EMBSYS 110, Spring 2017

- Week 9
  - Optimization (continued)
  - Bootloader and firmware upgrade
  - Low power modes
  - Open source toolchain
  - Other statechart tools

Optimization for Code – Multiple Instances

- Now we move on to optimization for code space.
- More than often you would need more than one instance of the same class.
- For example in our demo project we have an active object class for the user button named UserBtn. (Similarly we have one for the user LED named UserLed.)
- It serves well to interface with the user button and user LED.
- What if we want to add more buttons and LED's?

Optimization for Code – Multiple Instances

- One way is to create another active object class for the new button/LED, such as MenuBtn or StatusLed, etc.
- Let’s check what will be different in each class. It turns out we will be duplicating a lot of code.
- A more optimized way is to parameterize the class such that the GPIO port/pin used for the button or LED are configuration parameters rather than hard-coded.
- The configuration parameters will be a members of the class, such as *m_port* and *m_pin*.
- Those parameters can either be passed in via the constructor, or be contained in a static constant array of configuration structures.

Optimization for Code – Multiple Instances

- Example:

```
typedef struct {
    uint8_t id;             // HSM ID
    GPIO_TypeDef *port;     // GPIO port
    uint32_t pin;           // GPIO pin
} LedConfig;

static const LedConfig CONFIG[] = {
    { USER_LED,    GPIOA, GPIO_PIN_5 },
    { USER_STATUS, GPIOC, GPIO_PIN_1 }
};
```

- Note that you may need more HW specific parameters (e.g. if PWM is used, you'll need to parameterize HW timer used).
- Note how a line in the structure array replaces an entire duplicated class.

Optimization for Code – Multiple Instances

- Another example of parameterizing UartAct (in UartAct.h):

```
class UartAct : public QActive {
  ...
  typedef struct {
      uint8_t id;
      USART_TypeDef *uart;    // Common parameters.
      IRQn_Type uartIrq;
      uint32_t uartPrio;
      GPIO_TypeDef *txPort;   // Tx parameters.
      uint32_t txPin;
      uint32_t txAf;
      DMA_Channel_TypeDef *txDmaCh;
      uint32_t txDmaReq;
      IRQn_Type txDmaIrq;
      uint32_t txDmaPrio;
      // Rx parameters (similar to above) ...
  } Config;
  static Config const CONFIG[];
  Config const *m_config;
}
```

Optimization for Code – Multiple Instances

- Definition of UART configuration in UartAct.cpp:

```
// Define UART configurations.
UartAct::Config const UartAct::CONFIG[] = {
    { // Common parameters.
      UART2_ACT, USART2, USART2_IRQn, USART2_IRQ_PRIO,
      // Tx parameters.
      GPIOD, GPIO_PIN_5, GPIO_AF7_USART2, DMA1_Channel7,
      DMA_REQUEST_2, DMA1_Channel7_IRQn, DMA1_CHANNEL7_PRIO,
      // Rx parameters.
      GPIOD, GPIO_PIN_6, GPIO_AF7_USART2, DMA1_Channel6,
      DMA_REQUEST_2, DMA1_Channel6_IRQn, DMA1_CHANNEL6_PRIO },
      // Add more configurations here...
};
```

Optimization for Code – Multiple Instances

- Matching a configuration structure in the constructor (in UartAct.cpp):

```
UartAct::UartAct(uint8_t id, char const *name, ...) :
    Active((QStateHandler)&UartAct::InitialPseudoState, id, name, ...),
    m_config(NULL),
    ...
    uint32_t i;
    for (i = 0; i < ARRAY_COUNT(CONFIG); i++) {
        if (CONFIG[i].id == id) {
            m_config = &CONFIG[i];
            break;
        }
    }
}
```

Optimization for Code – Multiple Instances

- UART initialization based on configuration parameters (no longer hardcoded):

```
void UartAct::InitUart() {
    switch((uint32_t)(m_config->uart)) {
        case (uint32_t)USART2: __HAL_RCC_USART2_CLK_ENABLE(); break;
        // Add more cases here...
        default: FW_ASSERT(0); break;
    }
    // UART TX GPIO pin configuration.
    GPIO_InitTypeDef  gpioInit;
    gpioInit.Pin       = m_config->txPin;
    gpioInit.Alternate = m_config->txAf;
    HAL_GPIO_Init(m_config->txPort, &gpioInit);
    // Configure the DMA handler for TX
    m_txDmaHandle.Instance       = m_config->txDmaCh;
    m_txDmaHandle.Init.Direction = DMA_MEMORY_TO_PERIPH;
    m_txDmaHandle.Init.Request   = m_config->txDmaReq;
    HAL_DMA_Init(&m_txDmaHandle);
    __HAL_LINKDMA(&m_hal, hdmatx, m_txDmaHandle);
    ...
}
```

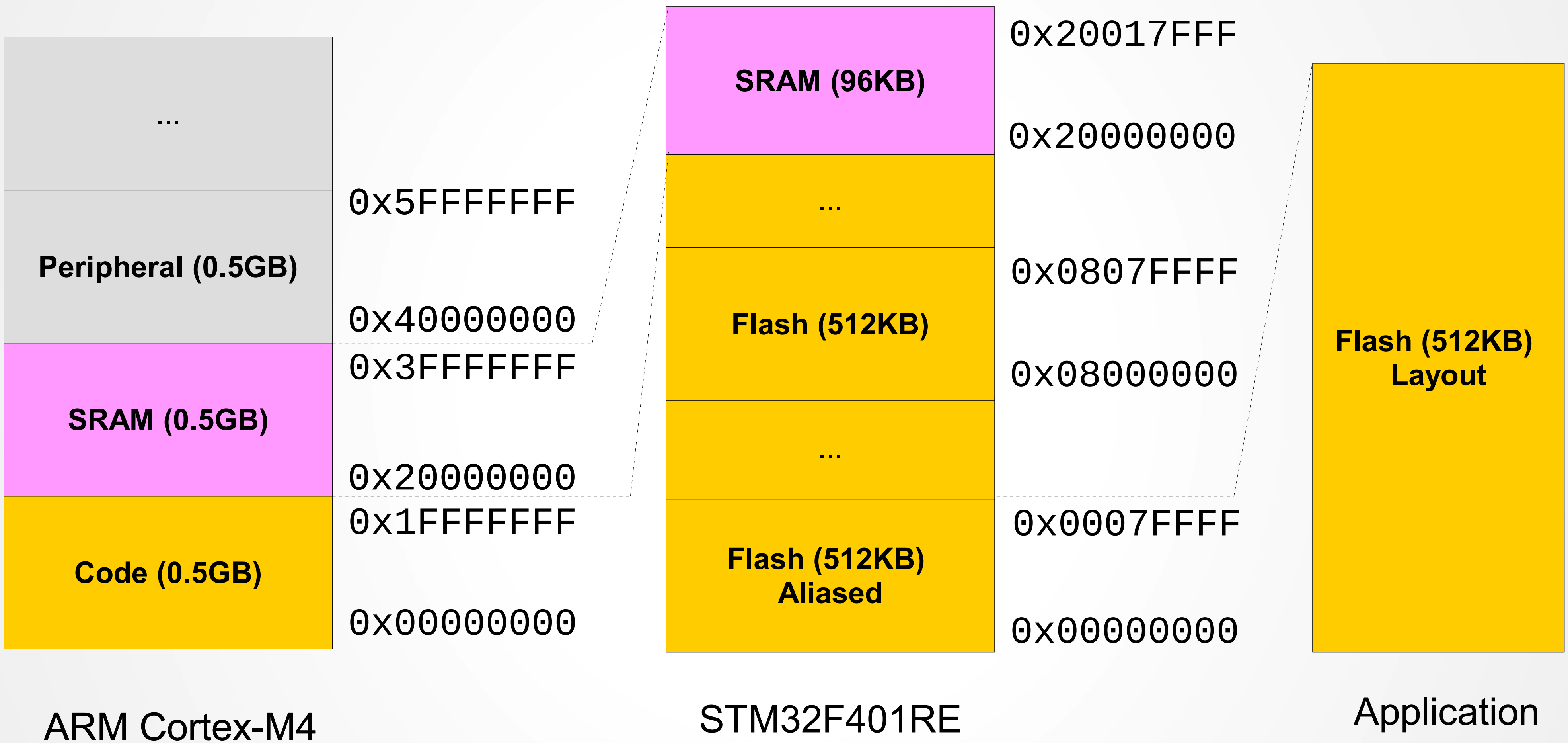
Optimization – Non-blocking Architecture

- With QP we can encapsulate multiple HSM (regions) in a single active object.
- In essence it allows multiple HSM's to share a single *thread*.
- This is possible due to the fundamental non-blocking run-to-completion nature of HSM's.
- Question – Why is it not possible with blocking design?
- Question – What is the difference between making an HSM a region than creating its own active object?
- Less threads → Less memory overhead.
- Less threads → Less context switching overhead (faster)
- Compare to Node.js which also uses a non-blocking architecture.

Bootloader and Firmware Upgrade

- For any embedded systems, it is important to understand one's memory map.
- For example, we need to understand it for:
  - Debugging purpose. Given an address like 0x08006924 or 0x20006AF0, we should have an idea whether it is in Flash or SRAM.
  - Designing bootloader and upgrade strategy.
- There are different levels of memory maps, from the highest level defined by Cortex-M4, through the middle level defined by the chip vendor (STM32F401) and finally to your application-specific Flash layout.

Bootloader and Firmware Upgrade



Bootloader and Firmware Upgrade

- Let's verify our memory map with the map file.
- The map files can be found here:

  Projects\MyApp\EWARM\stm32f401xe_flash.icf

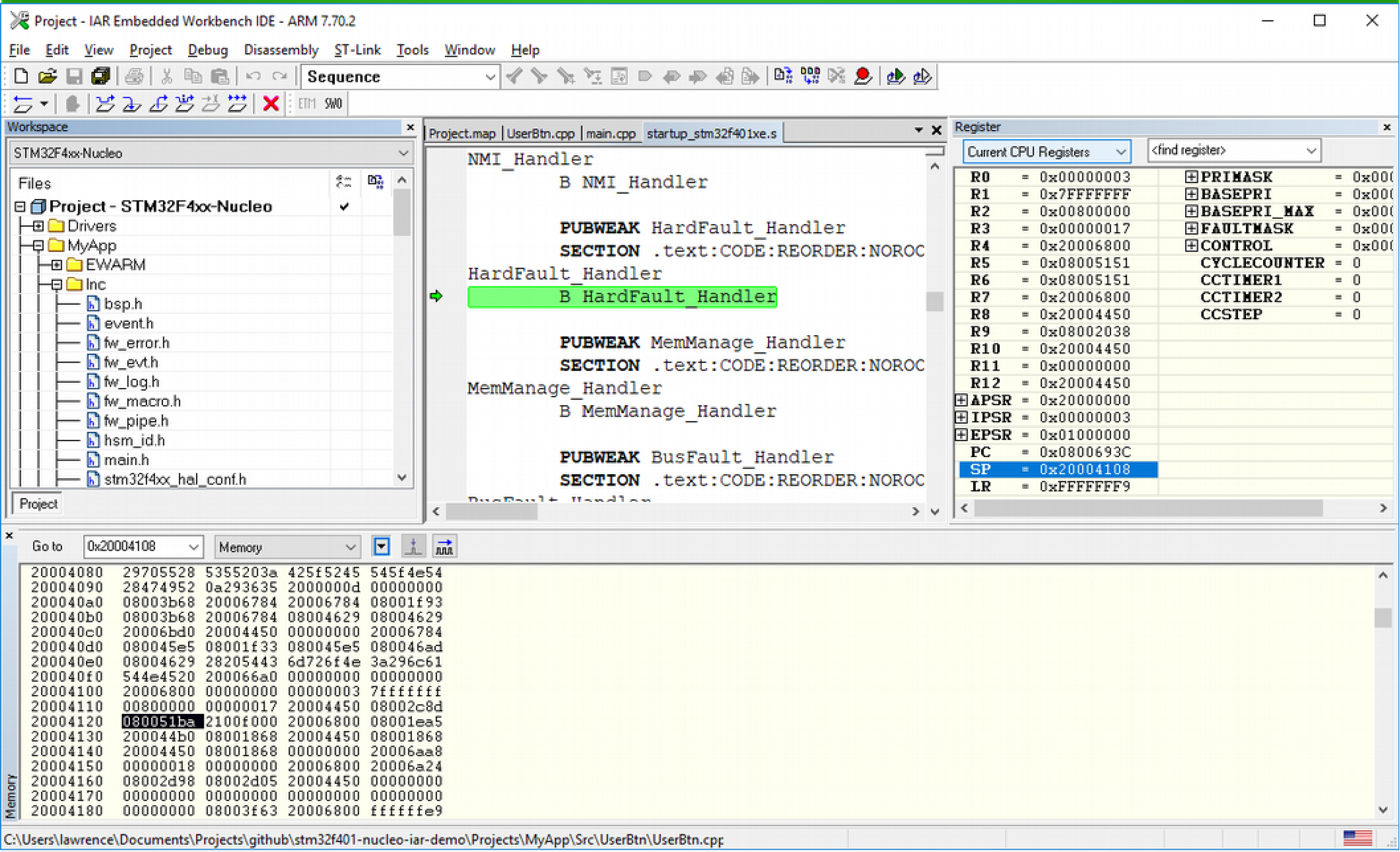
```
.intvec ro code  0x08000000   0x194  startup_stm32f401xe.o [1]
.text   ro code  0x08000194   0x840  System.o [1]
.text   ro code  0x080009d4    0x34  qf_qact.o [1]
 ...
.data   inited   0x20000000     0x4  stm32f4xx_nucleo.o [1]
.data   inited   0x20000004     0x4  System.o [1]
.data   inited   0x20000008    0x14  system_stm32f4xx.o [1]
```

Bootloader and Firmware Upgrade

- A debugging example:

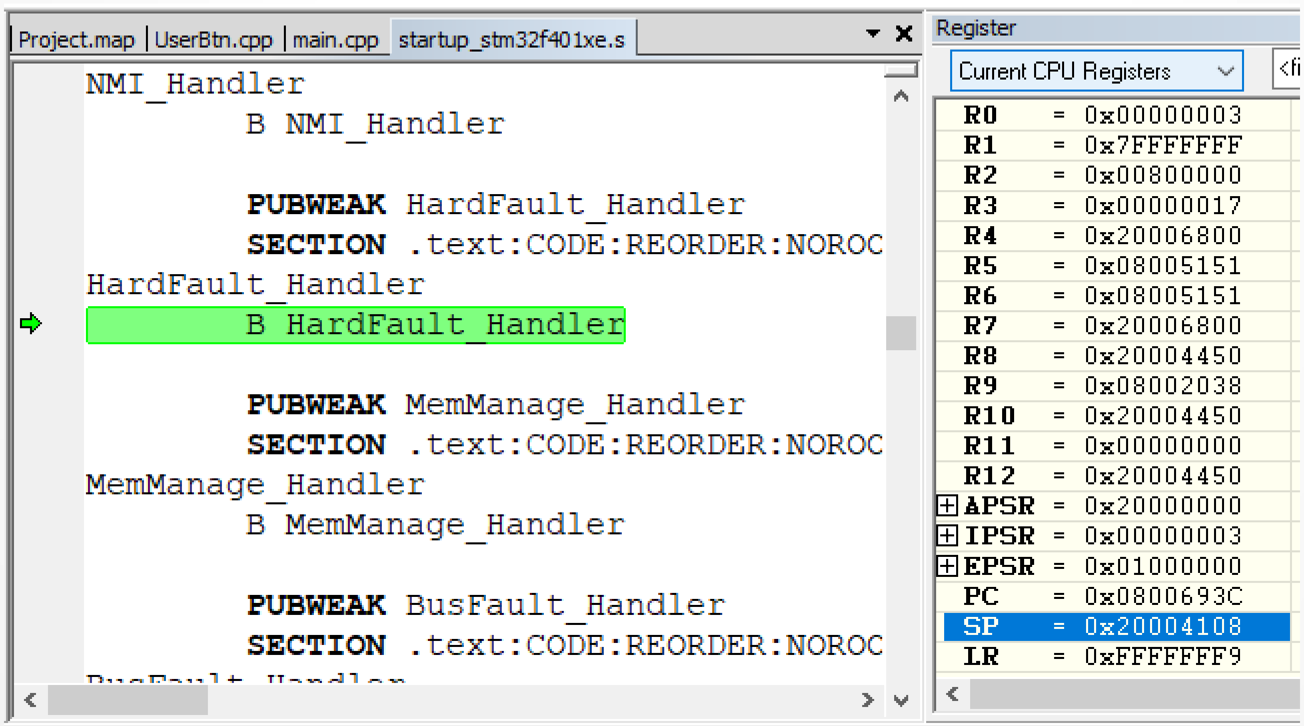
```
QState UserBtn::Up(UserBtn * const me, QEvt const * const e) {
    QState status;
    switch (e->sig) {
        case USER_BTN_TRIG: {
            LOG_EVENT(e);
            EnableGpioInt();
            if (HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_13) == GPIO_PIN_RESET) {
                ...
                *(uint32_t *)(0x7fffffff) = 0x1234;
            }
            status = Q_HANDLED();
            break;
        }
```

Bootloader and Firmware Upgrade



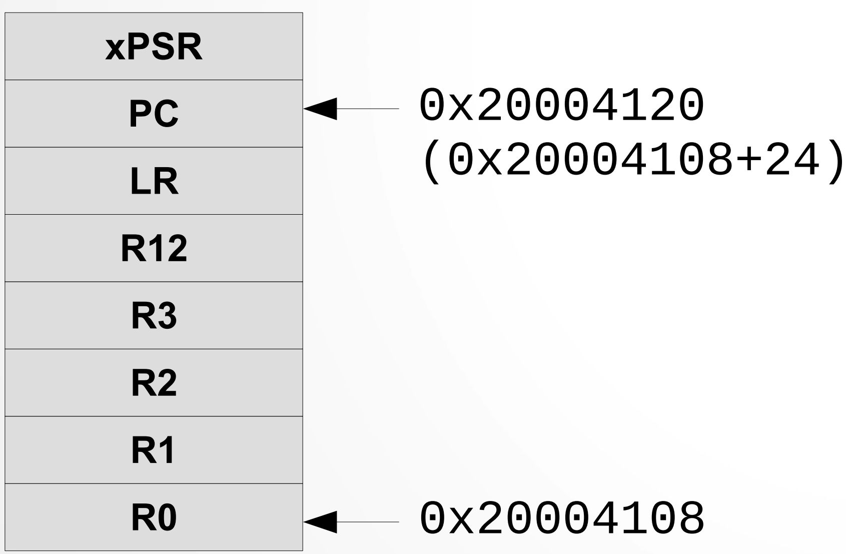
Bootloader and Firmware Upgrade

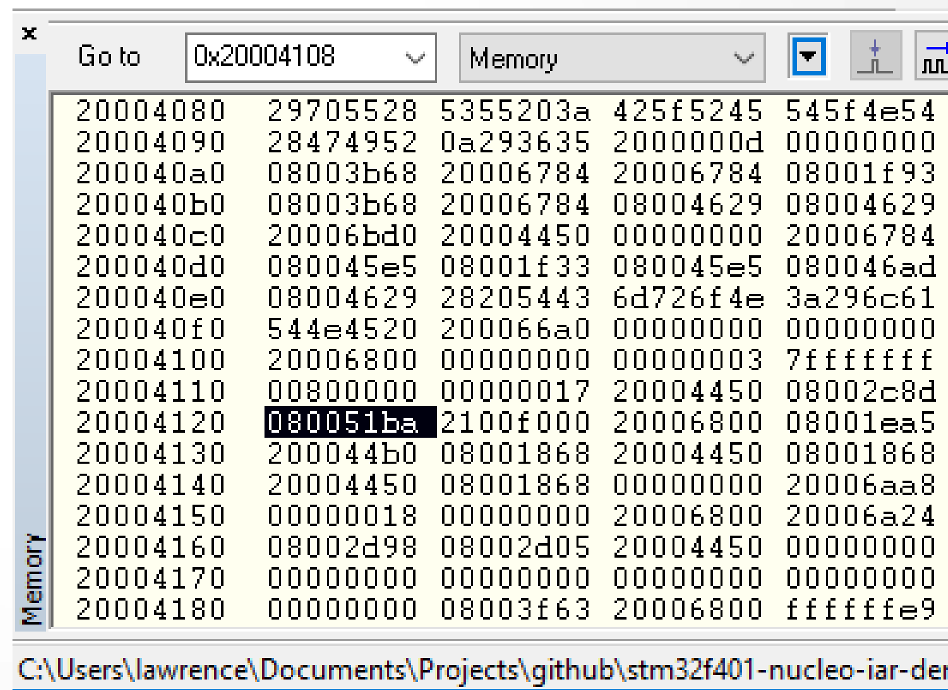
- System is hung. Break the execution and it shows it is at HardFault_Handler.
- Exception stack frame is automatically pushed to the current stack, the top of which is pointed to by the Stack Pointer (SP).
- SP is at 0x20004108 in the Register window.



Bootloader and Firmware Upgrade

- Now we need to review how an exception stack frame looks like as we saw in the first class.
- The exception stack frame captures the register contents when hardfault occurs.
- From the Memory window we can see the PC (Program Counter) when exception occurs is 0x080051ba.





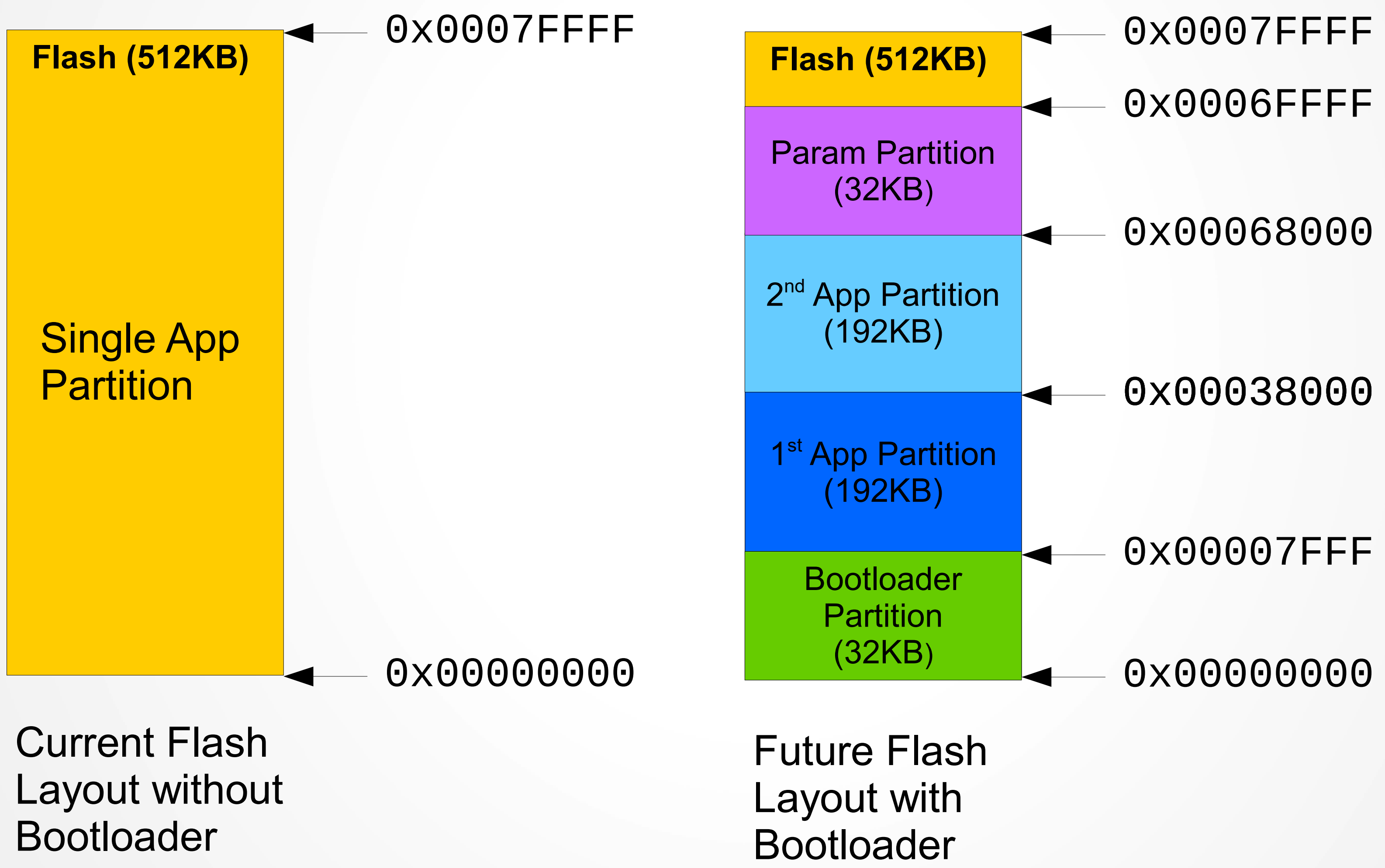
Bootloader and Firmware Upgrade

- The last piece of information to find out is which function the address 0x080051ba corresponds to.
- We can make use of the map file Projects\MyApp\EWARM\stm32f401xe_flash.icf.
- Even though we don’t know the exact instruction causing the hardfault, we have successfully narrow it down to the function userBtn::Up().

```
APP::UserBtn::Stopped(APP::UserBtn*, QP::QEvt const *)
                        0x0800507d    0x5a  Code  Gb  UserBtn.o [1]
APP::UserBtn::Up(APP::UserBtn *, QP::QEvt const *)::__FUNCTION__
                        0x080052c4     0x4  Data  Lc  UserBtn.o [1]
APP::UserBtn::Up(APP::UserBtn*, QP::QEvt const *)
                        0x08005151    0x74  Code  Gb  UserBtn.o [1]
APP::UserBtn::UserBtn()
                        0x08004fa1    0x42  Code  Gb  UserBtn.o [1]
APP::UserBtn::~UserBtn()
                        0x08006919     0xc  Code  Gb  UserBtn.o [1]
```

Bootloader and Firmware Upgrade

- Now let's look at our flash layout:

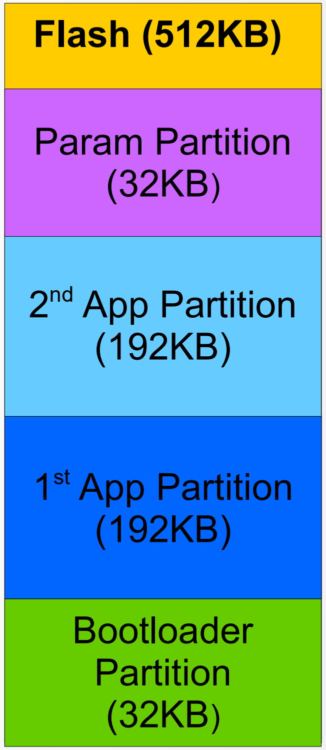


Bootloader and Firmware Upgrade

- Currently, just like any ST demo projects, we don't use a bootloader.
- We just use the ST-Link debugger to download our application to the beginning of flash.
- When we talk about an address we need to be careful about where it is based from.
- For example, it can be based from the beginning of flash (as on the previous page). That is 0x00000000 is the beginning of flash.
- Alternative, it can be based from the beginning of the processor memory map. That is 0x08000000 is the beginning of flash (as shown in the map file), which is also aliased to 0x00000000.
- In our discussion here, we use the first interpretation.

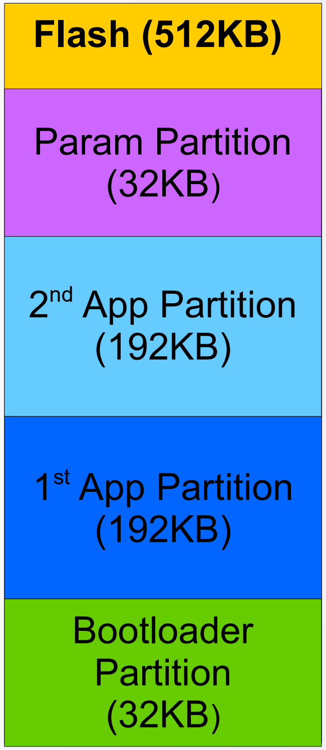
Bootloader and Firmware Upgrade

- In real production code we typically use a bootloader.
- A bootloader is relatively stable, well-tested. It usually does not require update in the field. When it is updated, there is a chance that it got corrupted and the device would be “boat-anchored”.
- A bootloader checks which App Partition is current and verifies its integrity (via CRC). It may optionally authenticate its signature.
- Finally a bootloader jumps to the verified/authenticated partition to start execution.
- If verification or authentication fails, a bootloader may notify the user or start recovery.
- What would happen if a bootloader jumps to a corupted applicaton without verification, and it hangs right away?



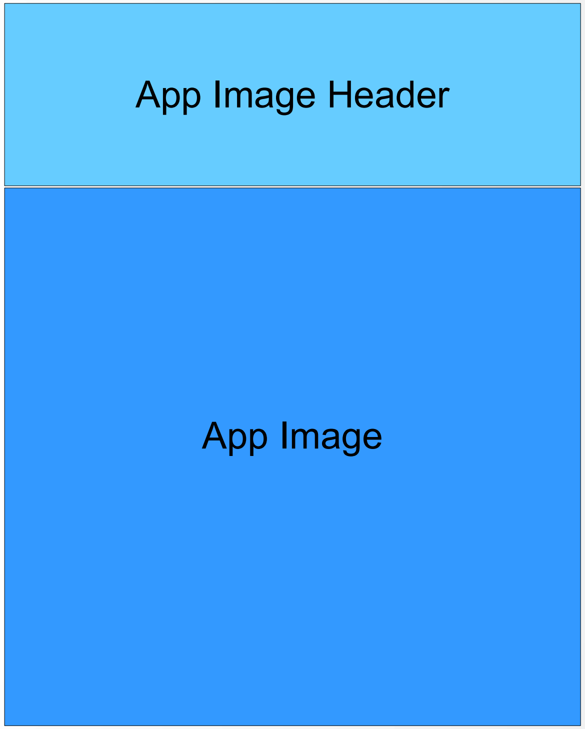
Bootloader and Firmware Upgrade

- We need two application partitions.
- When the 1st App Partition is current (i.e. it is currently executing from it), it downloads and saves the new application image to the 2nd App partition, and vice versa.
- Firstly, it avoids overwriting the flash sector that it is currently executing from. Flash doesn't like it.
- Secondly, it avoids corrupting the current App Partition in case the new application image is invalid (due to download or authentication error). It allows the current application to verify and authenticate the new image before switching over to it. It is for fail-safe upgrade.
- The remaining space can be used for configuration and manufacturing parameters, etc.



Bootloader and Firmware Upgrade

- We need a header for each App Image. Typical fields in the header include:
  - Magic word.
  - Version number.
  - Offset to the App Image (to skip over the header).
  - Length of the App Image.
  - CRC32 of the App Image for integrity check.
  - Signature of the App Image for authentication.

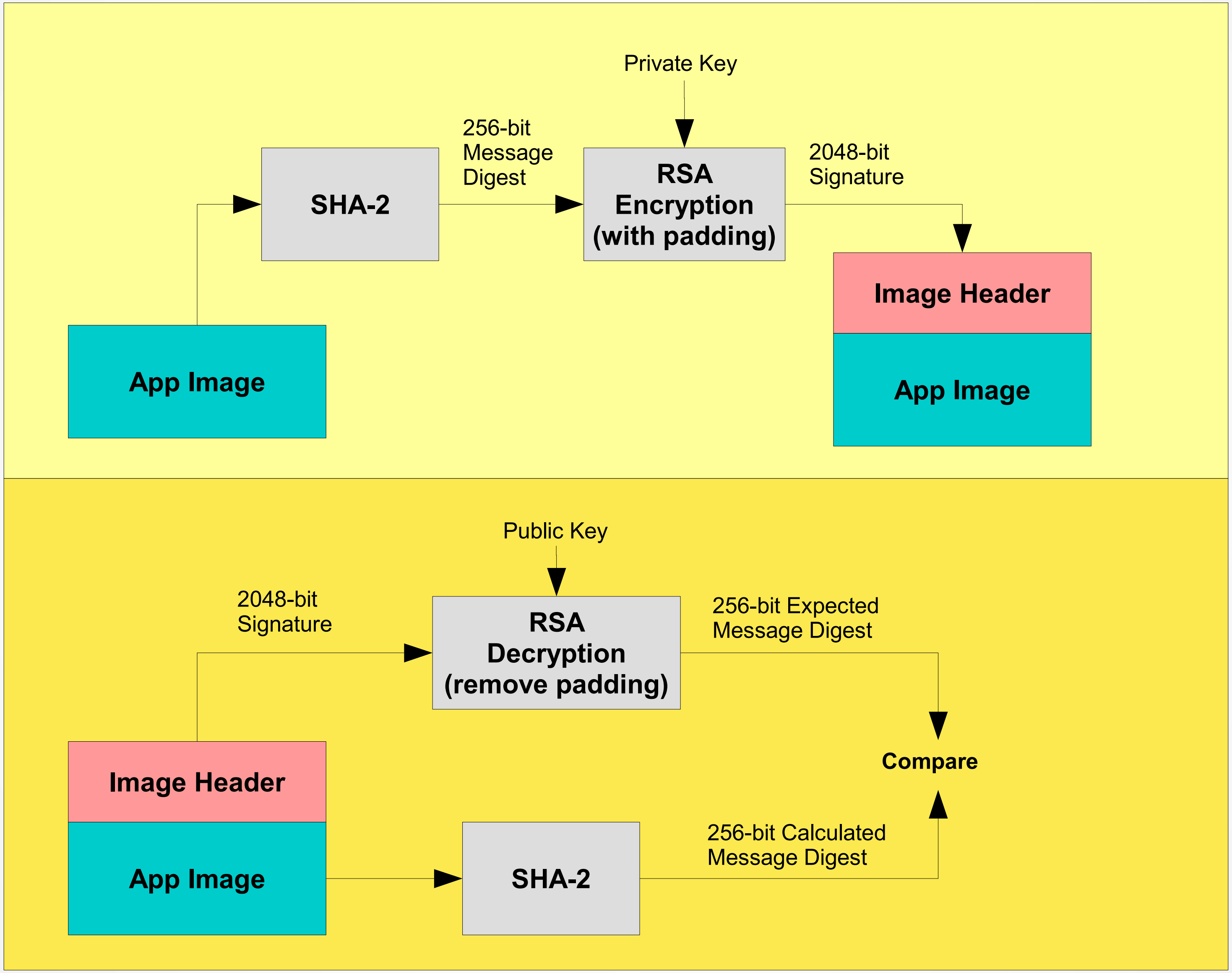


App Partition Layout (192KB)

Bootloader and Firmware Upgrade

- App Image authentication:
  1. Generate a fixed length message digest (MD) of the upgrade image using a secure hash algorithm, e.g. SHA-2 (which stands for Secure Hash Alogrithm 2). It supports different MD lengths, such as 256-bit, ie. 32 bytes.
  2. The key feature of MD is that the probability of having the same MD when the image is altered is super low. Compare to checksum or CRC32.
  3. Encrypt the MD with public-key cryptography such as RSA, using the private key that is only accessible to the manufacturer. Typical block size is 1024-bit (128 bytes) or 2048-bit (256 bytes).
  4. The encrypted MD is the ***signature*** of the App Image stored in the header.
  5. Open source cryto library (mbed TLS) - https://tls.mbed.org/

Bootloader and Firmware Upgrade



Low Power Modes

- STM32F401 has three low-power modes:
  - Sleep mode
    - Cortex-M4 with FPU core stopped.
    - Peripherals kept running.
    - Entered whenever no tasks/active object are running (RTOS/QP idle loop)
  - Stop mode
    - All clocks in the 1.2V domain stopped.
    - SRAM and register contents preserved.
    - Entered when system has been idle for some time, but fast wake-up is required.
  - Standby mode
    - Voltage regulator disabled.
    - 1.2V domain powered off.
    - SRAM and register contents lost (except in backup domain).
    - Entered when system is expected to be idle for a long period of time. Wake-up is slow since it requires a full reboot.

Low Power Modes

Table 15. Low-power mode summary

Mode name	Entry	Wakeup	Effect on 1.2 V domain clocks	Effect on V_{DD} domain clocks	Voltage regulator
Sleep (Sleep now or Sleep-on-exit)	WFI or Return from ISR	Any interrupt	CPU CLK OFF no effect on other clocks or analog clock sources	None	ON
	WFE	Wakeup event			
Stop	PDDS bit + STOP mode configuration + SLEEPDEEP bit + WFI, Return from ISR or WFE	Any EXTI line (configured in the EXTI registers, internal and external lines)	All 1.2 V domain clocks OFF	HSI and HSE oscillators OFF	Main regulator or Low-Power regulator (depends on *PWR power control register (PWR_CR)*)
Standby	PDDS bit + SLEEPDEEP bit + WFI, Return from ISR or WFE	WKUP pin rising edge, RTC alarm (Alarm A or Alarm B), RTC Wakeup event, RTC tamper events, RTC time stamp event, external reset in NRST pin, IWDG reset			OFF

Open Source Tools

- IAR full license is expensive, but you can get away by using:
  - VisualGDB Embedded ($89)
    - See https://visualgdb.com/buy/
  - Eclipse IDE with GCC and openocd (many to choose from)
    - System Workbench for STM32
      - Third-party but referenced by STM32 website
      - http://www.st.com/en/development-tools/sw4stm32.html
    - Roll-your-own
      - Works for me.
      - http://www.carminenoviello.com/
    - CooCox (http://coocox.org/software.html)
  - mbed (https://www.mbed.com/en/)

Other Statechart Tools

- Statechart is a graphical design method that applies to embedded systems and beyond.
- QP is just one of the many tools that help translate statechart diagrams to code. Others include:
  - Yakindu (free community version)
    - https://www.itemis.com/en/yakindu/state-machine/
  - VisualState by IAR (free 30-state evaluation)
    - https://www.iar.com/iar-embedded-workbench/add-ons-and-integrations/visualstate/
  - SCION-CORE (not for embedded, but for Javascript)
    - Can be used with **Node.js** for web server and with **React** for web client.
    - Now you can use statecharts from web front-end all the way to embedded (IoT)
    - https://github.com/jbeard4/SCION-CORE
    - See Jacob Beard's master thesis "Developing Rich, Web-Based User Interfaces with the **S**tate**c**harts **I**nterpretation and **O**ptimization E**n**gine"

Other Statechart Tools (SCION)

```
var statechartModel = {
        states : [
           {
              id : 'idle',
              onEntry : function(){
                 rectNode.textContent='idle';
              },
              transitions : [
                 {
                    event : 'mousedown',
                    target : 'dragging',
                    onTransition : function(event){
                       eventStamp = firstEvent = event.data;
                    }
                 }
              ]
           },
```

Other Statechart Tools (SCION)

```
    {
        id : 'dragging',
        onEntry : function(){
            rectNode.textContent='dragging';
        },
        transitions : [
            {
                event : 'mouseup',
                target : 'idle'
            },
            {
                event : 'mousemove',
                target : 'dragging',
                onTransition : function(event){
                    var dx = eventStamp.clientX - event.data.clientX;
                    var dy = eventStamp.clientY - event.data.clientY;

                    rectNode.style.left = (rectX -= dx) + 'px';
                    rectNode.style.top = (rectY -= dy) + 'px';

                    eventStamp = event.data;
                }
            }
        ]
    }
  ]
};
```

Other Statechart Tools (SCION)