

PSoC™ Roadshow Technical Workshop

Day 1 – PSoC™ 4 and ModusToolbox™

Objective: Understand the basic workflow of ModusToolbox™ leveraging low-level peripheral driver APIs, ModusToolbox™ Device Configurator, and CAPSENSE middleware

Prerequisites and Important Information

Installed Software:

- ModusToolbox™ Tools Package v3.0.0

Hardware being used:

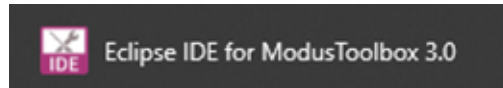
- PSoC™ 4100S Plus Prototyping Kit ([CY8CKIT-149](#))

Provided workshop collateral:

- **PSoC Roadshow Workshop Session 1 Lab Guide.pdf** (this document)
- Files to be referenced during workshop
 - **PSoC_4_Code_Snippets.c** – reference file containing blocks of code to be copied into project
 - **game.c / game.h** – additional project files to add game mechanics into workshop project
 - **final_main.c** – reference file containing final main.c implementation

Section 1 – Importing and Exploring a ModusToolbox code example

1. Open **Eclipse IDE for ModusToolbox v3.0**



2. Create a New Application

- Create new Workspace directory within User directory (example c:\users\username\psoc4_workshop)
- From Quick Panel select **New Application**, this action will launch the Project Creator
- Select the board support package: **PSoc™ 4 BSP -> CY8CKIT-149**
- Press **Next**, review available Code Examples
- Check the template application: **Getting Started -> Hello World**
- (Optional) Change project name to **MyProject** (or preferred name)
- Select **Create** to clone the application, create the eclipse project, and import the required BSP files and libraries

3. Review key element of the Eclipse IDE for ModusToolbox

- Use Quick Panel to start **building** the MyProject application

4. Review mtb_shared folder structure. Some key elements to review are:

- Library folders with release subfolders, README files, and Documentation folders

5. Review project folder structure. Some key elements to review are:

- Top level project folder – main.c, Makefile, subfolders
- README files
- **deps** – dependency directory for application, including user selected libraries and BSP
- **libs** – mtb workflow files, make recipe, BSP folder (files managed with ‘make getlibs’)
- **COMPONENT_CUSTOM_DESIGN_MODUS/TARGET_CY8CKIT-149** - Board Support Package

6. Review **main.c** of MyProject code examples. Key element to notice:



- Device bring-up and initialization: **cybsp_init()**
- Configuration of SCB UART using PDL APIs (Init and Enable)
- Serial printout of text string
- Main while loop with GPIO toggle and delay function

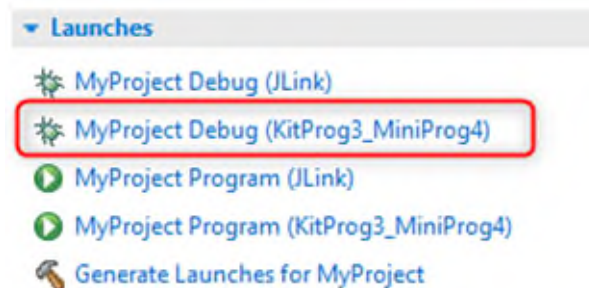
7. Add UART PutString command to clear Serial Terminal Console


- Locate provided **PSoc_4_Code_Snippets.c** file, and open in Eclipse IDE (drag-and-drop) or preferred editor.
- Add **code to “clear” serial terminal** to **main.c** file (copy/paste) just before existing PutString function

```
Cy_SCB_UART_PutString(CYBSP_UART_HW, "\x1b[2J\x1b[;H");
```

Section 2 – Debugging with Eclipse IDE for ModusToolbox

1. Open a Serial Terminal within the Eclipse IDE for ModusToolbox
 - Ensure development board is connected to PC using the USB cable
 - Select the Terminal panel (in lower section of IDE)
 - Select the Open a Terminal icon  (or press Ctrl-Alt-Shift-T)
 - Choose terminal: **Serial**
 - **Configure** Serial Port (likely the last entry in the list)
 - Baud rate: **115200** (default)
 - Recommended: Relocate the Terminal panel to a new location within the IDE
2. **Flash** application to development board and launch a **debug** connection
 - Ensure the application has finished compiling with no error within the Console
 - Ensure MyProject is selected (the active project within Eclipse)
 - Select the appropriate  **MyProject Debug (KitProg3_MiniProg4)** launch action from the Quick Panel



- Once a debug session has been established, the IDE will switch to the **Debug tab** and an initial breakpoint at main will be set. There will now be an active Debug toolbar available.
3. Explore Debugging options with in Eclipse IDE for ModusToolbox
 - Setting **breakpoints** in application, within the main for loop
 - **Step Over** and **Step Into** functions
 - Open Peripherals views, paying attention to:
 - **SCB[3]** (DEBUG UART), initialized in main
 - **GPIO -> PRT[3]** (USER LED), initialized during cybsp_init, and toggled within main loop
 - **Resume** action to run until breakpoints
 - Viewing **variables** and creating **expressions**
 - **Restarting** the target while maintaining debug session
 - **Terminating** a Debug session and returning to Project Explorer

Section 3 – Extending to a Pushbutton controlled LED with Low-Level Drivers

1. Configuring a GPIO input signal for use with generated PDL code

- Ensure the correct project is selected, and open **Device Configurator** (under BSP Configurators)
- Review features and panels available within Device Configurator, note existing configurations: SCB UART, CAPSENSE, and USER_LED
- From the Pins tab enable the USER_BTN signal
 - Use the filter text entry to search for “USER”
 - Locate the CYBSP_USER_BTN resource, enable with check box, and add new macro **SW1**
 - Review Parameter panel and configure as Digital Input
 - Set Drive Mode to **Resistive Pull-Up, Input buffer on**
 - Set Interrupt Trigger Type to **Falling Edge**
- Review **Code Preview** panel and observe define macros (based on **SW1_**)
- Generate code by saving the design.modus file (**File -> Save / Ctrl-S**)
- Exit the Device Configurator (**File -> Exit / Alt-F4**)

2. Register interrupt callback function for GPIO Pushbutton

- Review PDL documentation, exact code snippet is also available within [PSoC 4 Code Snippets.c](#)

```
cy_stc_sysint_t ButtonIRQ_cfg = { .intrPriority = 3, .intrSrc = SW1_IRQ };  
Cy_SysInt_Init( &ButtonIRQ_cfg, Button_Callback );  
NVIC_EnableIRQ( SW1_IRQ );
```
- Initialize and Enable interrupt for GPIO
 - Configuration structure, with priority and source
 - Initialization with callback function
 - Enabling of interrupt source with NVIC controller
- Create Button callback function, cut/paste **GPIO_Inv_xxxx** function from FOR loop



```
void Button_Callback(void)  
{  
    Cy_GPIO_ClearInterrupt(SW1_PORT, SW1_PIN);  
    Cy_GPIO_Inv(CYBSP_USER_LED1_PORT, CYBSP_USER_LED1_PIN);  
}
```

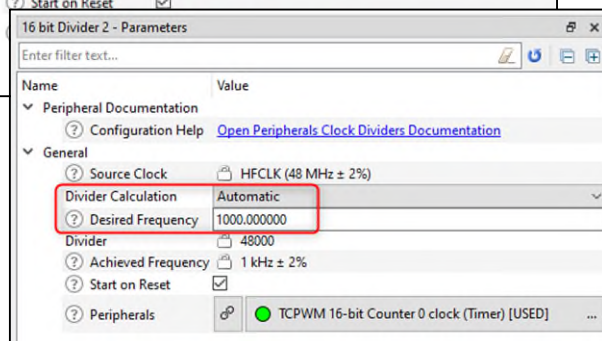
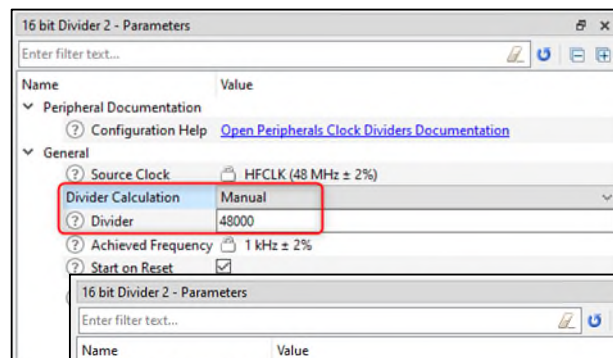
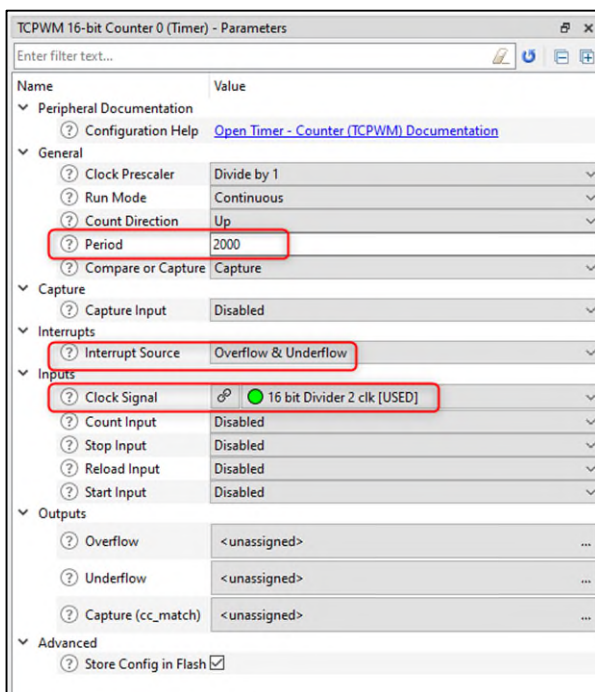
3. Compile and Flash application

- Ensure correct project is selected (the active project within Eclipse)
- Select the appropriate **MyProject Program (KitProg3_MiniProg4)** launch action from the Quick Panel
- Observe results in terminal and the LED toggling with each button press.

Section 4 – Add hardware timer with callback function

1. Generate configuration structure for a TCPWM Timer

- Ensure the correct project is selected, and open **Device Configurator** (under BSP Configurators)
- From the Peripheral tab enable the **TCPWM 16-bit Counter 0** digital peripheral block
 - Locate the TCPWM 16-bit Counter 0 resource and enable with check box (selecting Counter personality)
 - Provide the macro name **Timer**
 - Specify and configure the appropriate clock settings for the Timer
 - Note the new Task that has been added to the Notice List, and the corresponding icon in the Parameters tab
 - Assign a Clock Signal divider to this peripheral (16 bit Divider 2 clk)
 - Use the link icon  to jump to the selected clock divider settings
 - Change the frequency of the clock to **1kHz**
 - Option 1) Divider value of 48000
 - Option 2) Change Divider Calculation to Automatic and Desired Frequency to 1000
 - Return to Timer peripheral setting using the link icon 
 - Set **Period** to 2000, this will result in a timer that reached an overflow event every 2.001 seconds
 - Set **Interrupt Source** to Overflow & Underflow



- Review **Code Preview** panel and observe define macros (based on **Timer_**)
- Generate code by saving the design.modus file (**File -> Save / Ctrl-S**)
- Exit the Device Configurator (**File -> Exit / Alt-F4**)

2. Add code to Initialize, Register callback, Enable, and Start Timer

- Review PDL documentation, exact code snippet is also available within [PSoC 4 Code Snippets.c](#)
- Similar to the UART initialization and enabling, **Cy_TCPWM_Counter_xxxx** functions are used

```
Cy_TCPWM_Counter_Init(Timer_HW, Timer_NUM, &Timer_config);  
Cy_TCPWM_Counter_Enable(Timer_HW, Timer_NUM);
```

- Similar to the GPIO interrupt callback registering, **Cy_SysInt_xxxx** and **NVIC_EnableIRQ** functions are used

```
cy_stc_sysint_t TimerIRQ_cfg = { .intrPriority = 3, .intrSrc = Timer_IRQ };  
Cy_SysInt_Init(&TimerIRQ_cfg, Timer_Callback);  
NVIC_EnableIRQ(Timer_IRQ);
```

- Start the Timer using **Cy_TCPWM_TriggerStart**

```
Cy_TCPWM_TriggerStart(Timer_HW, Timer_MASK);
```

3. Create Timer callback function

```
volatile bool timer_expired = false;  
void Timer_Callback(void)  
{  
    Cy_TCPWM_ClearInterrupt(Timer_HW, Timer_NUM, CY_TCPWM_INT_ON_TC);  
    timer_expired = true;  
}
```

4. Update main FOR loop, to clear **timer_expired** flag and toggle LED (copy code from Button Callback)

```
for(;;)  
{  
    if (timer_expired) {  
        timer_expired = false;  
        Cy_GPIO_Inv(CYBSP_USER_LED1_PORT, CYBSP_USER_LED1_PIN);  
    }  
}
```

5. Compile and Flash application

- Ensure correct project is selected (the active project within Eclipse)
- Select the appropriate **<Project Name> Program (KitProg3_MiniProg4)** launch action from the Quick Panel
- Observe results in terminal and the LED blinky every 2 seconds

Section 5 – Reviewing CAPSENSE (CSD / CSX) configuration

1. Review CAPSENSE device configuration

- Ensure the correct project is selected, and open **Device Configurator**
- From Peripherals tab ensure **System -> CSD (CapSense)** is enabled and selected
- Review Code Preview panel
 - Most defines here are for use with the CSD Low-Level Drivers, this lab will be using the recommended CAPSENSE Middleware
 - Note the **CYBSP_CSD_HW**
- Review the Parameters panel
 - Launch **CapSense Configurator** from the Parameters panel

2. Review CAPSENSE Configurator

- **Basic Tab** with 4 defined widgets supported on the CY8CKIT-149
- **Advanced tab** and sub-tabs for General, CSD and/or CSX settings, and **Widget Details**
 - **Widget Details** tab with hardware and threshold parameters associated to each widget / signal
- **Pins Tab**, replicating settings found in the Device Configurator

3. Re-generate code by saving and exiting out of the CAPSENSE Configurator and Device Configurator

Section 6 – Initializing and Configuring CAPSENSE based on generated contexts

1. Modify main.c to include required (and recommended) header file, code can be copied from [PSoC 4 Code Snippets.c](#)

```
#include "cycfg_capsense.h"
```

2. **Build** the project, to pick up the included header files

3. Initialize CAPSENSE, referencing same code from API documentation

- Open CAPSENSE Middleware Documentation
- Navigate to: **API Reference -> High-level functions -> Cy_CapSense_Init**
- Review **Functional Usage** section, copying code sections into main.c file
 - Block of Initialization Code (*Place with main function*)
 - ISR Config structure for PSoC 4 Fourth Gen (*Place above init code*)
 - CapSense interrupt handler (*Place above main function*)
- Update interrupt handler to pass correct CapSense_HW define
 - **CYBSP_CSD_HW** (*from Device Configurator*)

Section 7 – Scanning and processing touch events with CAPSENSE middleware

1. Review available code examples, referenced in CAPSENSE middleware Quick Start Guide

- Code used in this workshop was derived from PSoC 6 Code Example

2. Within the FOR loop add code that will:

- Check if CAPSENSE is not currently busy (ongoing scanning)
- If not busy, will process any scanned data and return a status
- If successful, will then call a function to process and detected touch events
- Begin the next scan to complete the loop

The following code can be copied from the PSoC_4_Code_Snippets.c file and pasted within the main FOR loop:

```
/* Scan and Process CAPSENSE widgets */
if (CY_CAPSENSE_NOT_BUSY == Cy_CapSense_IsBusy(&cy_capsense_context)) {
    if (CY_CAPSENSE_STATUS_SUCCESS == Cy_CapSense_ProcessAllWidgets(&cy_capsense_context) ) {
        process_touch();
    }
    Cy_CapSense_ScanAllWidgets(&cy_capsense_context);
}
```

3. Each CAPSENSE touch event will correspond to controlling an LED. The GPIOs connected to the LEDs need to be enabled within the **Device Configurator**.

- Using Device Configurator configure the following Pins and **Strong Drive, Input buffer on**
 - P5[2] – CYBSP_LED_BTN0
 - P5[5] – CYBSP_LED_BTN1
 - P5[7] – CYBSP_LED_BTN2

4. Add process_touch code to respond to CAPSENSE touch events

- Copy reference code from PSoC_4_Code_Snippets.c file (process_touch function)
 - This function includes code only for Button 0
 - Carefully replicate code and edit to support Button 1 & 2
 - button0 -> button1 / button2
 - BTN0 -> BTN1 / BTN2
- Create function prototype above main function
void process_touch(void);

5. Compile and Flash application

- Ensure correct project is selected (the active project within Eclipse)
- Select the appropriate **<Project Name> Program (KitProg3_MiniProg4)** launch action from the Quick Panel
- Observe results in terminal and the LED blinky every 2 seconds, while CAPSENSE touch pads toggle corresponding LEDs

Section 8 – Add game logic and random LED selection

1. Remove toggling of USER_LED

- Remove `Cy_GPIO_Inv(...)` from `Button_Callback` function
- Remove `Cy_GPIO_Inv(...)` from main FOR loop (after timer expire check)

2. Define “Game Over” conditions and add print outs and enters Sleep mode

- After `timer_expires`, add check if any **CYBSP_LED_BTNx** is currently ON using **`Cy_GPIO_Read(...)`**
 - Code can be copied from PSoC_4_Code_Snippets.c file
 - If true, print out “Too Slow” using **`Cy_SCB_UART_PutString(...)`** and enter sleep using **`Cy_SysPm_CpuEnterSleep();`**

- Replace turning ON each LED_BTNx with function that prints out “Wrong Button” and enters sleep. Example:

```
if (Cy_GPIO_Read(CYBSP_LED_BTN2_PORT, CYBSP_LED_BTN2_NUM) == CYBSP_LED_STATE_ON) {  
    Cy_GPIO_Write(CYBSP_LED_BTN2_PORT, CYBSP_LED_BTN2_NUM, CYBSP_LED_STATE_OFF);  
} else {  
    Cy_SCB_UART_PutString(CYBSP_UART_HW, "Wrong Button\r\n");  
    Cy_SysPm_CpuEnterSleep();  
}
```

3. Turn ON random LED_BTNx using True Random Number Generator Crypto IP

- Add header file for Crypto APIs (top of `main.c` file)

```
#include "cy_crypto.h"
```

- Create variable to store random number within main function

```
uint32_t rndNum = 0;
```

- Initialize Crypto IP block (refer to PDL documentation for more details)

```
Cy_Crypto_Enable(CRYPTO);
```

- If timer expires AND game is not over (due to LEDs still being on), generate random number. This is the ELSE condition after checking any LED is still ON

```
Cy_Crypto_Trng(CRYPTO, 32UL, &rndNum);
```

- Add conditional statement to turn on LED_BTNx based on random number. Code can be copied from the PSoC_4_Code_Snippets.c file.

```
if (rndNum%3 == 0) {  
    Cy_GPIO_Write(CYBSP_LED_BTN0_PORT, CYBSP_LED_BTN0_NUM, CYBSP_LED_STATE_ON);  
} else if (rndNum%3 == 1) {  
    Cy_GPIO_Write(CYBSP_LED_BTN1_PORT, CYBSP_LED_BTN1_NUM, CYBSP_LED_STATE_ON);  
} else {  
    Cy_GPIO_Write(CYBSP_LED_BTN2_PORT, CYBSP_LED_BTN2_NUM, CYBSP_LED_STATE_ON);  
}
```

Section 9 – Low Power APIs and callback functions

1. Review PDL documentation for System Power Management (SysPm) APIs

- Code used in this workshop was derived from PSoC 4 Power Modes code example and PDL documentation

2. Create declaration of SysPm callback structure (this should be a global variable)

- Copy / Paste block of code from PSoC_4_Code_Snippets.c
 - Function prototype for **Sleep_Callback**
 - Declaration of callback parameters: **callbackParams**
 - Declaration of callback structure: **sleep_cb**

```
cy_stc_syspm_callback_t sleep_cb =
{
    &Sleep_Callback,    /* Callback function */
    CY_SYSPM_SLEEP,    /* Callback type */
    0,                  /* Skip mode */
    &callbackParams,    /* Callback params */
    NULL, NULL,         /* For internal usage */
};
```

3. Register Callback

- Within main function, the callback structure is registered using **Cy_SysPm_RegisterCallback(...)**

4. Add callback function

- Copy / Paste **Sleep_Callback** function from PSoC_4_Code_Snippets.c file. This function can be added to the end of the main.c file.
- Review code within case statement
 - CY_SYSPM_BEFORE_TRANSITION
 - Turn off all LEDs (LED_BTNx and USER_LED)
 - Stop hardware timer
 - CY_SYSPM_AFTER_TRANSITION
 - Turn on USER_LED
 - Restart hardware timer

5. Compile and Flash application

- Ensure correct project is selected (the active project within Eclipse)
- Select the appropriate **<Project Name> Program (KitProg3_MiniProg4)** launch action from the Quick Panel
- Observe results in terminal and on board
 - A random LED will be turned on, this associated CAPSENSE touch pad must be pressed before the timer
 - If a wrong pad is touched or the LED is not cleared in time, the device enters a low-power state
 - Pressing the SW1, causes a system interrupt. This wakes the device and the game resets

Section 10 – Gamification (making is a challenge)

1. Add additional source and header files

- Locate provided **game.c** / **game.h** files.
- Drag-and-Drop files into Eclipse IDE's Project Explorer view (into **MyProject** folder)
- Confirm that files will be “**Copied**” into project

2. Explore included code

- **#define's**
- **game_init**
 - Initialization of GPIO pins, done in place of using Device Configurator
 - Resets the score and timer period
- **game_advance**
 - Increments score when called
 - If a level threshold is reached, the period of the timer is decreased
- **game_over**
 - Prints out the final score
 - Resets the level indicator LEDs
 - Wait to confirm there are no UART or CAPSENSE scanning functions ongoing, this is helpful for a clean power mode transition
- **game_reset**
 - Resets the score and timer period


3. Integrate gamification functions

- Initial game just prior to the FOR loop within main
 - Add **game_init();**
- Advance game score and level difficulty when timer expires with all LEDs cleared
 - Add **game_advance();**
- End game when either wrong button is pressed, or timer expires with LEDs not cleared
 - Add **game_over();** (there will be four instances)
- Reset the game state when exiting Low-Power Sleep mode (part of the Sleep_Callback function)
 - Add **game_reset();**

4. Compile and Flash application

- Ensure correct project is selected (the active project within Eclipse)
- Select the appropriate **<Project Name> Program (KitProg3_MiniProg4)** launch action from the Quick Panel
- Play game! Events and Final score will be printed to serial terminal. Pressing SW1 will restart the game after a game over.

Section 11 – Bonus Content: Breathing LED in Sleep Mode

1. Launch **Device Configurator**
2. In Pin tab, enable **CYBSP_LED5**, this will be our resulting LED output
 - Configure as Strong Drive, Input buffer off
3. In Pin tab, enable **CYBSP_LED6**
 - Default setting (this allows generated code to configure the HSIOM setting required for SMART I/O)
4. In Pin tab, enable **Smart I/O 2** (we are using 2 because it is in the same port as the LED we intend to use)
 - Name BREATHING
5. Launch **Smart I/O Configurator** from Smart I/O 2 Parameter Panel
 - Set I/O 0 to Output
 - Set Chip 0 to Input (Async) / TCPWM 16-bit Counter 2 pwm
 - Set Chip 2 to Input (Async) / TCPWM 16-bit Counter 3 pwm
 - Configure LUT0 to use inputs (Chip 0 / Chip 2 / Chip 2)
 - Set LUT0 to be XNOR (Hex value 0x81) – We are using XNOR due to our LED being Active Low
 - Save and Close
6. Jump to TCPWM 16-bit Counter 2 pwm settings 
7. In Peripheral tab, configure **TCPWM 16-bit Counter 2**
 - Enable with PWM personality and name “**PWM2**”
 - Parameters:
 - Clock Prescaler: **Divide by 4**
 - Configure Clock to use “**16 bit Divider 3 clk**”
 - Use default **Period** and **Compare**
8. In Peripheral tab, configure **TCPWM 16-bit Counter 3**
 - Enable with PWM personality and name “**PWM3**”
 - Parameters:
 - Clock Prescaler: **Divide by 4**
 - Configure Clock to use “**16 bit Divider 3 clk**” (same as PWM2)
 - Set **Period** to **32708** (reducing by 60)
 - Set **Compare** to **16354** (reducing by 30)
9. Save and Close **Device Configurator**
10. Refer to [PSoC 4 Code Snippets.c](#) for code to Initialize and Enable Breathing LED (Smart I/O and PWMs)
11. Refer to [PSoC 4 Code Snippets.c](#) for code to Start and Stop the PWM signals within **Sleep_Callback**
12. Compile and Flash application
 - Went device enters Sleep, a breathing LED will driven on LED5 (with no CPU intervention)