MSP430X port - basic checkpointing design documentation

David Garriou, Jean-Luc Béchennec

December 11, 2019

1 Goals and ideas

1.1 Consistency

Our main goal is to keep a system within a coherent state even after a recovery from power loss. In other words, the kernel execution must be atomic with respect to power loss.

1.1.1 Consistency of kernel space and user space

When the system recovers all of its code, user space and kernel space, must be consistent. `OS_VAR` section contains the kernel data structures. It means that we must not transfer `OS_VAR` section to FRAM otherwise the system would recover with two differents (not consistent) states, the last user space state and the latest kernel space state.

We have to save both kernel and user space variables at the same time (during a single checkpoint).

1.2 Big picture

Big picture of what to do:

1. A periodic task, let's call it `energy_task`, will check for remaining energy in the super capacitor;
    (a) user space execution;
    (b) remaining energy consists in mesuring VCC voltage (on the terminals of the super capacitor);
    (c) if the remaining energy is lower than a defined threshold, we must realize a checkpoint

2 Normal MSP430X startup sequence

The MSP430X startup sequence is as follow:

1. After a reset, the `tpl_reset_handler` executes (see `tpl_startup.S` file). It:
   - (a) stops the watchdog timer;
   - (b) disables the interrupts;
   - (c) sets up the stack;
   - (d) calls `tpl_continue_reset_handler` C function.
2. `tpl_continue_reset_handler` (see `tpl_startup.c` file) does:
   - (a) initialization of the .bss section to 0 (uninitialized variables) and initialization of the .data section by copying the initial values from FRAM (initialized variables)[1];
   - (b) clock setup by calling `tpl_set_mcu_clock`;
   - (c) initialization of the MPU;
   - (d) a call to `main`.
3. `main` is responsability of the user but usually it:
   - (a) initializes application level devices;
   - (b) calls `StartOS`.
4. `StartOS`:
   - (a) calls `tpl_init_machine` that calls;
     - i. `tpl_init_machine_generic` that calls:
       - A. `tpl_init_mpu` which is not implemented yet (and would be redundant with 2c).
     - ii. `tpl_init_machine_specific` that calls:
       - A. `tpl_set_systick_timer`.
   - (b) calls `tpl_start_os` that does a system call to `tpl_start_os_service` that.
     - i. calls `tpl_init_os`;
     - ii. calls `tpl_enable_counters`;
     - iii. calls `StartupHook` if any;
     - iv. calls `tpl_start_scheduling`.

     when returning a task is schedule.

[1]Why not use the DMA to do that, it would use less energy

3 Modified sequence to restore a checkpoint

Following items shall be modified when restarting from a checkpoint:

- item 2a should be replaced by a copy from the checkpoint data in FRAM to SRAM.
- item 2b should use the replaced by an init with the clock frequency when the checkpoint was done. This can be done by having a variable (in .data segment) to store the clock frequency so that it would restored as part of the checkpoint data.
- item 2d should be replaced by a call to a mandatory function used to initialize the devices for the application (UART for instance) and to a new service, let's call it `RestartOS`. `RestartOS` would:

  1. Call `tpl_init_machine`;
  2. Call `tpl_restart_os` that does a system call to `tpl_restart_os_service` that does a `tpl_start`. `tpl_start` moves the highest priority task from the ready list to the elected slot of `tpl_kern`. Conditions shall be `NEED_SWITCH` true and `NEED_SAVE` false.

When returning from the `RestartOS` service, the highest priority task is scheduled and the system continues execution.

A boolean variable stored in FRAM, let's call it `tpl_checkpoint_available`, shall be used to select, when true, the modified sequence instead of the normal one.

4 Checkpointing

A new service is necessary, let's call it `Hibernate`. When called, `Hibernate`, terminates the caller. It copies the SRAM to the FRAM (checkpoint), stops the Systick, stops application interrupts (a user function shall be provided for that), programs the RTC to emit an interrupt every x seconds, enables interrupts and goes in LPM3 (Lowest power mode that preserves the RAM).

The RTC ISR checks the voltage of the MCU. If it is above a threshold (`RESUME_FROM_HIBERNATE_THRESHOLD`), it exists from LPM3. If this never happens, after a while, the MCU will power off. When in the future the MCU restart, what is described in section 3 applies.

If `Hibernate` resumes because the RTC ISR exited from LPM3 then it disables interrupt, starts application interrupts, starts the Systick, stop the RTC. It calls `tpl_start` to elect the highest priority task and returns.

In addition, a periodic basic task, `energy_task` checks (every 5 seconds ? more ?)the voltage. If the voltage drops below a threshold (`HIBERNATE_THRESHOLD`), `energy_task` calls `Hibernate` (and terminates):

MB: should be defined in the .oil, as it depends on both supercapacitor value, and application through external peripherals

```
TASK(energy_task)
{
  uint16 voltage = readPowerVoltage();
  if (voltage < HIBERNATE_THRESHOLD) {
    Hibernate();
  }
  else {
    TerminateTask();
  }
}
```

`HIBERNATE_THRESHOLD` shall be chosen so that the worst voltage drop between 2 executions of `energy_task` plus the voltage drop due to checkpointing is lower than the threshold.

`HIBERNATE_THRESHOLD` shall be lower than `RESUME_FROM_HIBERNATE_THRESHOLD`.

4.1 OIL modifications

An OS attribute is added:

```
BOOLEAN CHECKPOINT = FALSE;
```

If TRUE, some complementary information can be given:

```
BOOLEAN CHECKPOINT = TRUE {
  HIBERNATE_THRESHOLD = 1900; //in mV
  ENERGY_TASK_PERIOD  = 5000; //in ms
};
```

4.2 Non volatile data declaration

Non Volatile data is stored at the beginning of the FRAM. The MPU is configured at startup to let a section in R/W mode. However, limitations on the MPU hardware implementation require to use a memory area that is *a multiple of 1024 bytes*. The code section just follows, with R/X rights.

An OS variable in non volatile memory should be defined as:

```
#define OS_START_SEC_NON_VOLATILE_VAR_16BIT
#include "tpl_memmap.h"
VAR(uint16, OS_VAR) nvdata = 0;
#define OS_STOP_SEC_NON_VOLATILE_VAR_16BIT
#include "tpl_memmap.h"
```

for a task related non volatile variable the definition is (task is `serial_TX` here:

```
#define APP_Task_serial_TX_START_SEC_VAR_NON_VOLATILE_32BIT
#include "tpl_memmap.h"
VAR(uint32_t,AUTOMATIC) dataFRAM = 100;
#define APP_Task_serial_TX_STOP_SEC_VAR_NON_VOLATILE_32BIT
```

```
#include "tpl_memmap.h"
```

Note:

At this date, the non volatile data is initialized only when flashing the firmware. It should be updated so that the value can be updated on a non-restoring startup (no checkpoint).

4.3 DMA

Both checkpoint save and checkpoint restore are done using the DMA. The MSP430FR5994 has 6 DMA channels. Channel 0 is used. The block transfer mode is used. DMA is triggered by software. It copies the whole content of the SRAM (8kB, from 0x1C00 to 0x3BFF) to the FRAM. It would be safer to use double buffering because if a checkpoint cannot complete before the loss of power for any reason, the previous checkpoint will not be corrupted. An int variable in FRAM with value equal to 0, 1 or -1 is used to select the buffer. Let's call it `tpl_checkpoint_buffer`. If the value is -1, no checkpoint is saved. This is the initial value. If value is 0 a checkpoint is saved in buffer 0 and if value is 1, a checkpoint is saved in buffer 1. Saving a checkpoint would be as follow:

```
int buffer = (tpl_checkpoint_buffer + 1) & 1;
tpl_save_checkpoint(buffer);
tpl_checkpoint_buffer = buffer;
```

If power is lost during the `tpl_save_checkpoint` function, `tpl_checkpoint_buffer` is not updated and the checkpoint is not committed.

The content of `tpl_checkpoint_buffer` is used to select the normal startup described at section **??** or the startup from a checkpoint described at section 3.

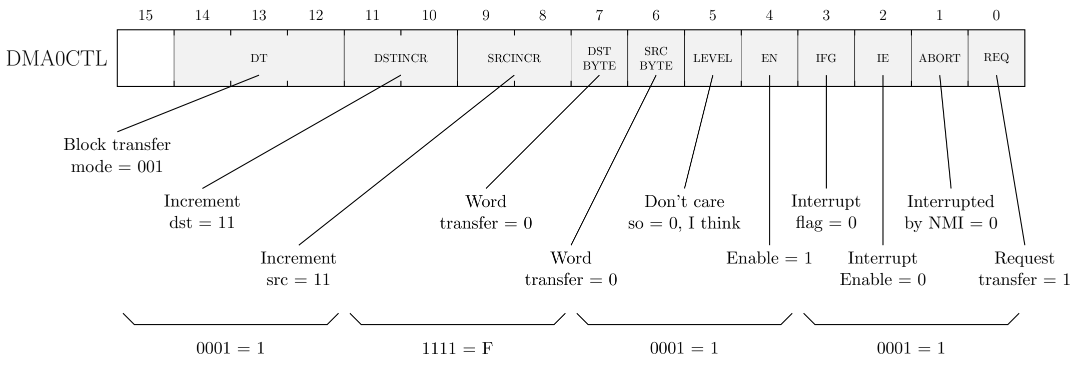
```
#define OS_START_SEC_CONST_16BIT
#include "tpl_memmap.h"
static CONST(sint16, OS_CONST) NO_CHECKPOINT = -1;
#define OS_STOP_SEC_CONST_16BIT
#include "tpl_memmap.h"

#define OS_START_SEC_NON_VOLATILE_VAR_16BIT
#include "tpl_memmap.h"
VAR(sint16, OS_VAR) tpl_checkpoint_buffer = -1;
#define OS_STOP_SEC_NON_VOLATILE_VAR_16BIT
#include "tpl_memmap.h"

if (tpl_checkpoint_buffer == NO_CHECKPOINT) {
  /* normal startup */
}
else {
  /* startup from checkpoint */
}
```

The 2 buffers are located above the 64kB limit, at the end of the FRAM for instance. The first one from 0x040000 to 0x041FFFh and the second one from 0x042000 to 0x043FFFh. The SRAM starts at address 0x001C00 and ends at address 0x003BFF.

The function to save the SRAM to a buffer using the DMA has to be written in assembly because writing to start address and end address registers shall be done using extended instructions when address is above the 64kB limit:

```
#include "tpl_assembler.h"

.equ SRAM_START      0x1C00
.equ BUFFER0_START   0x040000
.equ BUFFER1_START   0x042000
.equ CHECKPOINT_SIZE 0x1000 /* 4k words  */
.equ START_TRANSFER  0x1F11 /* see below */

/*
 * tpl_save_checkpoint
 * Argument is the buffer: 0 or 1
 */
.global tpl_save_checkpoint
.type   tpl_save_checkpoint, %function

tpl_save_checkpoint:

  mov    #0, &DMACTL0  /* DMA Channel 0 is triggered by software (DMAREQ) */
  mov    #SRAM_START, &DMA0SA      /* Source address is SRAM address      */
  tst    REG_RETARG
  jnz    buffer1
  movx.a #BUFFER0_START, &DMA0DA   /* Destination address is buffer 0     */
  jmp    cont
buffer1:
  movx.a #BUFFER1_START, &DMA0DA   /* Destination address is buffer 1     */
cont:
  mov    #CHECKPOINT_SIZE, &DMA0SZ /* Size of the transfer                */

  mov    #START_TRANSFER, &DMA0CTL /* Start the transfer                  */
  ret
```

`START_TRANSFER` explanation



5 New services

5.1 Service `RestartOS`

5.2 Service `Hibernate`

6 Peripherals

Specific code for peripherals re-initialization. TODO They have to be re-initialized before main call.