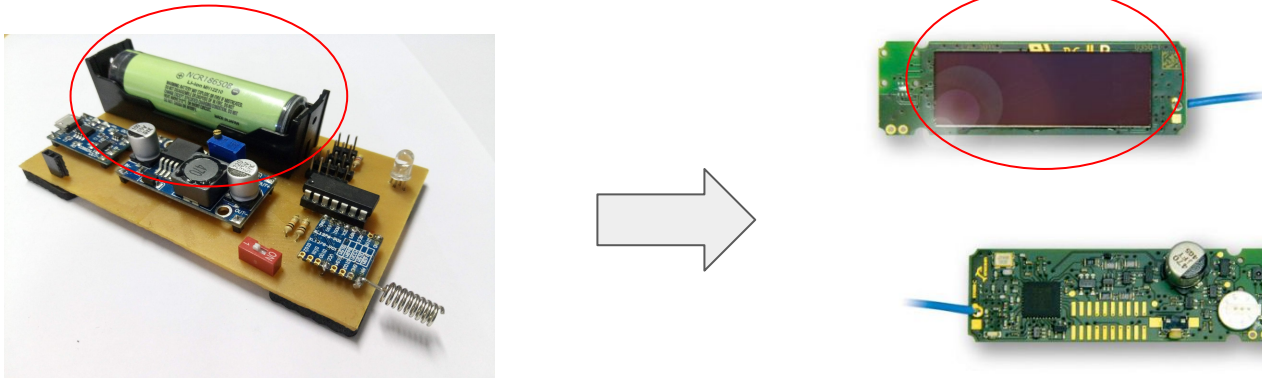


Intermittent Computation with FreeRTOS on MSP430

Prof. Li-Pin Chang
ESSLAB@NYCU

From WSN to EH Devices

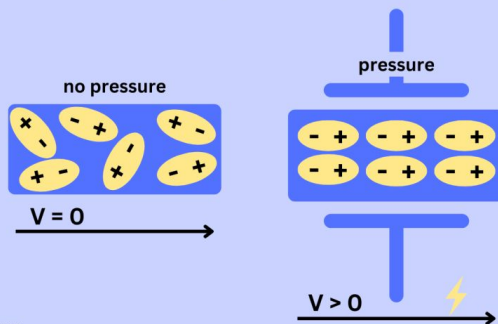
- Wireless Sensor Nodes
 - Smart building, smart healthcare, smart transportation, etc
 - Deployed in a large amount & remote locations
 - Battery replacement is not feasible or not possible
- Batteryless, energy-harvesting devices
 - Collect energy from the environment for long-term operation
 - Use capacitors as energy storage



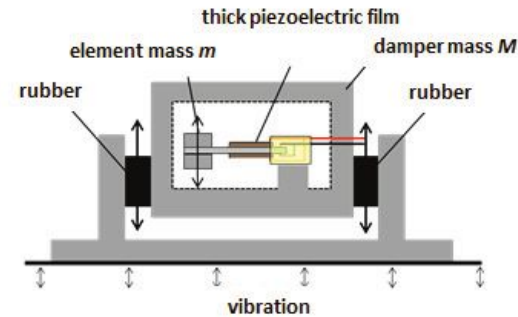
Piezoelectricity / Vibration Generator

Piezoelectricity and the Piezoelectric Effect

Piezoelectricity is the generation of an electric charge in certain materials in response to a mechanical stress.



sciencenotes.org



Generators



Reversal piezoelectric effect: buzzer

<https://sciencenotes.org/piezoelectricity-and-the-piezoelectric-effect/>

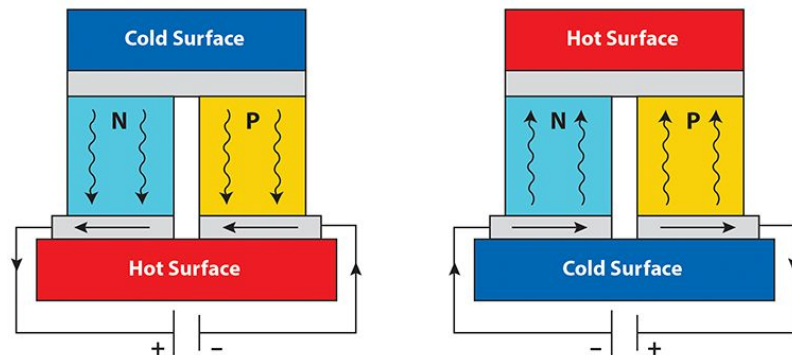
https://www.researchgate.net/figure/Piezoelectric-type-of-vibration-powered-generator_fig1_314190507

https://www.researchgate.net/figure/Experimental-use-of-piezoelectric-floor-tiles-C-Scholer-et-al-2009_fig4_323838138

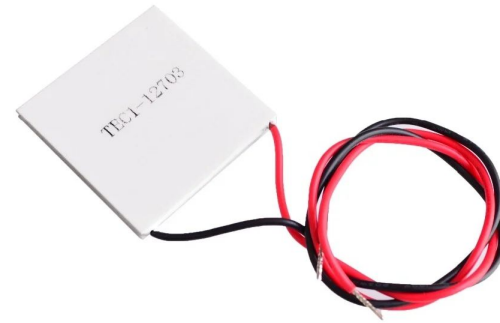
Peltier Effect / Thermoelectric Generators

The Peltier effect is a temperature difference created by applying a voltage between two electrodes connected to a sample of semiconductor material.

This phenomenon can be useful when it is necessary to transfer heat from one medium to another on a small scale.



Like Piezoelec, this process is reversible

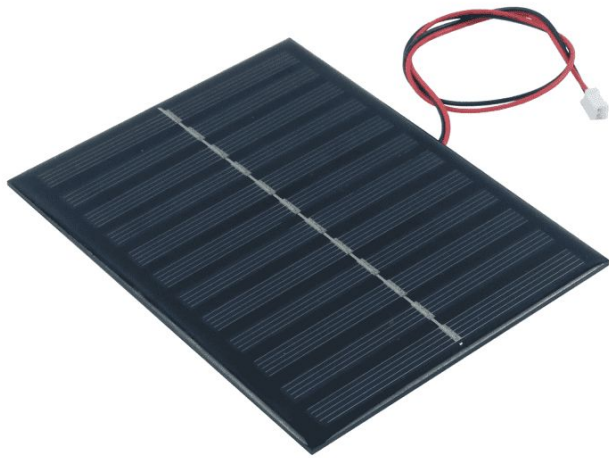


Cooler: consumes power



Generator: stirring cup

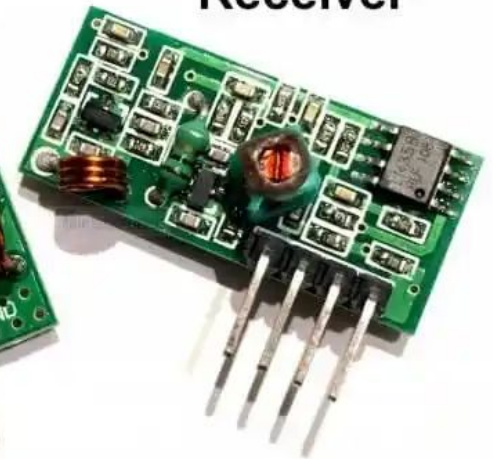
Solar Energy / RF Energy



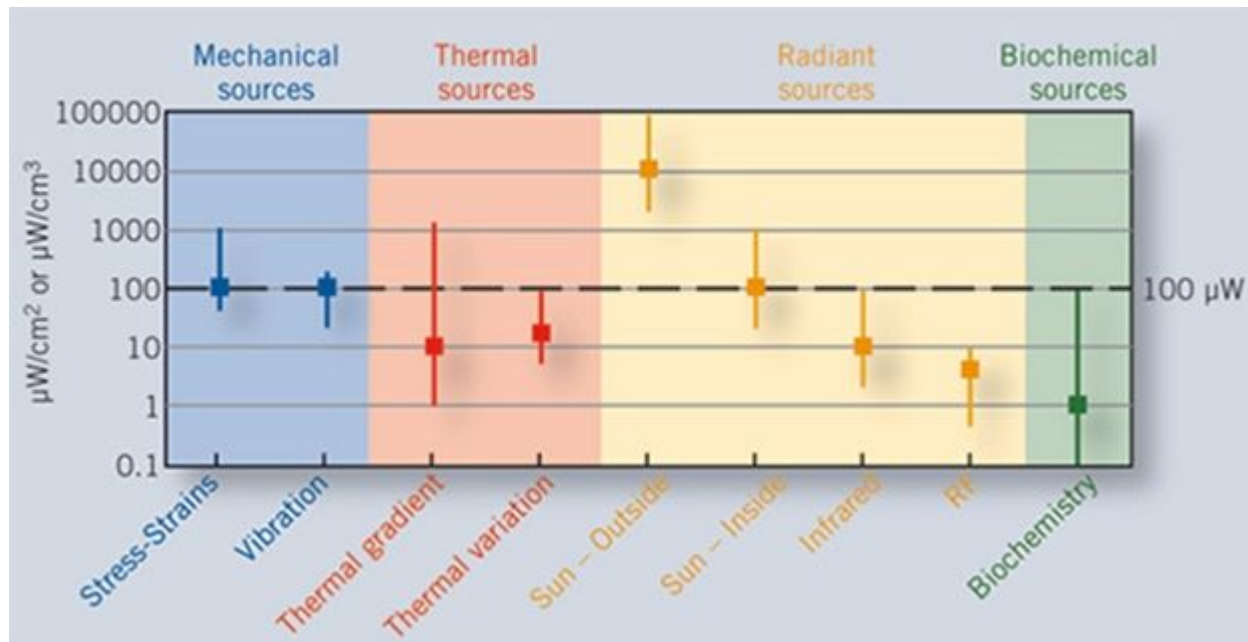
Transmitter



Receiver

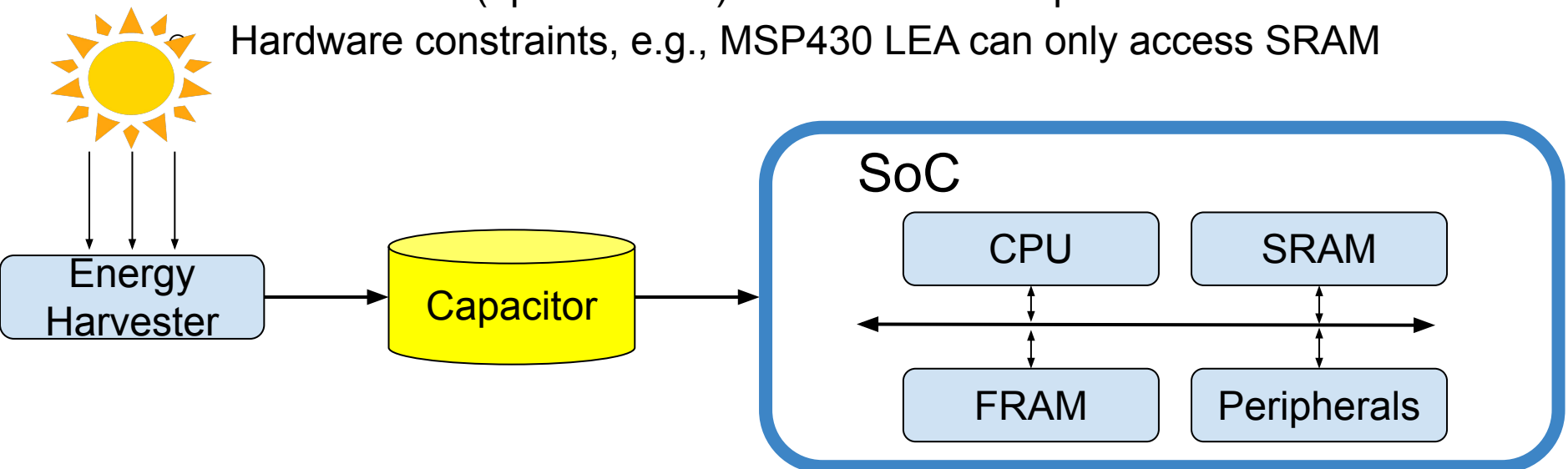


Comparison of Ambient Energy



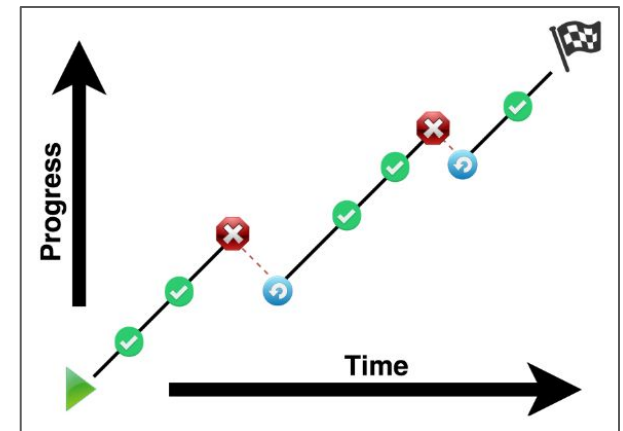
System Model

- Program context
 - CPU registers (volatile)
 - SRAM (volatile)
 - NVRAM, e.g., FRAM (non-volatile)
 - I/O status (volatile)
 - Some variables must be allocated to volatile memory (SRAM)
 - Faster reference (low latency) e.g., on MSP430, SRAM is synchronous with the CPU (up to 24MHz) but FRAM rates up to 8MHz
- Hardware constraints, e.g., MSP430 LEA can only access SRAM



Checkpoint

- Committing a checkpoint
 - Taking a snapshot of the program context with respect to time instant t
 - Performed by system software when necessary (periodic, low power, etc)
 - The backup storage must be persistent (non-volatile)
- Restoring a checkpoint
 - Restore the program context to the previous snapshot for time instant t
 - Occurs when the device recovers from power interruption
 - Losing a certain amount of program progress
- Trade-off between ckpt and progress loss
- Types of checkpoints
 - Continuous
 - Just in time
 - Atomic task



Continuous Checkpoint

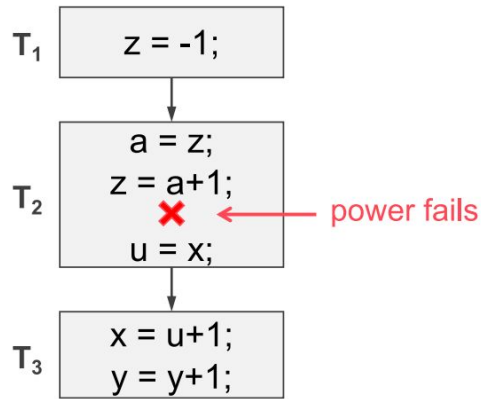
- Checkpoints are committed continuously
- System-level checkpoints
 - commit→Full backup/full restore of the entire program context
 - restore→Discarding post-checkpoint progress
- Compiler-assisted checkpoints
 - Backing up WAR (Write-After-Read)-affected variables only
 - Instructions involving WAR-free variables are idempotent

b=a
a=1

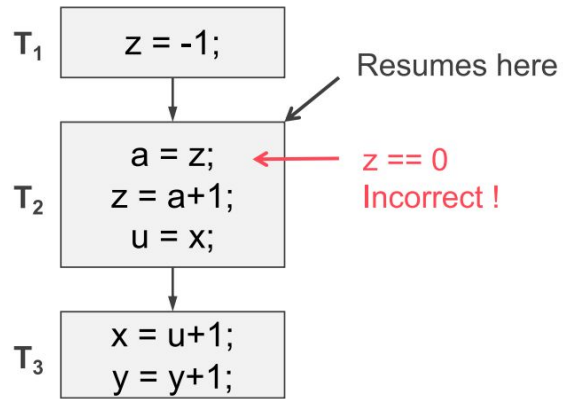
Not idempotent

a=1
b=1
c=b

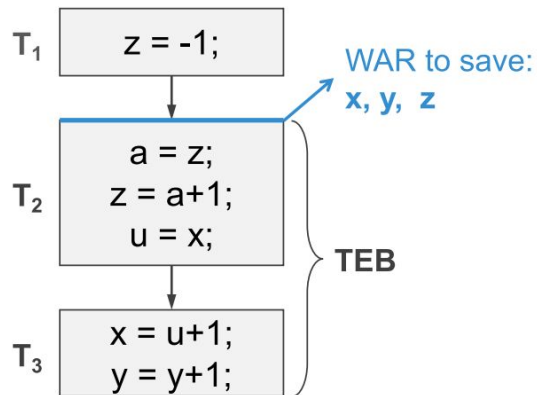
idempotent



(a)



(b)



(c)

(a) (b) no backup, (c) backup WAR variables a program block

Just-In-Time Checkpoint

- When the energy level is critically low, commits a checkpoint and *halts*
 - No post-checkpoint progress, no waste of energy
- Require to monitor the voltage of the capacitor
 - Often use an ADC or a voltage comparator
 - The monitoring can be expensive (ADC) and/or inaccurate (capacitor ages over time)

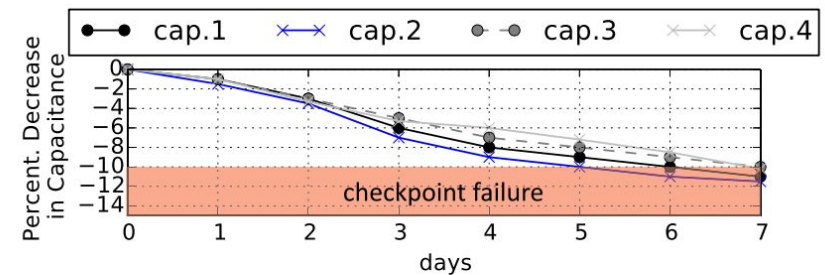
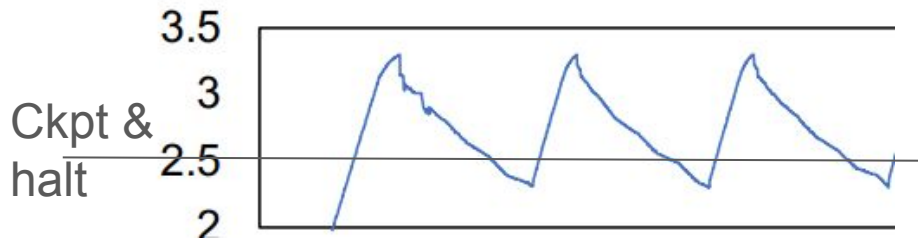
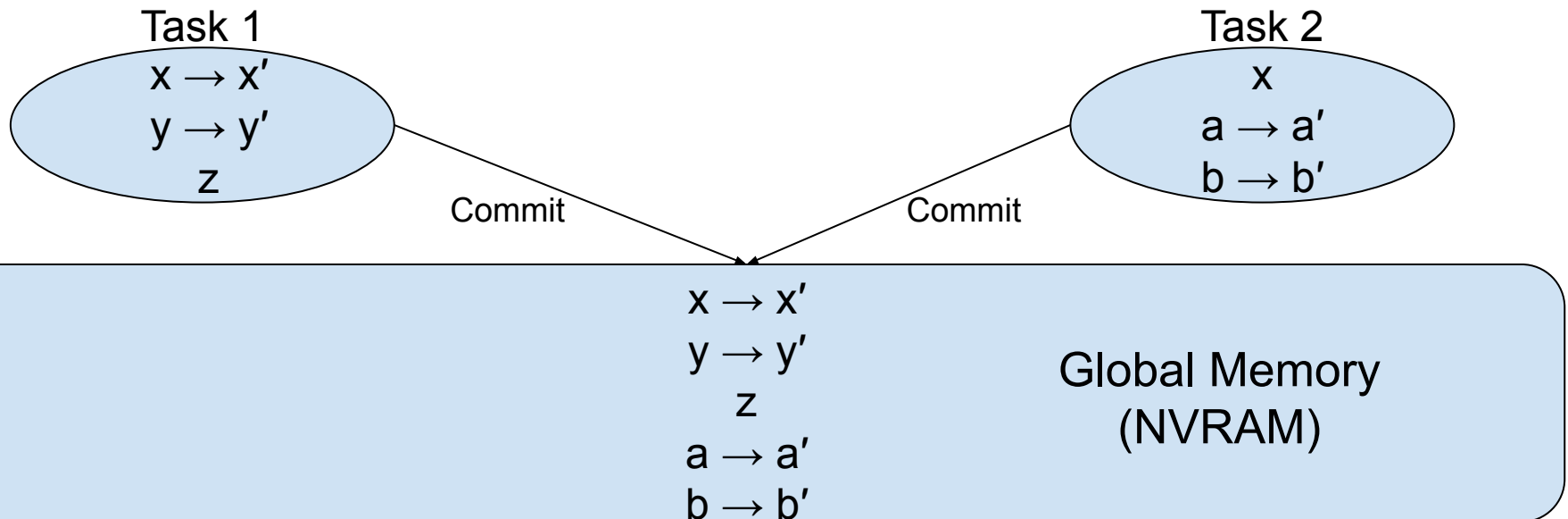


Fig. 1. Capacitor degradation in real energy harvesting systems. Within seven days, a capacitors can be severely degraded causing capacitor error.

Atomic Task

- Modifying variables locally and committing to the global memory when done [alpaca]
- Pros: Less complicated (from the system software perspective)
- Cons:
 - Not easy to handle resource synchronization; a large task can delay a set of dependent small tasks
 - Solution: splitting a big task into FSM → harder to program

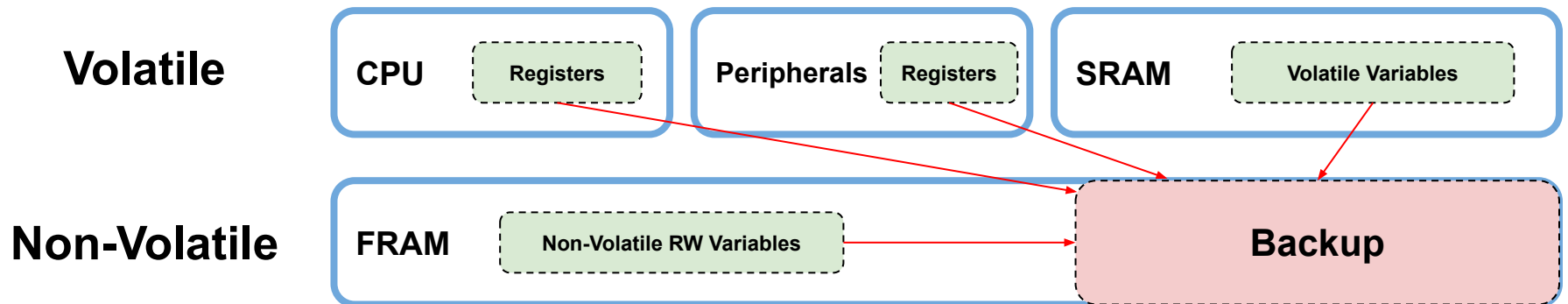


Checkpoint Commit and Restore

- Checkpoint is a snapshot of the program context with respect to a time instant t
 - Commit a checkpoint: to backup the program context
 - Restore a checkpoint: to restore the most recent checkpoint
- The following contents must be backed up and restored
 - Those can be modified after a checkpoint (NVRAM)
 - Those will be lost on power interruption (SRAM, NVRAM)
- The following need not be backed up or restored
 - RO variables in NVRAM

Program Context - Backup

- CPU Registers
 - All CPU registers (R0~R15)
- I/O registers
 - TAxR (timer counter register) → to be discussed...
- Volatile memory (SRAM)
 - Volatile contents (global variables) : .bss and .data sections
- Non-Volatile RW memory (FRAM)
 - ucHeap (task stacks + TCBs)



Backup Space

- MSP430FR5994 features 8KB SRAM and 256KB FRAM (embedded)
- Reserve a portion of the FRAM as the backup space
 - Ping-pong buffer: one for the currently-writing backup and the other for the read-only backup; to avoid power interruption in the middle of backup

```
#pragma PERSISTENT(backup_ucHeap)
uint8_t backup_ucHeap[2][UCHEAP_SIZE] = {0}; // One for currently-writing backup, one for read-only backup.

// Other backups
...

#pragma PERSISTENT(CW_backup_index)
uint8_t CW_backup_index = 0; // index of currently-writing backup

void commit()
{
    // Backing up ucHeap
    for (...)
    {
        backup_bss[CW_backup_index][i] = ...;
    }

    // Backing up other sections ...
    ...

    CW_backup_index ^= 1; // Swap the index of currently-writing backup only if backup is finished.
}
```

Checklist: Checkpoint Commit

- Memory map
- SRAM backup
- RW FRAM contents backup
- Peripheral state
- CPU register backup

Memory Mapping

- Memory mapping
 - A section contains variables, e.g., uninitialized globals are in .bss
 - A section is mapped to a certain type of memory, e.g., .bss is mapped to SRAM
 - You can change the mapping, of course
- There are two ways to explicitly declare memory mapping
 - Use linker command file to specify the mapping of sections
 - Use C directive `#pragma` to specify the mapping of a variable

Memory Map: Linker Command File

- Memory allocation involves two parts: **MEMORY** and **SECTIONS**
- **MEMORY**
 - Divides the memory into **regions** of RAM, FRAM, etc.

```
MEMORY
{
    ...

    RAM                : origin = 0x1C00, length = 0x1000
    FRAM                : origin = 0x4000, length = 0xBF80
    FRAM2               : origin = 0x10000, length = 0x33FF8
    ...
}
```

- **SECTIONS**
 - Maps **sections** (segments) to regions

```
SECTIONS
{
    ...

    .TI.persistent : {} > FRAM2                                /* For #pragma persistent */
    .bss           : {} > RAM , START(__bss__), END(__bssEnd__) /* Global & static vars */
    .data          : {} > RAM , START(__data__), END(__dataEnd__) /* Global & static vars */
    .TI.noinit     : {} > FRAM2                                /* For #pragma noinit */
    .stack         : {} > RAM (HIGH) , START(__stack__), END(__stackEnd__) /* Software system stack */
    ...
}
```

Memory Map: Linker Command File

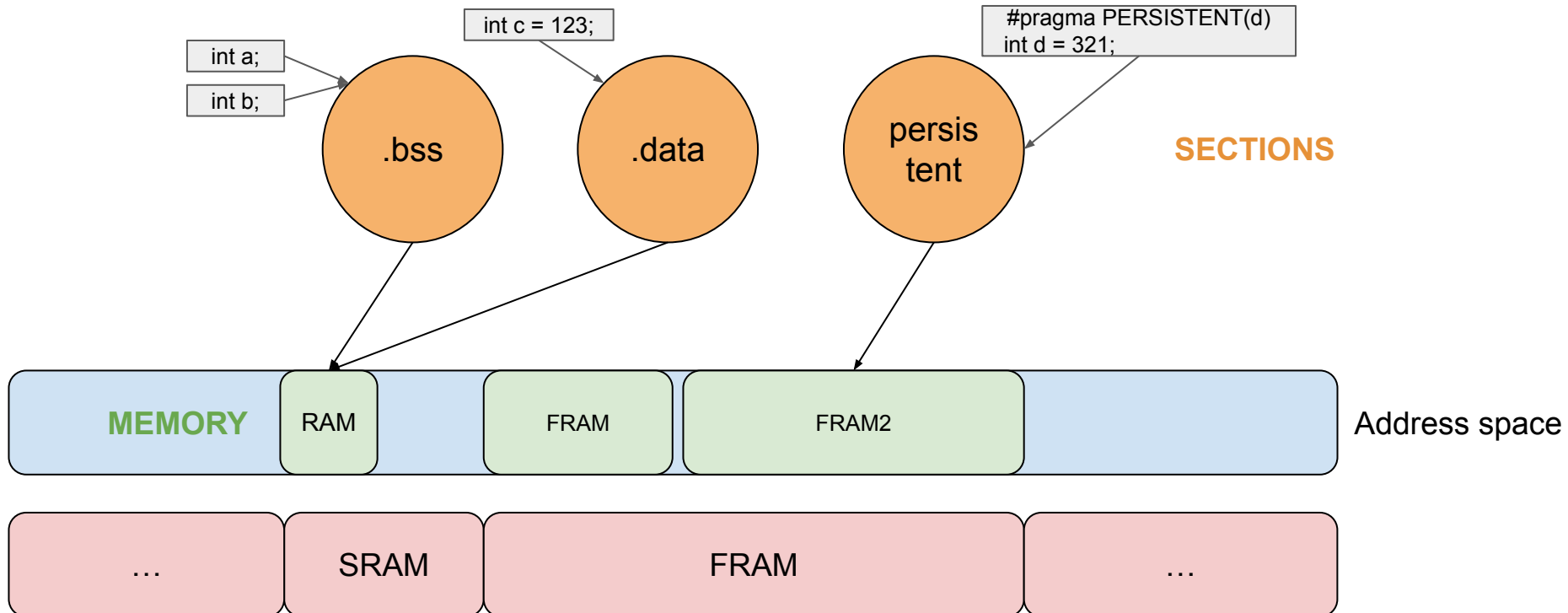
```
MEMORY
{
    ...

    RAM          : origin = 0x1C00, length = 0x1000
    FRAM          : origin = 0x4000, length = 0xBF80
    FRAM2         : origin = 0x10000, length = 0x33FF8
    ...
}
```

(1)		MSP430FR5994, MSP430FR5964
Memory (FRAM) Main: interrupt vectors and signatures Main: code memory	Total size	256KB 00FFFFh to 00FF80h 043FFFh to 004000h
RAM (shared with LEA on MSP430FR599x)		4KB 003BFFh to 002C00h
RAM		4KB 002BFFh to 001C00h
Device descriptor (TLV) (FRAM)		256 bytes 001AFFh to 001A00h
Information memory (FRAM)	Info A	128 bytes 0019FFh to 001980h
	Info B	128 bytes 00197Fh to 001900h
	Info C	128 bytes 0018FFh to 001880h
	Info D	128 bytes 00187Fh to 001800h
Bootloader (BSL) memory (ROM)	BSL 3	512 bytes 0017FFh to 001600h
	BSL 2	512 bytes 0015FFh to 001400h
	BSL 1	512 bytes 0013FFh to 001200h
	BSL 0	512 bytes 0011FFh to 001000h
Peripherals	Size	4KB 000FFFh to 000020h
Tiny RAM	Size	22 bytes 000001Fh to 00000Ah
Reserved	Size	10 bytes 0000009h to 0000000h

Memory Map: Linker Command File

```
SECTIONS
{
    ...
    .TI.persistent : {} > FRAM2                                /* For #pragma persistent */
    .bss           : {} > RAM      , START(__bss__),  END(__bssEnd__)          /* Global & static vars */
    .data          : {} > RAM      , START(__data__),  END(__dataEnd__)        /* Global & static vars */
    .TI.noinit     : {} > FRAM2                                /* For #pragma noinit */
    .stack         : {} > RAM (HIGH) , START(__stack__),  END(__stackEnd__)        /* Software system stack */
    ...
}
```



Memory Map: C Directive #pragma

- #pragma DATA_SECTION(*variable*, *section*)

- *variable* will be allocated into *section test*

- ```
#pragma DATA_SECTION(tmp, ".test")
uint8_t tmp;
```

- #pragma PERSISTENT(*variable*)

- *variable with initial value* to be allocated to **FRAM**

- ```
#pragma PERSISTENT(tmp)  
uint8_t tmp = 123;
```

- #pragma NOINIT(*variable*)

- *variable without initial value* to be allocated to **FRAM**

- ```
#pragma NOINIT(tmp)
uint8_t tmp;
```

PERSISTENT and NOINIT are like non-volatile versions of .data and .bss

# Memory Map: Linker map file (.map)

- Linker map file tells the start and length of MEMORY

| name  | origin   | length   | used     | unused   | attr | fill |
|-------|----------|----------|----------|----------|------|------|
| ...   |          |          |          |          |      |      |
| RAM   | 00001c00 | 00001000 | 000007a4 | 0000085c | RWIX |      |
| ...   |          |          |          |          |      |      |
| FRAM  | 00004000 | 0000bf80 | 00000a4c | 0000b534 | RWIX |      |
| ...   |          |          |          |          |      |      |
| FRAM2 | 00010000 | 00033ff8 | 0000ad1a | 000292de | RWIX |      |

# Memory Map: Linker map file (.map)

- Linker map file shows the actual mapping *after* linking
- Linker map file also tells the start and length of vars & func in each **SECTIONS**

| output section | page | origin   | length   | attributes/<br>input sections                      |
|----------------|------|----------|----------|----------------------------------------------------|
| .bss           | 0    | 00001df8 | 000001ac | UNINITIALIZED                                      |
|                |      | 00001df8 | 000000a0 | (.common:__TI_tmprnams)                            |
|                |      | 00001e98 | 0000005a | tasks.obj (.bss:pxReadyTasksLists)                 |
|                |      | 00001ef2 | 00000012 | timers.obj (.bss:xActiveTimerList1)                |
|                |      | 00001f04 | 00000012 | timers.obj (.bss:xActiveTimerList2)                |
|                |      | 00001f16 | 00000012 | tasks.obj (.bss:xDelayedTaskList1)                 |
|                |      | 00001f28 | 00000012 | tasks.obj (.bss:xDelayedTaskList2)                 |
|                |      | 00001f3a | 00000012 | tasks.obj (.bss:xPendingReadyList)                 |
|                |      | 00001f4c | 00000012 | tasks.obj (.bss:xSuspendedTaskList)                |
|                |      | 00001f5e | 00000012 | tasks.obj (.bss:xTasksWaitingTermination)          |
|                |      | 00001f70 | 00000008 | serial.obj (.bss)                                  |
|                |      | 00001f78 | 00000008 | tasks.obj (.bss)                                   |
|                |      | 00001f80 | 00000008 | timers.obj (.bss)                                  |
|                |      | 00001f88 | 00000008 | heap_4.obj (.bss:xStart)                           |
|                |      | 00001f90 | 00000008 | (.common:parmbuf)                                  |
|                |      | 00001f98 | 00000004 | rts430x_lc_ld_eabi.lib : memory.c.obj (.bss)       |
|                |      | 00001f9c | 00000004 | (.common:progress)                                 |
|                |      | 00001fa0 | 00000004 | (.common:snapshot_reg)                             |
| .TI.persistent | *    | 00010000 | 000068ce |                                                    |
|                |      | 00010000 | 00003800 | main.obj (.TI.persistent:ucHeap)                   |
|                |      | 00013800 | 000030b8 | checkpoint.obj (.TI.persistent:snapshot)           |
|                |      | 000168b8 | 0000000a | checkpoint.obj (.TI.persistent:dma_param)          |
|                |      | 000168c2 | 00000008 | checkpoint.obj (.TI.persistent:rtc_calendar_param) |
|                |      | 000168ca | 00000004 | my_timer.obj (.TI.persistent)                      |
|                |      |          |          |                                                    |

Variable

# Checklist: Checkpoint Commit

- Memory map
- SRAM backup
- RW FRAM contents backup
- Peripheral state
- CPU register backup



# SRAM Backup

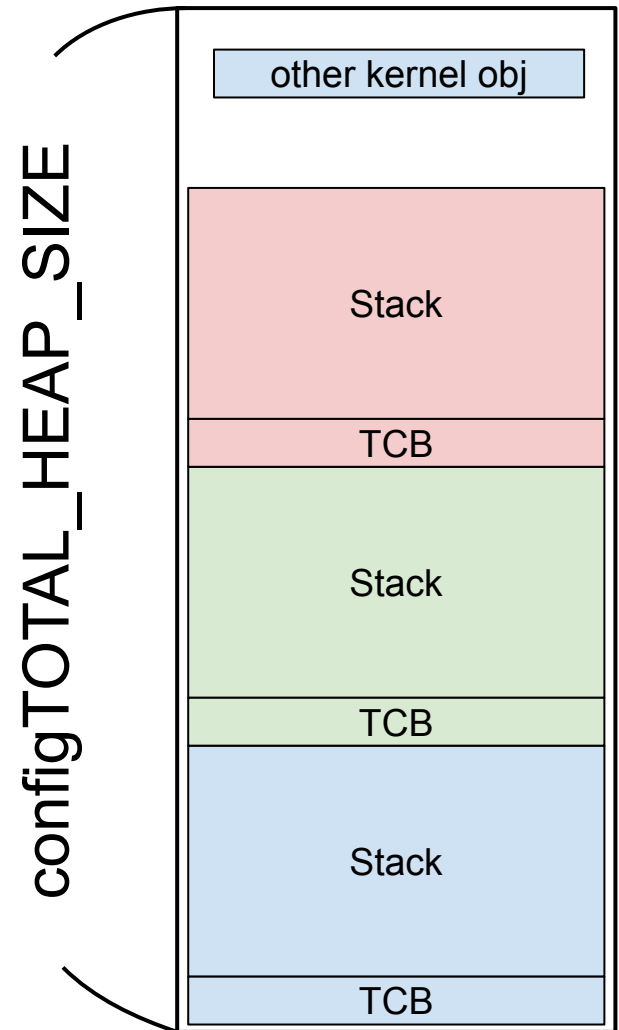
- Kernel variables/objects in .data and .bss (both from SRAM)
  - Various lists
    - Ready task list, Delayed task list, Suspended task list
    - Timer list
    - These are the “anchors” of the lists, not the whole list
  - pxCurrentTCB, uxTaskNumber, etc (see .map)
- 
- All the above evaporate when power goes off, so you have to backup **.bss and .data** sections
  - See .cmd or .map for their starting address and length

# Checklist: Checkpoint Commit

- Memory map
- SRAM backup
- RW FRAM contents backup
- Peripheral state
- CPU register backup

# ucHeap

- The largest RW space in FRAM
- FreeRTOS kernel objects
  - Tasks
    - TCB
    - stack
  - Queues
  - Semaphores
  - Mutexes
  - Event groups
- Dynamic memory allocation
  - `pvPortMalloc()`
  - `pvPortFree()`
- They are *runtime* objects so they are not shown in `.map`



# ucHeap

- In FRAM, only ucHeap needs to be backed up
- The address and length of ucHeap is in the linker map file

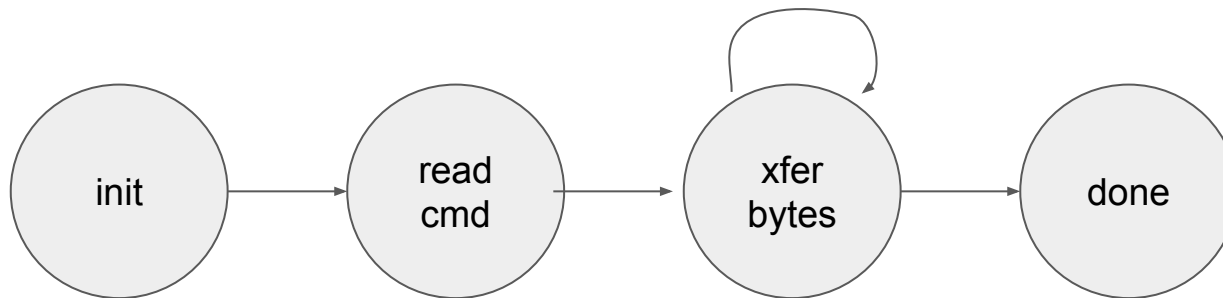
| output<br>section | page | origin   | length   | attributes/<br>input sections    |
|-------------------|------|----------|----------|----------------------------------|
| -----             | ---- | -----    | -----    | -----                            |
| .TI.persistent    |      |          |          |                                  |
| *                 | 0    | 00010000 | 000068ce |                                  |
|                   |      | 00010000 | 00003800 | main.obj (.TI.persistent:ucHeap) |

# Checklist: Checkpoint Commit

- Memory map
- SRAM backup
- RW FRAM contents backup
- Peripheral state
- CPU register backup

# Peripheral State

- Possible I/O peripherals
  - Clock, GPIO, DMA
- Peripherals often maintain an internal state
  - Rate of a clock
  - Mode of DMA
  - Baud rate of UART
- In our case, our program does not involve stateful I/O operations, and we just re-initialize peripheral states without backup/restore



# Peripheral State

- The following states will be re-initialized on system boot (no need to backup/restore)
  - WDT
  - DMA
  - Clock
  - Timer
    - Especially timer for task scheduling
  - e.t.c

```
int main(void)
{
 prvSetupHardware();

 restore();

 ...
}
```

```
static void prvSetupHardware(void)
{
 /* Stop Watchdog timer. */
 WDT_A_hold(__MSP430_BASEADDRESS_WDT_A__);

 /* Disable RTC */
 RTC_C_holdClock(RTC_C_BASE);

 /* Set PJ.4 and PJ.5 for LFXT. */
 GPIO_setAsPeripheralModuleFunctionInputPin(GPIO_PORT_PJ,
 GPIO_PIN4 + GPIO_PIN5,
 GPIO_PRIMARY_MODULE_FUNCTION);

 /* Set DCO frequency to 8 MHz. */
 CS_setDCOFreq(CS_DCORSEL_0, CS_DCOFSEL_6);

 /* Set external clock frequency to 32.768 KHz. */
 CS_setExternalClockSource(32768, 0);

 /* Set ACLK = LFXT. */
 CS_initClockSignal(CS_ACLK, CS_LFXTCLK_SELECT, CS_CLOCK_DIVIDER_1);

 /* Set SMCLK = DCO with frequency divider of 1. */
 CS_initClockSignal(CS_SMCLK, CS_DCOCLK_SELECT, CS_CLOCK_DIVIDER_1);

 /* Set MCLK = DCO with frequency divider of 1. */
 CS_initClockSignal(CS_MCLK, CS_DCOCLK_SELECT, CS_CLOCK_DIVIDER_1);

 /* Start XT1 with no time out. */
 CS_turnOnLFXT(CS_LFXT_DRIVE_0);

 /* Disable the GPIO power-on default high-impedance mode. */
 PMM_unlockLPM5();

 /* Setup DMA */
 dma_param.channelSelect = DMA_CHANNEL_0;
 dma_param.transferModeSelect = DMA_TRANSFER_BLOCK;
 dma_param.transferUnitSelect = DMA_SIZE_SRCWORD_DSTWORD;
 DMA_init(&dma_param);

 /* Setup Timer */
 timer_init();
 timer_start();

 ...
}
```

# Checklist: Checkpoint Commit

- Memory map
- SRAM backup
- RW FRAM contents backup
- Peripheral status backup
- CPU register backup



# CPU Registers Backup

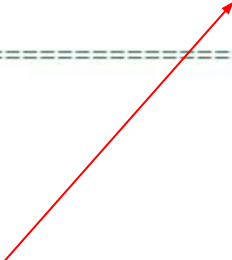
```
=====
; backupReg
; =====
backupReg: .asmfunc

 dint
 nop

 mov_x r1, backup_register + 4*1
 mov_x r2, backup_register + 4*2
 mov_x r3, backup_register + 4*3
 mov_x r4, backup_register + 4*4
 mov_x r5, backup_register + 4*5
 mov_x r6, backup_register + 4*6
 mov_x r7, backup_register + 4*7
 mov_x r8, backup_register + 4*8
 mov_x r9, backup_register + 4*9
 mov_x r10, backup_register + 4*10
 mov_x r11, backup_register + 4*11
 mov_x r12, backup_register + 4*12
 mov_x r13, backup_register + 4*13
 mov_x r14, backup_register + 4*14
 mov_x r15, backup_register + 4*15

 ret_x
.endasmfunc
```

```
#pragma PERSISTENT(backup_register)
uint32_t backup_register[16] = {0};
```



- You need to revise this code to support ping-pong buffer
- Where to backup R0?

# Backup & Restore of PC (R0)

- Method 1
  - Backup: save R0 right before `ret_x`. When executing `mov_x R0, mem_addr`, R0 should already refer to `ret_x`
  - Restore: load R0 in the last step
- Method 2
  - No need to backup R0 as there is a return address of `backupReg` on the stack frame (provided that task stack has been restored)
  - Execute `ret_x` as the last step of restoration, the program will return to the caller of `backupReg`

```
; =====
; backupReg
; =====
backupReg: .asmfunc

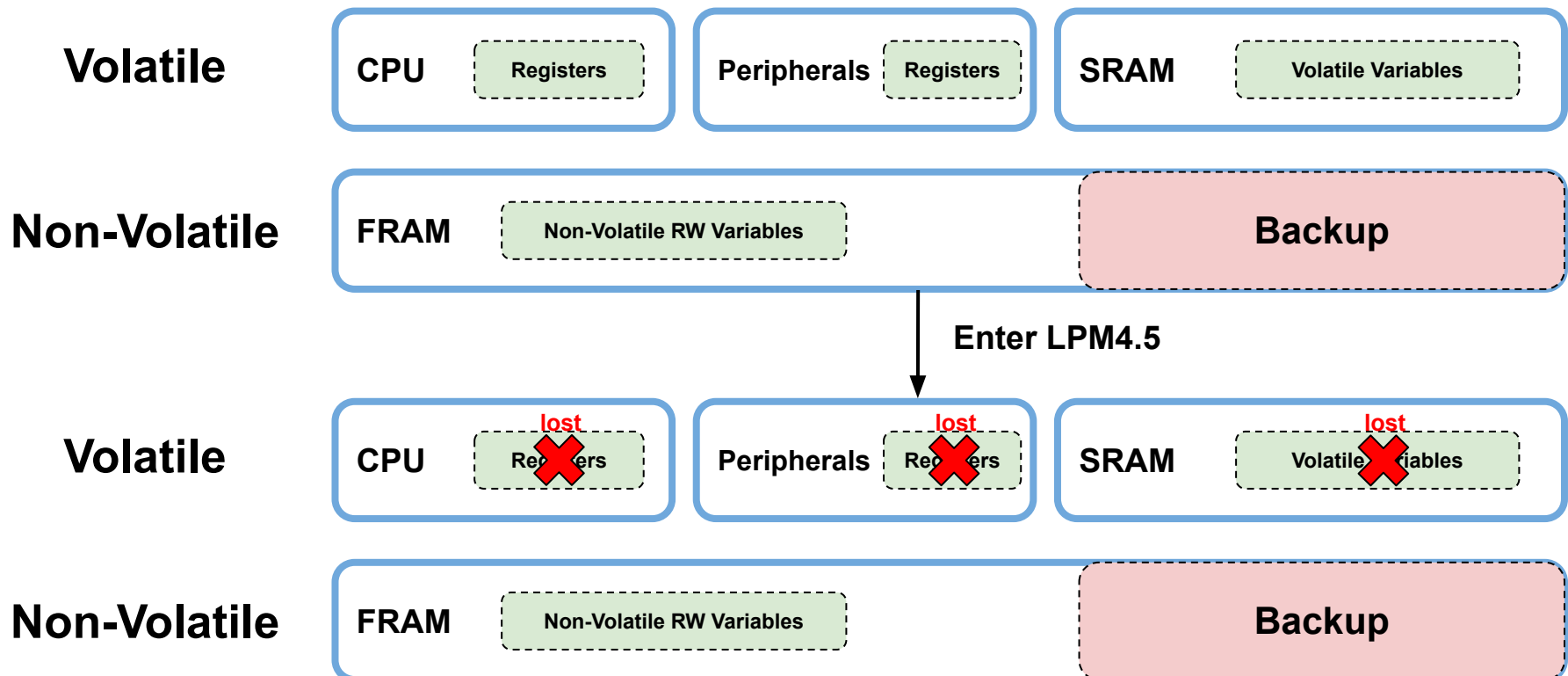
dint
nop

mov_x r1, backup_register + 4*1
mov_x r2, backup_register + 4*2
mov_x r3, backup_register + 4*3
mov_x r4, backup_register + 4*4
mov_x r5, backup_register + 4*5
mov_x r6, backup_register + 4*6
mov_x r7, backup_register + 4*7
mov_x r8, backup_register + 4*8
mov_x r9, backup_register + 4*9
mov_x r10, backup_register + 4*10
mov_x r11, backup_register + 4*11
mov_x r12, backup_register + 4*12
mov_x r13, backup_register + 4*13
mov_x r14, backup_register + 4*14
mov_x r15, backup_register + 4*15

ret_x
.endasmfunc
```

# LPM (Low Power Mode) 4.5

- Instead of powering off the board, we let the board go into a deep sleep mode (LPM 4.5), for which the CPU, SRAM, and IO are un-powered (and thus volatile context will be gone)



# How to enter LPM4.5

Reuse the following code

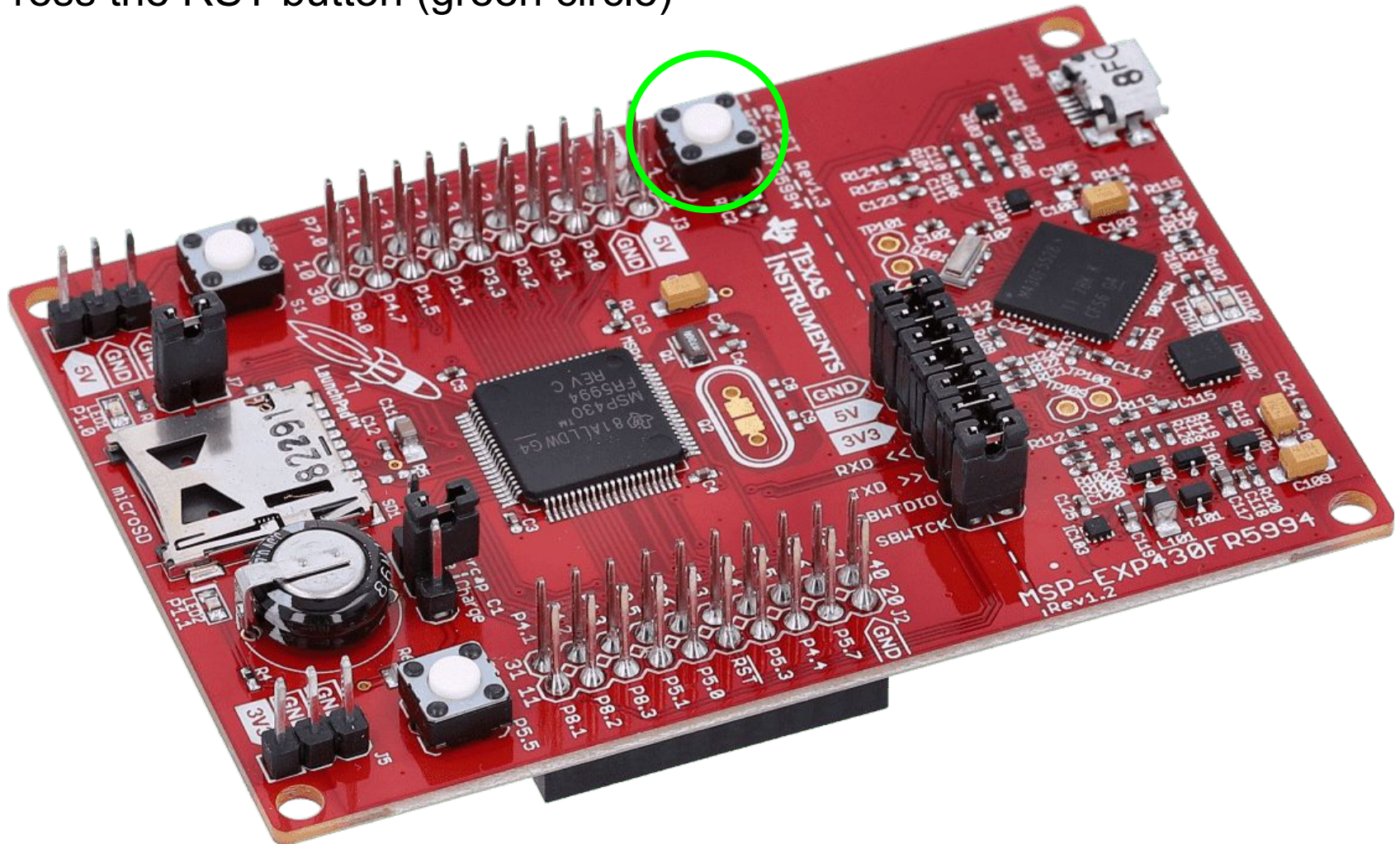
1. Clear GIE bit
2. Unlock PMMCTL0 register
3. Set PMMREGOFF bit in the PMMCTL0 register
4. Disable SVS
5. Lock PMMCTL0
6. Disable CPU, oscillator, system clocks

```
bic #GIE, SR ; Clear GIE bit
mov.b #PMPW_H, &PMMCTL0_H ; Unlock PMMCTL0 register
bis.b #PMMREGOFF, &PMMCTL0_L ; Set PMMREGOFF bit in the PMMCTL0 register
bic.b #SVSHE, &PMMCTL0_L ; Disable SVS
mov.b #000h, &PMMCTL0_H ; Lock PMMCTL0
bis #CPUOFF+OSCOFF+SCG0+SCG1, SR ; Disable CPU, oscillator, system clocks
```

checkout [user guide](#) for more detail !

# Waking up from LPM4.5

Press the RST button (green circle)



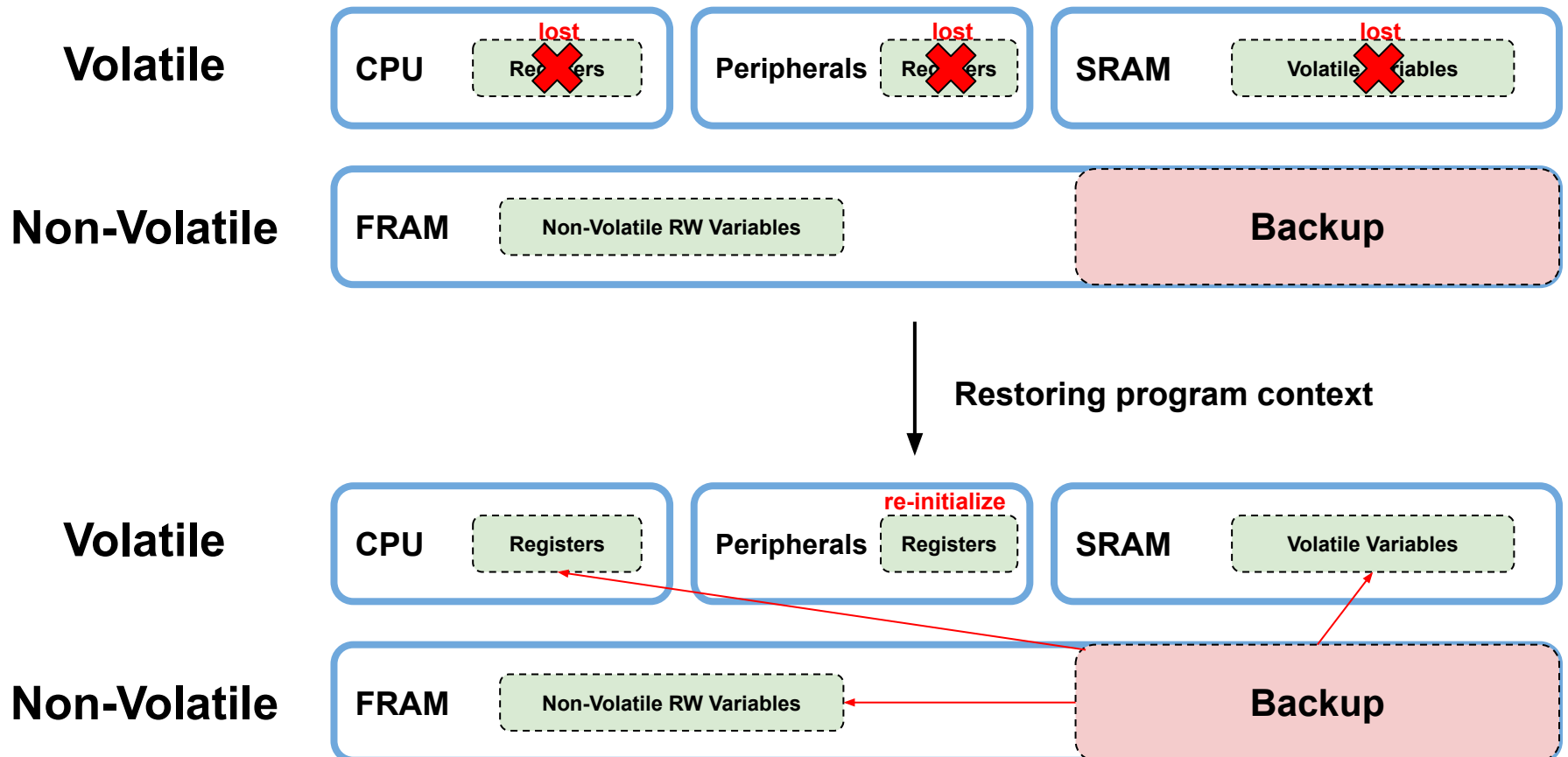
# Checkpoint Restoration

- Check if we are recovering from a power fail
- If so, find out the valid program context backup
- Restore the following contents (in order)
  - ucHeap
  - SRAM
  - CPU registers
- The restoration procedure must be idempotent (in case power fails again during restoration)



# Checkpoint Restoration

- PC points to the entry point of program
- Need to **restore** program context from last checkpoint



# Backup Slices



# The Curse of Stagnation

- Power fails are so frequent and the program cannot reach the next checkpoint
  - Reboot → restore. .. compute ... commit a checkpoint
  - Also known as the no-progress problem
- Possible causes
  - Slow restoration
  - Slow commit
  - No charging?
- Solutions
  - Use finer-granularity memory backup and restore
  - Shorten the checkpoint interval
- Trade-off
  - More metadata, may increase the overhead of checkpointing
  - Extra Backing up memory ....

# Capacitor Issues

CapOS

Recovery

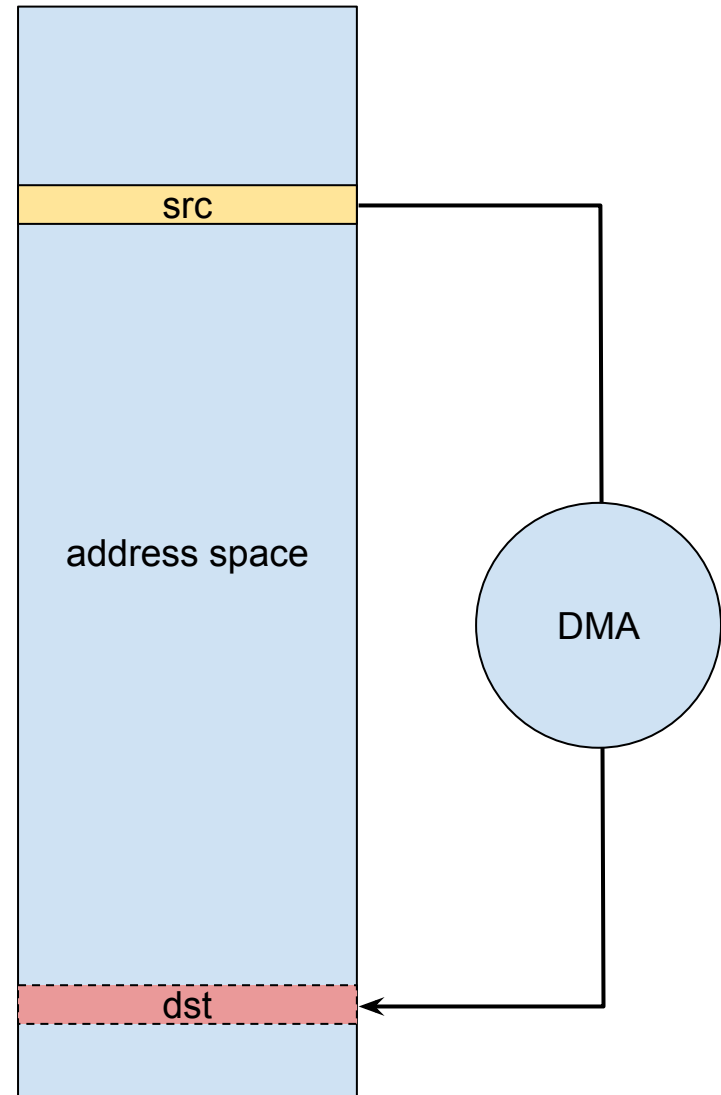
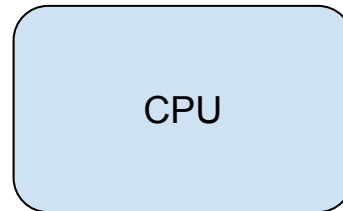
Multiple capacitor (small+large)

Capraba

capacitor attack [elstin]

# DMA

- Transfer data without CPU intervention
  - FRAM → FRAM
  - SRAM → FRAM
  - ADC → SRAM
- DMA transfer unit
  - Byte
  - Word
- DMA transfer can be triggered by
  - Timers
    - Timer interrupt
  - Serial Communication
    - SPI interrupt
    - UART interrupt
  - DMA
    - DMA interrupt



# DMA transfer modes

**Table 11-1. DMA Transfer Modes**

| <b>DMADT</b> | <b>Transfer Mode</b>          | <b>Description</b>                                                                                                        |
|--------------|-------------------------------|---------------------------------------------------------------------------------------------------------------------------|
| 000          | Single transfer               | Each transfer requires a trigger. DMAEN is automatically cleared when DMAxSZ transfers have been made.                    |
| 001          | Block transfer                | A complete block is transferred with one trigger. DMAEN is automatically cleared at the end of the block transfer.        |
| 010, 011     | Burst-block transfer          | CPU activity is interleaved with a block transfer. DMAEN is automatically cleared at the end of the burst-block transfer. |
| 100          | Repeated single transfer      | Each transfer requires a trigger. DMAEN remains enabled.                                                                  |
| 101          | Repeated block transfer       | A complete block is transferred with one trigger. DMAEN remains enabled.                                                  |
| 110, 111     | Repeated burst-block transfer | CPU activity is interleaved with a block transfer. DMAEN remains enabled.                                                 |

[ref.](#)

# DMA: Sample code

- From [user guide](#)

```
// Initialize and Setup DMA Channel 0
/*
 * Base Address of the DMA Module
 * Configure DMA channel 0
 * Configure channel for repeated block transfers
 * DMA interrupt flag will be set after every 16 transfers
 * Use DMA_startTransfer() function to trigger transfers
 * Transfer Word-to-Word
 * Trigger upon Rising Edge of Trigger Source Signal
 */
DMA_initParam param = {0};
param.channelSelect = DMA_CHANNEL_0;
param.transferModeSelect = DMA_TRANSFER_REPEATED_BLOCK;
param.transferSize = 16;
param.triggerSourceSelect = DMA_TRIGGERSOURCE_0;
param.transferUnitSelect = DMA_SIZE_SRCWORD_DSTWORD;
param.triggerTypeSelect = DMA_TRIGGER_RISINGEDGE;
DMA_init(¶m);
/*
 * Base Address of the DMA Module
 * Configure DMA channel 0
 * Use 0x1C00 as source
 * Increment source address after every transfer
 */
DMA_setSrcAddress(DMA_CHANNEL_0,
 0x1C00,
 DMA_DIRECTION_INCREMENT);

/*
 * Base Address of the DMA Module
 * Configure DMA channel 0
 * Use 0x1C20 as destination
 * Increment destination address after every transfer
 */
DMA_setDstAddress(DMA_CHANNEL_0,
 0x1C20,
 DMA_DIRECTION_INCREMENT);

// Enable transfers on DMA channel 0
DMA_enableTransfers(DMA_CHANNEL_0);

while(1)
{
 // Start block transfer on DMA channel 0
 DMA_startTransfer(DMA_CHANNEL_0);
}
```