Start-up procedure

> ⚠ Note that in this description, time advances down along the page.

> ⚠ Bear in mind as well that this description applies to the *Gen 1* design.

> ⚠ Items in square brackets ([]) are optional.

Initial instruction

- Execution after reset starts at `0xfffffffc`
    - A branch instruction (either b or ba; 26 bit range) to some boot code is loaded here
        - The Xilinx example branches to in block RAM (`bram`) at `0xffffff00`
        - The RTEMS example branches to `download_entry` (but I'm not sure how)
    - Potentially a `sc` (system call) instruction could be loaded here? Any advantage to this?
        - Probably not as the corresponding `ivor` register (PPC 440) is not loaded yet
        - The PPC 405 doesn't have `ivor` registers, so it would continue executing at the system call vector

`dlEntry.s` (`download_entry()`)

This file is considered part of an RTEMS BSP and can be found in `${RTEMS_ROOT}/src/c/src/lib/libbsp/powerpc/virtex5/dlentry`. What's written here is written for the PPC 440 found in Xilinx Virtex 5 parts. The Virtex 4 version is similar.

- In our case, the boot code starts at `startup`
    - Other names are `start`, `download_entry` and `__rtems_entry_point`
- Boot code vaguely follows the "Initialization Software Requirements" outlined in the PowerPPC 440x5 Embedded Processor Core User's Manual v7.1 from IBM
    - Why only "vaguely"?
- Clear MSR
- Disable debug events
- Configure instruction and data cache registers
- Set up decrementer and timer registers
- Clear exception registers ECR and XER
- Invalidate instruction and data caches
- Clear the CPU reservation bit
- Set up CCR0, CCR1, MMUCR, CRF and CTR
- Set up TLB pages
- Set up debug events
- Set up EABI and SYSV environment
- Clear out BSS section
- Load vector offset register
- Set up TOC (i.e., `r2`)
- Set up initial stack (i.e., `r1`)
- Set up argument registers r3, r4 and r5
- *Branch* to `boot_card()`

`boot_card()`

While the RTEMS structure provides for allowing this function to be supplied by the RTEMS BSP, we use the version that the distribution comes with. It is found in the `${RTEMS_ROOT}/src/c/src/lib/libbsp/shared` directory called `bootcard.c`.

In the following, functions prefixed with `bsp_` are supplied by the RTEMS BSP.

- Command line is in the first and only argument
    - In our system this is always a null pointer
- Disable interrupts
- Store command line
- Call `bsp_start()`
- Determine RTEMS work area and heap location and size
- Initialize RTEMS data structures
- Initialize the C library
    - This also installs the heap
- Call `bsp_pretasking_hook()`
- [Enable RTEMS debugging capabilities]
- RTEMS initialization before loading device drivers

- Call `bsp_predriver_hook()`
- Initialize device drivers
- Call `bsp_postdriver_hook()`
- Start multitasking
    - Before starting the first task call any C++ static constructors.
    - Thread with entry point `Init` runs
    - Not clear how this returns. Perhaps when the last task is deleted?
- Call `bsp_cleanup()`
- Return to the start code
    - Not clear what's in the `lr` at this point, i.e., where do we return to?

RTEMS BSP

This constitutes our contributions to RTEMS. The code here sets up the processor and board for *generic* use. Files related to it can be found in `${RTEMS_ROOT}/src/c/src/lib/libbsp/powerpc/virtex5/...`.

The functions prefixed with `app_` are supplied by the RTEMS *application*, i.e., the RCE project, in our case.

1. `bsp_start()`
    - Set up default character output function
    - Get CPU type and revision cached
    - Initialize device driver parameters
        - Rate of timer source for `clock.c`
        - `bsp_timer_internal_clock` ?
        - `bsp_timer_average_overhead` ?
        - `bsp_timer_least_valid` ?
    - Initialize default raw exception handlers
    - Call `app_bsp_start()`
    - Return to `boot_code()`
2. `bsp_pretasking_hook()`
    - Call `app_bsp_pretasking_hook()`
    - Return to `boot_code()`
3. `bsp_predriver_hook()`
    - Call `app_bsp_predriver_hook()`
    - Return to `boot_code()`
4. `bsp_postdriver_hook()`
    - Call `rtems_libio_supp_helper()` to open `/dev/console` for `stdin`, `stdout` and `stderr`, if it exists
    - Call `app_bsp_postdriver_hook()`
    - Return to `boot_code()`
5. `bsp_cleanup()`
    - Call `app_bsp_cleanup()`
    - Return to `boot_code()`

RCE BSP

This is the portion of the BSP that is specific to the RCE project. It can be found in `release/rce/init/src/Init.cc`.

1. `app_bsp_start()`
    - This routine should set up the processor and board as needed for the task at hand, i.e., it is *not* generic.
    - Replace the character output function with one that writes to the `syslog`
    - Return to `bsp_start()`
2. `app_bsp_pretasking_hook()`
    - Initialize `RceDebug`
    - Initialize `RcePic`
    - Return to `bsp_pretasking_hook()`
3. `app_bsp_predriver_hook()`
    - Initialize `RceEthernet`
    - Initialize `RceBsdnet`
    - Return to `bsp_predriver_hook()`
4. `app_bsp_postdriver_hook()`
    - Return to `bsp_postdriver_hook()`
5. `app_bsp_cleanup()`
    - Return to `bsp_cleanup()`

`RceDebug`

- Set up an RTEMS extension that creates and manages the `syslog`
- Return to `app_bsp_pretasking_hook()`

`RcePic`

- Set up a single PIC Manager
    - Set up a vector of PEBs ?
    - Set up a vector of ECDs ?
    - Set up a vector of FLBs ?
    - Set up a vector of PIBs ?

    - Install a BOOK-E Critical exception handler
    - Install an External Interrupt handler
- Return to `app_bsp_pretasking_hook()`

`RceEthernet`

- Create a single empty linked list of Ethernet drivers
- Return to `app_bsp_predriver_hook()`

`RceBsdnet`

- Create a single empty linked list of Ethernet handlers
- Return to `app_bsp_predriver_hook()`

`Init` task

This task is automatically launched by the act of enabling multitasking in `boot_card()`

- Launch `init_executive()` task
- Delete the `Init` task

`init_executive()`

This function runs in its own RTEMS task that was launched by `Init` task. This forms the *intent* of the loaded executable. Other possibilities exist, but generally, this will be one of the `core` executables.

- Announce what's running
- Configure the network from DHCP
- Set up the dynamic linker
- [Start the shell]
- [Start the debugger daemon (`gdb` stub)]
- Create a `Task`
- Determine what the `Task` should run
    - Read metadata from flash
    - Read the front panel rotary switch
- Dynamically link the code
- Run the `Task`