Лабораторна робота № 10

Вивчення технології CapSense мікроконтролерsd PSoC 6

Мета роботи: Вивчити функціонування CapSense мікроконтролерів PSOC 6.

Теоретичні відомості

Ємнісна сенсорна технологія вимірює зміни ємності між пластиною (давачем) та оточуючим середовищем для детектування присутності пальця на сенсорній поверхні або поряд з нею.

Основи CapSense.

Типовий давач CapSense складається з мідної прокладки правильної форми та розміру, вигравіруваної на поверхні друкованої плати. Непровідна накладка використовується як сенсорна поверхня для кнопки, як зображено на рис. 11.1.

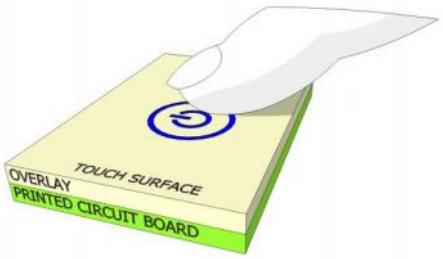

Рис. 11.1. Ємнісний сенсорний давач (давач дотику)

Touch surface – поверхня сенсора.

Overlay – плівка накладена на поверхню давача.

Printed circuit board (PCB) – друкована плата.

Доріжки друкованої плати та переходи з'єднують сенсорні площадки з PSoC GPIO, які сконфігуровані як сенсорні виводи CapSense. Як зображено на рис. 11.2, власна ємність кожного електроду моделюється як CSX, а взаємна ємність між електродами моделюється як CMXY. Електрична схема CapSense, вбудована в PSoC, перетворює ці значення ємності в еквівалентні цифрові значення. Ці цифрові відліки потім обробляються ЦП мікрокронтролера для виявлення дотику.

Для CapSense також потрібно зовнішній конденсатор CMOD для визначення власної ємності, а також конденсатори CINTA та CINTB для вимірювання взаємної ємності. Ці зовнішні конденсатори підключені між окремим виводом GPIO та "землею". Якщо екрануючий електрод використовується для допусків на рідину або для вимірювання великих віддалей давачем наближення, тоді потрібний додатковий конденсатор CTANK.

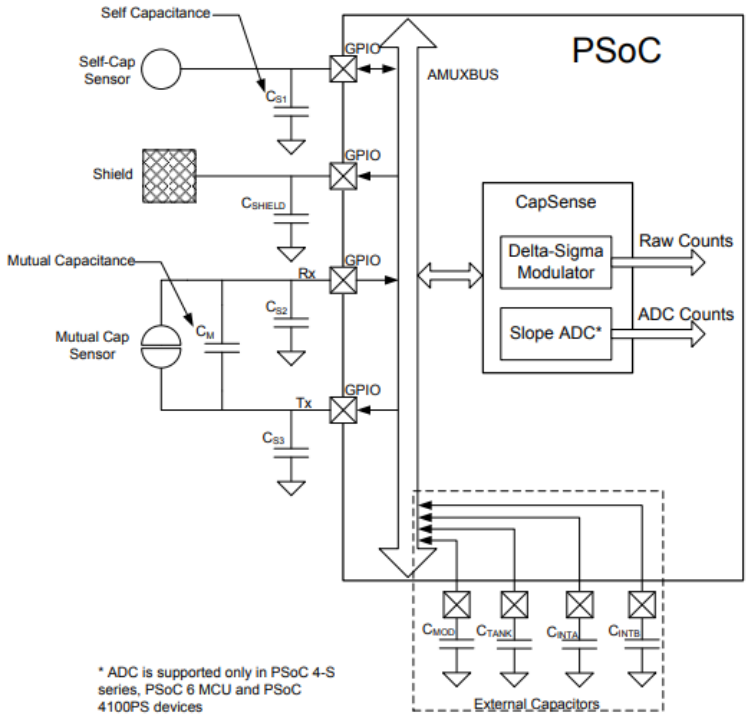

Рис. 11.2. Пристрій PSoC, давачі та зовнішні конденсатори

Ємність давача при відсутності дотику називається паразитною ємністю C_P. Паразитна ємність виникає внаслідок електричного поля між давачем (включаючи сенсорну панель, доріжки і перехідні отвори) та іншими провідниками в системі, такими як заземляючі поверхні, доріжки і будь-який метал в корпусі виробу. GPIO та внутрішні ємності PSoC також вносять вклад в паразитну ємність. Але ці внутрішні ємності зазвичай дуже малі в порівнянні з ємністю давача.

Зондування власної ємності.

На рис. 11.3 зображено, як вивід GPIO з'єднується з сенсорною панеллю з допомогою доріжок та перехідних отворів для вимірювання власної ємності. Як правило, площадка заземлення оточує сенсорну панель, щоб ізолювати її від інших давачів та доріжок. Хоча на рис. 11.3 зображені деякі лінії поля навколо сенсорної панелі, але фактичний розподіл електричного поля є дуже складним.

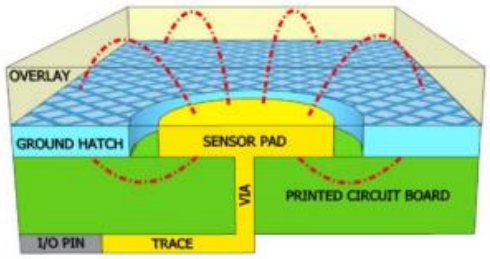

Рис. 11.3. Паразитна ємність

Коли на поверхневій плівці давача присутній палець, провідний організм людини та велика маса тіла людини утворюють заземлену провідну площину, паралельну сенсорній панелі (рис. 11.4).

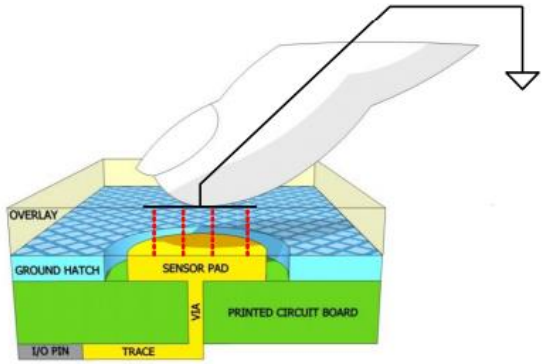

Рис. 11.4. Ємність пальця

Таке розміщення утворює конденсатор з паралельними пластинами. Ємність між сенсорною панеллю та пальцем має вигляд:

$$C_F = \frac{\varepsilon_0 \varepsilon_r A}{d}, \quad (11.1)$$

де: ε_0 - діелектрична проникливість вакууму ($\varepsilon_0 = 8.8541878176 * 10^{-12}$ Ф/м;) ε_r – відносна діелектична проникливість поверхневої плівки давача; A – площа дотику пальця та сенсорної панелі; d – товщина поверхневої плівки.

Ємність C_F – це ємність пальця. Паразитна ємність C_P та ємність C_F пальця з'єднані паралельно одна одній, так як обидві представляють ємність між контактом давача та землею. Тому загальна ємність C_S давача, коли палець присутній на давачі, є сумою ємностей C_P та C_F.

$$C_s = C_p + C_F. \quad (11.2)$$

При відсутності дотику ємність C_S рівна C_P.

PSoC перетворює ємність C_S в еквівалентні цифрові значення, які називаються необробленими значеннями. Так як дотик пальця збільшує загальну ємність контакту давача, збільшення необроблених значень вказує на дотик пальця.

CapSense мікроконтролера PSoC 4 підтримує значення паразитної ємності до 65 пФ для ємності на 0,3 пФ і до 35 пФ для ємності пальця 0,1 пФ.

Зондування взаємної ємності.

На рис 11.5 зображено розміщення кнопок давача для вимірювання взаємної ємності. Вимірювання взаємної ємності зводиться до вимірювання ємності між двома електродами, які називаються передаючим (Tx) та приймаючим (Rx) електродами. В системі визначення взаємної ємності цифровий сигнал напруги, який переключається між VDDIO2 або VDDD3 (якщо VDDIO не підтримується пристроєм) та GND, подається на вивід Tx, і вимірюється величина заряду, отриманого на виводі Rx. Кількість заряду, отриманого на електроді Rx, прямо пропорційна взаємній ємності (CM) між двома електродами.

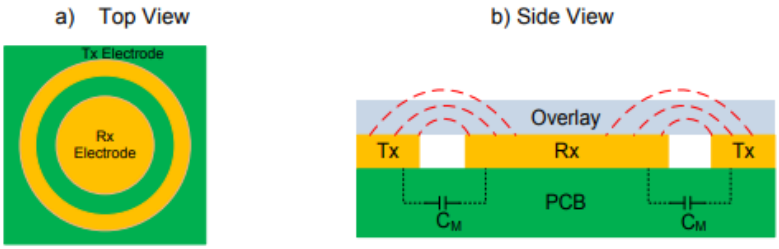

Рис. 11.5. Робота зондування взаємної ємності

Якщо палець поміщається між електродами Tx та Rx, взаємна ємність зменшується до C1M, як зображено на рис. 11.6. Через зменшення взаємної ємності заряд, отриманий на Rx електроді, також зменшується. Система CapSense вимірює величину заряду на Rx електроді для виявлення стану дотику/ відсутності дотику.

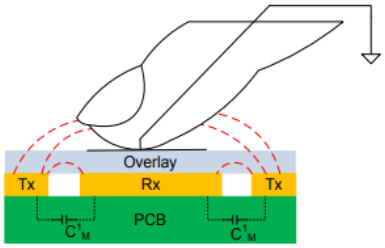

Рис. 11.6. Взаємна ємність з дотиком пальця

Характерні особливості ємнісного зондування мікроконтролерів PSoC 6.

- Забезпечує краще в своєму класі відношення сигнал/шум (SNR).
- Підтримує методи зондування з власною ємністю (CSD) та взаємною ємністю (CSX).
- Має технологію автоматичного налаштування SmartSense для розпізнавання CSD, щоб не використовувати складний процес ручного налаштування.
- Підтримує різні віджети, такі як кнопки, матричні кнопки, повзунки, сенсорні панелі та давачі наближення.
- Забезпечує наднизьке енергоспоживання та стійку до рідини ємнісну сенсорну технологію.
- Містить вбудований графічний тюнер для налаштування, тестування та відладки в реальному часі.
- Забезпечує чудову несприйнятливість до зовнішнього шуму та низького випромінювання.
- Забезпечує кращу в своєму класі стійкість до впливу рідин.
- Містить вбудовану бібліотеку самоперевірки (built-in self-test (BIST)) для реалізації вимог класу B для CapSense.
- Підтимує жести одним та двома пальцями.
- Підтримує одноконтурний АЦП.

CapSense – це ємнісне зондування, запропоноване та реалізоване фірмою Cypress. Ємнісне зондування може використовуватися в багатьох додатках та проектах, де звичайні механічні кнопки можуть бути замінені витонченими інтерфейсами, щоб змінити спосіб взаємодії користувачів з електронними системами. До них відносяться побутова техніка, автомобілі, IoT та промислові додатки. CapSense підтримує декілька інтерфейсів (віджетів з використанням

методів розпізнавання власної ємності CSX та взаємної ємності CSD, забезпечуючи високу продуктивність).

Це рішення для компонента CapSense включає в себе майстер налаштування для створення та налаштування віджетів CapSense, API для управління компонентом з вбудованого програмного забезпечення та додаток CapSense Tuner для налаштування, тестування та відладки. Це забезпечує просте проектування користувацьких інтерфейсів в додатках користувачів.

Технологія CapSense стала популярною для заміни традиційних користувацьких та механічних інтерфейсів. В ній задіяно менше деталей, що економить затрати і підвищує надійність без зношення. Основними перевагами CapSense в порівнянні з іншими рішеннями є висока продуктивність в важких умовах оточуючого середовища та несприйнятливість до широкого спектру джерел шуму.

Технологія CapSense використовується для:

- детектування близькості для інноваційного користувацького досвіду та оптимізації енергоспоживання;
- заміна для детектування близькості на основі IR- випромінювання, чутливого до шкіри та кольору;
- безконтактне вимірювання рівня рідини в різних областях використання;
- безпечні операції з шкідливими матеріалами.

Обмеження:

Цей компонент підтримує всі пристрої з підтримкою CapSense в сімействі пристроїв PSoC 6. Але деякі функції є обмеженими:

- CapSense Tuner не підтримує функції АЦП. Ця функція буде додана в наступній версії компонента.
- Ця версія компонента підтримує детектування жестів для одного віджета за раз.

Тюнер CapSense.

Компонент CapSense надає графічний додаток Tuner для відладки і налаштування системи CapSense. Для роботи з додатком Tuner в нього додається компонент зв’язку та після цього набір регістрів компонента надається додатку Tuner. Додаток Tuner працює з компонентами зв’язку EZI2C та UART.

Для редагування параметрів використовується додаток Tuner і застосовуються нові налаштування до пристрою з допомогою кнопки "To Device". Це можна зробити при використанні Manual або SmartSense (тільки для апаратних параметрів) режимів для тюнінга.

- Для редагування порогових параметрів використовується режим SmartSense (тільки для апаратних параметрів).
- Для редагування всіх параметрів використовується Manual режим.
- Якщо SmartSense (повне автоналаштування) вибрано для режима налаштування CSD, користувач має параметри доступу тільки для читання (крім параметра ємності пальця).

Кнопка " To Device " є доступною, коли на панелі "Graph Setup" включена функція Synchronized і довільний параметр в тюнері може бути змінений. Синхронізоване управління може бути включене, коли потік FW регулярно викликає функцію CapSense_RunTuner(). Якщо ця функція відсутня в коді додатку, то режим синхронізованого зв’язку відключений.

Реалізація додатку з використанням технології CapSense.

Розглянемо приклад додатку, в якому реалізований проект з користувацьким інтерфейсом, використовуючи дисплей E-INK з повзунком CapSense та сенсорними кнопками.

1. Підключимо плату стенду до комп’ютера з допомогою кабелю USB через розняття USB (J10).
2. Відкриємо проект CE218133_EINK_CapSense в середовищі PSoC 4.2 PSoC Creator. Виконаємо його компіляцію та програмування прошивки в мікроконтролер PSoC 6 стенду.

Дисплей E-INK оновлюється та показує екран запуску на протязі 3 сек, а потім на ньому відображається меню, в якому перерахована важлива інформація про набір та програмне забезпечення (рис. 11.7). Червоний світлодіод LED9 включається, якщо дисплей E-INK не виявлено. У цьому випадку потрібно перевірити з'єднання між екраном E-INK Display Shield та платою стенду, а потім виконати скидання програмного забезпечення (натиснути кнопку Res).

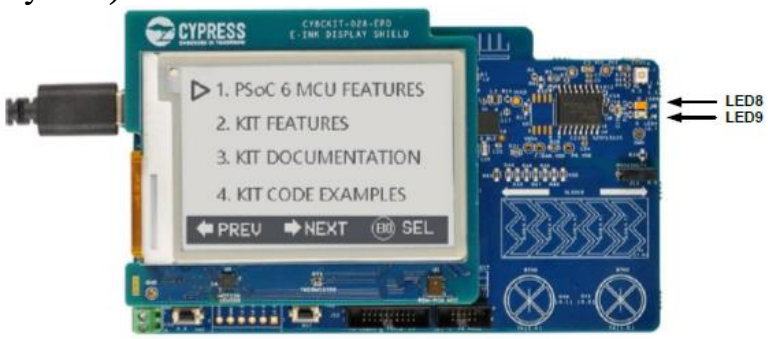

Рис. 11.7. Вигляд меню проекту на екрані дисплею E-INK

3. Для навігації по меню, як показано на рис. 11.8, використовуватимемо сенсорний повзунок та кнопки CapSense.

Для дисплею E-INK потрібно близько секунди, щоб оновити інформацію на ньому після сенсорного вводу. Помаранчевий світлодіод LED8 вмикається, коли дисплей зайнятий. Сенсорні входи не обробляються, коли дисплей зайнятий.

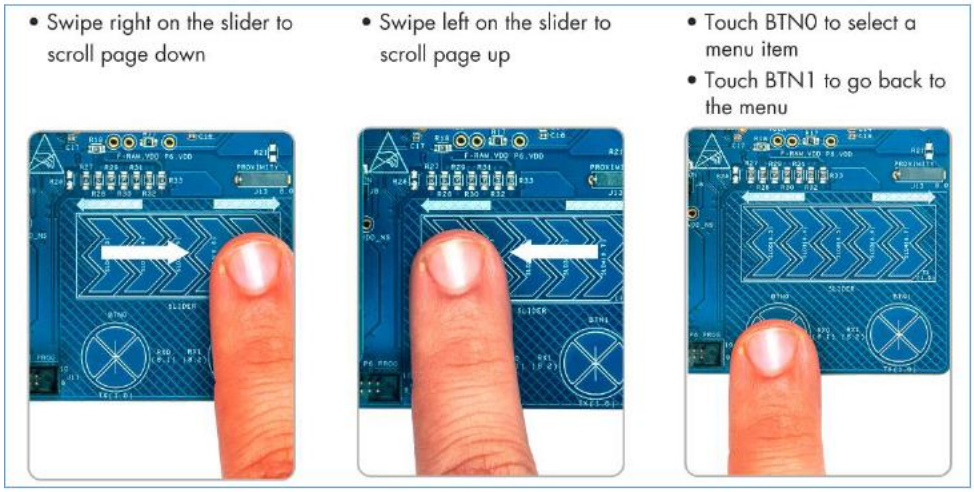

Рис. 11.8. Параметри навігації по меню

Реалізація проекту.

E-INK (електронне чорнило) - це технологія відображення на папері, що характеризується високою контрастністю, широкими кутами огляду та мінімальною потужністю споживання в режимі очікування. На відміну від звичайних підсвічуваних плоских панелей, що випромінюють світло, E-INK відображає світло, як папір. Це робить дисплей E-INK зручнішим для відображення інформації на ньому та для читання, забезпечує ширший кут огляду, ніж більшість світлодіодних дисплеїв. Тому читати інформацію з E-INK дисплеїв зручно навіть при сонячному світлі.

У цьому проекті використовується екран дисплея E-INK CY8CKIT-028-EPD разом із платою стенду. Екран E-INK має 2,7-дюймовий дисплей E-INK з роздільною здатністю 264 × 176 пікселів.

Для цього проекту на рис. 11.9, рис. 11.10, рис. 11.11 зображені схеми в PSoC Creator 4.2.

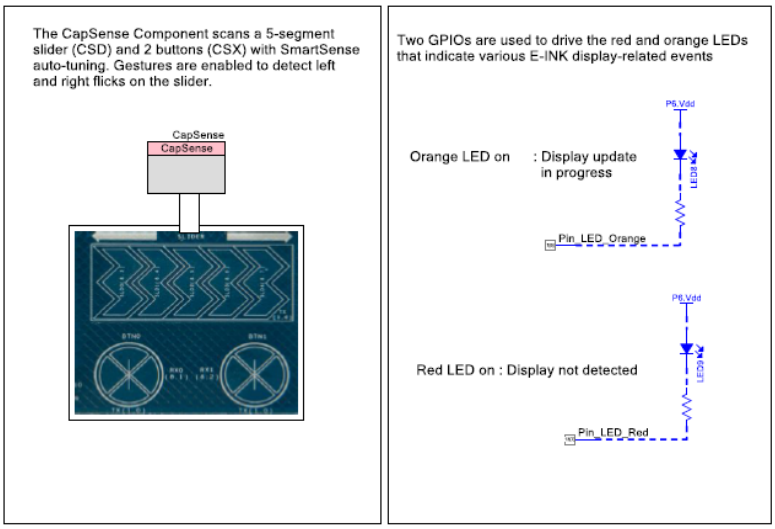

Рис. 11.9. Схема компонента CapSense та світлодіодів LED8, LED9

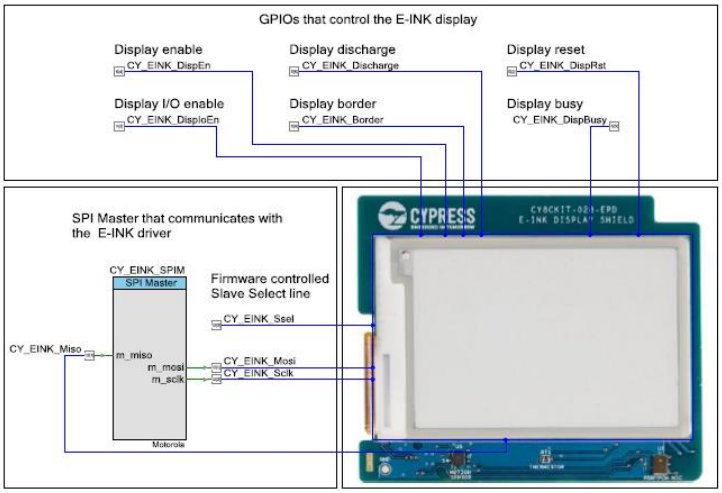

Рис. 11.10. Схема підключення E-INK дисплею в проекті

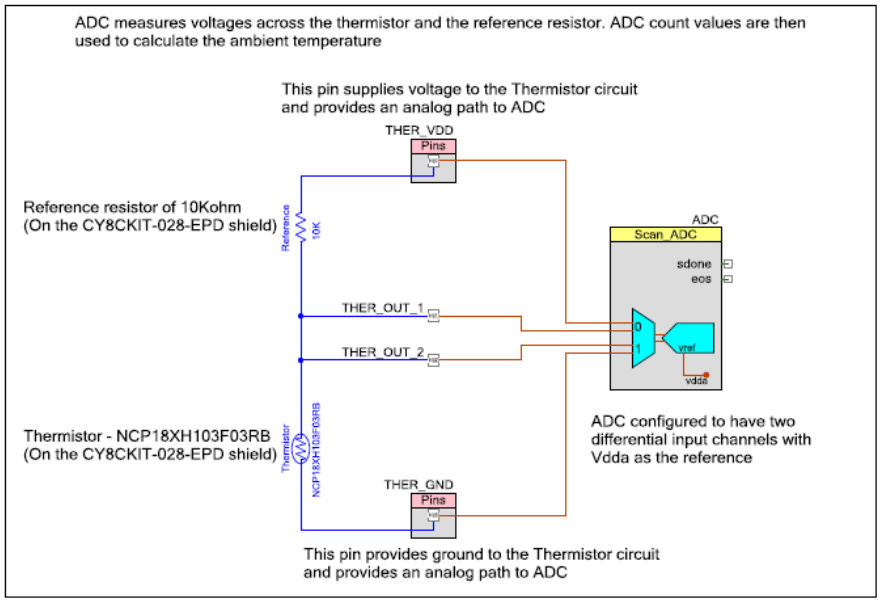

Рис. 11.11. Схема температурної компенсації для екрану E-INK

Основна вкладка компонента CapSense зображена на рис. 11.12.

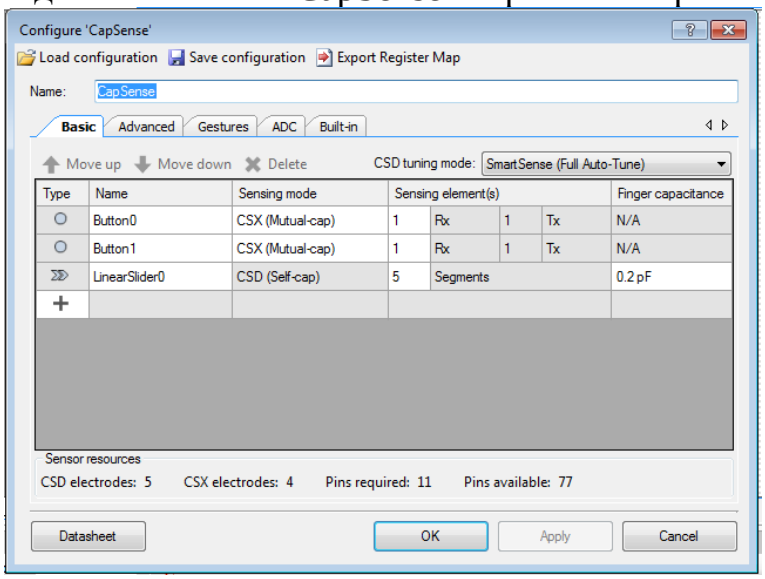

Рис. 11.12. Вигляд основної вкладки компонента CapSense

2. Реалізація проекту, який демонструє роботу PSoC 6 з RTOS з використанням CapSense та UART для управління інтенсивністю червоного світлодіода з допомогою широтно-імпульсного модулятора PWM.

Компонента CapSense:

- встановлює інтенсивність світлодіода на 0% з допомогою кнопки BTN0;
- встановлює інтенсивність світлодіода на 100% з допомогою кнопки BTN1;
- встановлює інтенсивність в межах (0 – 100 %) з допомогою повзунка CapSense.

Компонента UART:

- встановлює інтенсивність світлодіода на 0% при натисканні на клавішу ′0′ клавіатури;
- встановлює інтенсивність світлодіода на 100% при натисканні на клавішу ′1′ клавіатури;
- встановлює інтенсивність світлодіода на 50% при натисканні на клавішу ′5′ клавіатури;
- виводить довідкову інформацію при натисканні на клавішу ′?′ клавіатури.

Інтенсивністю засвічування світлодіода можна керувати з допомогою широтно-імпульсного модулятора (PWM) та високочастотних вхідних тактових імпульсів, які подаються на вхід PWM. Якщо шпаруватість PWM є "малою", то інтенсивність засвічування світлодіода є низькою. Якщо ж шпаруватість є "великою", то інтенсивність буде "високою". При шпаруватості PWM 0% світлодіод не світитиме, при шпаруватості PWM 50% яскравість засвічування буде середньою, а при шпаруватості PWM 100% яскравість засвічування буде максимальною. Встановимо тактову частоту вхідних імпульсів на 1 МГц, а період PWM на 100. Значення порівняння в PWM встановимо в діапазоні від 0 до 100, що відповідатиме яскравості засвічування в відсотках.

Компонент CapSense має два режими вимірювання: взаємно ємність та власна ємність. Стенд CY8CKIT-062-BLE має дві сенсорні кнопки та 5-ти сегментний сенсорний повзунок.

При реалізації цього проекту потрібно вирішити наступні завдання:

- UART – читання даних з клавіатури та передача повідомлень в компонент PWM;
- CapSense – зчитування положення пальців з повзунка та кнопок і передача повідомлень в компонент PWM;
- PWM – буде мати вхідну чергу. При отриманні цілочисельного повідомлення буде встановлено значення порівняння PWM з отриманим цілим числом (0> = msg <= 100).

Створимо новий проект, реалізувавши в ньому схему, зображену на рис. 11.13.

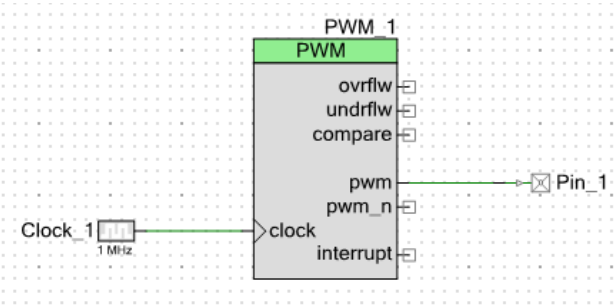

Рис. 11.13. Схема проекту з компонентами PWM та UART

Вкладка конфігурування виводу Red зображена на рис. 11.14.

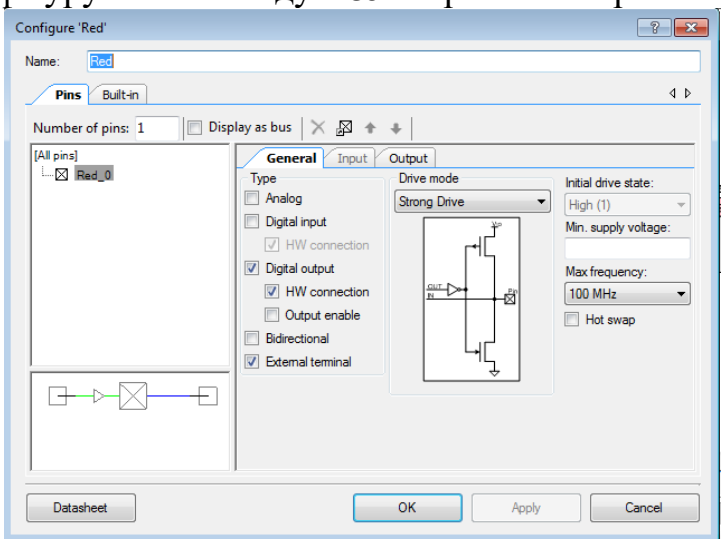

Рис. 11.14. Вкладка конфігурування виводу Red

Налаштуємо період (Period 0) TCPWM на 100 і параметр порівняння (Compare 0) на 50 (рис. 11.15). Це забезпечить блимання світлодіода з частотою 10 кГц.

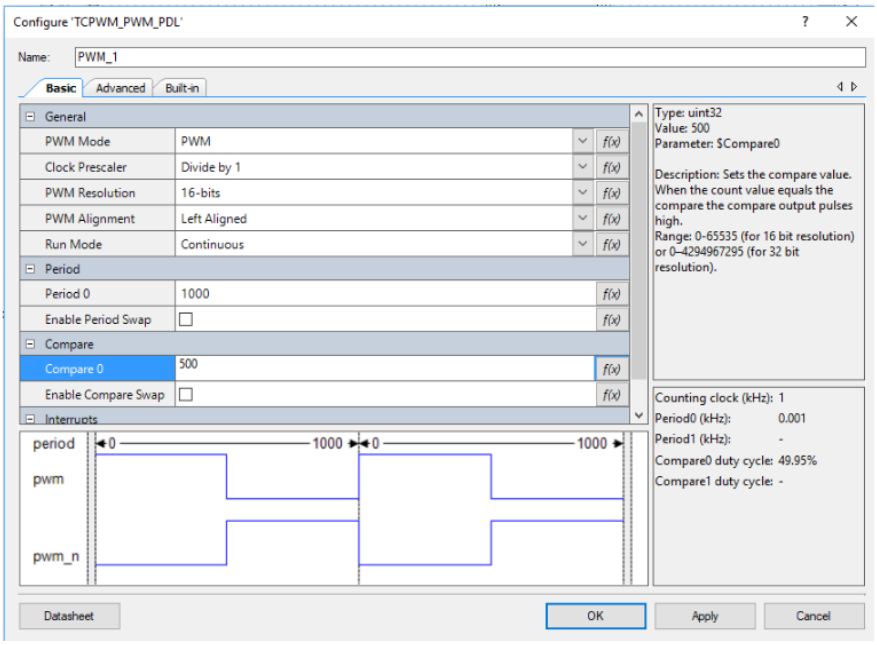

Рис. 11.15. Налаштування параметрів компонента PWM_1

Під’єднаємо вивід Red до порту P0.3.

Виконаємо редагування головного файлу *.c. В цьому мікроконтролері їх є два. Один має назву "main_cmop.c" для Cortex-M0+, а другий - "main_cm4.c"

для Cortex-M4. Для роботи компонента TCPWM потрібно його включити з допомогою функції PWM_1_Start (). Зробимо це в програмі "main_cm4.c".

```
// Lab_Work_11.
#include "project.h"

int main(void)
{
    __enable_irq(); /* Enable global interrupts. */

   PWM_1_Start();

    for(;;)
    {
        /* Place your application code here. */
    }
}
```

Змінимо налаштування проекту, додавши друк для відладки та FreeRTOS. PSoC Creator має ці дві функції. Функція printf названа як "Retarget I/O". Тому потрібно налаштувати ввід / вивід printf на периферію в PSoC. Найкращий периферійний пристрій для printf – це UART, підключений до комп'ютера через kitprog2.

Для того, щоб додати printf та FreeRTOS в проект, потрібно клацнути правою кнопкою мишки на Project та вибрати "Build Settings…" (рис. 11.16).

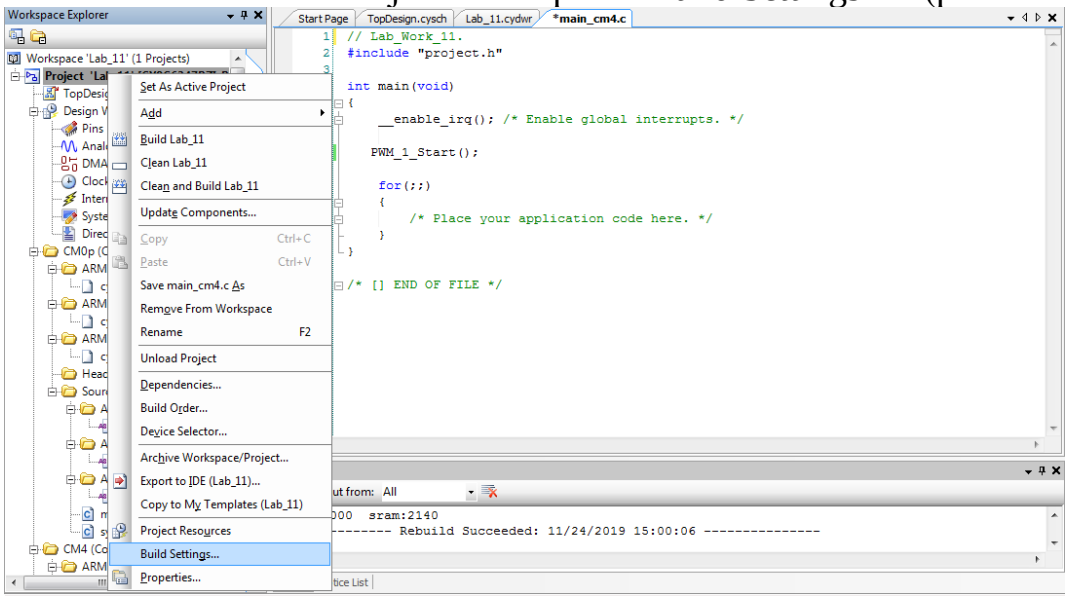

Рис. 11.16. Вигляд вікна вибору "Build Settings…"

При цьому відкриється вікно, зображене на рис. 11.17.

Клацнемо на "Peripheral Driver Library" (бібліотека периферійних драйверів) та виберемо "FreeRTOS" та "Memory Management heap_4". При створенні проекту PSoC Creator додасть їх обидва в наш проект.

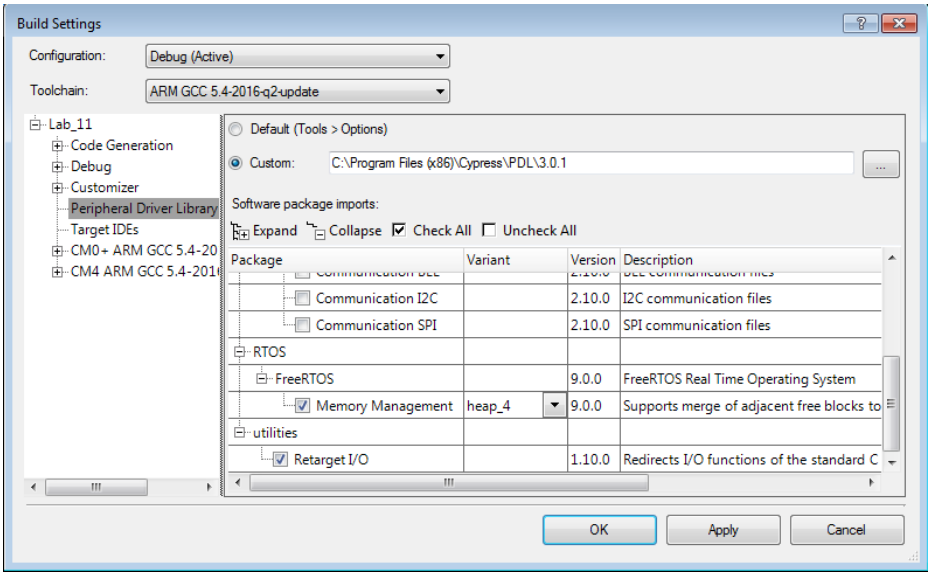

Рис. 11.17. Вигляд вікна налаштування FreeRTOS

Виберемо "Build->Generate Application" та звернемо увагу, що PSoC Creator додав декілька файлів в проект, включаючи FreeRTOS, stdio_user.c / .h та retarget_i (рис. 11.18).

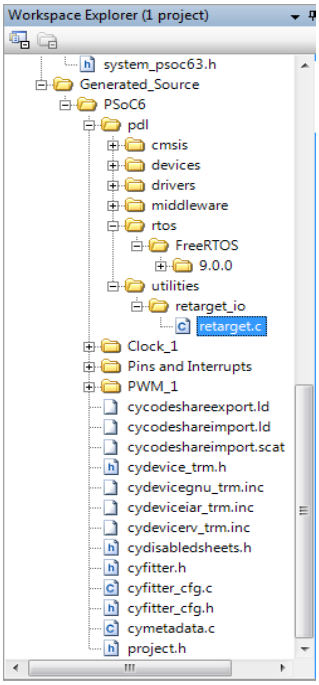

Рис. 11.18. Вигляд вікна Workspace Explorer після виконання Generate Application

Додамо компонент UART до схеми та назначимо для нього виводи. Для того, щоб використати функції printf, getchar та putchar, потрібно під єднати стандартні бібліотеки stdin і stdout до UART, підключеного до KitProg2, який з’єднаний з комп’ютером. Відкривши програму терміналу (Putty), можна прочитати та написати в PSoC.

Під’єднаємо виводи UART до виводів мікроконтролера PSoC 6 (рис. 11.19).

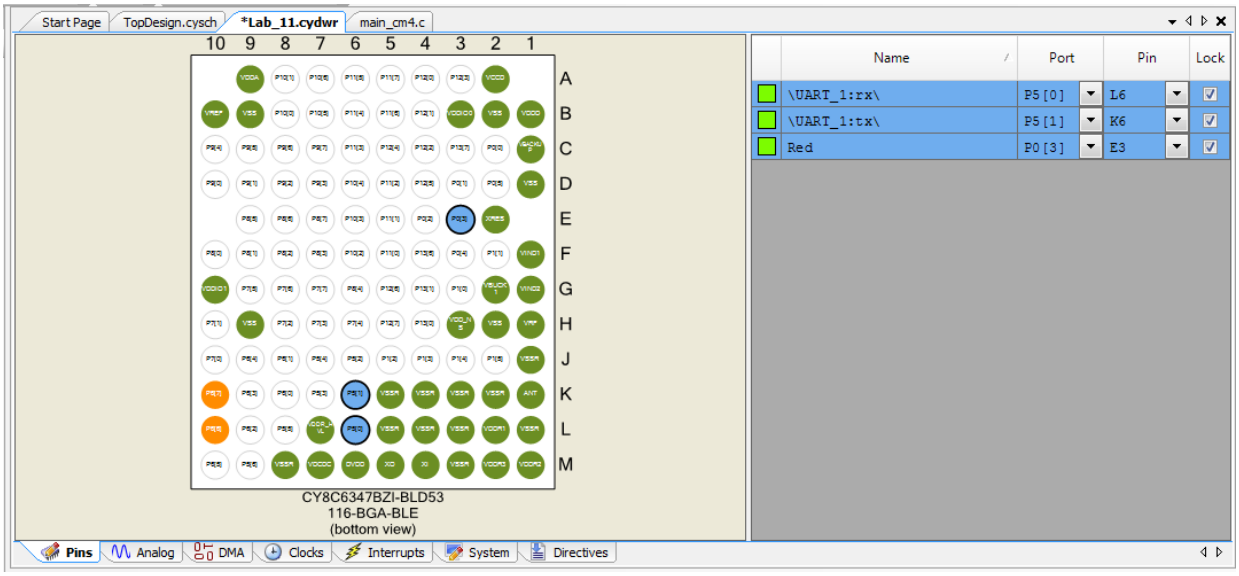

Рис. 11.19. Вигляд вікна з виводами під'єднання компонента UART

Відкриємо файл "stdio_user.h" та добавимо в нього «include <project.h>» (рядок 149) та змінимо IO_STDOUT_UART і IO_STDIN_UART на UART_1_HW. Ім'я «UART_1» в рядках 155-156 (рис. 11.20).

```
148 */
149 #include <project.h>
150 #include "cy_device_headers.h"
151
152 /* Must remain uncommented to use this utility */
153 #define IO_STDOUT_ENABLE
154 #define IO_STDIN_ENABLE
155 #define IO_STDOUT_UART      UART_1_HW
156 #define IO_STDIN_UART       UART_1_HW
157
158 #if defined(IO_STDOUT_ENABLE) || defined(IO_STDIN_ENABLE)
159 #if defined(IO_STDOUT_UART) || defined(IO_STDIN_UART)
160 #include "scb/cy_scb_uart.h"
161 #endif /* IO_STDOUT_UART || IO_STDIN_UART */
162 #endif /* IO_STDOUT_ENABLE || IO_STDIN_ENABLE */
163
164 /* Controls whether CR is added for LF */
165 #ifndef STDOUT_CR_LF
166 #define STDOUT_CR_LF    0
167 #endif /* STDOUT_CR_LF */
168
169 #if defined(__cplusplus)
170 extern "C" {
171 #endif
172
```

Рис. 11.20. Фрагмент коду файлу "stdio_user.h"

Відкриємо файл FreeRTOSConfig.h (рис. 11.21) та задокументуємо попередження в рядку 83 перед тим як змінити розмір пам'яті в рядку 109:

```
#define configTOTAL_HEAP_SIZE                    (48*1024).
```

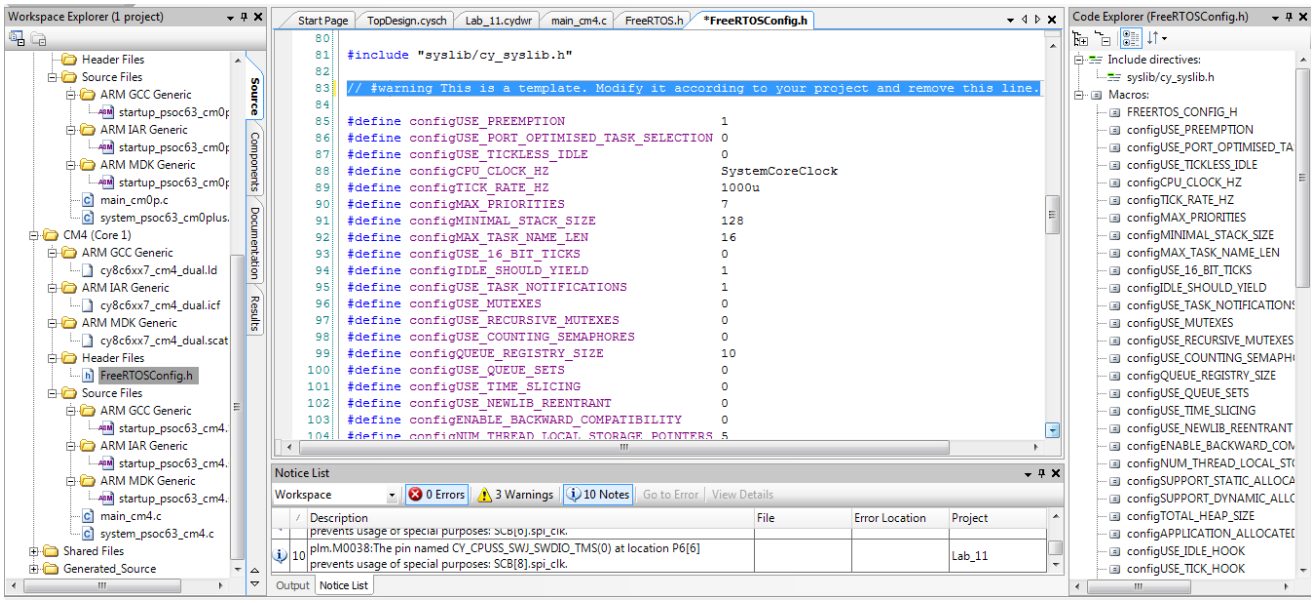

Рис. 11.21. Зміни в файлі FreeRTOSConfig.h

Додамо фрагмент програми до нашого проекту. Це функція UartTask, яку використовує FreeRTOS для контролю Uart.

```
// Lab_Work_11.
#include "project.h"
#include <stdio.h>
#include "FreeRTOS.h"
#include "task.h"

void uartTask(void *arg)
{
    (void)arg;
    char c;
    setvbuf(stdin,0,_IONBF,0);
    while(1)
    {
        c = getchar();
        switch(c)
        {
            case 's': // Stop the PWM
                printf("Stopped PWM\r\n");
                Cy_TCPWM_PWM_Disable(PWM_1_HW,PWM_1_CNT_NUM);

            break;
            case 'S': // Start the PWM
                printf("Started PWM\r\n");
                Cy_TCPWM_PWM_Enable(PWM_1_HW,PWM_1_CNT_NUM);
                Cy_TCPWM_TriggerStart(PWM_1_HW, PWM_1_CNT_MASK);
            break;
        }
    }
}

int main(void)
{
    __enable_irq(); /* Enable global interrupts. */

   PWM_1_Start();
   UART_1_Start();

    printf("\033[2J\033[H"); // Clear Screen
    printf("  --- Work Lab_11. ---\r\n");
```

```
    printf("  --- Test UART.   ---\r\n");

    // Mask a FreeRTOS Task called uartTask
    xTaskCreate(uartTask,"UART Task",configMINIMAL_STACK_SIZE,0,3,0);
    vTaskStartScheduler();
    for(;;)
    {
        /* Place your application code here. */
    }
}
```

Виконаємо налаштування компоненти CapSense. Додамо дві кнопки в дизайн та виконаємо конфігурацію "CSX (Mutual-cap)" (рис. 11. 22).

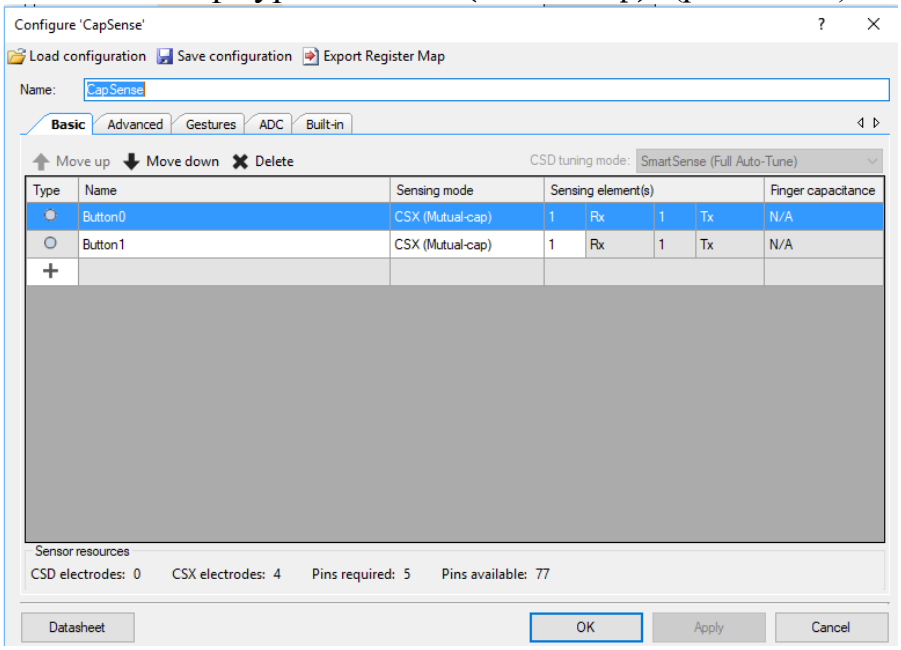

Рис. 11.22. Вигляд головної вкладки компоненти CapSense

На стенді CY8CKIT-062-BLE є загальна лінія передачі. Це дозволяє зекономити 1 вивід. Для цього потрібно повідомити компоненту CapSense, що використовується спільний Tx. Для цього потрібно натиснути Advanced -> Widget Details. Потім вибираємо "Button1_tx" і встановлюємо Sensor Connection / Ganging до "Button0_Tx" (рис. 11.23).

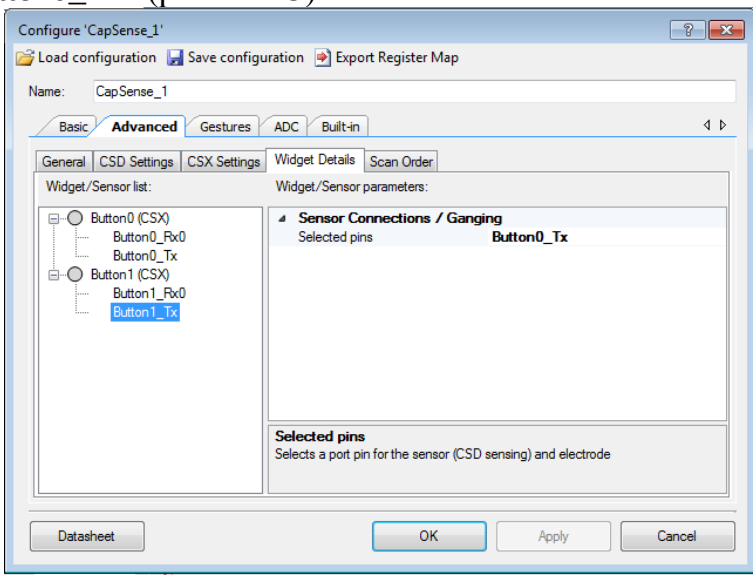

Рис. 11. 23. Налаштування компонента CapSense

Встановимо виводи для компонента CapSense (рис. 11.24).

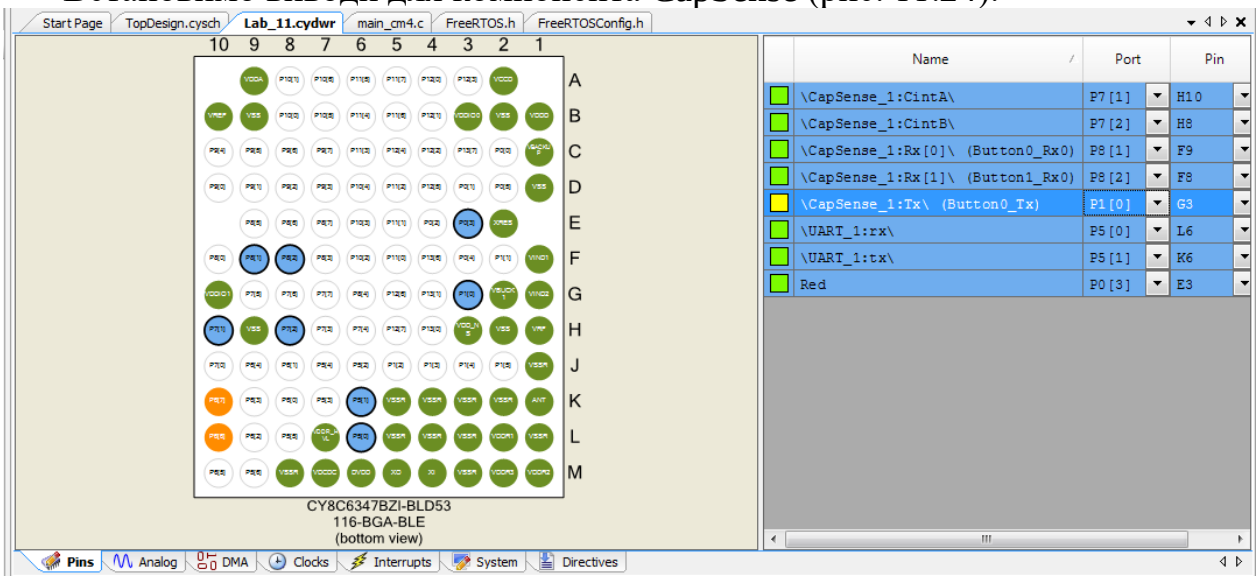

Рис. 11.24. Вигляд вікна з виводами під’єднання компонентів проекту

Запустимо "Generate Application", який дозволить PSoC Creator розмістити та маршрутизувати проект, а також ввести всі необхідні драйвери.

Створимо нову функцію, яка матиме назву capsenseTask. Використаємо 4-ри змінних для зберігання попереднього та поточного станів кнопок.

Додамо сенсорний повзунок CapSense на схему і встановимо виводи. Для того, щоб слайдер працював, потрібно додати відмет LinearSlider, і натиснувши на ′+′ та вибрати LinearSlider (рис. 11.25).

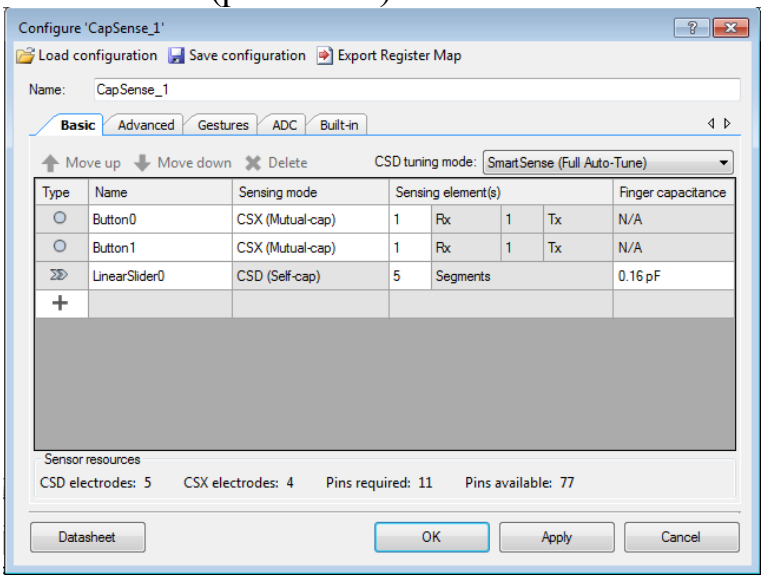

Рис. 11.25. Вигляд вікна для налаштування лінійного слайду

Встановимо виводи для лінійного слайдера компонента CapSense (рис. 11.26).

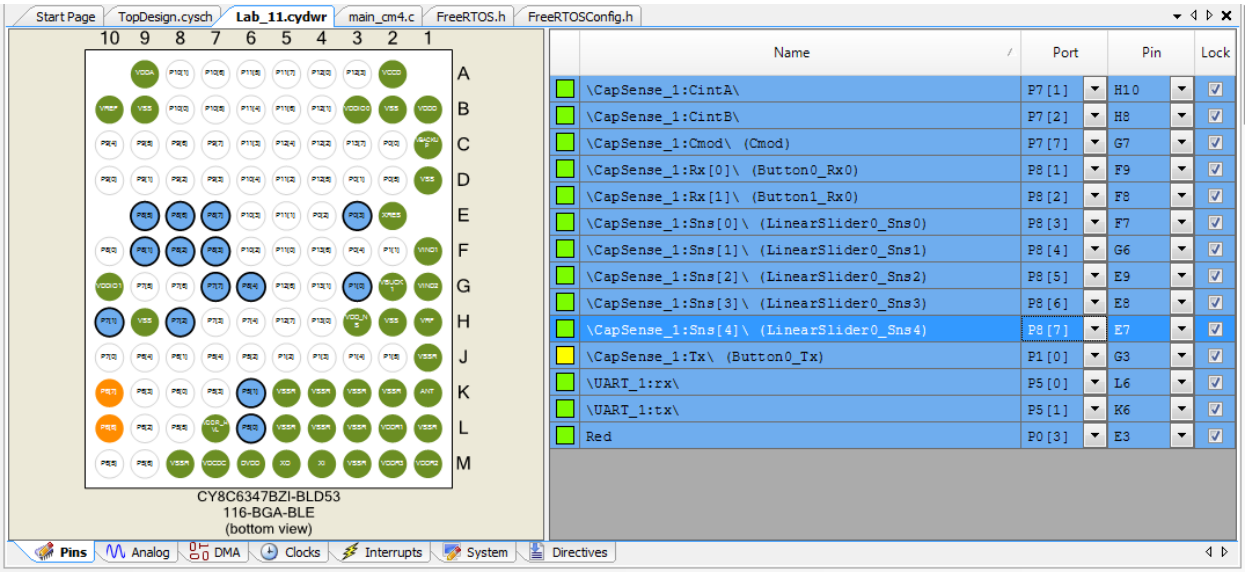

Рис. 11.26. Вигляд вікна з виводами під'єднання компонентів проекту

```
// Lab_Work_11.

#include "project.h"
#include <stdio.h>
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"

QueueHandle_t pwmQueueHandle; // used to get meesages to set PWM % (compare)

// This task controls the PWM
// Receives uint32 % from pwmQueue
void pwmTask(void *arg)
{
    (void)arg;
    uint32_t msg;

    printf("Starting PWM Task\r\n");
    pwmQueueHandle = xQueueCreate(1,sizeof(uint32_t));

    while(1)
    {
        xQueueReceive(pwmQueueHandle,&msg,portMAX_DELAY);
        Cy_TCPWM_PWM_SetCompare0(PWM_1_HW,PWM_1_CNT_NUM,msg);
   }
}

// uartTask - the function which handles input from the UART
void uartTask(void *arg)
{
    (void)arg;

    char c;
    uint32_t msg;

    printf("Started UART Task\r\n");
    setvbuf(stdin,0,_IONBF,0);
      while(1)
      {
            if(Cy_SCB_UART_GetNumInRxFifo(UART_1_HW))
            {
                c = getchar();
            switch(c)
```

```
            {
                  case '0': // send the stop message
                        msg = 0;
                        xQueueSend(pwmQueueHandle,&msg,portMAX_DELAY);
                  break;
                  case '1': // Send the start message
                        msg = 100;
                        xQueueSend(pwmQueueHandle,&msg,portMAX_DELAY);
                  break;
                      case '5':
                            msg = 50;
                            xQueueSend(pwmQueueHandle,&msg,portMAX_DELAY);
                      break;
                      case '?': // Print Help
                            printf("s - stop PWM\r\nS - start PWM\r\n");
                      break;
                }
            }
            taskYIELD();
    }
}

// capsenseTask
// Read buttons and slider using CapSense and send messages to pwmQueue
void capsenseTask(void *arg)
{
    (void)arg;

    uint32_t msg;

    int b0prev=0;
    int b1prev=0;
    int b0current=0;
    int b1current=0;
    int sliderPos;

    printf("Starting CapSense Task\r\n");

    CapSense_Start();
    CapSense_ScanAllWidgets();

    for(;;)
    {
        if(!CapSense_IsBusy())
        {
            CapSense_ProcessAllWidgets();
            sliderPos=CapSense_GetCentroidPos(CapSense_LINEARSLIDER0_WDGT_ID);
            if(sliderPos<0xFFFF) // If they are touching the slider then send
the %
            {
                msg = sliderPos;
                xQueueSend(pwmQueueHandle,&msg,portMAX_DELAY);
            }
            b0current = CapSense_IsWidgetActive(CapSense_BUTTON0_WDGT_ID);
            b1current = CapSense_IsWidgetActive(CapSense_BUTTON1_WDGT_ID);

            if(b0current && b0prev == 0) // If they pressed btn0
            {
                msg = 0;
                xQueueSend(pwmQueueHandle,&msg,portMAX_DELAY);
            }
            if( b1current && b1prev == 0) // If they pressed btn0
            {
               msg = 100;
```

```
                xQueueSend(pwmQueueHandle,&msg,portMAX_DELAY);
            }
            b0prev = b0current;
            b1prev = b1current;

            CapSense_UpdateAllBaselines();
            CapSense_ScanAllWidgets();
        }
        else
            taskYIELD();

    }
}
int main(void)
{
    __enable_irq(); /* Enable global interrupts. */

    /* Place your initialization/startup code here */

    PWM_1_Start();
    UART_1_Start();

    printf("\033[2J\033[H"); // VT100 Clear Screen
    printf("PWM Capsense & UART Project\r\n");

    // Mask a FreeRTOS Task called uartTask
    xTaskCreate(pwmTask,"PWM Task",configMINIMAL_STACK_SIZE*8,0,3,0);
    xTaskCreate(uartTask,"UART Task",configMINIMAL_STACK_SIZE*8,0,3,0);
    xTaskCreate( capsenseTask, "CapSense Task",2048*2,0,3,0);

    vTaskStartScheduler();  // Will never return

    for(;;) // It will never get here
    {
    }
}
```