1. Overall architecture of NuttX

1.1 What is an RTOS?

1.1.1 An RTOS as a library

NuttX, as with all RTOSs, is a collection of various features bundled as a library. It does not execute except when either (1) the application calls into the NuttX library code, or (2) an interrupt occurs. There is no meaningful way to represent an architecture that is implemented as an library of user managed functions with an diagram.

You can however, pick any subsystem of an RTOS an represent that in some fashion.

1.1.2 Kernel Threads

There are some RTOS functions that are implemented by internal threads [to be provided].

1.2 The Scheduler

1.2.1 Schedulers and Operating Systems

An operating system is a complete environment for developing applications. One important component of an operating system is a scheduler: That logic that controls when tasks or threads execute. Actually, more than that; the scheduler really determines what a task or a thread is! Most tiny operating systems, such as FreeRTOS are really not operating "systems" is the since or providing a complete operating environment. Rather these tiny operating systems consist really only of scheduler. That is how important the scheduler is.

1.2.1 Task Control Block (TCB)

In NuttX a thread is any controllable sequence of instruction execution that has its own stack. Each task is represented by a data structure called a task control block or `TCB`. That data structure is defined in the header file `include/nuttx/sched.h`.

1.2.2 Task Lists

These TCBs are retained in lists. The state of a task is indicated both by the `task_state` field of the TCB and by a series of task lists. Although it is not always necessary, most of these lists are prioritized so that common list handling logic can be used (only the `g_readytorun`, the `g_pendingtasks`, and the `g_waitingforsemaphore` lists need to be prioritized).

All new tasks start in a non-running, uninitialized state:

`volatile dq_queue_t g_inactivetasks;`

> This the list of all tasks that have been initialized, but not yet activated. NOTE: This is the only list that is not prioritized.

When the task is initialized, it is moved to a read-to-run list. There are two lists representing ready-to-run threads and several lists representing blocked threads. Here are the read-to-run threads:

`volatile dq_queue_t g_readytorun;`

This is the list of all tasks that are ready to run. The head of this list is the currently active task; the tail of this list is always the idle task.

`volatile dq_queue_t g_pendingtasks;`

This is the list of all tasks that are ready-to-run, but cannot be placed in the `g_readytorun` list because: (1) They are higher priority than the currently active task at the head of the g_readytorun list, and (2) the currently active task has disabled pre-emption. These tasks will stay in this holding list until pre-emption is again enabled (or the until the currently active task voluntarily relinquishes the CPU).

Tasks in the `g_readytorun` list may become blocked. In this cased, their TCB will be moved to one of the blocked lists. When the block task is ready-to-run, its TCB will be moved back to either the `g_readytorun` to to the `g_pendingtasks` lists, depending up if pre-emption is disabled and upon the priority of the tasks.

Here are the block task lists:

`volatile dq_queue_t g_waitingforsemaphore;`

This is the list of all tasks that are blocked waiting for a semaphore.

`volatile dq_queue_t g_waitingforsignal;`

This is the list of all tasks that are blocked waiting for a signal (only if signal support has not been disabled)

`volatile dq_queue_t g_waitingformqnotempty;`

This is the list of all tasks that are blocked waiting for a message queue to become non-empty (only if message queue support has not been disabled).

`volatile dq_queue_t g_waitingformqnotfull;`

This is the list of all tasks that are blocked waiting for a message queue to become non-full (only if message queue support has not been disabled).

`volatile dq_queue_t g_waitingforfill;`

This is the list of all tasks that are blocking waiting for a page fill (only if on-demand paging is selected).

(Reference `nuttx/sched/os_internal.h`).

1.2.3 State Transition Diagram

The state of a thread can then be easily represented with this simple state transition diagram:

[to be provided]

1.2.4 Scheduling Policies

In order to be a real-time OS, an RTOS must support `SCHED_FIFO`. That is, strict priority scheduling. The thread with the highest priority runs.... Period. The thread with the highest priority is always associated with the TCB at the head of the `g_readytorun` list.

NuttX supports one additional real-time scheduling policy: `SCHED_RR`. The RR stands for round-robin and this is sometimes called round-robin scheduling. In this case, NuttX supports *timeslicing*: If a task with `SCHED_RR` scheduling policy is running, then when each timeslice elapses, it will give up the CPU to the next task that is at the same priority. Note: (1) If there is only one task at this priority, `SCHED_RR` and `SCHED_FIFO` are the same, and (2) `SCHED_FIFO` tasks are never pre-empted in this way.

1.2.5 Task IDs

Each task is represented not only by a TCB but also by a numeric task ID. Given a task ID, the RTOS can find the TCB; given a TCB, the RTOS can find the task ID. So they are functionally equivalent. Only the task ID, however, is exposed at the RTOS/application interfaces.

1.3 NuttX Tasks

1.3.1Processes vs. Threads

In larger system OS such as Windows or Linux, you will often here the name *processes* used to refer to threads managed by the OS. A process is more than a thread as we have been discussing so far. A process is a protected *environment* that hosts one or more threads. By environment we mean the set of resources set aside by the OS but in the case of the protected environment of the process we are specifically referring its *address space*.

In order to implement the process' address space, the CPU must support a *memory management unit* (MMU). The MMU is used to enforce the protected process environment.

However, NuttX was designed to support the more resource constrained, lower-end, deeply embedded MCUs. Those MCUs seldom have an MMU and, as a consequence, can never support processes as are support by Windows and Linux. So NuttX does not support processes. NuttX will support an MMU but it will not use the MMU to support processes. NuttX operates only in a *flat* address space. (NuttX will use the MMU to control the instruction and data caches and to support protected memory regions).

1.3.2 NuttX Tasks and Task Resources

All RTOSs support the notion of a *task*. A task is the RTOS's moral equivalent of a process. Like a process, a task is a thread with an environment associated with it. This environment is like environment of the process but does not include a private address space. This environment is private and unique to a task. Each task has its own environment

This task environment consists of a number of resources (as represented in the TCB). Of interest in this discussion are the following. Note that any of these task resources may be disabled in the NuttX configuration to reduce the NuttX memory footprint:

1. **Environment Variables**. This is the collection of variable assignments of the form:

   VARIABLE=VALUE

2. **File Descriptors**. A file descriptor is a task specific number that represents an *open* resource (a file or a device driver, for example).

3. **Sockets**. A socket descriptor is like a file descriptor, but the open resource in this case is a network socket.

4. **Streams**. Streams represent standard C buffered I/O. Streams wrap file descriptors or sockets a provide a new set of interface functions for dealing with the standard C I/O (like `fprintf()`, `fwrite()`, etc.).

In NuttX, a task is created using the interface `task_create()`. Reference: http://nuttx.sourceforge.net/NuttxUserGuide.html#taskcreate

1.3.3 The Pseudo File System and Device Drivers

A full discussion of the NuttX file system belongs elsewhere. But in order to talk about task resources, we also need to have a little knowledge of the NuttX file system

NuttX implement a *Virtual Files System* (VFS) that may be used to communicate with a number of different entities via the standard `open()`, `close()`, `read()`, `write()`, etc. interfaces. Like other VFSs, the NuttX VFS will support file system mount points, files, directories, device drivers, etc.

Also, as with other VFSs, the NuttX file system will support psuedo-file systems, that is, file systems that appear as normal media but are really presented under programmatic control. In Linux, for example, you have the `/proc` and the `/sys` psuedo-file systems. There is no physical media underlying the pseudo-file system.

The NuttX root file system is always a psuedo-file system. This is just the opposite from Linux. With Linux the root file system must always be some physical block device (if only an initrd ram disk). Then once you have mounted the physical root file system, you can mount other file systems -- including Linux psuedo-filesystems like `/proc` or `/sys`. With NuttX, the root file system is always a pseudo-file system that does not require any underlying block driver or physical device. Then you can mount real filesystem in the pseudo-filesystem.

This arrangement makes life much easier for the tiny embedded world (but also has a few limitations --- like where you can mount file systems).

NuttX interacts with devices via device drivers – that is via software that controls hardware and conforms to certain NuttX conventions (see include/nuttx/fs/fs.h). Device drivers are represented by *device nodes* in the pseudo-file system. By convention, these device nodes are created in the `/dev` directory.

Now that have digressed a little to introduce the NuttX file system and device nodes, we can return to our discussion of task resources.

1.3.4 `/dev/console` and Standard Streams

There are three special cases of I/O: `stdin`, `stdout`, and `stderr`. These are type FILE* and correspond to file descriptors 0, 1, and 2 respectively. When the very first thread is created (called the IDLE thread), the special device node `/dev/console` is opened. `/dev/console` provides the `stdin`, `stdout`, and `stderr` for the initial task

1.3.5 Inheritance of the Task Environment and I/O Redirection

When one task creates a new task, that new task inherits the task resources of its parent. This includes all of the environment variables, file descriptors, and sockets (NOTE: This inheritance can be limited by special options in the NuttX configuration).

So, if nothing special is done, then every task will use `/dev/console` for the standard I/O. However, a task may close file descriptor 0 through 2 and open a new device for standard I/O. Then any children tasks that are created will inherit that new re-directed standard I/O as well. This mechanism is used throughout NuttX. As examples:

- In the THTTPD server to redirect socket I/O to standard I/O for CGI tasks,
- In the Telnet server so that new tasks inherit the Telnet session.

1.3.6 Tasks vs. Pthreads

Systems like Linux also support POSIX *pthreads*. In the Linux environment, the process is created with one thread running in it. But by using interfaces like `pthread_create()`, you can create multiple threads that run and share the same process resources.

NuttX also supports POSIX pthreads and the NuttX pthreads also support this behavior. That is, the NuttX POSIX pthreads also share the resources of the parent task. However, since NuttX does not support process address environments, the difference is not so striking. When a task creates a pthread, the newly create pthread will share the environment variables, file descriptors, sockets, and streams of the parent task.

Note: These task resources are reference counted and will persist as long as thread in the task group is still active.

1.3.7 Process IDs/task IDs/pthread IDs

The term *process ID* is standard (usually abbreviated as `pid`) and used to identify a task in NuttX. So, more technically, this number is a task ID as was described above. Pthreads are also described by a `pthread_t` ID. In NuttX, the `pthread_t` ID is also the same task ID.

2. The NuttX Initialization Sequence

2.1 Overview

At the highest level, the NuttX initialization sequence can be represented in three phases:

1. The hardware-specific power-on reset initialization,
2. NuttX RTOS initialization, and
3. Application Initialization.

This initialization sequence is really quite simple because the system runs in single-thread mode up until the point the that is starts the application. That means that the initialization sequence is just simple, straight-line function calls.

Just before starting the application, the system goes to multi-threaded mode and things can get more complex.

Each of these will be discussed in more detail in the following paragraphs.

2.2 Power-On Reset Initialization.

2.2.1 Overview

The software begins execution when the processor is reset. This usually at power-on, but all resets are basically the same where they occur because of power-on, pressing the reset button, or on a watchdog timer expiration. The software that executes when the processor is reset is unique to the particular CPU architecture and is not a common part of NuttX. The kinds of things that must be done by the architecture-specific reset handling includes:

1. Putting the processor in its operational state. This may include things like setting CPU modes; initializing co-processors, etc.
2. Setting up clocking so that the software and peripherals operate as expected,
3. Setting up the C stack pointer (and other processor registers)
4. Initializing memory, and
5. Starting NuttX.

2.2.2 Memory Initialization

In C implementations, there are two general classes of variable storage. First there are the initialized variables. For example, consider the global variable `x`:

```
int x = 5;
```

The C code must be assured that after reset, the variable `x` has the value 5. Initialized variable of this kind are retained in a special memory section called *data* (or `.data`).

Other variables are not initialized. Like the global variable `y`:

```
int y;
```

But the C code will still expect y to have an initial value. That initial value will be zero. All uninitialized variables of this this type have have the value zero. These uninitialized variables are retained in a section called *bss* (or `.bss`).

When we say that the reset handling logic initializes memory, we mean two things:

1. It provides the (initial) values of the initialized variables by copying the values from FLASH into the `.data` section, and
2. It resets all of the uninitialized variables to zero. It clears the `.bss` section.

2.2.3 STM32 F4 Reset

Lets walk through reset sequence of one particular processor. Let's look at the NuttX initialization for the STM32 F4 MCU. This reset logic can be found in to files:

1. `nuttx/arch/arm/src/stm32_vectors.S`
2. `nuttx/arch/arm/src/stm32_start.c`

`nuttx/arch/arm/src/stm32_vectors.S`

The roll of stm32_vectors.S in this reset sequence is very small. This file provides all of the STM32 exception vectors and power-on reset is simply another exception vector. Some important things to note about this file:

1. `.section  .vectors, "ax"`. This pseudo operation will place all of the vectors into a special section call .vectors. On of the STM32 F4 linker scripts is located at nuttx/configs/stm3240g-eval/nsh/ld.script. In that file, you can see that , you can see that the section .vectors is forced to lie at the very beginning of FLASH memory.

   The STM32 F4 can be configured to boot in different ways via strapping. If it is strapped to boot from FLASH, then the STM32 FLASH memory will be aliased to address 0x0000 0000 when the reset occurs. The is the address of the power-up reset interrupt vector.

2. The first two 32-bit entries in the vector table represent the power-up exception vector (which we know will be positioned at address 0x0000 0000 when the reset occurs). Those two entries are:

   ```
   .word IDLE_STACK /* Vector  0: Reset stack pointer */
   .word __start    /* Vector  1: Reset vector */
   ```

The Cortex-M family is unique in the way that is handles the reset vector. Notice that there are two values: the stack pointer for the start-up thread (the IDLE thread), and the entry point in the IDLE thread. When the reset occurs, the the stack pointer is automatically set to the first value and then the processor jumps to reset entry point `__start` specified in the second entry. This means that the reset exception handling code can be implemented in C rather than assembly language.

`nuttx/arch/arm/src/stm32_start.c`

The reset vector `__start` lies in the file `stm32_start.c` and does the real, low-level architecture-specific initialization. This initialization includes:

- `stm32_clockconfig();` Initialize the PLLs and peripheral clocking needed by the board.
- `stm32_fpuconfig();` If the STM32 F4's hardware floating point is initialized, then configure the FPU and enable access to the FPU co-processors.
- `stm32_lowsetup();` Enable the low-level UART. This is done very early in initialization so that we can get serial debug output to the console as soon as possible. If you are doing a board bring-up this is very important.
- `stm32_gpioinit();` Perform any GPIO remapping that is needed (this is a stub for the F4, but the F1 family requires this step).
- `showprogress('A');` This simply outputs the character 'A' on the serial console (only if `CONFIG_DEBUG` is enabled). If debug is enabled, you will always see the letters ABDE output on the console. That output all comes from this file.

Next the memory is initialized:

- The .bss section is set to zero (Letter 'B' is then output if CONFIG_DEBUG is enabled), then
- The .data section is set to its initial values (The letter 'C' is output if debug is enabled),

Then board-specific logic is initialized:

- `stm32_boardinitialize();` This function resides with the board-specific logic. For the case of the STM3240G-EVAL board, this board initialization logic can be found at `configs/stm3240g-eval/src/up_boot.c`.

For the case of the STM3240G-EVAL board, the stm32_boardinitialize() does the following operations:

- `stm32_spiinitialize();` Initialize SPI chip selects if SPI is enabled.
- `stm32_selectsram();` Configure the STM32 FSMC to support external SRAM if external SRAM support is enabled.
- `up_ledinit();` Initialize the on-board LEDs if they are used.

When `stm32_boardinitialize()` returns to `__start()`, the low-level, architecture-specific initialization is complete and NuttX is started:

- `os_start();` This is the NuttX entry point. It performs the next phase of RTOS-specific initialization and then brings up the application.

The operations performed by `os_start()` are discussed in the next paragraph.

2.3 NuttX RTOS Initialization.

2.3.1 `os_start()`

When the ow-level, architecture-specific initialization is complete and NuttX is started by calling the function `os_start()`. This function resides in the file `nuttx/sched/os_start.c`. The operations performed by `os_start()` are summarized below. Note that many of these features can be disabled from the NuttX configuration file and in that case those operations are not performed:

1. Initializes some NuttX global data structures,
2. Initializes the TCB for the IDLE (i.e, the thread that the initialization is performed on),
3. `kmm_initialize();` Initialize the memory manager (in most configurations, `kmm_initialize()` is an alias for the common `mm_initialize()`).
4. `irq_initialize();` Initialize the interrupt handler subsystem. This initializes only data structures; CPU interrupts are still disabled.
5. `wd_initialize();` Initialize the NuttX watchdog timer facility,
6. `clock_initialize();` Initialize the system clock,
7. `timer_initialize();` Initialize the POSIX timer facilities,
8. `sig_initialize();` Initialize the POSIX signal facilities,
9. `sem_initialize();` Initialize the POSIX semaphore facilities,
10. `mq_initialize();` Initialize the POSIX message queue facilities,
11. `pthread_initialize();` Initialize the POSIX pthread facilities,
12. `fs_initialize();` Initialize file system facilities,
13. `net_initialize();` Initialize networking facilities,

Up to this point, all of the initialization steps have only been software initializations. Nothing has interacted with the hardware. Rather, all of these steps simply prepared the environment so that things like interrupts and threads can function properly. The next phases depend upon that setup.

14. `up_initialize();` The processor specific details of running the operating system will be handled here. Such things as setting up interrupt service routines and starting the clock are some of the things that are different for each processor and hardware platform. See below for a specific example of the initialization steps performed by the ARM version of this function.

Then,

15. `lib_initialize();` Initialize the C libraries. This is done last because the libraries may depend on the above.
16. `sched_setupidlefiles();` This is the logic that opens `/dev/console` and creates `stdin`, `stdout`, and `stderr` for the IDLE thread. All tasks subsequently created by the IDLE thread will inherit these file descriptors.
17. `os_bringup();` Create the initial tasks. This will be described more below.
18. And finally enter the IDLE loop. After completing the initialization, the roll of the IDLE thread changes. It is now becomes the thread that executes only when there is nothing else to do in the system (hence, the name IDLE thread).

2.3.2 IDLE Thread Activities.

As mention, the IDLE thread is the thread that executes only when there is nothing else to do in the system. It has the lowest priority in the system. It always has the priority 0. It is the only thread that is permitted to have the priority 0. And it can never be blocked (otherwise, what would run then?).

As a result, the IDLE thread is always in the `g_readytorun` list and, in fact, since that list is prioritized, can guaranteed to always be the final entry at the tail of the `g_readytorun` list.

The IDLE is an an infinite loop. But this does not make it a "CPU hog." Since it is the lowest priority, the it can be suspended whenever anything else needs to run.

The IDLE thread does two things in this infinite loop:

1. If the worker was not started (see `os_bringup()` below), then the IDLE thread will perform memory clean-up. Memory clean is required to handle deferred memory deallocation. Memory allocations must be deferred when the memory is freed in a context where the software does not have access to the heap and, hence, cannot truly free the memory (such as in an interrupt handler). In this case, the memory is simply put into a list of freed memory and, eventually, cleaned up by the IDLE thread.

   NOTE: The worker thread's primary function is as the "bottom half" for extended device driver processing. If the worker thread was started, then it will run at a higher priority than the IDLE thread. In this case, the worker thread will take over responsibility for cleaning up these deferred allocations.

2. `up_idle();` Then the loop calls `up_idle()`. The operations performed by `up_idle()` are architecture- and board-specific. In general, this is the location where CPU-specific reduced power operations may be performed.

2.3.3 `os_bringup()`

This function is called at the very end of the initialization sequence in `os_start()`, just before entering the IDLE loop. This function is located in `nuttx/sched/os_bringup.c`. This function starts all of the required threads and tasks needed to bring up the system. This function performed the following specific operations:

1. If on-demand paging is configured, this function will start the page fill task. This is the task that runs in order to satisfy page faults in processors that have an MMU and in configurations where on-demand paging is enabled.
2. The, if so configured, this function starts the worker thread. The worker thread may be used to execute any processing deferred to the worker thread via APIs provided in `include/nuttx/wqueue.h`. The worker thread's primary function is as the "bottom half" for extended device driver processing but can be used for a variety of purposes.
3. Finally, `os_bringup()` will start the application task. Be default this is the task whose entry has the name `user_start()`. user_start() is provided by application code and when it runs, it begins the application-specific phase of the initialization sequence as described below.

NOTE: The default `user_start()` entry point can be changed to use one of the named applications used by NSH. This is a start-up option that is not often used and will not be discussed further here.

2.3.2 STM32 F4 up_initialize()

All ARM-based MCUs share a common `up_initialize()` implementation provided at `nuttx/arch/arm/common/up_initialize.c`. The operations perform by this common ARM initialization will, however, call into facilities provided by the particular ARM chip. For the STM32 F4, those facilities would be provided by logic in files as `nuttx/arch/arm/src/stm32`. The common ARM initialization sequence is:

1. `up_calibratedelay();` One operation that must be performed during a CPU port is the calibration of timing delay loops. If `CONFIG_ARCH_CALIBRATION` is defined, then `up_initialize()` will perform some specific operations for the calibration of the delay loop. This, however, is not part of the normal initialization sequence. `up_calibratedelay()` is implemented within `up_initialize.c`.
2. `up_addregion();` The basic heap was set up during processing by os_start(). However, if the board supports multiple, discontiguous memory regions, any addition memory regions can by added to the heap by this function. For the STM32 F4, `up_addregion()` is implemented in `nuttx/arch/arm/src/stm32/stm32_allocateheap.c`.
3. `up_irqinitialize();` This function initialize the interrupt subsystem. For the STM32 F4, `up_irqinitialize()` is implemented in `nuttx/arch/arm/src/stm32_irq.c`.
4. `up_pminitialize();` If `CONFIG_PM` is defined, the function must initialize the power management subsystem. This MCU-specific function must be called *very* early in the initialization sequence *before* any other device drivers are initialized (since they may attempt to register with the power management subsystem). There is no implementation of `up_pminitialize()` for any STM32 platform.
5. `up_dmainitialize();` Initialize the DMA subsystem. For the STM32 F4, this DMA initialization can be found in `nuttx/arch/arm/src/stm32_dma.c` (which includes `nuttx/arch/arm/src/stm32f4xxx_dma.c`).
6. up_timerinit(); Initialize the system timer interrupt. For the STM32 F4, this function initializes the ARM Cortex-M SYSTICK timer and can be found at `nuttx/arch/arm/src/stm32_timerisr.c`.
7. `devnull_register();` Registers the standard `/dev/null`.
8. Then this function initializes the console device (if any). This means calling one of (1) `up_serialinit();` for the standard serial driver (found at `nuttx/arch/arm/src/stm32_serial.c` for the STM32 F4), (2) `lowconsole_init();` for the low-level, write-only serial console (found at `nuttx/drivers/serial/lowconsole_init.c`), or (2) `ramlog_sysloginit()` for the RAM console (found at `nuttx/drivers/ramlog.c`).
9. `up_netinitialize();` Initialize the network. For the STM32 F4, this function is in `nuttx/arch/arm/src/stm32_eth.c`.
10. `up_usbinitialize();` Initialize USB (host or device). For the STM32 F4, this function is in `nuttx/arch/arm/src/stm32_otgfsdev.c`.
11. `up_ledon(LED_IRQSENABLED);` Finally, `up_initialize()` illuminates board-specific LEDs to indicate the IRQs are now enabled.

2.3.2 STM32 F4 IDLE thread

The default STM32 F4 IDLE thread is located at `nuttx/arch/arm/src/stm32_idle.c`. This default version does very little:

1. It includes a example, “skeleton” function that illustrates that kinds of things that you can do if `CONFIG_PM` is enabled (this example code is not fully implemented in the default IDLE logic).
2. The it executes the Cortex-M thumb2 instruction *`wfi`* which causes the CPU to sleep until the next interrupt occurs.

3.4 Application Initialization

At the conclusion of the OS initialization phase in `os_start()`, the user application is started by creating a new task at the entry point `user_start()`. There must be exactly one entry point called `user_start()` in every application built on top of NuttX. Any additional initialization performed in the `user_start()` function is purely application dependent.

3.4.1 A Simple Hello World Application

The simplest user application would be the “Hello, World!” example. See `apps/examples/hello`. Here is the whole example:

```
int user_start(int argc, char *argv[])
{
  printf("Hello, World!!\n");
  return 0;
}
```

In this case, no additional application initialization is needed. It just “says hello” and exits.

3.5 Building NuttX with Board-Specific Pieces Outside the Source Tree

There are at least four ways to build NuttX with board-specific pieces outside the source tree:

1. ***make export***. There is a make target called *make export*. It will build NuttX, then bundle all of the header files, libraries, start-up objects, and other build components into a .zip file. You can can move that .zip file into any build environment you want. You even build NuttX under a DOS CMD window.

   This make target is documented in the top level `nuttx/README.txt` file.

2. **Replace `apps/`.** You can replace the entire `apps/` directory. If there is nothing in the `apps/` directory that you need, you can define `CONFIG_APPS_DIR` in your `.config` file so that it points to a different, custom application directory. You can copy any pieces that you like from the old `apps/` directory to your new, custom apps directory as necessary.

   This is documented in `NuttX/configs/README.txt` and `nuttx/Documentation/NuttxPortingGuide.html` (Online at http://nuttx.sourceforge.net/NuttxPortingGuide.html#apndxconfigs under Build options). And in the `apps/README.txt` file.

3. **`external` Sub-directory**. If you like the random collection of stuff in the apps/ directory but just want to expand the existing components with your own, external sub-directory then there is an easy way to that too: You just create the symbolic link at apps/external that redirects to your application sub-directory. The apps/Makefile will always automatically check for the existence of an apps/external directory and if it exists, it will automatically incorporate it into the build (you do not have to add it to the apps/.config file).

   This feature of the apps/Makefile is documented only here.

   You can, for example, create a script called `install.sh` that installs a custom application, configuration, and board specific directory. This script might do something like the following:

   - Copy your `MyBoard` directory to `configs/MyBoard`.
   - Add a symbolic link to `MyApplication` at `apps/external`
   - Configure NuttX usually by executing:

     ```
     tools/configure.sh MyBoard/MyConfiguration
     ```

     or simply by copying

     ```
     defconfig->nutt.config,
     setenv.sh->nuttx/setenv.sh, Make.defs->nuttx/Make.defs,
     appconfig->apps/.config
     ```

   Using the 'external' link makes it especially easy to add a 'built-in' application an existing configuration.

4. **Custom Links**. You add as many symbolically linked directories to apps/ as you would like:

   - Add symbolic links in `apps/` to as many other directories as you want.
   - Then just add the (relative) paths to the links in your `appconfig` file (that becomes the `apps/.config` file).

   That is basically the same as option #3 but doesn't use the magic `external` link and allows to add as many linked sub-directories as you want. The top-level `apps/Makefile` will always to build whatever in finds in the `apps/.config` file (plus the `external` link if present).

4 The NuttShell (NSH)

4.1 Overview

The NuttShell (NSH) is a simple shell application that may be used with NuttX. It is described here: http://nuttx.sourceforge.net/NuttShell.html. It supports a variety of commands and is (very) loosely based on the bash shell and the common utilities used in Unix shell programming. That reference provides a good overview of NSH and will not be duplicated here. Instead, the paragraphs in this section will focus on customizing NSH: Adding new commands, changing the initialization sequence, etc.

4.2 The NSH Library and NSH Initialization

NSH is implemented as a library that can be found at `apps/nshlib`. As a library, it can be custom built into any application that follows the NSH initialization sequence described below. As an example, the code at `apps/examples/nsh/nsh_main.c` illustrates how to start NSH and the logic there was intended to be incorporated into your own custom code. Although code was generated simply as an example, in the end most people just use this example code as their application `main()` function. That initialization performed by that example is discussed in the following paragraphs.

4.2.1 NSH Initialization sequence

The NSH start-up sequence is very simple. As an example, the code at `apps/examples/nsh/nsh_main.c` illustrates how to start NSH. It simple does the following:

1. If you have C++ static initializers, it will call your implementation of `up_cxxinitialize()` which will, in turn, call those static initializers. For the case of the STM3240G-EVAL board, the implementation of `up_cxxinitialize()` can be found at `nuttx/configs/stm3240g-eval/src/up_cxxinitialize.c`.
2. This function then calls `nsh_initialize()` which initializes the NSH library. `nsh_initialize()` is described in more detail below.
3. If the Telnet console is enabled, it calls `nsh_telnetstart()` which resides in the NSH library. `nsh_telnetstart()` will start the Telnet daemon that will listen for Telnet connections and start remote NSH sessions.
4. If a local console is enabled (probably on a serial port), then `nsh_consolemain()` is called. `nsh_consolemain()` also resides in the NSH library. `nsh_consolemain()` does not return so that finished the entire NSH initialization sequence.

4.2.2 `nsh_initialize()`

The NSH initialization function, nsh_initialize(), be found in `apps/nshlib/nsh_init.c`. It does only three things:

1. `nsh_romfsetc();` If so configured, it executes an NSH start-up script that can be found at `/etc/init.d/rcS` in the target file system. The `nsh_romfsetc()` function can be found in `apps/nshlib/nsh_romfsetc.c`. This function will (1) register a ROMFS file system, then (2) mount the ROMFS file system. `/etc` is the default location where a read-only, ROMFS file system is mounted by `nsh_romfsetc()`.

The ROMFS image is, itself, just built into the firmware. By default, this rcS start-up script contains the following logic:

```
# Create a RAMDISK and mount it at XXXRDMOUNTPOUNTXXX

mkrd -m XXXMKRDMINORXXX -s XXMKRDSECTORSIZEXXX XXMKRDBLOCKSXXX
mkfatfs /dev/ramXXXMKRDMINORXXX
mount -t vfat /dev/ramXXXMKRDMINORXXX XXXRDMOUNTPOUNTXXX
```

Where the `XXXX*XXXX` strings get replaced in the template when the ROMFS image is created:

- `XXXMKRDMINORXXX` will become the RAM device minor number. Default: 0
- `XXMKRDSECTORSIZEXXX` will become the RAM device sector size
- `XXMKRDBLOCKSXXX` will become the number of sectors in the device.
- `XXXRDMOUNTPOUNTXXX` will become the configured mount point. Default: `/etc`

By default, the substituted values would yield an rcS file like:

```
# Create a RAMDISK and mount it at /tmp

mkrd -m 1 -s 512 1024
mkfatfs /dev/ram1
mount -t vfat /dev/ram1 /tmp
```

This script will, then:

- Create a RAMDISK of size 512*1024 bytes at /dev/ram1,
- Format a FAT file system on the RAM disk at /dev/ram1, and then
- Mount the FAT filesystem at a configured mountpoint, /tmp.

This rcS template file can be found at `apps/nshlib/rcS.template`. The resulting ROMFS file system can be found in `apps/nshlib/nsh_romfsimg.h`.

2. `nsh_archinitialize();` Next any architecture-specific NSH initialization will be performed (if any). For the STM3240G-EVAL, this architecture specific initialization can be found at `configs/stm3240g-eval/src/up_nsh.c`. This it does things like: (1) Initialize SPI devices, (2) Initialize SDIO, and (3) mount any SD cards that may be inserted.

3. `nsh_netinit();` The `nsh_netinit()` function can be found in `apps/nshlib/nsh_netinit.c`.

4.4 NSH Commands
4.4.1 NSH Command Overview

NSH supports a variety of commands as part of the NSH program. All of the NSH commands are listed in the NSH documentation at http://nuttx.sourceforge.net/NuttShell.html#cmdoverview. Not all of these commands may be available at any time, however. Many commands depend upon certain NuttX configuration options. You can enter the help command at the NSH prompt to see the commands actual available:

```
nsh> help
```

For example, if network support is disabled, then all network-related commands will be missing from the list of commands presented by '`nsh> help`'. You can the specific command dependencies in the table at:
file:///c:/cygwin/home/patacongo/projects/nuttx/nuttx/trunk/nuttx/Documentation/NuttShell.html#cmddependencies.

4.4.1 Adding New NSH Commands

New Commands can be added to the NSH very easily. You simply need to add two things:

1. The implementation of your command, and
2. A new entry in the NSH command table

Implementation of Your Command. For example, if I want to add a new a new command called `mycmd` to NSH, you would first implement the `mycmd` code in a function with this prototype:

```
int cmd_mycmd(FAR struct nsh_vtbl_s *vtbl, int argc, char **argv);
```

The `argc` and `argv` are used to pass command line arguments to the NSH command. Command line parameters are passed in a very standard way: `argv[0]` will be the name of the command, and `argv[1]` through `argv[argc-1]` are the additional arguments provided on the NSH command line.

The first parameter, `vtbl`, is special. This is a pointer to session-specific state information. You don't need to know the contents of the state information, but you do need to pass this `vtbl` argument when you interact with the NSH logic. The only use you will need to make of the `vtbl` argument will be for outputting data to the console. You don't use `printf()` within NSH commands. Instead you would use:

```
void nsh_output(FAR struct nsh_vtbl_s *vtbl, const char *fmt, …);
```

So if you only wanted to output “Hello, World!” on the console, then your whole command implementation might be:

```
int cmd_mycmd(FAR struct nsh_vtbl_s *vtbl, int argc, char **argv);
{
      nsh_output(vtbl, “Hello, World!”);
      return 0;
}
```

The prototype for the new command should be placed in `apps/examples/nshlib/nsh.h`.

Adding You Command to the NSH Command Table. All of the commands support by NSH appear in a single table called:

```
const struct cmdmap_s g_cmdmap[]
```

That can be found in the file `apps/examples/nshlib/nsh_parse.c`. The structure `cmdmap_s` is also `defined in apps/nshlib/nsh_parse.c`:

```
struct cmdmap_s
{
  const char *cmd;        /* Name of the command */
  cmd_t       handler;    /* Function that handles the command */
  uint8_t     minargs;    /* Minimum number of arguments (including command) */
  uint8_t     maxargs;    /* Maximum number of arguments (including command) */
  const char *usage;      /* Usage instructions for 'help' command */
};
```

This structure provides everything that you need to describe your command: It name (`cmd`), the function that handles the command (`cmd_mycmd()`), the minimum and maximum number of arguments needed by the command, and a string describing the command line. That last string is what is printed when enter "`nsh> help`".

So, for you sample commnd, you would add the following the to the g_cmdmap[] table.

```
    { "mycmd", cmd_mycmd, 1, 1, NULL },
```

This entry is particularly simply because `mycmd` is so simple. Look at the other commands in `g_cmdmap[]` for more complex examples.

4.5 NSH "Built-In" Applications

4.5.1 Overview

In addition to these commands that are a part of NSH, external programs can also be executed as NSH commands. These external programs are called "Built-In" Applications for historic reasons. That terminology is somewhat confusing because the actual NSH commands as described above are truly "built-into" NSH whereas these applications are really external to NuttX.

These applications are built-into NSH in the sense that they can be executed by simply typing the name of the application at the NSH prompt. Built-in application support is enabled with the configuration option CONFIG_NSH_BUILTIN_APPS. When this configuration option is set, you will also be able to see the built-in applications if you enter "nsh> help". They will appear at the bottom of the list of NSH commands under:

```
    Builtin Apps:
```

Note that no detailed help information beyond the name of the built-in application is provided.

4.5.2 Named Applications

Overview. The underlying logic that supports the NSH built-in applications is called "Named Applications". The named application logic can be found at `apps/namedapp`. This logic simply does the following:

1. It supports registration mechanism so that named applications can dynamically register themselves at build time, and
2. Utility functions to look up, list, and execute the named applications.

Named Application Utility Functions. The utility functions exported by the named application logic are prototyped in `apps/include/apps.h`. These utility functions include:

- `int namedapp_isavail(FAR const char *appname);` Checks for availability of application registered as `appname` during build time.

- `const char *namedapp_getname(int index);` Returns pointer to a name of built-in application pointed by the `index`. This is the utility function that is used by NSH in order to list the available built-in applications when “`nsh> help`” is entered.

- `int exec_namedapp(FAR const char *appname, FAR const char **argv);` Executes built-in named application registered during compile time. This is the utility function used by NSH to execute the built-in application.

Autogenerated Header Files. Application entry points with their requirements are gathered together in two files when NuttX is first built:

1. `apps/namedapp/namedapp_proto.h`: Prototypes of application task entry points.
2. `apps/namedapp/namedapp_list.h`: Application specific information and start-up requirements

Registration of Named Applications. The NuttX build occurs in several phases as different build targets are executed: (1) *context* when the configuration is established, (2) *depend* when target dependencies are generated, and (3) *default* (`all`) when the normal compilation and link operations are performed. Named application information is collected during the make *context* build phase.

An example application that can be “built-in” is be found in the apps/examples/hello directory. Let's walk through this specific cause to illustrate the general way that built-in applications are created and how they register themselves so that they can be used from NSH.

`apps/examples/hello`. The main routine for apps/examples/hello can be found in `apps/examples/hello/main.c`. When `CONFIG_EXAMPLES_HELLO_BUILTIN` is defined, this main routine simplifies to:

```
int hello_main(int argc, char *argv[])
{
  printf("Hello, World!!\n");
  return 0;
}
```

This is the built in function that will be registered during the *context* build phase of the NuttX build. That registration is performed by logic in `apps/examples/hello/Makefile`. But the build system gets to that logic through a rather tortuous path:

1. The top-level context make target is in `nuttx/Makefile`. All build targets depend upon the *context* build target. For the `apps/` directory, this build target will execute the *context* target in the `apps/Makefile`.
2. The `apps/Makefile` will, in turn, execute the *context* targets in all of the configured sub-directories. In our case will include the `Makefile` in `apps/examples`.
3. And finally, the `apps/examples/Makefile` will execute the *context* target in all configured `example` sub-directores, getting us finally to `apps/examples/Makefile` (which is covered below).
4. At the conclusion of the *context* phase, the `apps/Makefile` will touch a file called `.context` in the `apps/` directory, preventing any further configurations during any subsequent *context* phase build attempts.

NOTE: Since this context build phase can only be executed one time. Any subsequent configuration changes that you make will, then, not be reflected in the build sequence. That is a common area of confusion. Before you can instantiate the new configuration, you have to first get rid of the old configuration. The most drastic way to this is:

```
make distclean
```

But then you will have to re-configuration NuttX from scratch. But if you only want to re-build the configuration in the apps/ sub-directory, then there is a less labor-intensive way to do that. The following NuttX make command will remove the configuration only from the apps/ directory and will let you continue without re-configuring everything:

```
make apps_distclean
```

Logic for the context target in `apps/examples/hello/Makefile` registers the hello_main() appliation in the namedapp's `namedapp_proto.h` and `namedapp_list.h` files. That logic that does that in `apps/examples/hello/Makefile` is abstracted below:

1. First, the `Makefile` includes `apps/Make.defs`:

   ```
   include $(APPDIR)/Make.defs
   ```

   This defines a macro called REGISTER that adds data to the *namedapp* header files:

   ```
   define REGISTER
           @echo "Register: $1"
           @echo "{ \"$1\", $2, $3, $4 }," >> "$(APPDIR)/namedapp/namedapp_list.h"
           @echo "EXTERN int $4(int argc, char *argv[]);" >> "$(APPDIR)/namedapp/namedapp_proto.h"
   endef
   ```

   When this macro runs, you will see the output in the build "`Register: hello`", that is a sure sign that the registration was successful.

2. The make file then defines the application name (`hello`), the task priority (default), and the stack size that will be allocated in the task runs (2Kb).

   ```
   APPNAME         = hello
   PRIORITY        = SCHED_PRIORITY_DEFAULT
   STACKSIZE       = 2048
   ```

3. And finally, the `Makefile` invokes the `REGISTER` macro to added the `hello_main()` named application. Then, when the system build completes, the `hello` command can be executed from the NSH command line. When the `hello` command is executed, it will start the task with entry point `hello_main()` with the default priority and with a stack size of 2Kb.

   ```
   .context:
   ifeq ($(CONFIG_EXAMPLES_HELLO_BUILTIN),y)
           $(call REGISTER,$(APPNAME),$(PRIORITY),$(STACKSIZE),$(APPNAME)_main)
           @touch $@
   endif
   ```

Other Uses of Named Application. The primary purpose of named applications is to support command line execution of applications from NSH. However, there are two other uses of named applications that should be mentioned.

1. **Named Start-Up `main()` function**. A named application can even be used as the main, start-

up entry point into your embedded software. When the user defines this option in the NuttX configuration file:

```
CONFIG_BUILTIN_APP_START=<application name>
```

that application will be invoked immediately after system starts instead of the normal, default `user_start()` entry point. Note that `<application name>` must be provided just as it would have been on the NSH command line. For example, `hello` would result in `hello_main()` being started at power-up.

This option might be useful in some develop environments where you use NSH only during the debug phase, but want to eliminate NSH in the final product. Setting `CONFIG_BUILTIN_APP_START` in this way will bypass NSH and execute your application just as if it were entered from the NSH command line.

2. ***binfs***. *binfs* is a tiny file system located at `apps/namedapp/binfs.c`. This provides an alternative what of visualizing installed named applications. Without *binfs*, you can see the installed named applications using the NSH help command. *binfs* will create a tiny pseudo-file system mounted at `/bin`. Using *binfs*, you can see the available named applications by listing the contents of `/bin` directory. This gives some superficial Unix compatibility, but does not really and any new functionality.

4.5.3. Synchronous Built-In Applications

By default, built-in commands started from the NSH command line will run asynchronously with NSH. If you want to force NSH to execute commands then wait for the command to execute, you can enable that feature by adding the following to the NuttX configuration file:

```
CONFIG_SCHED_WAITPID=y
```

The configuration option enables support for the standard `waitpid()` RTOS interface. When that interface is enabled, NSH will use it to wait, sleeping until the built-in application executes to completion.

Of course, even with `CONFIG_SCHED_WAITPID=y` defined, specific applications can still be forced to run asynchronously by adding the ampersand (&) after the NSH command.

4.5.4 Application Configuration File

The appconfig File. A special configuration file is used to configure which applications are to be included in the build. The source for this file is saved at `configs/<board>/<configuration>/appconfig`. The existence of the `appconfig` file in the board configuration directory is sufficient to enable building of applications.

The `appconfig` file is copied into the `apps/` directory as `.config` when NuttX is configured. `.config` is included by the top-level `apps/Makefile`. As a minimum, this configuration file must define files to add to the `CONFIGURED_APPS` list like:

```
CONFIGURED_APPS += examples/hello
```

Changes in the Works. There are changes in the works that will obsolete the `appconfig` file. These changes will implement an automated configuration system for NuttX. One consequence of this new configuration system is that the `appconfig` file will become obsolete and will be replaced by a new mechanism for selecting applications. This new mechanism is not yet available, but is dicussed here: http://tech.groups.yahoo.com/group/nuttx/message/1604.

4.6 Customizing NSH Initialization

4.6.1 Ways to Customize NSH Initialization

There are three ways to customize the NSH start-up behavior. Here they are presented in order of increasing difficulty:

1. You can extend the initialization logic in `configs/stm3240g-eval/src/up_nsh.c.` The logic there is called each time that NSH is started and is good place in particular for any device-related initialization.
2. You replace the sample code at `apps/examples/nsh/nsh_main.c` with whatever start-up logic that you want. NSH is a library at `apps/nshlib`. `apps.examplex/nsh` is just a tiny start-up function (`user_start()`) that that runs immediately and starts NSH. If you want something else to run immediately then you can write your write your own custom `user_start()` function and then start other tasks from your custom `user_start()`.
3. NSH also supports a start-up script that executed when NSH first runs. This mechanism has the advantage that the start-up script can contain any NSH commands and so can do a lot of work with very little coding. The disadvantage is that is is considerably more complex to create the start-up script. It is sufficiently complex that is deserves its own paragraph.

4.6.2 NuttShell Start up Scripts

First of all you should look at file:///c:/cygwin/home/patacongo/projects/nuttx/nuttx/trunk/nuttx/Documentation/NuttShell.html#startupscript. Most everything you need to know can be found there. That information will be repeated and extended here for completeness.

NSH Start-Up Script. NSH supports options to provide a start up script for NSH. The start-up script contains any command support by NSH (i.e., that you see when you enter 'nsh> help'). In general this capability is enabled with `CONFIG_NSH_ROMFSETC=y`, but has several other related configuration options as described with the NSH-specific configuration settings (file:///c:/cygwin/home/patacongo/projects/nuttx/nuttx/trunk/nuttx/Documentation/NuttShell.html#nshconfiguration). This capability also depends on:

- `CONFIG_DISABLE_MOUNTPOINT=n`. If mount point support is disabled, then you cannot mount *any* file systems.
- `CONFIG_NFILE_DESCRIPTORS` < 4. Of course you have to have file descriptions to use any thing in the file system.
- `CONFIG_FS_ROMFS` enabled. This option enables ROMFS file system support

Default Start-Up Behavior. The implementation that is provided is intended to provide great flexibility for the use of Start-Up files. This paragraph will discuss the general behavior when all of the configuration options are set to the default values.

In this default case, enabling `CONFIG_NSH_ROMFSETC` will cause NSH to behave as follows at NSH start-up time:

- NSH will create a read-only RAM disk (a ROM disk), containing a tiny ROMFS filesystem containing the following:

  ```
  `--init.d/
       `-- rcS
  ```

  Where rcS is the NSH start-up script.

- NSH will then mount the ROMFS filesystem at `/etc`, resulting in:

  ```
  |--dev/
  |   `-- ram0
  `--etc/
      `--init.d/
          `-- rcS
  ```

- By default, the contents of rcS script are:

  ```
  # Create a RAMDISK and mount it at /tmp

  mkrd -m 1 -s 512 1024
  mkfatfs /dev/ram1
  mount -t vfat /dev/ram1 /tmp
  ```

- NSH will execute the script at `/etc/init.d/rcS` at start-up (before the first NSH prompt. After execution of the script, the root FS will look like:

  ```
  |--dev/
  |   |-- ram0
  |   `-- ram1
  |--etc/
  |   `--init.d/
  |       `-- rcS
  `--tmp/
  ```

Example Configurations. Here are some configurations that have `CONFIG_NSH_ROMFSETC=y` in the NuttX configuration file. There might provide useful examples:

```
configs/hymini-stm32v/nsh2
configs/ntosd-dm320/nsh
```

```
configs/sim/nsh
configs/sim/nsh2
configs/sim/nx
configs/sim/nx11
configs/sim/touchscreen
configs/vsn/nsh
```

In most of these cases, the configuration sets up the *default* `/etc/init.d/rcS script`. The default script is here: `apps/nshlib/rcS.template`. (The funny values in the template like `XXXMKRDMINORXXX` get replaced via `sed` at build time). This default configuration creates a ramdisk and mounts it at `/tmp` as discussed above.

If that default behavior is not what you want, then you can provide your own custom `rcS` script by defining `CONFIG_NSH_ARCHROMFS=y` in the configuration file. The only example that uses a custom `/etc/init.d/rcS` file in the NuttX source tree is this one: `configs/vsn/nsh`. The `configs/vsn/nsh/defconfig` file also has this definition:

`CONFIG_NSH_ARCHROMFS=y` -- Support an architecture specific ROMFS file.

Modifying the ROMFS Image. The contents of the `/etc` directory are retained in the file `apps/nshlib/nsh_romfsimg.h` OR, if `CONFIG_NSH_ARCHROMFS` is defined, `include/arch/board/rcs.template`). In order to modify the start-up behavior, there are three things to study:

1. **Configuration Options.** The additional `CONFIG_NSH_ROMFSETC` configuration options discussed with the other NSH-specific configuration settings (file:///c:/cygwin/home/patacongo/projects/nuttx/nuttx/trunk/nuttx/Documentation/NuttShell.html#nshconfiguration).

2. **`tools/mkromfsimg.sh` Script**. The script `tools/mkromfsimg.sh` creates `nsh_romfsimg.h`. It is not automatically executed. If you want to change the configuration settings associated with creating and mounting the `/tmp` directory, then it will be necessary to re-generate this header file using the `tools/mkromfsimg.sh` script.

   The behavior of this script depends upon several things:

   - The configuration settings then installed configuration.

   - The `genromfs` tool (available from http://romfs.sourceforge.net) or included within the NuttX buildroot toolchain. There is a snapshot here: misc/tools/genromfs-0.5.2.tar.gz.

   - The `xxd` tool that is used to generate the C header files (xxd is a normal part of a complete Linux or Cygwin installation).

   - The file `apps/nshlib/rcS.template` (OR, if `CONFIG_NSH_ARCHROMFS` is defined `include/arch/board/rcs.template`.

3. **`rcS.template`**. The file `apps/nshlib/rcS.template` contains the general form of the

rcS file; configured values are plugged into this template file to produce the final rcS file.

rcS.template. The default rcS.template, apps/nshlib/rcS.template, generates the standard, default apps/nshlib/nsh_romfsimg.h file.

If CONFIG_NSH_ARCHROMFS is defined in the NuttX configuration file, then a custom, board-specific nsh_romfsimg.h file residing in configs/<board>/include will be used. NOTE when the OS is configured, include/arch/board will be linked to configs/<board>/include.

As mention above, the only example that uses a custom /etc/init.d/rcS file in the NuttX source tree is this one: configs/vsn/nsh. The custom script for the configs/vsn case is located at configs/vsn/include/rcS.template.

All of the startup-behavior is contained in rcS.template. The role of mkromfsimg.sh script is to (1) apply the specific configuration settings to rcS.template to create the final rcS, and (2) to generate the header file nsh_romfsimg.h containg the ROMFS file system image. To do this, mkromfsimg.sh uses two tools that must be installed in your system:

1. The genromfs tool that is used to generate the ROMFS file system image.
2. The xxd tool that is used to create the C header file

You can find the generated ROMFS file system for the configs/vsn case here: configs/vsn/include/rcS.template