupts and start multi-tasking.

os_link

Syntax :
```
usys  os_link (link_t *upper_link, link_t *low_rx_link,
               link_t *low_tx_link) ;
```

Description : Connect Lower Tx or Rx Modules/Devices to a Upper Module using the Stack-Link interface.

Input/s : `upper_link` Stack-Link interface of the Upper Module with it's Lower Internal interface initialized. This includes...

`li_tx` Upper Module's Lower Internal Transmit Function. If the Module doesn't support this this functionality, this value will be set to `NULL`

`li_rx` Upper Module's Lower Internal Receive Function. If the Module doesn't support this this functionality, this value will be set to `NULL`

`low_rx_link` Stack-Link interface of the lower Module which sends data to the Upper Module. It needs to have it's upper internal functions initialized. This includes

`ui_tx` Lower Module's Upper Internal Transmit Function. If the Module doesn't support this this functionality, this value will be set to `NULL`

`low_tx_link` Stack-Link interface of the lower Module which receives data from the Upper Module. It needs to have it's upper internal functions initialized. This includes

`ui_rx` Lower Module's Upper Internal Receive Function. If the Module doesn't support this this functionality, this value will be set to `NULL`

NOTE: A Module will have it's Internal (Upper and Lower) Transmit and Receive Functions already initialized in it's Stack-Link structure `xxx_link` which is available in file `<base>/inc/modules.h` and a Driver will have it's Upper Transmit and Receive functions already initialized in it's Stack-Link interface in file `<base>/inc/devices.h`

Return/s : Possible return codes are...

`GOOD` Call was successful and Stack-Links were established. Specifically, the following Stack-Link function pointers were initialized. Specifically...

In the `upper_link` structure...

`le_tx`	Points to Lower Module/Driver Upper Internal Receiver function which is `ui_rx`
`le_rx`	Points to Lower Module/Driver Upper Internal Transmit function which is `ui_tx`

In the `low_rx_link` structure...

`ue_tx`	Points to Upper Module's Lower Internal Receiver function which is `li_rx`

In the `low_tx_link` structure...

`ue_rx`	Points to Upper Module's Lower Internal Transmitter function which is `li_tx`

`BAD` `mod_link` was `NULL`

Definition File : `akalon.h`

Notes : This function conveniently links Modules/Devices to each if these Modules/Devices conform to the Stack-Link Format. This is normally done in the BSP in function `bsp_post_init()`

os_pause

Syntax : `void os_pause (void) ;`

Description : Stops the Akalon OS. This disables interrupts and multi-tasking.

Input/s : None.

Return/s : None.

Definition File : `akalon.h`

Notes : This function can be used when an atomic operation is needed. It's very important to restart the Akalon OS (ie by calling `os_restart()`) when the atomic operation is completed. Also, make sure to keep this duration to a minimum.

Calling this function from an interrupt context has not effect.

This function is non-blocking.

os_restart

Syntax	:	`void os_restart (void) ;`
Description	:	Re-start the Akalon OS.
Input/s	:	None.
Return/s	:	None. .
Definition File	:	`akalon.h`
Notes	:	**WARNING:** DO NOT call this function from an interrupt context since this function might enable interrupts and cause undesirable system behavior.
		This function is non-blocking.

sem_del

Syntax	:	`usys sem_del (usys id) ;`	
Description	:	Delete a Semaphore.	
Input/s	:	`id`	ID of the Semaphore.
Return/s	:	Possible return codes are...	
		`GOOD`	Call was successful. Semaphore is deleted.
		`E_ID`	Invalid Semaphore `ID`
Definition File	:	`akalon.h`	
Notes	:	**WARNING:**	This call is currently not implemented.

sem_get

Syntax	:	`usys sem_get (usys id, usys time_out) ;`	
Description	:	Get/Take/Pend a Semaphore.	
Input/s	:	`id`	ID of the Semaphore
		`time_out`	Wait time for obtaining Semaphore. Possible choices are... 1) time in system tics. 2) `DONT_WAIT` Do not wait if the Semaphore cannot be obtained. 3) `WAIT_FOREVER` Wait until the Semaphore is available.
Return/s	:	Possible return codes are...	
		`GOOD`	Call was successful. Semaphore received.
		`E_ID`	Invalid Semaphore `ID`
		`E_RES`	Semaphore is not available.
		`E_TIME`	Timeout.
Definition File	:	`akalon.h`	
Notes	:	**WARNING:**	This function might Block. Therefore, do not call it from an interrupt context.

sem_give

Syntax	:	`usys  sem_give (usys id) ;`	
Description	:	Gives/Posts a Semaphore.	
Input/s	:	`id`	ID of the Semaphore.
Return/s	:	Possible return codes are...	
		`GOOD`	Call was successful. Semaphore given/posted.
		`BAD`	Semaphore's count is already equal to `max_count`
		`E_ID`	Invalid Semaphore `ID`
Definition File	:	`akalon.h`	
Notes	:	This call can be called from any context.	

If a higher priority task is waiting for the Semaphore, the calling task is suspended and the higher priority task is executed.

sem_info

Syntax :
```
usys  sem_info (usys id, usys *count_max, usys *count_now,
                char *name) ;
```

Description : Provides information on a Semaphore.

Input/s :

`id`	ID of the Semaphore
`count_max`	Caller instantiated location to store the maximum count of the Semaphore
`count_now`	Caller instantiated location to store the current count of the Semaphore
`name`	Caller instantiated location to store the Semaphore's Name. Size of name (in bytes) cannot exceed definition `NAME_SIZE` (in file `akalon.h`)

Return/s : Possible return codes are...

`GOOD` Call was successful. The following information is returned...

`count_max`	Maximum Count of Semaphore. For Binary Semaphores, this value will be 1.
`count_now`	Semaphore's current count.
`name`	Semaphore's Name.

`E_ID` Invalid Semaphore ID

Definition File : `akalon.h`

Notes : This call can be called from any context and is non-blocking.

sem_new

Syntax	:	`usys sem_new (usys count_max, usys count_start, char *name, usys *id) ;`	
Description	:	Create a Semaphore.	
Input/s	:	`count_max`	Maximum Count of Semaphore. Set to 1 for a *Binary* Semaphore.
		`count_start`	Semaphore's starting count (Ex. for a full Binary Semaphore this value will be 1 while an empty binary semaphore, this value will be 0)
		`name`	Semaphore's Name. Size of name (in bytes) cannot exceed definition `NAME_SIZE` (in file `akalon.h`)
		`id`	Caller instantiated location to store the new Semaphore's ID
Return/s	:	Possible return codes are...	
		`GOOD`	Call was successful. The following information is returned...
			`id` ID of new Semaphore.
		`E_VALUE`	`count_start` is greater than `count_max`
		`E_MEM`	Insufficient Memory to complete Call.
Definition File	:	`akalon.h`	
Notes	:	**WARNING:**	This function might Block. Therefore, do not call it from an interrupt context.

task_delay

Syntax	:	`void task_delay (usys tics) ;`	
Description	:	Delay the Task.	
Input/s	:	`tics`	Number of system tics to delay Task. Valid values are 1 to the Processor's natural maximum.
Return/s	:	Possible return codes are...	
		`GOOD`	Call was successful and the Task was Deleted.
		`E_ID`	Incorrect Task `ID`.
Definition File	:	`akalon.h`	
Notes	:	**WARNING:**	This function cannot be called from an interrupt context (duh !!!)
		WARNING:	This function IS blocking. (duh !!!)
		WARNING:	Do not call this function with interrupts disabled.

task_del

Syntax	:	`usys  task_del (usys id) ;`	
Description	:	Delete a Task.	
Input/s	:	`id`	ID of Task to Delete.
Return/s	:	Possible return codes are...	
		`GOOD`	Call was successful and the Task was Deleted.
		`E_ID`	Invalid Task `id`.
Definition File	:	`akalon.h`	
Notes	:	**WARNING:**	This call is not implemented.

task_id_get

Syntax : `usys  task_id_get (void) ;`

Description : Return the Task ID of the caller.

Input/s : none

Return/s : Possible return are...

Task ID of the calling Task. If the caller is not associated with a Task, then Zero is returned.

Definition File : `akalon.h`

Notes : This function can be called from any context and is non-blocking.

task_info

Syntax :

```
usys  task_info (usys id, usys *priority, usys *time_slice,
                 usys *stack_size, usys *max_msgs,
                 usys *max_msg_size, char *name) ;
```

Description : Provides status/information on a Task.

Input/s :

`id`	ID of the Task
`priority`	Caller instantiated location to store the Task's priority
`time_slice`	Caller instantiated location to store the Task's time slice
`stack_size`	Caller instantiated location to store the Task's Stack size
`max_msgs`	Caller instantiated location to store the Task's maximum number of Messages.
`max_msg_size`	Caller instantiated location to store the Task's maximum message size
`name`	Caller instantiated location to store the Task's Name. Size of name (in bytes) cannot exceed definition `NAME_SIZE` (in file `akalon.h`)

Return/s : Possible return codes are...

`GOOD` Call was successful. The following information in the `task_t` structure will be filled out...

`stack_size`	Size of stack in bytes.
`time_slice`	Currently not implemented.
`max_msgs`	Maximum number of messages the Task can receive.
`max_msg_size`	Maximum size of a message in bytes.
`entry_point`	Hex address of Task Entry function.
`inputs[3]`	Three inputs for the task Entry function.
`name`	Task's Name.

`E_ID` Invalid Task `ID`

Definition File : `akalon.h`

Notes : This function can be called from any context and is non-blocking.

task_new

Syntax :

```
usys  task_new (usys priority, usys time_slice, usys stack_size,
                usys max_msgs, usys max_msg_size,
                void (*entry_func)(),
                usys arg0, usys arg1, usys arg2,
                char *name, usys *id) ;
```

Description : Create a Task.

Input/s :

`priority`	Priority of Task. Possible values are 1 to the Processor's natural maximum.
`time_slice`	Execution time (i.e time slice) in system tics. Only used when multiple Tasks are at the same priority. Currently not implemented.
`stack_size`	Size of stack in bytes.
`max_msgs`	Maximum number of messages the Task can receive.
`max_msg_size`	Maximum size of a message in bytes.
`entry_point`	Function pointer to the Task's Entry Function Point.
`arg<0..2>`	Inputs into the Task's Entry function.
`name`	Task's Name. Size of name (in bytes) cannot exceed definition `NAME_SIZE` (in file `akalon.h`)
`id`	Caller instantiated location to store the new Task's ID

Return/s : Possible return codes are...

`GOOD`	Call was successful and the Task was created. The following information is returned... `id` ID of the Task.
`BAD`	Cannot allocate and initialize the Task's Mailbox.
`E_MEM`	Insufficient Memory to complete Call.

Definition File : `akalon.h`

Notes : **WARNING:** This function might Block. Therefore, do not call it from an interrupt context.

If the newly created Task is at a higher priority (i.e. lower priority number) than the calling Task's priority, then the newly created task is executed before the call returns.

task_sleep

Syntax	:	`void task_sleep (void) ;`
Description	:	Put the calling Task to sleep.
Input/s	:	None.
Return/s	:	None.
Definition File	:	`akalon.h`
Notes	:	**WARNING:** This API call CANNOT be called from an Interrupt context.
		After the calling Task is sleeping, it can be woken by the API call `task_wake()`

task_wake

Syntax	:	`void task_wake (usys id) ;`
Description	:	Put the calling Task to sleep.
Input/s	:	`id` ID of task to be woken.
Return/s	:	Possible return codes are...
		`GOOD` Call was successful and the Task was woken.
		`E_ID` Invalid Task ID.
Definition File	:	`akalon.h`
Notes	:	This API call that CAN be made from an Interrupt Context (i.e. it can be called from an Interrupt Service Routing)
		If the task that is woken is at a higher priority (i.e. has a lower priority number) than the task that called this function, the higher priority task will be executed before this API call is returned.

timer_delete

Syntax :	`usys timer_delete (usys id) ;`	
Description :	Delete a Timer.	
Input/s :	`id`	ID of Timer to Delete.
Return/s :	Possible return codes are...	
	`GOOD`	Timer Deleted.
	`E_ID`	Invalid Timer ID.
Definition File :	`akalon.h`	
Notes :	**WARNING:**	This Function might Block. Therefore, do not call it from an interrupt context.

timer_info

Syntax :

```
usys  timer_info (usys id, usys *tics, usys *type, char *name,
                  usys *tics_now, usys *is_active) ;
```

Description : Provides information on a Timer.

Input/s :

`id`	ID of Timer.
`tics`	Caller instantiated location to store the Timer's Tics
`type`	Caller instantiated location to store the Timer Type
`name`	Caller instantiated location to store the Timer's Name. Size of name (in bytes) cannot exceed definition `NAME_SIZE` (in file `akalon.h`)
`tics_now`	Caller instantiated location to store the current Tics
`is_active`	Caller instantiated location to store the current state of the Timer

Return/s : Possible return codes are...

`GOOD` Call was successful. The following information is returned...

`tics`	Initial tic count.
`type`	Timer type. Possible options are... `TIMER_ONESHOT` = One shot Timer. This type requires a restart (with the `timer_start()` call) after the expiration of the timer. `TIMER_REPEAT` = Repeat Timer. This type doesn't require to be restarted after the expiration of the Timer.
`name`	Timer's Name.
`tics_now`	Current tic count.
`is_active`	Current state of Timer. Possible options are... `YES` = Timer is active `NO` = Timer is not-active

`E_ID` Invalid Timer ID.

Definition File	:	`akalon.h`
Notes	:	This Function can be called from any context and is non-blocking.

timer_new

Syntax : `usys timer_new (usys tics, usys type, void (*timer_func)(), char *name, usys *id) ;`

```
usys  timer_new (usys tics, usys type, void (*timer_func)(),
                 char *name, usys *id) ;
```

Description : Creates a Timer.

Input/s :

`tics` Timer interval in system tics.

`type` Timer Type. Possible options are...

- `TIMER_ONESHOT` One shot timer. This type requires a restart with API `timer_start()` after the expiration of the Timer.
- `TIMER_REPEAT` Repeat Timer. This type doesn't require a restart after the expiration of the Timer.

`timer_func` Function to be called when Timer expires.

WARNING: Function is called from an interrupt context. Therefore, the entry function SHOULD NOT call any functions that might block (ex. printf).

`name` Timer's Name. Size of name (in bytes) cannot exceed definition `NAME_SIZE` (in file `akalon.h`)

`id` Caller instantiated location to store the new Timer's ID

Return/s : Possible return codes are...

`GOOD` Timer Created. The following The following information is returned...

- `id` ID of new Timer.

`E_MEM` Insufficient Memory to complete Call.

Definition File : `akalon.h`

Notes : **WARNING:** This Function might Block. Therefore, do not call it from an interrupt context.

timer_start

Syntax	:	`usys timer_start (usys id) ;`
Description	:	Starts a Timer.
Input/s	:	`id` ID of Timer to be started.
Return/s	:	Possible return codes are...
		`GOOD` Timer Started.
		`E_ID` Invalid Timer ID.
Definition File	:	`akalon.h`
Notes	:	This Function can be called from any context and is non-blocking.

timer_stop

Syntax :	`usys timer_stop (usys id) ;`	
Description :	Stops a Timer.	
Input/s :	`id`	ID of Timer to be stopped.
Return/s :	Possible return codes are...	
	`GOOD`	Timer Stopped.
	`E_ID`	Invalid Timer ID.
Definition File :	`akalon.h`	
Notes :	This Function can be called from any context and is non-blocking.	
	This call resets the Timer but doesn't restart it (i.e. To restart, call `timer_start()`).	

timer_tic

Syntax : `void  timer_tic (void) ;`

Description : Provides the system tic to the Akalon OS.

Input/s : None.

Return/s : None.

Definition File : `akalon.h`

Notes : This Function is normally called by the BSP from an interrupt context.

7 Missing Information

- Index
- Processor