CODE TIME TECHNOLOGIES

Abassi RTOS

Debug / Monitoring Shell

Disclaimer

Table of Contents

List of Figures

List of Tables

1 Introduction

This document describes the Debug / Monitoring shell add-on provided with Abassi[1] [R1] (including mAbassi [R2] and µAbassi [R3]). The shell complements software development environments by providing a way to access the RTOS services in the application. The Shell is part of the application so it can be also be used to monitor and debug in the field the RTOS part of an application.

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
`Shell.h`	Include file for the debug / monitoring shell
`Shell.c`	"C" file for the debug / monitoring shell
`SubShell.c`	"C" file template to add application specific debug / monitoring commands

1.2 Limitations

- The debug / monitoring shell requires the `OS_NAMES` build options to be defined and set to a non-zero value. If it is set to a zero value a compile time error is generated.
- The shell can handle statically defined / allocated descriptors but it cannot deal with them on a name basis. They can only be accessed using their addresses.
- The shell is single user only. Creating multiple instances (tasks) running the shell will create conflicts between multiple users and possibly an application crash.
- At the present time all input and output are performed through `stdin` and `stdout`. In a future release the capability to use alternate I/Os will be added.

1.3 Features

The debug / monitoring shell provides a supplemental way to debug / control / inspect an application using Abassi. It is not a source code debugger alike the Eclipse GUI but it is instead it's a debugger for the RTOS. It allows the inspection (information dump) of all the services created and used in application and it also allows the modification on the services and performing some operations on the services, for example posting a semaphore. It also provides a file system shell to perform basic operation alike `mkdir`, `ls` or `cat`. Application specific commands can easily be added to the Shell making them accessible through the Shell user interface..

The input and output of the shell are done through `stdin` and `stdout`, which typically are mapped through the system call layer to an UART on the target platform. If a re-direction is desired, refer to the Abassi UART driver [R4] as it explains how to redirect `stdin`, `stdout`, and `stderr` to another device.

1.4 Target Set-up

All there is to do to configure and enable the use of the shell is to include the following files in the build:

- `Shell.c`
- `SubShell.c` (If application specific commands are added)

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

and to set-up the include search directory path making sure the file `Shell.h` is found and to define the build options as required. The shell may or may not, depending on the target platform, be independent from other include files.

The shell is in fact a task and the shell is included and runs in an application by creating a task for it and letting the task run. The priority of the shell task should be set to either a very high priority or a very low priority. When it is set to a very high priority, the shell will likely impact the real-time processing but on the positive side it should always be accessible even under heavy CPU utilization by the application. On the opposite end, running the shell at a very low priority won't impact much (or not at all if it's alone at the lowest priority) but under heavy application load the access to the shell could become be intermittent if not completely inaccessible.

The shell task can be created and resumed using for example:

Table 1-2 Shell set-up

```
#include "Shell./h"

TSKcreate("Shell", SHELL_PRIO, SHELL_STACK_SIZE, &OSshell, 1);
```

1.5 Build Options

The debug / monitoring shell relies on a few build options for its configuration and they are listed in the following table:

Table 1-3 Build options

Build Option	Default	Description
SHELL_CMD_LEN	128	Maximum number of character the shell processes on a single command line
SHELL_FILES	0	Boolean to enable/disable the access to the file system commands
SHELL_HISTORY	0	Selects if a command line history is available and if available how many previous commands it memorizes
SHELL_INPUT	10	Specifies how the shell gets the command line characters
SHELL_LOGIN	0	Boolean to enable/disable the login accesses credentials; i.e. username & password
SHELL_USE_SUB	0	Boolean to enable/disable application specific commands add-on
SH_USERNAME_# SH_PASSWORD_# SH_RDONLY_#	Not defined	Triplets of build options specifying a username password and if that user is logged in RW access or RO access.

All build options can be set (overloaded) through the command line. Using a fictitious build option `SHELL_BUILD_OPTION`, the default value assigned to `DMA_BUILD_OPTION` can be overloaded by using the compiler command line option `–D DMA_BUILD_OPTION` and specifying the new value (`1234`), as shown in the following example:

Table 1-4 Command line set of `OS_BUILD_OPTON` (ASM)

```
…cc  … -D SHELL_BUILD_OPTION=256  …
```

All default build options are set as show on the previoss table and can be directly changed by editing the `#define` value assigned in the target specific `".c"` file:

Table 1-5 `OS_BUILD_OPTION` modification

```
#ifndef DMA_BUILD_OPTION
  #define DMA_BUILD_OPTION       1234           /* Comment….                          */
#endif
```

1.5.1 SHELL_CMD_LEN

The build option `SHELL_CMD_LEN` specifies the maximum number of characters the shell can process on a command line. If more characters than `SHELL_CMD_LEN` are "typed", the excess characters are ignored.

1.5.2 SHELL_FILES

The build option `SHELL_FILES` is a Boolean controlling is file system commands are include in the shell or not. By default this build option is set to 0, meaning the file system commands are not available in the shell. To include the file system commands define the build option `SHELL_FILES` and set it to a non-zero value. The file system commands are only useable with the System Call Layer.

1.5.3 SHELL_HISTORY

Alike most modern command line systems the shell support the capability to memorize past commands and recall and edit them as needed. The build option `SHELL_HISTORY` specifies if past commands are memorized and if they are, how many can be memorized. Command history is not supported if the build option `SHELL_HISTORY` is not defined or if it is define with a value of 0 or less. To support command history, define and set the build option `SHELL_HISTORY` to a positive value. The positive value is the maximum number of commands the shell can memorize. When `SHELL_HISTORY` commands have been typed and a new command is typed, the oldest command held in the history buffer is deleted to make room for the newest command.

Past commands, up to `SHELL_HISTORY` of them, can be recalled and edited. The recall and editing recognizes the arrow key of a VT100 terminal and it also recognized the equivalent EMACS movement control characters. Use the command `help edit` in the shell for more details.

1.5.4 SHELL_INPUT

The shell supports 2 type of handling for the input command: one is blocking and the other one is polling. When the shell is waiting for a command line with blocking, it uses the standard "C" function `getchar()` and the task gets blocked until a <CR> is encountered. The polling method relies on the system call `GetKey()` function which is non-blocking. When polling is used the shell task will sleep for a specified time duration when no characters are available; that way the task will not consume excessive processing resource waiting for a new character. To use the blocking (`getchar()`) method, define the build option `SHELL_INPUT` and set it to a value of 0 or less. By default the shell uses the polling method (`GetKey()`) with sleep time of 10 *m*s. To use polling with a different sleep time, define the build option `SHELL_INPUT` and set its value to the desired sleep time specified in *m*s; e.g. for 50 *m*s set the value of `SHELL_INPUT` to 50.

Selecting blocking (`getchar()`) vs. polling (`GetKey()`) is related to the priority the shell task is running at. When the task is running at a high priority it is desirable for it to be blocked until the whole command line has been typed. In that case, the UART driver should be configured to use circular with interrupts to minimize the processing resource usage. When the shell task is running at a low priority, then the task sleep time allows the shell task to relinquish the CPU to other task that are at the same or lower priority. If there wasn't a sleep time for the input then the shell task would remain running and not allow the other tasks at the same or lower priority to run.

1.5.5 SHELL_LOGIN

The build option `SHELL_LOGIN` controls if login credentials are required to access the shell. If SHELL_LOGIN is not defined or is defined and set to a value less or equal to 0 then no access credentials are required. If it is defined and set to a value greater than 0 then access credentials are required with the possibility of doing an automatic "logout" after a pre-programmed time of input inactivity. If the value is set to 1, no automatic logout occurs and any value greater than 1 specifies the inactivity timeout in seconds.

The shell supports two type of accesses: read-write and read-only and the type of access is user specific. There are 2 default username / password already hard coded and one of them has full read-write access when the other is restricted to read-only operations. Up to 10 triplets of username / password / access type can be added with the build option `SH_USERNAME_#`, `SH_PASSWORD_#`, and `SH_RDONLY_#` (See section 1.5.7).

Inactivity time out is only useable when the selected shell input is `GetKey()` because `getchar()` is a blocking operation with no timeout capabilities (See section 1.5.4).

1.5.6 SHELL_USE_SUB

It is possible to add application-specific commands to the shell by defining and setting the build option `SHELL_USE_SUB` to a non-zero value. A template with two examples dummy commands is provided in the file `SubShell.c`. When adding commands one must make sure there are no conflicts with already existing commands: the shell commands and the file system commands. If a conflict arises, the shell or file system command will be the one executed and not the application specific command.

1.5.7 SH_USERNAME_#, SH_PASSWORD_# and SH_RDONLY_#

When the build option `SHELL_LOGIN` is defined and set to a value greater than 0 login credentials are required to access the shell. Up to 10 triplets of username / password and access type can be added through the triplets `SH_USERNAME_#`, `SH_PASSWORD_#`, and `SH_RDONLY_#`. # in the build option name can have any values 0 to 9 and it is not necessary to have continuous numbering but it is necessary to always define the whole triplet. `SH_USERNAME_#`, `SH_PASSWORD_#` must be defined as strings, which typically involve to use the form \"name\" on the compiler command line. The build option `SH_RDONLY_#` is a Boolean when set to zero allows the username specified in the triplet to have full read and write access. If `SH_RDONLY_#` is set to a non-zero value then the associated username has restricted access only allow it to perform read operations, in other words the user cannot change anything in the RTOS.

2 Commands

All shell commands have an intrinsic help and all there is to do in the shell to get a full details on how to use a command. For example, the help for the command `sem` (semaphore monitoring / debug) is shown in the following table:

Table 2-1 Example of help

```
Abassi> help sem
sem  : semaphore information / processing
usage:
       sem
            show all the semaphores in the application
       sem ##
            dump the semaphore descriptor field memory offsets
       sem <SemName>
            dump info on the semaphore <SemName>
       sem <SemName> post
            post the semaphore <SemName>
                - component SEMpost()
       sem <SemName> wait
            wait on the semaphore <SemName> with a timeout of 0
                - component SEMwait()
       sem <SemName> wait #
            wait on the semaphore <SemName> with a timeout of #
                - component SemWait()
       sem <SemName> reset
            reset the count of the semaphore <SemName>
                - component SEMreset()
       sem <SemName> value #
            set the count of the semaphore <SemName> to #
                - component - none - direct field update
       sem <SemName> abort
            abort the blocking of all tasks on the semaphore <SemName>
                - component SEMabort()
       sem <SemName> order FCFS
            set First-Come-First-Served unblocking order for semaphore <SemName>
                - component SEMsetFCFS()
       sem <SemName> order prio
            set priority unblocking order for the semaphore <SemName>
                - component SEMnotFCFS()
```

Some expressions are used in the help display:

#	Numerical value, either decimal or hexadecimal (“C” representation)
##	This is NOT an expression, ## must be used on the command line
<Name>	Indicate to use the name of an existing service in the application

The available commands for the shell and file system at the time the document was written:

Table 2-2 List of commands

```
List of commands:

help : me
edit : command line control characters
evt  : event operations
```

```
exit : exit from the shell
grp  : group information / processing
log  : logging - not supported (OS_LOGGING_TYPE <= 0)
mblk : memory blocks - not supported (OS_MEM_BLOCK == 0)
mbx  : mailbox information / processing
mem  : memory information / setting
mtx  : mutex information / processing
sem  : semaphore information / processing
sys  : system information
task : task information / processing
tim  : timer services information / processing


File commands:
cat   : Redirect a file to stdout or redirect stdin to a file
cd    : Change directory
chmod : Change a file / directory access modes
cp    : Copy a file
du    : Show disk usage
errno : Read or reset errno
fmt   : Format a drive
        fmt # [FAT16|FAT32|exFAT]
ls    : List the current directory contents
mkdir : Make a new directory
mnt   : Mount a drive to a mount point e.g. mnt 0 /
mv    : Move / rename a file
perf  : Throughput measurements
pwd   : Show current directory
rm    : Remove / delete a file
rmdir : Remove / delete a directory
umnt  : Unmount a mount point
```

3 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] uAbassi RTOS – User Guide, available at http://www.code-time.com
[R4] Abassi RTOS – UART Support, available at http://www.code-time.com