CODE TIME TECHNOLOGIES

# Abassi RTOS

## UART Support

## Disclaimer

Code Time Technologies Inc. provides this document “AS IS” without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.
Code Time Technologies Inc. does not warrant that the contents of this document will meet your requirements or that the document is error-free. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the document. Code Time Technologies Inc. may make improvements and/or changes in the product(s) and/or program(s) described in the document at any time. This document does not imply a commitment by Code Time Technologies Inc. to supply or make generally available the product(s) described herein.

# Table of Contents

# List of Figures

# List of Tables

# 1 Introduction

This document describes the UART driver available for Abassi[1] [R1] (including mAbassi [R2] and µAbassi [R3]). The standalone use of the UART driver is also described in here.

The UART driver is a key module used by the System Calls Layer as it handles the stdio I/O; for more information, refer to [R4].

## 1.1 Distribution Contents

The set of files supplied with this distribution are listed in Table 1-1 below:

**Table 1-1 Distribution**

| File Name | Description |
|---|---|
| `???_uart.h` | Include file for the UART driver (??? is target dependent) |
| `???_uart.c` | "C" file for the Abassi UART driver (??? is target dependent) |
| `???_uart_CMSIS.c` | "C" file for the CMSIS RTOS API UART driver (??? is target dependent and only available with the CMSIS API) |
| `SAL.h` | Include file for the standalone abstraction layer (supplied with standalone package only) |
| `SAL.c` | "C" file for the standalone abstraction layer (supplied with standalone package only) |
| `ISRhandler_???.s` | "ASM" add-on file for the standalone version only. It contains support for both the driver and the demo application. |

## 1.2 Limitations

The standalone version of the driver does not use nor support interrupts, mailboxes, or semaphores. All wait operations are performed using polling.

## 1.3 Features

The UART driver API is kept the same across all target platforms. Target specific extra functionality is not described in this document; refer to the code itself and embedded comments.

The UART driver can operate using exchange mailboxes or not using them. For example, if the UART device has large enough FIFOs, then it may be preferable to not use mailboxes. (Mailboxes are not available for µAbassi or the standalone version.) When using mailboxes, the blocking of the driver occurs when the transmit mailbox is full or the receive mailbox is empty. Mailboxes are filled (RX) and emptied (TX) by the driver though an interrupt handler.

When not using mailboxes, the driver can be set to either perform pure polling (with optional periodic sleep) or to block on a semaphore when no RX characters are available or when the TX FIFO is full. Polling can configured to use circular buffers, where the circular buffers are filled / emptied in the ISR handler therefore performing transfers in the "background".

[1] When Abassi is mentioned in this document, unless explicitly stated, it always means Abassi, mAbassi and µAbassi.

NOTE: The UART driver is designed for use in a multi-task environment and, as such, relies on a protection mechanism to provide exclusive access to the UART peripheral. If the UART driver is not used with mutex protection, there is a slight possibility for a lock-up or erroneous character exchange if two or more tasks concurrently use the driver. This restriction also includes the use of the driver inside an interrupt. Refer to the build options `UART_FULL_PROTECT` (Section 2.1.8) and `UART_SINGLE_MUTEX` (Section 2.1.10) for more information on the internal protection capabilities offered by the driver.

This UART driver has been optimized to be usable under as many conditions as possible. For example, here is a short list of where or when the UART driver can be used:

- Called within an interrupt
- Initialized and used before Abassi is even started
- Used with interrupts disabled
- Used with the Abassi logging feature

Basically, all possible operating conditions under which an application can operate using Abassi are handled. Whenever possible, task blocking (through semaphores, mailboxes, etc.) is used, if requested to wait, when waiting for room in the output UART / mailbox or awaiting the arrival of a new character / mailbox. Any operating condition that would render the task blocking problematic, such as possibly locking-up or crashing the application, will make the driver fall back to using pure polling. When mailboxes are used, polling occurs only once a transmit mailbox is full or a receive mailbox is empty. Using this fall back mechanism greatly increases the usability of the UART driver and makes it able to operate under all conditions, plus it removes most usage restrictions from the user.

IMPORTANT: If the UART driver is using interrupts (either semaphores or mailboxes), then on a multi-core platform it is strongly advisable to make sure the UART interrupt is targeted to all cores. If only one core is the target to the UART interrupt, then a lock-up condition could exist.

Also provided by the UART driver are input and output filtering. The filtering allows character substitution or removal. For example, in "C" it is typical to only use the character `'\n'` (New Line) as the end of line. When this is sent as is on a terminal, a `'\r'` (Carriage Return) is missing and the displayed text is not as expected. The filter can be set-up upon initialization to replace all outgoing `'\n'` with the sequence `'\r' '\n'`. Equivalently, when entering text on a terminal, it is typical to use the `'\r'` as the end of line character. The filtering can be set to substitute all incoming `'\r'` by `'\n'` to make the input text compatible with "C".

# 2 Target Set-up

All there is to do to configure and enable the use of the UART driver in an application based on Abassi is to include the following file in the build:

- `???_uart.c` (For Abassi & standalone)
- `???_uart_CMSIS.c` (For CMSIS RTOS API)
- `SAL.c` (For Standalone)
- `ISRhandler_???.s` (For Standalone)

and set-up the include search directory order making sure the file `???_uart.h` is found (and `SAL.h` for the standalone) and set-up build options as required.

If interrupts are used, one or multiple UART interrupt handlers (`UARTintHndl_n()`, Section 3.1.5) must be attached to the interrupt controller. In Abassi this is simply done using the `OSisrInstall()` component.

The UART driver may or may not, depending on the target platform, be independent from other include files.

## 2.1 Build Options

There are a few build options that allow the UART driver to be configured for the requirements of the target application. The following two tables list all of them (there is an alternate token naming, refer to section 2.2.4) :

**Section Table 2-1 Build Options**

| Token Name | Default | Description |
|---|---|---|
| `OS_PLATFORM` | `Target dependent` | Number indicating the target platform. Refer to `???_uart.h` to see the list of supported platforms and the default one. |
| `UART_MAX_DEVICES` | `Target dependent` | Number of UART device(s) supported by the target platform |
| `UART_LIST_DEVICE` | `Target dependent` | Bit field selecting the UART device(s) to use. The default value is dependent on the build option `OS_PLATFORM` |
| `UART_CLK` | `Target dependent` | Frequency of the UART module clock |
| `UART_CIRC_BUF_RX` | `0` | If using a circular buffer in the RX direction, size of the buffer |
| `UART_CIRC_BUF_TX` | `0` | If using a circular buffer in the TX direction, size of the buffer |
| `UART_SLEEP` | `0` | When using polling, sleep time when no RX chars available or when the TX FIFO is full. |
| `UART_FULL_PROTECT` | `0` | Boolean used to activate the UART driver internal protection for exclusive device access |
| `UART_FIFO_ROOM` | `2500` | Sets the FIFO threshold when interrupts are used |

| | | |
|---|---|---|
| UART_ARG_CHECK | 1 | Boolean to check ot not check the validity of the function arguments |
| UART_TOUT | -1 or 1s | Timeout when using a blocking mechanism |

**Table 2-2 Build Options (RTOS only)**

| File Name | Default | Description |
|---|---|---|
| UART_SINGLE_MUTEX | -1 | Boolean controlling if a single mutex is shared amongst all devices or if each device has their own mutex |
| UART_SLEEP | 0 | Boolean controlling to use or not use sleep (time blocking) when polling is used |
| UART_MULTIPLE_DRIVER | 0 | To use drivers for different UART types at the same time. |

### 2.1.1 OS_PLATFORM

The build option OS_PLATFROM informs the UART driver about the target platform it will operate on. There are two benefits ensuing from the presence of this build option:

- The UART driver implicitly knows the total number of UART devices on the platform.
- The UART driver is able to configure and reset the UART devices without the application intervention.

### 2.1.2 UART_MAX_DEVICES

The build option UART_MAX_DEVICES informs the UART driver of how many UART devices are on the target platform. If this build option is not set, then the UART driver will rely on the build option UART_LIST_DEVICE (Section 2.1.3). If the build option UART_LIST_DEVICE is also not set, then the UART driver will rely on the OS_PLATFORM value (Section 2.1.1).

UART_MAX_DEVICES can exceptionally be set to a value of zero; this creates "do-nothing stub" functions for uart_init() (Section 3.1.1), uart_recv() (Section 0), and uart_send() (Section 3.1.4). These do-nothing functions always return 0, and having do-nothing stubs is useful with the System Call Layer when none of the stdio (stdin, stdout, stderr) resources are used.

### 2.1.3 UART_LIST_DEVICE

The build option UART_LIST_DEVICE informs the UART driver about the UART devices that are used. When the target platform has multiple UART devices, enabling only the devices used by the application offers two benefits:

- Minimize the data memory required by the driver, as there is no need to reserve memory for the queue descriptors / buffers or optional mutexes of unused devices.
- When a single UART device is used, the UART registers are accessed through direct memory access, possibly making the driver more real-time efficient.

This build option is a bit field, where the bit position represents the UART device number. When the corresponding bit is cleared (set to 0) it specifies the device is not used; when the corresponding bit is set to 1 then the device is used. The following table shows the valid combinations for a target platform with 2 UART devices:

**Table 2-3 Build Options**

| UART_LIST_DEVICE | UART #0 | UART #1 |
|---|---|---|
| 1 | In use | Not used |
| 2 | Not used | In use |
| 3 | In use | In use |

If the build option `UART_LIST_DEVICE` is not externally defined, the default value will be set according to the build option `UART_MAX_DEVICES` (Section 2.1.2). If the build option `UART_MAX_DEVICES` is not set, then `UART_LIST_DEVICE` will be set according to the build option `OS_PLATFORM` (Section 2.1.1) and will make available all the UART devices on the target platform.

`UART_LIST_DEVICE` can exceptionally be set to a value of zero; this creates "do-nothing stub" functions for `uart_init()` (Section 3.1.1), `uart_recv()` (Section 0), and `uart_send()` (Section 3.1.4). These do-nothing functions always return 0, and having do-nothing stubs is useful with the System Call Layer when none of the stdio (stdin, stdout, stderr) resources are used.

### 2.1.4 UART_CLK

This build option overrides the clock frequency the UART operates. The build option value is in Hz and the default value is target platform dependent.

### 2.1.5 UART_CIRC_BUF_RX

When the build option `UART_CIRC_BUF_RX` is defined and set to a positive value (`> 0`), it specifies the size of a circular buffer to be used in the RX directions. The circular buffer can only be used in blocking mode: the interrupt handler fills the circular buffer when new characters are available. To use the circular buffer, the argument `RXsize` of the function `uart_init()` (Section 3.1.1) must be set to a non-zero value (the value itself is irrelevant). By defining and setting `UART_CIRC_BUF_RX` to a positive value (`> 0`), it overloads the use of mailboxes (`RXsize > 0`) or it overloads the per character transfers with blocking on semaphore (`RXsize < 0`).

It is detrimental to use `RXsize == 0` when `UART_CIRC_BUF_RX` defined and set to a positive value (`> 0`), as the UART driver operates in the polling mode. There will be memory allocated for the circular buffer although that memory will never be used.

### 2.1.6 UART_CIRC_BUF_TX

When the build option `UART_CIRC_BUF_TX` is defined and set to a positive value (`> 0`), it specifies the size of a circular buffer to be used in the TX directions. The circular buffer can only be used in blocking mode: the back ground fills the circular buffer and the interrupt handler retrieve characters from it when there is room in the UART device. To use the circular buffer, the argument `TXsize` of the function `uart_init()` (Section 3.1.1) must be set to a non-zero value (the value itself is irrelevant). By defining and setting `UART_CIRC_BUF_TX` to a positive value (`> 0`), it overloads the use of mailboxes (`TXsize > 0`) or it overloads the per character transfers with blocking on semaphore (`TXsize < 0`).

It is detrimental to use `TXsize == 0` when `UART_CIRC_BUF_TX` defined and set to a positive value (`> 0`), as the UART driver operates in the polling mode. There will be memory allocated for the circular buffer although that memory will never be used.

### 2.1.7 UART_SLEEP

This build option is supported in both the standalone and the RTOS version but is only useful in an RTOS based application. When using polling, if there are no characters available in the RX FIFO or if the TX FIFO is full, it is possible to put the calling task to sleep with the use of the `UART_SLEEP` build option. When `UART_SLEEP` is set to a positive value (> 0), it specifies the number of timer tick to put the task to sleep. `UART_SLEEP` is ignored if the RTOS is not configured to use a timer (`OS_TIMER_US`) [R1] or if timeouts are set to infinity (`OS_TIMEOUT <= 0`) [R1].

### 2.1.8 UART_FULL_PROTECT

The build option `UART_FULL_PROTECT` allows the activation of an RTOS independent mechanism for exclusive access protection of the UART device(s). Setting the build option `UART_FULL_PROTECT` to a non-zero value will configure the driver to use its RTOS independent internal protection; setting it to a zero value will not enable the internal protection. One must be aware this protection mechanism relies on interrupts disabling / restore plus spinlocks when in a multi-core environment. As such, it is possible the interrupts could be disabled for a fair amount of time. Refer to Section 2.2 for more information.

If full protection is enabled (`UART_FULL_PROTECT != 0`), then the build option `UART_SINGLE_MUTEX` (Section 2.1.10) is ignored and the UART does not use mutexes.

### 2.1.9 UART_FIFO_ROOM

This build option only applies to a limited number of UART drivers. The value specified by `UART_FIFO_ROOM` indicates the FIFO threshold in µs when the UART generates an interrupt. This is when the RX FIFO has reached the point where it has just enough room left to received `UART_FIFO_ROOM` µs of character duration, or when the TX FIFO is left with less than `UART_FIFO_ROOM` µs of character duration to transmit. Time units are used instead of the number of characters, as the goal is to set the FIFO thresholds according to the worst-case response to a UART interrupt in the target application. There are many factors that influence the UART response time, but this is outside the scope of this document.

By default, `UART_FIFO_ROOM` is set to 2500, which is for a 2.5 *m*s maximum interrupt response time. The duration of a UART character takes into account all bits for the programmed Baud rate and protocol: the start bit, the number of character bits, the parity bit if used, and the number of stop bits. Depending on the size of the RX FIFO and the calculated character duration, it could be possible that the resulting threshold does not match the requested time, because a safety margin is used.

### 2.1.10 UART_SINGLE_MUTEX

In an RTOS environment (either Abassi or with the CMSIS RTOS API), the driver can provide exclusive access protection to the UART device(s) through its internal mutex(es). There are two methods available to provide the mutex based exclusive access to the UART device(s):

- Using a single mutex shared amongst all devices
- Using one mutex per device

Setting the build option `UART_SINGLE_MUTEX` to a positive value will configure the driver to use a single mutex, shared amongst all the UART devices. Setting it to a zero value will assign to each UART device its own mutex. Setting it to a negative value (default value) will not activate the internal use of mutexes by the driver. Refer to Section 2.2 for more information.

### 2.1.11 UART_SLEEP

The build option `UART_SLEEP`, which is only available in an RTOS environment (either Abassi or the CMSIS RTOS API), is used to enable or not enable the task sleep when the driver is configured to perform polling during the device initialization. Depending on the Baud rate and if the UART device has a FIFO, and also depending of that FIFO size, one must make sure the minimum granularity of sleep time (this is set by the RTOS timer tick period) is not too short for proper operation. Take for example a communication with a Baud rate at 115200 and a UART device with 64 character FIFOs. The FIFOs can be filled/emptied in about 5.5ms. If the RTOS timer tick is set to 10*m*s, or even 5*m*s, it simply results in the problem where there is not enough time resolution to guarantee the exact duration of the task sleep.

Setting the build option `UART_SLEEP` to a non-zero value will configure the driver to put the task into sleep when any modules are configured for polling. Setting it to a zero value (default value) will not put tasks to sleep. This build option is global therefore it is not possible to set one device to use sleep and another to not use sleep.

### 2.1.12 UART_MULTIPLE_DRIVER

See section 2.2.4.

### 2.1.13 UART_ARG_CHECK

The build option `UART_ARG_CHECK` configures the UART driver on how it handles the API function arguments. If the build option is set to a non-zero value (default setting) the function arguments are validated with error report upon invalidity. If the build option is set to a value of zero, none of the function arguments are validated.

### 2.1.14 UART_TOUT

When the UART driver is set-up to use interrupts upon initialization (`uart_init()`), either mailboxes or binary semaphores are used as the blocking mechanism. By default, `UART_TOUT` is set to 1 second (`OS_TICK_PER_SEC` [R1]) meaning the mailboxes or binary semaphores block with a timeout of 1 s. This is a safety mechanism that guarantees the UART driver will never lock-up an application upon issues with interrupts. The timeout duration (argument to the service call) is set by the value assigned to the build option UART_TOUT.

## 2.2 Set-up

The UART driver can operate under most contexts in an application (Section 1.3). But to properly operate, there are some basic requirements unique to each situation. The following sub-sections describe the basic required set-up.

### 2.2.1 RTOS (Not used in ISRs)

The exclusive access protection with mutexes is all there is to consider when using the UART driver in a multi-tasking environment when it is not used in an interrupt handler. If a single task uses the UART driver, then no exclusive access protection is required. When used across multiple tasks, the best exclusive access protection choice is to rely on an I/O library that internally uses mutexes to control the accesses of the standard I/O devices. That is only if the library uses the same mutex on the input and the output. When this is not the case, or when the library does not use mutexes, then the build option `UART_SINGLE_MUTEX` should be set to non-negative value. Setting it to zero will make the UART driver use a single mutex per devices. When set to a positive value, the UART driver uses a single mutex for all UART devices.

It is always possible to not use a mutex for the exclusive access protection mechanism by setting the build option `UART_FULL_PROTECT` (Section 2.1.8) to a non-zero value. Doing so involves the use of interrupt disabling / restore (plus spinlocks in a multi-core platform). Using mutexes is preferable as the choice for the protection mechanism implemented because using the interrupt disable / restore does intrinsically impact the real-time response of the application.

In the Abassi RTOS environment, the UART driver does automatically fall back to a non-blocking mode when it is called with any of these conditions true:

- Called before the `OSstart()` was done
- Called from within the kernel
- Called with the interrupts disabled
- Called within an interrupt handler

Falling back to a non-blocking mode means that if the driver is set to use semaphores, the semaphore is not used and the driver performs polling. If the driver is set to use mailboxes, then mailboxes are still used but without blocking with a periodic manual operation of the UART mailbox interrupt to get the mailbox properly used by the UART device. Please be aware that when the UART driver is initialized before `OSstart()` is done, the driver operates in polling only. When this is the set-up, if semaphores or mailboxes operation are desired, a re-initialization of the driver is required (Section 3.1.1).

### 2.2.2 RTOS (Used in ISRs)

When the UART driver is used inside the interrupt handler(s) of an application built on an RTOS, then it becomes impossible to fully rely on mutexes for the exclusive access protection of the UART device, as mutexes (locking mutex could block a task) should never be used in interrupts. Therefore, only the basic interrupt disable / restore (plus spinlocks in a multi-core platform) can be relied on to provide exclusive accesses protection to the UART device. This interrupt disabling / restore + spinlocks must use the mechanism provided by the UART driver by setting the build option `UART_FULL_PROTECT` (Section 2.1.8) to a non-zero value, as it deals with all cases of access to the UART device. When the build option `UART_FULL_PROTECT` is set to a non-zero value, the build option `UART_SINGLE_MUTEX` is internally forced to a negative value, meaning to not associate internal mutexes with the UART driver.

Another requirement when using the UART driver in an interrupt handler is to always request a non-waiting sending/receiving by setting argument `Len` in `uart_send` (Section 3.1.4) and `uart_recv` (Section 0) to a negative value. If the UART driver is not called with the indication to not wait, the driver will ignore the request to wait and internally fall back to a no-wait condition when it is used within an interrupt handler.

### 2.2.3 Stand-Alone

When the UART driver is used in a stand-alone application, mutexes are not available, therefore the build option `UART_SINGLE_MUTEX` must be set to a negative value to inform the UART driver to not rely on mutexes. If the UART driver is not used in the interrupt handlers, then the build option `UART_FULL_PROTECT` (Section 2.1.8) can be left to its default value of zero, meaning no interrupt disable / restore protection. If the UART driver is used in both the background and the interrupts handlers, then the build option `UART_FULL_PROTECT` should be set to a non-zero value.

### 2.2.4 Multiple Drivers

It is possible to use 2 or more drivers for different UARTs. Example of the need for this is a processor with on-chip UART(s) on a board with different type of UART(s), or a SocFPGA with added UART(s) in the FPGA fabrics that are of different type than the processor system UART(s). To use multiple drivers the build option `UART_MULTIPLE_DRIVER` must be defined and set to a non-zero value. This changes the API names of the driver by pre-pending the UART type to the function names. For example, if `dw_uart.c` is used, the APIs are named as following:

**Table 2-4 BAPI remapping**

| Original | Multiple |
|---|---|
| `uart_filt()` | `dw_uart_filt()` |
| `uart_init()` | `dw_uart_init()` |
| `uart_recv()` | `dw_uart_recv()` |
| `uart_send()` | `dw_uart_send()` |
| `UARTintHndl_#()` | `dw_UARTintHndl_#()` |

The prefix is always the prefix in the file name; e.g. `dw_uart.c` prefix is dw and `cd_uart.c` is cd.

All build options if not prefixed apply to all the drivers. To set build options on a per-driver basic al there is to do is used the build option that has been pre-fixed with the same prefix used in the API but in uppercase. For example to set each of the multiple drivers to support 3 devices each, the build option `UART_MAX_DEVICES` should defined and set to 3. If the `dw_uart` and the `ns16550_uart` drivers are use together and, as for the example below, there are 2 devices for the `dw` and 3 for the `ns16550`, then `DW_UART_MAX_DEVICES` should defined and set to 2 and `NS16550_UART_MAX_DEVICES` should defined and set to 3.

A custom wrapper must be provided. The following code shows such a driver for the `dw_uart` (2 UARTs accessed as device #0 and #4) and the `ns16550_uart` (3 UARTs accessed as devicde #2, #3, and #4):

**Table 2-5 Multiple UART wrapper example (`uart.c`)**

```
#include "uart.h"

/* ------------------------------------------------------------------------------ */

int uart_filt(int Dev, int Enable, int Filter)
{
int RetVal;

   RetVal = (Dev < 2)
          ?      dw_uart_filt(Dev,   Enable, Filter)
          : ns16550_uart_filt(Dev-2, Enable, Filter);

   return(RetVal);
}

/* ------------------------------------------------------------------------------ */

int uart_init(int Dev, int Baud, int Nbits, int Parity, int Stop, int RXsize,
              int TXsize,int Filter)
{
int RetVal;

   RetVal = (Dev < 2)
          ?      dw_uart_init(Dev,   Baud, Nbits, Parity, Stop, RXsize, TXsize, Filter)
          : ns16550_uart_init(Dev-2, Baud, Nbits, Parity, Stop, RXsize, TXsize, Filter);

   return(RetVal);
}

/* ------------------------------------------------------------------------------ */

int uart_recv(int Dev, char *Buff, int Len)
{
int RetVal;

   RetVal = (Dev < 2)
          ?      dw_uart_recv(Dev,   Buff, Len)
          : ns16550_uart_recv(Dev-2, Buff, Len);

   return(RetVal);
}


/* ------------------------------------------------------------------------------ */

int uart_send(int Dev, const char *Buff, int Len)
{
int RetVal;

   RetVal = (Dev < 2)
          ?      dw_uart_send(Dev,   Buff, Len)
          : ns16550_uart_send(Dev-2, Buff, Len);

   return(RetVal);
}

/* ------------------------------------------------------------------------------ */

void UARTintHndl_0(void) {      dw_UARTintHndl_0(); }
void UARTintHndl_1(void) {      dw_UARTintHndl_1(); }
void UARTintHndl_2(void) { ns16550_UARTintHndl_0(); }
void UARTintHndl_3(void) { ns16550_UARTintHndl_1(); }
void UARTintHndl_4(void) { ns16550_UARTintHndl_2(); }
```

```
/* EOF */
```

**Table 2-6 Multiple UART wrapper example (`uart.h`)**

```
#ifndef __UART_H__
#define __UART_H__       1

#include "dw_uart.h"                          /* DW  UART driver                    */
#include "ns16550_uart.h"                     /* NS16550 UART driver                */

#ifndef UART_MULTI_DRIVER                     /* It must be defined and set to !=0  */
  #define UART_MULTI_DRIVER      0          /* Set 0 to trigger the error message */
#endif

#if ((UART_MULTI_DRIVER) == 0)
  #error "UART_MULTI_DRIVER must be defined and set to a non-zero value"
#endif

#define UART_MAX_DEVICES ((DW_DW_UART_MAX_DEVICES)+(NS16550_DW_UART_MAX_DEVICES))
#define UART_LIST_DEVICE ((DW_UART_LIST_DEVICE)|((NS16550_UART_LIST_DEVICE)<<2))

/* ------------------------------------------------------------------------------- */

int uart_filt(int Dev, int Enable, int Filter);
int uart_init(int Dev, int Baud, int Nbits, int Parity, int Stop, int RXsize,
              int TXsize,int Filter);
int uart_recv(int Dev, char *Buff, int Len);
int uart_send(int Dev, const char *Buff, int Len);

/* ------------------------------------------------------------------------------- */

extern void UARTintHndl_0(void);
extern void UARTintHndl_1(void);
extern void UARTintHndl_2(void);
extern void UARTintHndl_3(void);
extern void UARTintHndl_4(void);

#endif

/* EOF */
```

Enabling the multiple drivers can also be used to redirect any of the 3 `stdio` when the System Call layer [R4] is used. The way to do this is to "play" with the UART device number assigned to `stdin`, `stdout`, and `stderr` (respectively `G_UartDevIn`, `G_UartDevOut`, and `G_UartDevErr`). The UART device number is the information used by the wrapper to determine if the physical device to access is an UART or an other type of device. Any device number with a value of less than `UART_MAX_DEVICES` is for sure an UART device; all other values indicate another device type. If the other device is accessible through the System Call layer, then everything \become easy as all there is to do is to call `write()` or `read()` with the proper file descriptor value. The next 2 tables show the require operations.

If the other device is not accessible through the System Call layer, the implementation should be very close to the example provided.

NOTE: The filter enabled in the UART assigned to the `stdio` devices are not active when a redirected device is not an UART: the filters are always specific the UART device they have been set-up.

**Table 2-7 stdio redirection UART wrapper example (`uart.c`)**

```
#include “SysCall.h”

extern int G_UartDevOut;

…

   UartOut = G_UartDevOut;              /* Memo original UART device number          */
                                        /* Open the file and redirection             */
   fd = open(“Log.txt”, O_CREATE | O_RDWR, 0666);
   if (fd >= 0) {
       G_UartDevOut = UART_MAX_DEVICES+fd;
   }
   else {
       … error …
   }

…
                                        /* Close the file and original stdout        */
   G_UartDevOut = UartOut;
   close(fd);
```

**Table 2-8 stdio redirection UART wrapper example (`uart.c`)**

```
#include "uart.h"
#include “SysCall.h”

/* ---------------------------------------------------------------------------- */

int uart_filt(int Dev, int Enable, int Filter)
{
int RetVal = 0;

  if (Dev < UART_MAX_DEVICES) {
      RetVal = dw_uart_filt(Dev,   Enable, Filter);
  }

   return(RetVal);
}

/* ---------------------------------------------------------------------------- */

int uart_init(int Dev, int Baud, int Nbits, int Parity, int Stop, int RXsize,
              int TXsize,int Filter)
{
int RetVal = 0;

  if (Dev < UART_MAX_DEVICES) {
     RetVal = dw_uart_init(Dev, Baud, Nbits, Parity, Stop, RXsize, TXsize, Filter);
  }

   return(RetVal);
}

/* ---------------------------------------------------------------------------- */

int uart_recv(int Dev, char *Buff, int Len)
{
int RetVal;

   if (Dev < UART_MAX_DEVICES) {
       RetVal = dw_uart_recv(Dev, Buff, Len);
   }
   else {
       RetVal = read(Dev-UART_MAX_DEVICES, Buff, Len);
   }

   return(RetVal);
}


/* ---------------------------------------------------------------------------- */

int uart_send(int Dev, const char *Buff, int Len)
{
int RetVal;

   if (Dev < UART_MAX_DEVICES) {
       RetVal = dw_uart_send(Dev, Buff, Len);
   }
   else {
       RetVal = write(Dev-UART_MAX_DEVICES, Buff, Len);
   }

   return(RetVal);
}
```

```
/* ------------------------------------------------------------------------------- */

void UARTintHndl_0(void) {      dw_UARTintHndl_0(); }
void UARTintHndl_1(void) {      dw_UARTintHndl_1(); }
void UARTintHndl_2(void) {      dw_UARTintHndl_2(); }
void UARTintHndl_3(void) {      dw_UARTintHndl_3(); }

/* EOF */
```

# 3 API

In this section, the API of all common UART driver functions is provided. The next section gives examples on how to use the UART

### 3.1.1 uart_filt

**Synopsis**

```
#include "???_uart.h"

int uart_filt(int Dev, int Enable, int Filter);
```

**Description**

`uart_filt()` is a component to use for changing the filter setting (the setting that was don't during initialization with `uart_init()`). The UART controller to use is indicated by the argument `Dev`, if the filtering indicated by the argument `Filter` is to be enable, then the argument `Enable` must be set to a non-zero value and if the filtering is to be disable, then `Enable` must be set to 0.

**Arguments**

| | |
|---|---|
| `Dev` | Device's controller number (Number starting at 0) |
| `Enable` | If enabling or disabling the filtering operation indicated in the argument `Filter:`<br>`== 0` : disable<br>`!= 0` : : enable |
| `Filter` | Handling of \r, \n and \b (Is a set of bits OR any of the control masks). On some platform it can also be used to enable flow control with RTS/CTS & DTR/DSR etc.<br>`== 0` : do nothing and return the current filter setting; i.e. the binary OR of all filter masks that have been enable.<br>See the defines of input and output character filter control masks in `???_uart.h` |

**Returns**

| | |
|---|---|
| `int` | `== -1` : error<br><br>`!= -1` : original setting of the filter option(s) indicated by `Filter.`<br>It is not all the current filtering options but only the one that were specified by `Filter`. |

**Component type**

Function

**Options**

**Notes**

For example to disable the echoing of RX to TX (for password entry for example):

```
Original = uart_filt(0, 0, UART_ECHO);
if (Original != -1) {
       … read the password
       uart_filt(0, 1, Original);
}
```

Because the return value (when there is no error) is the original filter setting of UART_ECHO, the re-ebaling can be done without checking if UART_ECHO was originaly enable or not. If it was originally enabled, then the return value held in `Original` is `UART_ECHO`. If it wasn't enabled, the return value held in `Original` is 0, indicating no options to be enabled.

**See Also**

### 3.1.2 uart_init

**Synopsis**

```
#include "???_uart.h"

int uart_init(int Dev, int Baud, int Nbits, int Parity, int Stop,
              int RXsize, int TXsize, int Filter);
```

**Description**

`uart_init()` is the component used to initialize one UART device. The device's controller number is indicated by the argument `Dev`. The Baud rate is specified by the argument `Baud`, the number of bits per character by the argument `Nbits`, if parity is used, and what parity, by the argument `Parity`, and the number of stop bits with the argument `Stop`. The use of mailboxes, polling or semaphore blocking, is enabled and its size through the arguments `RXsize` and `TXsize`. The filtering and echoing are configured with the argument `Filter`.

**Arguments**

`Dev` Device's controller number (Number starting at 0)

`Baud` Baud rate of the serial link. The value specified is the baud rate as is. For example, if a baud rate of 115200 is required, the argument `Baud` is simply set to the numerical value of 115200

`Nbits` Number of bits per character. The range is target dependent and if a value out of range is specified, no error is reported and the resulting number of characters is target dependent

`Parity` Parity bit
- `== 0` : no parity bit
- `> 0` : even parity
- `< 0` : odd parity

`Stop` Number of stop bits
- `== 5` : 0.5 stop bit
- `== 10` : 1 stop bit
- `== 15` : 1.5 stop bit
- `== 20` : 2 stop bit

`RXsize` If using a mailbox in the receive direction and if so the size of the mailbox
- `< 0` : with `UART_CIRC_BUF_RX <= 0` (or undefined)
  blocking on a semaphore (requires the ISR handler)
  with `UART_CIRC_BUF_RX > 0`
  Circular buffer with blocking (requires the ISR handler)
- `== 0` : polling, not using mailboxes (ISR handler is not used)
- `> 0` : with `UART_CIRC_BUF_RX <= 0` (or undefined)
  mailboxes set to a size of `RXsize` (requires the ISR handler)
  with `UART_CIRC_BUF_RX > 0`
  Circular buffer with blocking (requires the ISR handler)

`TXsize` If using a mailbox in the transmit direction and if so the size of the mailbox
- `< 0` : with `UART_CIRC_BUF_TX <= 0` (or undefined)
  blocking on a semaphore (requires the ISR handler)
  with `UART_CIRC_BUF_TX > 0`
  Circular buffer with blocking (requires the ISR handler)
- `== 0` : polling, not using mailboxes (ISR handler is not used)
- `> 0` : with `UART_CIRC_BUF_TX <= 0` (or undefined)
  mailboxes set to a size of `TXsize` (requires the ISR handler)
  with `UART_CIRC_BUF_TX > 0`
  Circular buffer with blocking (requires the ISR handler)

| | |
|---|---|
| `Filter` | Handling of \r, \n and \b (Is a set of bits OR any of the control masks). On some platform it can also be used to enable flow control with RTS/CTS & DTR/DSR etc.<br>`== 0` : do nothing<br>See the defines of input and output character filter control masks in `???_uart.h` |

**Returns**

| | |
|---|---|
| `int` | `== 0` : success<br>`!= 0` : error |

**Component type**

Function

**Options**

**Notes**

As previously stated, it is possible to initialize and use the UART driver before Abassi is started. Doing so means it is possible to re-initialize the UART driver after Abassi is started in order to use mailboxes and other services.

The UART driver can operate with three different mechanisms, and a direction can use a different mechanism from the other one:

- Using polling with short sleep time when no characters are available
- Using a mailbox, which will block when the mailbox is empty
- Using a semaphore to block when no characters are available

The polling mechanism is selected by setting the argument `RXsize` and/or `TXsize` to zero when using `uart_init()`. When using polling, the UART driver does not rely on the UART ISRs or the UART ISR handler. Instead, it simply checks continuously to see if a character is available from the UART. When no characters are available, then, if possible, it will go into sleep for a short time and then check again. If the build option `UART_CIRC_BUF_RX` and/or `UART_CIRC_BUF_TX` are set to a positive value, then the polling mechanism uses a circular buffer. Although circular buffers can be used with and without interrupts, their use without interrupts provides no advantage.

The semaphore blocking mechanism is selected by setting the `RXsize` and/or `TXsize` to a negative value when using `uart_init()`. When using the semaphore blocking, the UART driver does rely on the UART ISRs and the UART ISR handler. In the receive direction, when there are no characters available from the UART, the driver will block, if possible, on the receive semaphore. If it is not possible to block, it falls back to pure polling. In the transmit direction, when there is no room left in the output FIFO, the driver will block, if possible, on the transmit semaphore. If it is not possible to block, it falls back to pure polling. The semaphore gets posted in the interrupt handler when the blocking condition is over. Compared to the polling mechanism, the use of the semaphore keeps the driver blocked until a new character is available or there is room in the transmit direction, while the polling keeps awakening the driver to check if a new character is available.

The mailbox mechanism is selected by setting the `RXsize` or `TXsize` to a positive when using `uart_init()`.

IMPORTANT:

When the mailboxes are used, it is critical to take into consideration the size of the interrupt queue (set by Abassi's build option `OS_MAX_PEND_RQST`) when dimensioning the UART mailbox queues. The transfer of the characters between the mailboxes and the UART are mostly done inside an interrupt handler. As such, each character exchanged results in a request that is inserted in Abassi's ISR queue. Therefore if the total size of the mailboxes (RX & TX) of all the UARTs devices used exceeds the size of Abassi's ISR queue, there is a risk of overflowing the ISR queue. This means for sure loss of characters and if the out-of-memory detection is enable (Abassi's build option `OS_OUT_OF_MEM`), then the application will freeze (due to the detection of the ISR queue overflow).

**See Also**

### 3.1.3 uart_recv

**Synopsis**

```
#include "???_uart.h"

int uart_recv(int Dev, char *Buff, int Len);
```

**Description**

`uart_recv()` is the component used to read the characters received by the UART. The UART controller to use is indicated by the argument `Dev`, and the buffer to deposit the received characters into is specified by `Buff`. The number of bytes to read from the UART is specified by the argument `Len`. With the argument `Len`, it is possible to specify if the driver should wait for all the characters indicated or to quit when there are no more characters presently available.

**Arguments**

| | |
|---|---|
| `Dev` | Device number (Number starting at 0) |
| `Buff` | Buffer that will hold the data read from the UART. This buffer must be sized to at least `Len` bytes |
| `Len` | Number of character to read from the UART. If the value of `Len` is negative, then it indicates to the driver to read up to –`Len` characters and quit when either –`Len` have been read or there are no more characters presently available. |

**Returns**

| | |
|---|---|
| `int` | Number of characters read and held in `Buff`. The reported number may be different than the number of characters received by the UART due to the filtering. |

**Component type**

Function

**Options**

**Notes**

**See Also**

`uart_send()`

### 3.1.4 uart_send

**Synopsis**

```
#include "???_uart.h"

int uart_send(int Dev, const char *Buff, int Len);
```

**Description**

`uart_send()` is the component used to send characters to the UART. The UART controller to use is indicated by the argument `Dev`, and the buffer holding the characters to send is specified by `Buff`. The number of bytes to send to the UART is specified by the argument `Len`. With the argument `Len`, it is possible to specify if the driver should wait for room in the UART (or UART FIFO) or to quit when there is no room presently available.

**Arguments**

| | |
|---|---|
| `Dev` | Device number (Number starting at 0) |
| `Buff` | Buffer that holds the data to send to the UART |
| `len` | Number of characters to send to the UART. If the value of `Len` is negative, then it indicates to the driver to send up to –`Len` characters and quit when either –`Len` have been sent or there is no more room in the UART to accept more characters. |

**Returns**

| | |
|---|---|
| `int` | Number of characters sent to the UART. The reported number may be different than the number of characters physically sent by the UART due to the filtering. |

**Component type**

Function

**Options**

**Notes**

**See Also**

`uart_recv()`

### 3.1.5 UARTintHndl_*n*

**Synopsis**

```
#include "???_uart.h"

void UARTintHndl_n(void);
```

**Description**

`UARTintHndl_n()` is the interrupt handler for the UART driver (not used by the standalone version). The ***n*** in the name is a numerical value that specifies the device number the interrupt handler is for.

**Arguments**

`void`

**Returns**

`void`

**Component type**

Function

**Options**

**Notes**

The interrupt handler should always be attached to the targeted UART device interrupt and the number of the interrupt handler MUST match the device number. If there is a mismatch, then the application will most likely crash. If the interrupt handler is not attached and the related interrupt enabled, the UART driver for this device will not operate at all (not applicable for the standalone version).

**See Also**

# 4 Examples

## 4.1 Initialization

The first step required when using the UART driver is to initialize the device controller to be used. The second step required (not applicable with the standalone version) is to attach the interrupt handler for the UART device and enable these interrupts. The following example is typical of how the UART should be configured when using the GCC library. It shows the initialization of controller #1 with the following characteristics:

- Baud rate of 115200
- 8 Data bits
- No parity
- 1 Stop bit (argument set to 10)
- Mailbox of 32 entries in the receive direction
- Mailbox of 128 entries in the transmit direction
- The filtering performs the following:
    - Replace all outgoing LF with the sequence CR and LF
    - Replace all incoming CR with LF
    - Echo to the output all received characters
    - Expand input BS (replace) with BS, WhiteSpace, BS

**Table 4-1 Initialization**

```
uart_init(1, 115200, 8, 0, 10, 32, 128, UART_FILT_OUT_LF_CRLF
                                      | UART_FILT_IN_CR_LF
                                      | UART_ECHO
                                      | UART_ECHO_BS_EXPAND);

OSisrInstall(UART_INT1, &UARTintHndl_1);
GICenable(UART_INT1, 128, 1);
```

The interrupt handler can only be used (and MUST be used) with Abassi and the CMSIS API. With the standalone version, do NOT attach/use the interrupt handler or enable the related interrupt.

## 4.2 UART transmission

The following example is simply the GCC library `_write_r()` (or `_write()` in some libraries) function used for all I/O writing. It calls directly `uart_send()` with device #1 (same device as shown in the initialization example). It passes the buffer and length information directly to `uart_send()`. As the buffer length is positive, the driver will send all the characters supplied, even if it has to wait for room in the UART (or UART FIFO).

**Table 4-2 UART transmission**

```
_ssize_t _write_r (struct _reent *rreenn, int fd, const void *vbuf, size_t len)
{
   if ((fd == 1) || (fd == 2))              /* Make sure it is stdout or stderr      */
       return(uart_send(1, (const char *)vbuf, (int)len));
   }
   else {                                   /* Is a file writing                     */
       …
   }
   return(0)
}
```

## 4.3 UART reception

The following example is simply the GCC library `_read_r()` function that is used for all I/O reading. It uses directly `uart_recv()` with device #1 (same device as shown in the initialization example). It passes the buffer and number of characters to read directly to `uart_recv()`. As the buffer length is positive, the driver will read all the requested characters, even if it has to wait for new characters to become available from the UART (or UART FIFO).

**Table 4-3 UART reception**

```
_ssize_t _read_r(struct _reent *rreenn, int fd, void *vbuf, size_t size)
{
   if (fd == 0)                             /* Make sure it is stdin                 */
       return(uart_recv(0, (char *)vbuf, (int)size));
   }
   else {                                   /* Is a file reading                     */
       …
   }
   return(0)
}
   …
```

The following example shows an implementation of the non-standard but omni-present `getkey()` function that is used to report if a character is available and what it is. It uses directly `uart_recv()` with device #1 (same device as shown in the initialization example). It passes the pointer to a single character and requests to read a length of `-1`, meaning to return any character currently available or nothing if none is available. The information if a character is available is reported through the return value of `uart_recv()`.

**Table 4-4 UART reception (no waiting)**

```
int getkey(void)
{
char     ccc;

   ccc = (char)0;
   uart_recv(1, &ccc, -1);

   return((int)ccc);
}
```

# 5 References

[R1] Abassi RTOS – User Guide, available at http://www.code-time.com
[R2] mAbassi RTOS – User Guide, available at http://www.code-time.com
[R3] µAbassi RTOS – User Guide, available at http://www.code-time.com
[R4] Abassi RTOS – System Calls Layer, available at http://www.code-time.com

# 6 Revision History

| Date | Version | Author/Editor | Description |
|---|---|---|---|
| 2016.04.06 | 1.1 | EV | First Draft |
| 2016.04.06 | 1.2 | AP | Review changes |
| 2016.04.11 | 1.3 | EV | Added semaphore blocking |
| 2016.04.27 | 1.4 | EV | Updated for final implementation |
| 2016.04.27 | 1.5 | AP | Review changes |
| 2016.06.19 | 1.6 | EV | `UART_FIFO_ROOM` and some other little things |
| 2016.06.19 | 1.7 | AP | Review changes |
| 2017.08.25 | 1.8 | EV | Lots of build options added |
| 2018.02.22 | 1.9 | EV | Multiple drivers at the same time |
| 2018.07.18 | 1.10 | EV | I/O redirection and uart_filt() with filter == 0 |
| 2018.08.28 | 1.11 | EV | Small changes |
| 2018.08.29 | 1.12 | EV | Circular buffer explanations were wrong |
| 2018.11.20 | 1.13 | EV | `UART_CLK` instead of `UART_CLOCK` + Alternate Token naming |