BIRZEIT UNIVERSITY

Department of Electrical & Computer Engineering
ENCS4110 – Computer Design Laboratory

Laboratory Manual

TM4C123 LaunchPad (ARM Cortex-M4)

Version: v1.0.0

Date: May 2026

Contents

1 Assembly Basics and Program Structure

Learning Objectives

After completing this experiment, you will be able to:

- Identify the main components of the ARM Cortex-M4 architecture, including general-purpose registers, stack pointer, link register, program counter, and program status register (xPSR).
- Write, assemble, and debug a minimal ARM assembly program containing a vector table and `Reset_Handler`.
- Use core data-processing, shift/rotate, and compare/test instructions to manipulate register data.
- Apply conditional execution and branching using condition codes (`EQ`, `NE`, `GT`, `LT`, etc.).
- Debug assembly programs in **Keil uVision5** using breakpoints, single-stepping, and register/memory inspection to analyze instruction effects.

Experiment Overview

This experiment introduces ARM Cortex-M4 assembly programming fundamentals, including register architecture, status flags, and memory organization. You will build minimal startup code with a vector table and Reset_Handler, practice data-processing and control-flow instructions, and use the Keil µVision5 debugger to single-step through code while inspecting registers and memory. By the end of this lab, you will understand the core building blocks of ARM assembly programming and be able to write, assemble, and debug low-level embedded code.

1.1 Theoretical Background

1.1.1 Cortex-M4 Architecture

1.1.1.1 Registers Overview

The Cortex-M4 architecture includes a set of general-purpose registers (R0-R12), a stack pointer (SP), a link register (LR), a program counter (PC), and a program status register (xPSR), all of which are 32-bit registers. The general-purpose registers are used for data manipulation and temporary storage during program execution. The SP is used to manage the call stack, while the LR holds the return address for function calls. The PC points to the next instruction to be executed, and the xPSR contains flags and status information about the processor state.

Program Status Register (xPSR) holds the current state of the processor, including condition flags (Negative, Zero, Carry, Overflow), interrupt status, and execution state. These flags are updated based on the results of arithmetic and logical operations, allowing for conditional branching and decision-making in programs.

1.1.1.2 Memory Mapping

The Cortex-M4 uses a flat memory model, where all memory locations are accessible through a single address space. This model simplifies programming and allows for efficient access to data and instructions. The memory is divided into several regions, including code memory (for storing instructions), data memory (for storing variables), and peripheral memory (for interfacing with hardware components). The architecture supports both little-endian and big-endian data formats, with little-endian being the default.

Table 1.1: Cortex-M4 Memory Regions (ARMv7-M)

Region	Address Range	Description
Code	`0x00000000`–`0x1FFFFFFF`	Flash memory for program code
SRAM	`0x20000000`–`0x3FFFFFFF`	On-chip static RAM for data
Peripheral	`0x40000000`–`0x5FFFFFFF`	Memory-mapped peripheral registers
External RAM	`0x60000000`–`0x9FFFFFFF`	External RAM (if implemented)
External Device	`0xA0000000`–`0xDFFFFFFF`	External devices/memory (if implemented)
Private Peripheral Bus	`0xE0000000`–`0xE00FFFFF`	Cortex-M4 internal peripherals (NVIC, SysTick, MPU, SCB)
System	`0xE0100000`–`0xFFFFFFFF`	System region (reserved/system-level)

1.1.2 Assembly Language Basics

Assembly language is a low-level programming language that provides a direct correspondence between human-readable instructions and the machine code executed by the processor. Each instruction encodes a specific operation, such as moving data, performing arithmetic or logic, or altering control flow. Because it maps so closely to hardware, assembly allows precise control of system resources and is commonly used in performance-critical routines or when direct access to hardware is required.

Assembly programs are typically composed of three main elements: *instructions*, *directives*, and *labels*.

Instructions Instructions are the executable commands that the CPU carries out. Examples include data movement (`MOV`), arithmetic (`ADD`, `SUB`), logical operations (`AND`, `ORR`), and control flow (`B`, `BL`). Each instruction directly translates to one or more machine code opcodes and determines the actual behavior of the program.

Directives Directives are commands to the assembler that guide how source code is translated into machine code but do not generate instructions themselves. Common examples include `AREA` (define a code or data section), `ALIGN` (align data to memory boundaries), `DCD` (allocate and initialize a word of storage), and `EXPORT` (export a symbol for linking). Directives organize program layout, control memory allocation, and manage symbol visibility.

Labels Labels are symbolic names that mark specific locations in code or data. They act as targets for jumps and branches or as references for data access. Labels improve program readability and maintainability by avoiding hard-coded addresses. For instance, a label like `loop_start` can be used as the destination of a branch instruction, and the assembler automatically computes the correct relative address.

1.1.2.1 Instruction Set Overview

The ARM Cortex-M4 instruction set is a subset of the ARMv7-M architecture, designed for efficient execution in embedded systems. It includes a variety of instructions for data processing, memory access, and control flow. Key categories of instructions include:

- **Data Processing Instructions**: These include arithmetic operations (e.g., `ADD`, `SUB`), logical operations (e.g., `AND`, `ORR`), and data movement instructions (e.g., `MOV`, `MVN`).
- **Load and Store Instructions**: Instructions like `LDR` (load register) and `STR` (store register) are used to transfer data between registers and memory.
- **Branch Instructions**: Control flow is managed using branch instructions such as `B` (branch), `BL` (branch with link), and conditional branches like `BEQ` (branch if equal).
- **Special Instructions**: These include instructions for system control, such as `NOP` (no operation), `WFI` (wait for interrupt), and instructions for manipulating the stack and handling exceptions.

1.1.2.2 General Instruction Format

Assembly source lines generally follow this shape:

```
[label]  OPCODE{<cond>}{S}  operands  ; comment
```

where curly braces `{}` denote optional components, and:

- `label`: optional symbolic name marking the current address.
- `OPCODE`: instruction mnemonic (e.g., `ADD`, `MOV`, `B`).
- `<cond>`: optional condition code (e.g., `EQ`, `NE`, `LT`, `GE`) that predicates execution.
- `S`: optional suffix indicating whether to update the condition flags (e.g., `ADDS`).
- `operands`: registers, immediates, or memory operands (e.g., `R0, R1, #1` or `[R2]`).
- Anything after a semicolon (`;`) is a comment ignored by the assembler.

```
loop_start   ADDS   R0, R0, #1      ; R0 = R0 + 1, update flags (N,Z,C,V)
```

Listing 1.1: Instruction format example

1.1.3 Basic Program Template (Boilerplate)

The minimal skeleton below shows a valid vector table in a `READONLY RESET` area, a `READWRITE` data area for variables, and a code area with the `Reset_Handler` entry point.

```
        AREA    RESET, CODE, READONLY      ; Vector table lives in read-only code
        EXPORT  __Vectors                  ; Make symbol visible to the linker
__Vectors
        DCD     0x20001000                 ; Initial SP value (top of stack in SRAM)
        DCD     Reset_Handler              ; Reset vector: entry address
        ALIGN
        ; -------------------- Read-Write Data --------------------
        AREA    M_DATA, DATA, READWRITE    ; Variables go here (RAM)
        EXPORT  COUNT                      ; Export if referenced by other modules
COUNT   DCD     0                          ; Initialized RW variable (word)
BUF     SPACE   16                         ; Uninitialized RW buffer (16 bytes)
        ALIGN
        ; -------------------- Application Code -------------------
        AREA    MYCODE, CODE, READONLY
        ENTRY
        EXPORT  Reset_Handler
Reset_Handler
        ; Example: COUNT++ and store to BUF[0]
        LDR     R0, =COUNT                 ; R0 <- &COUNT
        LDR     R1, [R0]                   ; R1 <- COUNT
        ADDS    R1, R1, #1                 ; R1 = R1 + 1 (update flags)
        STR     R1, [R0]                   ; COUNT <- R1
        LDR     R2, =BUF                   ; R2 <- &BUF
        STRB    R1, [R2]                   ; BUF[0] <- (low byte of R1)
STOP    B       STOP                       ; Stay here forever
        END
```

Listing 1.2: Cortex-M4 boilerplate with READWRITE data

What each directive does

- `AREA <name>, CODE|DATA, READONLY|READWRITE`: defines a section. Put the vector table and program text in `CODE, READONLY`; put variables in `DATA, READWRITE`.
- `EXPORT <symbol>`: makes a label visible to the linker/other modules.
- `DCD, DCW, DCB <values>`: allocate and initialize words, halfwords, or bytes.
- `SPACE <n>`: reserve n bytes of uninitialized storage in RAM.
- `ALIGN`: align to a suitable boundary (commonly 4 bytes for words).
- `ENTRY`: mark the entry point of the image for the toolchain.
- `END`: end of source file.

Notes

- The first two words in `__Vectors` must be the initial stack pointer value and the address of `Reset_Handler`.
- The assembler/linker places sections in appropriate memory regions based on the target device and linker script.
- Labels must start at the beginning of the line (no indentation), while instructions and directives should be indented for proper assembly.
- Variables in `READWRITE` areas are initialized to zero by default. While you can specify initial values using directives like `DCD`, the linker will place these in flash and copy them to RAM during startup, or they may be zeroed out during RAM initialization.

1.2 Procedure

1.2.1 Setting Up the Keil uVision5 Environment

Make sure you have the **Keil uVision5** IDE installed on your computer. If not, download and install it from the official Keil website (https://www.keil.com/demo/eval/arm.htm).

1.2.1.1 Creating a New Project

1. Open **Keil uVision5** and create a new project:
   - Go to `Project > New uVision Project...`
   - Choose a directory and name for your project (e.g., `Exp01_ARM_Assembly`).
2. Select the target device:
   - In the "Select Device for Target" dialog, choose ARM Cortex-M4 (ARMCM4) as we will be using it only for simulation.
   - Click "OK" to confirm.
3. Configure project settings:
   - Go to `Project > Options for Target 'Target 1'...`
   - Under the "Debug" tab, select "Use Simulator" as the debug driver.
4. Add a new assembly file to the project:
   - Right-click on "Source Group 1" in the Project window and select `Add New Item to Group 'Source Group 1'...`
   - In the "Add New Item" dialog, select "Assembly File" from the list.
   - Name the file (e.g., `main.s`) and click "Add".
5. Build the project:
   - Click on the "Build" button (or go to `Project > Build Target`), or use the shortcut `F7`.
   - Check the "Output" window for any errors or warnings. If there are errors, fix them in your assembly code and rebuild.
   - Once the build is successful, you should see a message indicating that the build was completed without errors.
6. Start a debugging session:
7. Click on the "Debug" button (or go to `Debug > Start/Stop Debug Session`), or use the shortcut `Ctrl + F5`.

1.2.1.2 Debugging and Running the Program

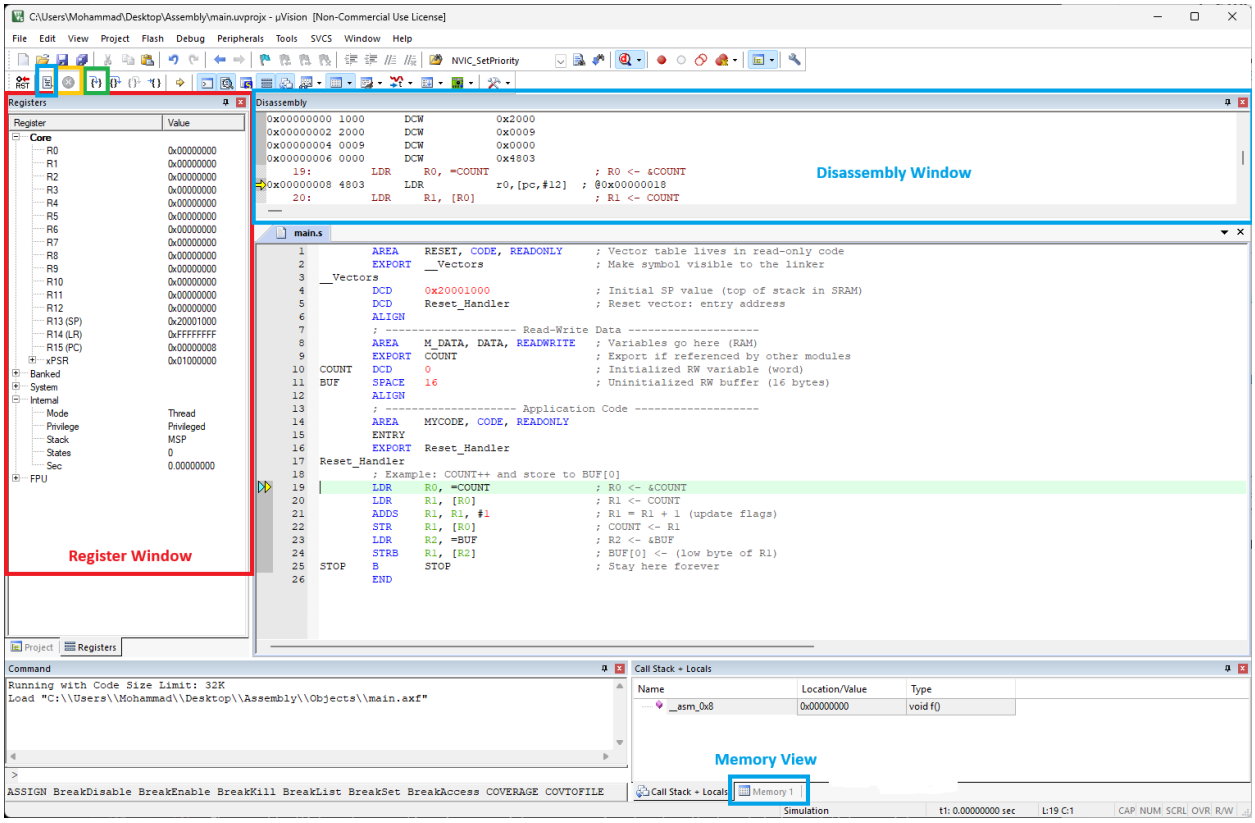


Figure 1.1: Keil uVision5 Debugging Interface

Figure 1.1 shows the Keil uVision5 debugging interface. You can run and debug your assembly program using two main approaches:

1. **Step by Step Execution**:
   - Use the "Step" button (or press `F11`) marked in green in Figure 1.1 to execute your program one instruction at a time.
   - Observe the changes in the registers and memory as you step through each instruction.
2. **Run the entire program**:
   - Use the "Run" button (or press `F5`) marked in blue in Figure 1.1 to execute your program continuously until it hits a breakpoint or completes execution.
   - After running, you must stop the execution using the "Stop" button marked in yellow in Figure 1.1.
   - Check the final values in the registers and memory to verify the program's behavior.

1.2.2 Examples

1.2.2.1 Example 1: Simple Arithmetic Operations and Flag Manipulation

The following example demonstrates basic arithmetic operations, flag manipulation.

```
        AREA    RESET, CODE, READONLY
        EXPORT  __Vectors
__Vectors
        DCD     0x20001000          ; Initial SP (example top-of-stack)
        DCD     Reset_Handler       ; Reset vector
        AREA    MYCODE, CODE, READONLY
        ENTRY
        EXPORT  Reset_Handler
Reset_Handler
; ========== Part A: Moving immediates (MOV, MOVW/MOVT, MOV32, LDR =) =========
        ; 1) Simple 8/12-bit immediate
        MOV     R2, #0x01           ; R2 = ?   (after step)
        ; 2) 16-bit low half into R5
        MOV     R5, #0x3210         ; R5 = ?   (low 16 bits set)
        ; 3) High half into R5: combine with (2)
        MOVT    R5, #0x7654         ; R5 = ?   (now 0x76543210)
        ; 4) 32-bit immediate with MOV32 macro (emits MOVW+MOVT)
        MOV32   R6, #0x87654321     ; R6 = ?
        ; 5) Literal load of a 32-bit immediate
        LDR     R7, =0x87654321     ; R7 = ?
; ========== Part B: ADD/SUB without and with flag updates =========
        MOV     R2, #0x02           ; R2 = ?
        MOV     R3, #0x03           ; R3 = ?
        ADD     R1, R2, R3          ; R1 = ?   (flags unchanged)
        ; Now set R3 to all ones, then add with flags
        MOV32   R3, #0xFFFFFFFF     ; R3 = ?
        ADDS    R1, R2, R3          ; R1 = ?   FLAGS? (N,Z,C,V)
        SUBS    R1, R2, R3          ; R1 = ?   FLAGS? (N,Z,C,V)
        ; Same add but without S (no flag update)
        MOV     R4, #0xFF           ; small value for contrast
        ADD     R1, R2, R4          ; R1 = ?   FLAGS? (should be unchanged)
        ; Now with S, so flags DO update
        ADDS    R1, R2, R4          ; R1 = ?   FLAGS? (N,Z,C,V)
; ========== Part C: Overflow / Zero / Negative flag demos =========
        ; Create an ADD overflow with two large positive numbers
        MOV32   R2, #0x7FFFFFFF     ; R2 = ?
        MOV32   R3, #0x7FFFFFFF     ; R3 = ?
        ADDS    R1, R2, R3          ; R1 = ?   Overflow expected? FLAGS?
        ; Create a ZERO result
        MOV     R2, #1
        SUBS    R1, R2, #1          ; R1 = ? (=0)  FLAGS? (Z should be 1)
        ; Create a NEGATIVE result
        SUBS    R1, R2, #2          ; R1 = ? (negative) FLAGS? (N should be 1)
STOP    B       STOP                 ; infinite loop
        END
```

Listing 1.3: Example 1: Simple Arithmetic Operations and Flag Manipulation

2 Data Processing and Memory Operations

Learning Objectives

After completing this experiment, you will be able to:

- Perform arithmetic, logical, and shift/rotate operations using data-processing instructions (including `Operand2` with the barrel shifter).
- Move data between registers and memory using load/store instructions with immediate, register-offset, and pre-/post-indexed addressing modes.
- Declare and initialize data objects (arrays, strings, buffers) with assembler directives, and use pointers to access and modify them.
- Trace how instructions affect registers and xPSR flags using the **Keil uVision5** debugger (breakpoints, single-step, register/memory views).

Experiment Overview

This experiment develops fluency with the ARM Cortex-M4 instruction set for data manipulation and memory access. You will practice arithmetic, logical, and shift/rotate instructions, apply various addressing modes for load/store operations, and define data structures with assembler directives. By the end of this lab, you will be able to write assembly routines that perform register-level computation and structured memory access, providing the foundation for flow control and procedure calls in later experiments.

2.1 Theoretical Background

As mentioned in Experiment 1, assembly instructions are split into three main categories: data processing, load/store, and branch instructions. This experiment focuses on data processing instructions, load/store instructions and their addressing modes. branch instructions and flow control will be covered in the next experiment.

2.1.1 Data Processing Instructions

Data processing instructions perform arithmetic and logical operations on data stored in registers. They can also manipulate the condition flags in the xPSR based on the results of the operations. Common data processing instructions take the following form:

```
{LABEL}  OPCODE{<cond>}{S} Rd, Rn, Operand2
```

where:

- `LABEL`: optional label for branching.
- `OPCODE`: the operation to be performed (e.g., `ADD`, `SUB`, `AND`, `ORR`).
- `<cond>`: optional condition code that predicates execution.
- `S`: optional suffix indicating whether to update the condition flags.
- `Rd`: destination register where the result is stored.
- `Rn`: first operand register.
- `Operand2`: second operand, which can be an immediate value limited to 8 bits, a register, or a barrel shifter operation (see Section 2.1.1.4).

2.1.1.1 Arithmetic Instructions

Arithmetic instructions perform basic mathematical operations. Some common arithmetic instructions include addition, subtraction, multiplication, and their variants. The following table summarizes some of the most commonly used arithmetic instructions in the ARM Cortex-M4 architecture.

Table 2.1: Common ARM Cortex-M4 Arithmetic Instructions

Instr.	**Syntax**	**Operation**	**Description**
ADD	`ADD{S} Rd, Rn, Operand2`	$Rd \leftarrow Rn + Operand2$	*Operand2* may be a register, an immediate, or a shifted register.
ADC	`ADC{S} Rd, Rn, Operand2`	$Rd \leftarrow Rn + Operand2 + C$	Adds carry-in C.
SUB	`SUB{S} Rd, Rn, Operand2`	$Rd \leftarrow Rn - Operand2$	Standard subtraction.
SBC	`SBC{S} Rd, Rn, Operand2`	$Rd \leftarrow Rn - Operand2 - \overline{C}$	Subtract with carry. If carry flag is clear, result is reduced by one. Used for multiword arithmetic.
RSB	`RSB{S} Rd, Rn, Operand2`	$Rd \leftarrow Operand2 - Rn$	Reverse subtract.
MUL	`MUL{S} Rd, Rn, Rm`	$Rd \leftarrow (Rn \times Rm)_{[31:0]}$	32×32 → low 32 bits.
MLA	`MLA Rd, Rn, Rm, Ra`	$Rd \leftarrow (Rn \times Rm) + Ra$	Multiply-accumulate.
MLS	`MLS Rd, Rn, Rm, Ra`	$Rd \leftarrow Ra - (Rn \times Rm)$	Multiply-subtract.
UMULL	`UMULL RdLo, RdHi, Rn, Rm`	$\{RdHi, RdLo\} \leftarrow Rn \times Rm$	Unsigned 32×32 → 64-bit product.
SMULL	`SMULL RdLo, RdHi, Rn, Rm`	$\{RdHi, RdLo\} \leftarrow Rn \times Rm$	Signed 32×32 → 64-bit product.
UDIV	`UDIV Rd, Rn, Rm`	$Rd \leftarrow Rn \div Rm$	Unsigned division.
SDIV	`SDIV Rd, Rn, Rm`	$Rd \leftarrow Rn \div Rm$	Signed division.

Note: C denotes the carry flag in `xPSR`. *Operand2* may be an immediate or a shifted register depending on the encoding.

2.1.1.2 Logical and Move Instructions

Logical instructions perform bitwise operations on data, while move instructions transfer data between registers or load immediate values. The following table summarizes some of the most commonly used logical and move instructions in the ARM Cortex-M4 architecture.

Table 2.2: Logical and Move Instructions

Instr.	Syntax	Operation	Description
AND	`AND Rd, Rn, Operand2`	$Rd \leftarrow Rn \,\&\, Operand2$	Bitwise AND.
ORR	`ORR Rd, Rn, Operand2`	$Rd \leftarrow Rn \mid Operand2$	Bitwise OR.
EOR	`EOR Rd, Rn, Operand2`	$Rd \leftarrow Rn \oplus Operand2$	Bitwise XOR.
BIC	`BIC Rd, Rn, Operand2`	$Rd \leftarrow Rn \,\&\, \neg Operand2$	Bit clear.
MVN	`MVN Rd, Operand2`	$Rd \leftarrow \neg Operand2$	Bitwise NOT of operand.
MOV	`MOV Rd, Operand2`	$Rd \leftarrow Operand2$	Register or immediate move.
MOVW	`MOVW Rd, #imm16`	$Rd[15{:}0] \leftarrow imm16$	Write low halfword.
MOVT	`MOVT Rd, #imm16`	$Rd[31{:}16] \leftarrow imm16$	Write high halfword (low preserved).

Note: C denotes the carry flag in `xPSR`. *Operand2* may be an immediate or a shifted register depending on the encoding.

In this experiment, you will work with bitwise logical instructions to manipulate individual bits within registers. Such operations are fundamental in microcontroller programming, where control and status registers often contain multiple configuration fields packed into a single 32-bit word. Understanding how to set, clear, toggle, or test specific bits without altering the rest of the register is essential for safely modifying hardware configurations and controlling peripherals.

Set and Clear Bits To set, clear, or toggle specific bits in a register, you can use the following logical instructions:

- `ORR Rd, Rn, #mask`: Sets bits in `Rd` where the corresponding bits in `mask` are 1.
- `BIC Rd, Rn, #mask`: Clears bits in `Rd` where the corresponding bits in `mask` are 1.
- `EOR Rd, Rn, #mask`: Toggles bits in `Rd` where the corresponding bits in `mask` are 1.

BIC is essentially an AND operation with the negated mask, i.e., `BIC Rd, Rn, #mask` is equivalent to `AND Rd, Rn, #~mask`.

Check Bits To check whether certain bits are set or cleared, you can use the `AND` instruction followed by a comparison:

- `AND Rd, Rn, #mask`: Isolates bits in `Rn` where the corresponding bits in `mask` are 1. The result is stored in `Rd`.
- You can then use `CMP Rd, #0` to determine if the result is zero (all masked bits cleared) or non-zero (at least one bit set).
- Alternatively, you can use `TST Rn, #mask`, which performs the AND operation and updates the condition flags without storing the result.

The `TST` instruction and its interaction with the condition flags will be explored in more detail in the next experiment, where you will learn how conditional execution and branching depend on flag states.

2.1.1.3 Shift and Rotate Instructions

Table 2.3: Shift and Rotate Instructions

Instr.	Syntax	Operation	Description
LSL	`LSL Rd, Rm, #sh|Rs`	$Rd \leftarrow Rm \ll sh$	Logical left shift by immediate or by register.
LSR	`LSR Rd, Rm, #sh|Rs`	$Rd \leftarrow Rm \gg sh$	Logical right shift (zero fill).
ASR	`ASR Rd, Rm, #sh|Rs`	$Rd \leftarrow Rm \gg sh$	Arithmetic right shift (sign fill).
ROR	`ROR Rd, Rm, #sh|Rs`	$Rd \leftarrow \mathrm{ROR}(Rm, sh)$	Rotate right by immediate or by register.
RRX	`RRX Rd, Rm`	$Rd \leftarrow \mathrm{ROR}_C(Rm, 1)$	Rotate right 1 bit through carry (uses C as incoming bit 31, outgoing bit $0 \rightarrow C$).

Note: Shift amount can be an immediate `#sh` (0–31) or a register `Rs` (low 8 bits used). For immediates: `LSL #0` = no shift; `LSR #0` is treated as shift by 32; `ASR #0` is treated as shift by 32; `ROR #0` means `RRX`.

Not all shift/rotate instructions are explicitly present in the ARMv7-M ISA. For example, there is no ROL (rotate left) or ASL (arithmetic shift left) instruction, as these operations can be achieved using existing shift instructions: ROL can be implemented using ROR with a complementary shift amount, and ASL is equivalent to LSL.

2.1.1.4 Barrel Shifter

The barrel shifter is a hardware feature that allows for efficient shifting and rotating of register values as part of data processing instructions. It can perform operations such as logical shifts (left or right), arithmetic shifts, and rotations on the second operand (`Operand2`) before it is used in the instruction without wasting extra instructions or cycles.
Examples of barrel shifter usage:

```
ADD    R0, R2, R1, LSL #2    ; R0 = R2 + (R1 << 2) using barrel shifter
SUB    R3, R4, R5, LSR #1    ; R3 = R4 - (R5 >> 1) using barrel shifter
ORR    R6, R7, R8, ROR #3    ; R6 = R7 | (R8 rotated right by 3)
```

Listing 2.1: Barrel shifter examples

2.1.2 Load and Store Instructions

Since the ARM Cortex-M4 follows the RISC design philosophy, it uses a load/store architecture. This means that arithmetic and logical instructions operate only on registers. Any data in memory must first be loaded into a register before processing, and results must be stored back to memory if they need to be preserved.

Table 2.4: Load and Store Instructions (Summary)

Instr.	Syntax Example	Description
LDR / STR	`LDR/STR Rt, [Rn, #off]`	Load/store a 32-bit word.
LDRB / STRB	`LDRB/STRB Rt, [Rn, #off]`	Load/store an 8-bit byte.
LDRH / STRH	`LDRH/STRH Rt, [Rn, #off]`	Load/store a 16-bit halfword.
LDRSB / LDRSH	`LDRSB/LDRSH Rt, [Rn, #off]`	Load signed byte/halfword and sign-extend to 32 bits.
LDRD / STRD	`LDRD/STRD Rt, Rt2, [Rn, #off]`	Load/store a 64-bit doubleword (two registers).

```
Reset_Handler
        LDR     R0, =XVAL          ; R0 = &XVAL (address of XVAL)
        LDR     R1, [R0]           ; R1 = 0x12345678 (load word from memory)
        ADR     R5, XVAL           ; R5 = PC-relative address of XVAL (if in range)

        LDRB    R2, [R0]           ; R2 = 0x78 (lowest byte of XVAL)
        LDRH    R3, [R0]           ; R3 = 0x5678 (lowest halfword of XVAL)

        MOV     R4, #0xFF
        LDR     R0, YPTR           ; R0 = contents of YPTR = &YVAL
        STRB    R4, [R0]           ; store 0xFF into YVAL (low byte only)

STOP    B       STOP

        AREA    CONSTANTS, DATA, READONLY
XVAL    DCD     0x12345678         ; word in memory
YPTR    DCD     YVAL               ; contains the address of YVAL

        AREA    MYDATA, DATA, READWRITE
YVAL    DCD     0

        END
```

Listing 2.2: Examples of Load and Store Instructions

2.1.2.1 Declaring Data in Memory

Data in assembly is defined using *assembler directives* that reserve and optionally initialize memory. Common directives include `DCD`, `DCW`, and `DCB`, which define words, halfwords, and bytes, respectively. These are typically placed within a `DATA` area to create arrays, lookup tables, buffers, or strings.

- `DCD` — Define Constant Word (32 bits per element)
- `DCW` — Define Constant Halfword (16 bits per element)
- `DCB` — Define Constant Byte (8 bits per element)
- `SPACE` — Reserve uninitialized memory (in bytes)
- `FILL` — Fill memory with a specified value for a given length

```
            AREA    CONSTANTS, DATA, READONLY
NUMBERS     DCD     10, 20, 30, 40        ; array of 32-bit integers
BYTES       DCB     1, 2, 3, 4            ; array of bytes
TEXT        DCB     "HELLO",0             ; null-terminated ASCII string
            AREA    MYDATA, DATA, READWRITE
BUFFER      SPACE   64                    ; reserve 64 bytes (uninitialized)
PATTERN     FILL    0xFF, 64              ; fill 64 bytes with 0xFF
```

Listing 2.3: Declaring arrays and strings in memory

Each label (e.g., `NUMBERS`, `TEXT`) marks the starting address of a data object in memory. You can load these addresses into registers using `LDR R0, =NUMBERS` or `ADR R0, TEXT`, then access individual elements through the appropriate addressing modes.
Note: Strings are stored as consecutive ASCII characters in memory. A terminating zero (`0x00`) is typically appended to indicate the end of the string, similar to C-style strings.

2.1.2.2 Understanding Pointer Declarations

The directive `YPTR DCD YVAL` reserves a 32-bit word at the label `YPTR` and initializes it with the address of `YVAL`. In other words, `YPTR` acts as a *pointer variable* that holds the address of another variable (`YVAL`). Executing `LDR Rn, YPTR` loads the 32-bit contents stored at `YPTR`—that is, the address of `YVAL`—into `Rn`, making `Rn` a pointer to `YVAL`.

Address	Label	Contents
0x2000	XVAL	0x12345678
0x2004	YPTR	0x2008 (address of YVAL)
0x2008	YVAL	0x00000000

2.1.2.3 Loading Addresses and Values: `LDR`, `LDR =`, and `ADR`

In ARM assembly, it is important to distinguish between loading a *value* from memory and loading the *address* of a label. Although these instructions look similar, their behavior and purpose differ depending on how the assembler interprets them.

- **`LDR Rn, label`** Loads the 32-bit *value* stored at the memory address identified by `label` into register `Rn`. The CPU performs a direct memory read:

  $$\texttt{Rn} \leftarrow [\texttt{label}]$$

  Example: `LDR R0, XVAL` loads the contents of `XVAL` (e.g., `0x12345678`) into `R0`.
- **`LDR Rn, =label`** Loads the *address* of `label` into `Rn`, rather than the data stored at that address. The assembler generates this by either constructing the address using `MOVW/MOVT` instructions or placing it in a nearby *literal pool* for PC-relative loading.[1]

  $$\texttt{Rn} \leftarrow \texttt{\&label}$$

  Example: `LDR R0, =XVAL` places the address of `XVAL` in `R0`.
- **`ADR Rn, label`** Loads the *address* of `label` into `Rn` by computing it relative to the current program counter. This method requires no literal pool or memory access but only works for nearby addresses (about ±4 KB in Thumb mode):

  $$\texttt{Rn} \leftarrow \texttt{PC} + \texttt{offset(label)}$$

2.1.3 Addressing Modes

Addressing modes define how the effective address or operand value is obtained by an instruction. The ARM Cortex-M4 supports several common addressing modes, summarized below:

Table 2.5: General Addressing Modes in ARM Cortex-M4

Mode	Syntax Example	Description
Immediate	`MOV R0, #10`	Operand is a constant value encoded in the instruction.
Register Direct	`MOV R0, R1`	Operand is taken directly from a register.
Register Indirect	`LDR R0, [R1]`	Register holds the address of the operand in memory.
Register Offset	`LDR R0, [R1, R2]`	Effective address = base register + offset register.
Immediate Offset	`LDR R0, [R1, #4]`	Effective address = base register + constant offset.
Pre-indexed	`LDR R0, [R1, #4]!`	Base updated first, then memory access.
Post-indexed	`LDR R0, [R1], #4`	Memory access first, then base register updated.

[1] For more details, see ARM Developer Documentation, "Literal pools and `LDR =const`". https://developer.arm.com/documentation/dui0473/m/dom1359731147760

```
; Immediate Offset
    LDR     R0, [R1, #4]      ; R0 = word at memory[R1 + 4]
; Register Offset
    LDR     R0, [R1, R2]     ; R0 = word at memory[R1 + R2]
; Pre-indexed
    LDR     R0, [R1, #4]!     ; R1 = R1 + 4, then load R0 = [R1]
; Post-indexed
    LDR     R0, [R1], #4      ; load R0 = [R1], then R1 = R1 + 4
```

Listing 2.4: Examples of Offset, Pre-indexed, and Post-indexed Addressing Modes

2.2 Procedure

2.2.1 Examples

2.2.1.1 Example 1 — Data Processing Instructions

This example demonstrates various arithmetic and bitwise operations on registers.

```
        AREA  RESET, CODE, READONLY
        EXPORT __Vectors
__Vectors
        DCD 0x20001000
        DCD Reset_Handler
        ALIGN

        AREA MYCODE, CODE, READONLY
        ENTRY
        EXPORT Reset_Handler
Reset_Handler
        ; Load values from memory into registers
        LDR     R1, NUM1            ; R1 = 50
        LDR     R2, NUM2            ; R2 = 12
        ; Arithmetic operations
        ADD     R3, R1, R2          ; R3 = 50 + 12 = 62
        SUB     R3, R3, #4          ; R3 = 62 - 4 = 58
        MUL     R4, R3, R2          ; R4 = 58 * 12 = 696
        ; Logical operations
        AND     R5, R4, #0xFF       ; R5 = 696 & 0xFF = 0xB8 (184)
        ORR     R5, R5, #0x01       ; R5 = 0xB8 | 0x01 = 0xB9 (185)
        BIC     R5, R5, #0x08       ; R5 = 0xB9 & ~0x08 = 0xB1 (177)
        EOR     R5, R5, #0x02       ; R5 = 0xB1 ^ 0x02 = 0xB3 (179)
        ; Shift and Barrel Shifter
        LSL     R8, R4, #4          ; logical left shift by 4 bits
        LSR     R9, R4, #8          ; logical right shift by 8 bits
        ADD     R5, R5, R5, LSL #1  ; R5 = R5 + (R5 << 1) = 3 * R5
        ; Store result in memory using a pointer
        LDR     R6, RP              ; R6 = address of RESULT
        STR     R5, [R6]            ; RESULT = R5
        ; Read back for verification
        LDR     R7, [R6]            ; R7 = RESULT
STOP    B       STOP

        AREA CONSTANTS, DATA, READONLY
NUM1    DCD     50              ; First integer
NUM2    DCD     12              ; Second integer
RP      DCD     RESULT          ; Pointer to RESULT variable

    AREA MYDATA, DATA, READWRITE
RESULT  DCD     0               ; Will hold the final computed value
    END
```

Listing 2.5: Arithmetic and bitwise operations example

2.2.1.2 Example 2 — Load/Store with Different Addressing Modes

This example demonstrates load and store instructions using various addressing modes.

```
        AREA    RESET, CODE, READONLY
        EXPORT  __Vectors
__Vectors
        DCD     0x20001000          ; Initial SP (example)
        DCD     Reset_Handler       ; Reset vector
        AREA    MYCODE, CODE, READONLY
        ENTRY
        EXPORT  Reset_Handler
Reset_Handler
        LDR     R0, =ARRAY          ; R0 = &ARRAY (base address)
        ; 1) Immediate Offset: EA = R0 + #8  (third element)
        ; -------------------------------------------------------
        LDR     R1, [R0, #8]        ; R1 = ARRAY[2] = ? (expect 30)
        LDR     R7, =OUT
        STR     R1, [R7, #0]        ; OUT[0] = R1
        ; 2) Pre-indexed: R2 = [R2 + #4], then R2 = R2 + #4
        ;    Use a scratch pointer so R0 remains the base.
        ; -------------------------------------------------------
        MOV     R2, R0              ; R2 = &ARRAY
        LDR     R3, [R2, #4]!       ; R2 -> &ARRAY[1], R3 = ARRAY[1] = ? (expect
    20)
        STR     R3, [R7, #4]        ; OUT[1] = R3
        ; After this, R2 now points at ARRAY[1].
        ; 3) Post-indexed: R4 = [R4], then R4 = R4 + #12
        ;    Load first element, then advance pointer to the 4th.
        ; -------------------------------------------------------
        MOV     R4, R0              ; R4 = &ARRAY
        LDR     R5, [R4], #12       ; R5 = ARRAY[0] = ? (expect 10), R4 ->
    &ARRAY[3]
        STR     R5, [R7, #8]        ; OUT[2] = R5
        ; 4) Register Offset: EA = R0 + R6
        ;    Offset register holds byte offset (multiple of 4 for words).
        ; -------------------------------------------------------
        MOV     R6, #12             ; byte offset to ARRAY[3]
        LDR     R8, [R0, R6]        ; R8 = ARRAY[3] = ? (expect 40)
        STR     R8, [R7, #12]       ; OUT[3] = R8
        ; read second element via register offset
        MOV     R6, #4              ; byte offset to ARRAY[1]
        LDR     R9, [R0, R6]        ; R9 = ARRAY[1] = ? (expect 20)
STOP    B       STOP
        AREA    CONSTS, DATA, READONLY
ARRAY   DCD     10, 20, 30, 40      ; four words at consecutive addresses
        AREA    MYDATA, DATA, READWRITE
OUT     DCD     0, 0, 0, 0          ; capture buffer for observed loads.
        END
```

Listing 2.6: Load/store with different addressing modes example

2.2.2 Tasks

2.2.2.1 Task 1 — Bitwise Register Manipulation

Start with `R0 = 0x12345678`. Perform the following operations and observe the results in the debugger (verify in hex and binary):

- Clear bits 4–7 (second hex nibble).
- Set bits 8–11 (force that nibble to `F`).
- Toggle bits 28–31 (highest nibble).

Hint: Use `BIC`, `ORR`, and `EOR` with appropriate masks.

2.2.2.2 Task 2 — Addressing Modes with an Array

Given:

```
ARRAY   DCD  0x11, 0x22, 0x33, 0x44
OUT     SPACE 16
```

Load each element using a different addressing mode, then store to `OUT`:

- 0x11 via *immediate offset*
- 0x22 via *pre-indexed*
- 0x33 via *post-indexed*
- 0x44 via *register offset*

Hint: Put `ARRAY`'s base in `R1` (e.g., `LDR R1, =ARRAY`). Verify `OUT` in memory after execution.

3 Control Flow and Subroutines

Learning Objectives

After completing this experiment, you will be able to:

- Implement conditional and unconditional branching using ARM branch instructions.
- Design and implement loop constructs (`for`, `while`) using compare and branch instructions.
- Create and call procedures with proper parameter passing and return mechanisms.
- Manage the stack for local variables, parameter passing, and nested procedure calls.
- Apply the ARM Procedure Call Standard (AAPCS) to ensure correct register usage and calling conventions.

Experiment Overview

This experiment introduces program control flow, procedure implementation, and stack management on the ARM Cortex-M4 processor. You will implement conditional branches and loops, write procedures following the ARM Procedure Call Standard (AAPCS), and handle nested calls with parameter passing using the stack. By the end of this lab, you will understand how to create modular assembly programs with proper control flow, calling conventions, and stack management.

3.1 Theoretical Background

3.1.1 Flow Control

Flow control instructions alter the sequential execution of instructions by changing the program counter (PC). These instructions enable the implementation of conditional statements, loops, and procedure calls that are fundamental to structured programming.

3.1.1.1 Condition Evaluation

Before implementing control flow, we must understand how to evaluate conditions and set processor flags. ARM provides dedicated instructions for comparing values and testing bit patterns that update the condition flags without storing results.

Table 3.1: ARM Cortex-M4 Compare and Test Instructions

Instr.	Syntax	Description / Usage
CMP	`CMP Rn, Operand2`	Compare `Rn` with `Operand2` (`Rn - Operand2`); updates flags (Z, N, C, V).
CMN	`CMN Rn, Operand2`	Compare negative (`Rn + Operand2`); used for checking against negative values.
TST	`TST Rn, Operand2`	Logical AND test (`Rn AND Operand2`); sets Z if all tested bits are 0.
TEQ	`TEQ Rn, Operand2`	Logical XOR test (`Rn EOR Operand2`); sets Z if operands are equal.

Usage Notes:

- These instructions only affect the condition flags (N, Z, C, V) — they do not store a result.
- `CMP`/`CMN` are arithmetic comparisons; `TST`/`TEQ` are logical bitwise comparisons.
- Many data-processing instructions can update flags by appending `S` (e.g., `ADDS`, `SUBS`).
- Common use: immediately followed by conditional branches such as `BEQ`, `BNE`, `BGT`, etc.

Examples:

```
CMP     R0, #10          ; Compare R0 with 10
BLT     LessThan10       ; Branch if R0 < 10
BGE     GreaterOrEqual   ; Otherwise, R0 >= 10
```

Listing 3.1: Arithmetic comparison using CMP

```
MOV     R1, #0x12        ; R1 = 0001 0010b
TST     R1, #0x10        ; Test if bit 4 is set
BEQ     BitClear         ; Branch if bit 4 = 0 (Z=1)
BNE     BitSet           ; Branch if bit 4 = 1 (Z=0)
```

Listing 3.2: Bit test using TST

```
MOV     R2, #0x55
MOV     R3, #0x55
TEQ     R2, R3           ; XOR -> result 0, sets Z=1
BEQ     ValuesEqual      ; Branch if equal
```

Listing 3.3: Equality check using TEQ

3.1.1.2 Conditional Branching

Branch instructions are the primary mechanism for implementing flow control in ARM assembly. They modify the program counter to jump to different parts of the code based on conditions or unconditionally.

Table 3.2: ARM Cortex-M4 Branch Instructions

Instr.	Syntax	Description / Usage
B	`B label`	Unconditional branch to `label` (always jumps)
B<cond>	`B<cond> label`	Conditional branch based on flags
BL	`BL label`	Branch with link: calls a subroutine, storing return address in `LR`.
BX	`BX Rm`	Branch to address in register, often `BX LR` to return from a subroutine.
CBZ	`CBZ Rn, label`	Branch if `Rn == 0`. Example: `CBZ R0, Done`.
CBNZ	`CBNZ Rn, label`	Branch if `Rn != 0`. Example: loop until counter reaches zero.

`CBZ/CBNZ` instructions have specific constraints:

- **Register**: operand must be a low register `R0-R7`.
- **Range**: branch is *forward-only*; the destination must be within 0–126 bytes after the instruction.
- **Flags**: does not update condition flags (`N, Z, C, V`).

For backward or longer jumps, use `CMP/TST` with conditional branches (`BEQ`, `BNE`, `BGT`, ...).

3.1.1.3 Conditional Execution

ARM assembly supports conditional execution, where most instructions can be conditionally executed based on the current state of the condition flags. This feature allows for efficient implementation of conditional statements without explicit branching.

Conditional Instruction Format: Most ARM instructions can be made conditional by appending a condition code suffix:

```
OPCODE{<cond>} Rd, Rn, Operand2
```

Examples:

- `ADDEQ R0, R1, R2` — Add only if equal (Z=1)
- `MOVNE R3, #10` — Move only if not equal (Z=0)
- `SUBGT R4, R4, #1` — Subtract only if greater than (signed)

Advantages of Conditional Execution:

- **Performance**: Eliminates branch instructions for simple conditional operations
- **Code density**: Reduces the number of instructions needed
- **Pipeline efficiency**: Avoids branch prediction penalties for simple conditions
- **Atomic operations**: Multiple related conditional operations can be grouped

3.1.1.4 How Branch Instructions Work

Branch instructions change the flow of execution by modifying the Program Counter (`PC`). When a branch is executed, the instruction encodes an *offset* which is added to the current value of the `PC`.

Offset calculation: The branch instruction contains a signed immediate value (positive or negative). The processor adds this offset (aligned to halfword boundaries) to the current `PC`.

- A *positive offset* causes a **forward branch** (jump to a higher memory address, later in the program).
- A *negative offset* causes a **backward branch** (jump to a lower memory address, earlier in the program).

Example: Suppose a branch instruction is located at address `0x100`, and the assembler encodes an immediate offset of `-0x08`. The effective target address will be:

$$0x100 + 4 + (-0x08) = 0xFC$$

This means the processor jumps **backward** to an earlier instruction. Such negative offsets are typically used to implement loops (e.g., repeat until zero).

3.1.1.5 Condition Codes

Conditional branches use condition codes that test the processor status flags (N, Z, C, V) set by previous instructions. These enable implementing high-level constructs like if-statements and loops.

Table 3.3: Common ARM Condition Codes

Cond.	Meaning	Description
EQ	Equal	Execute if $Z = 1$.
NE	Not equal	Execute if $Z = 0$.
CS/HS	Carry set / Unsigned higher or same	Execute if $C = 1$.
CC/LO	Carry clear / Unsigned lower	Execute if $C = 0$.
MI	Minus (negative)	Execute if $N = 1$.
PL	Plus (non-negative)	Execute if $N = 0$.
VS	Overflow set	Execute if $V = 1$.
VC	Overflow clear	Execute if $V = 0$.
HI	Unsigned higher	Execute if $C = 1$ *and* $Z = 0$.
LS	Unsigned lower or same	Execute if $C = 0$ *or* $Z = 1$.
GE	Greater or equal (signed)	Execute if $N = V$.
LT	Less than (signed)	Execute if $N \neq V$.
GT	Greater than (signed)	Execute if $Z = 0$ *and* $N = V$.
LE	Less or equal (signed)	Execute if $Z = 1$ *or* $N \neq V$.
AL	Always	Always execute (default if no condition).
NV	Never	Reserved / do not use.

3.1.2 Loop Patterns

Loops are fundamental control structures that repeat a block of code based on conditions. ARM assembly implements loops using combinations of compare instructions, conditional branches, and counters.

3.1.2.1 For Loop Structure

A typical for loop has the structure: initialization, condition check, body execution, and increment/decrement. This type of loop executes a known number of times.

```
        AREA M_DATA, DATA, READONLY
array   DCD 10, 20, 30, 40, 50    ; array of 5 integers
length  EQU 5                     ; number of elements (; just a constant, no memory)
```

Listing 3.4: Declaring Array and Length

```
        ; Initialization
        MOV     R0, #0                  ; i = 0
        MOV     R1, #0                  ; sum = 0
        LDR     R3, =array              ; load base address of array into R3

for_start
        ; Condition check
        CMP     R0, #length             ; compare i with length
        BGE     for_end                 ; if i >= length, exit loop

        ; Loop body
        LDR     R2, [R3, R0, LSL #2]    ; load array[i]; EA: R3 + (R0 * 4)
        ADD     R1, R1, R2              ; sum += array[i]

        ; Increment
        ADD     R0, R0, #1              ; i++
        B       for_start               ; repeat

for_end
```

Listing 3.5: For loop implementation pattern

3.1.2.2 While Loop Structure

While loops check the condition before executing the loop body, potentially executing zero times if the initial condition is false. This type of loop is useful when the number of iterations is not known in advance and depends on dynamic conditions.

```
        AREA M_DATA, DATA, READONLY
mystring DCB "Hello World!", 0    ; null-terminated string
```

Listing 3.6: Declaring Null-Terminated String

Note: `0` and `'0'` are two different values, as the former is actually zero, while the latter is the ASCII code for the character '0' (which is 48 in decimal).

```
    ; Intialization
    LDR     R0, =mystring   ; pointer to string
    MOV     R1, #0          ; character count = 0

while_start
                            ; Condition check
    LDRB    R2, [R0], #1    ; load current character and post-increment pointer
    CMP     R2, #0          ; check for null terminator
    BEQ     while_end       ; if zero, exit loop

                            ; Loop body - do something with R2


    B       while_start     ; repeat
while_end
```

Listing 3.7: While loop with string processing example

3.1.3 Procedures and Stack

3.1.3.1 ARM Architecture Procedure Call Standard (AAPCS)

Procedures are reusable blocks of code that encapsulate a specific task. They promote modular design, code reuse, and clearer program structure. In ARM assembly, procedures are implemented using branch-and-link instructions along with register usage conventions defined by the ARM Architecture Procedure Call Standard (AAPCS).

The AAPCS is the set of rules that define how functions exchange data and how registers must be preserved during a procedure call:

- **R0–R3**: Hold the first four parameters. `R0` also holds the return value. Caller-saved.
- **Stack**: Any additional parameters beyond the first four are passed on the stack.
- **R4–R11**: Must be preserved by the callee. If a procedure uses them, it must save and restore them.
- **SP (R13)**: Stack pointer, always points to the current top of the stack.
- **LR (R14)**: Link register holds the return address. Caller-saved.

Note: Callees are the procedures being called, while callers are the ones calling the procedure.

3.1.3.2 Procedure Templates

A procedure is entered with a `BL` (branch-with-link) instruction, which stores the return address in the link register `LR`. The callee returns by branching to `LR` (e.g., `BX LR`). By the AAPCS, the first four arguments are passed in `R0-R3` and the primary return value is placed in `R0`.

Basic Procedure Template

Example: simple procedure that expects two integers in `R0` and `R1` and returns their sum in `R0`.

```
AddTwo  PROC
        ADD R0, R0, R1    ; return R0+R1 in R0
        BX  LR
        ENDP
```

Listing 3.8: Basic procedure structure

Note: The `PROC` and `ENDP` directives help define the start and end of a procedure and they could be safely omitted.

Procedure with Preserved Registers When a procedure uses callee-saved registers (R4-R11), it must preserve their original values by saving them on the stack and restoring them before returning:

```
ProcessArray PROC
        PUSH    {R4-R6, LR}     ; Save used registers and LR
        MOV     R4, R0          ; Save array pointer
        MOV     R5, R1          ; Save array length
        MOV     R6, #0          ; Initialize counter

        ; Process array using R4, R5, R6...

        MOV     R0, R6          ; Return counter value
        POP     {R4-R6, PC}     ; Restore registers and return
        ENDP
```

Listing 3.9: Procedure using preserved registers

Nested Procedure Calls When one procedure calls another, the link register (`LR`) must be preserved, otherwise the return address would be lost. This is done by pushing `LR` onto the stack before making another call.

```
OuterProc
        PUSH    {LR}            ; Save return address
        BL      InnerProc       ; Call inner procedure
        MOV     R1, R0          ; Use return value
        POP     {PC}            ; Return to caller

InnerProc
        MOV     R0, #42         ; Return value
        BX      LR              ; Return
```

Listing 3.10: Nested procedure example

3.1.3.3 Stack Model (Full, Descending)

There are four common stack models based on two characteristics: whether the stack is full or empty, and whether it grows up (ascending) or down (descending).

- **Full Ascending**: SP points to the last used location; stack grows toward higher addresses.
- **Empty Ascending**: SP points to the next free location; stack grows toward higher addresses.
- **Empty Descending**: SP points to the next free location; stack grows toward lower addresses.

- **Full Descending**: SP points to the last used location; stack grows toward lower addresses.

The ARM Cortex-M4 uses a **full descending stack**, meaning:

- The stack pointer (`SP`) points to the last used location (full).
- The stack grows toward lower memory addresses (descending).

3.1.3.4 Stack Operations

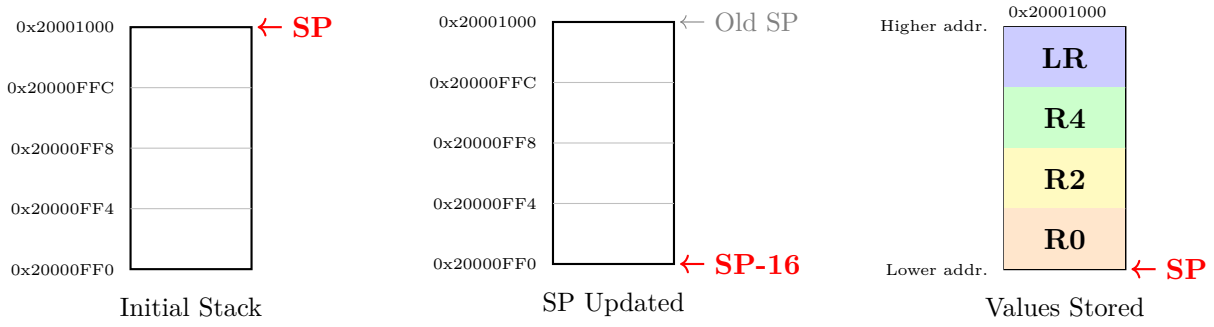
The Cortex-M4 provides `PUSH` and `POP` instructions that automatically update the stack pointer (`SP`) and allow saving or restoring multiple registers in a single instruction. These operations are essential for implementing procedure calls, local variables, and context switching.

PUSH The `PUSH` instruction saves one or more registers onto the stack in a single operation. When executed, the stack pointer (`SP`) is decremented to reserve space, and the specified registers are written to consecutive memory locations starting from the new `SP` value. **Rule:** `PUSH` stores registers on the stack, with the **lowest-numbered register** placed at the **lowest memory address** and the **highest-numbered register** placed at the **highest memory address**. This guarantees a consistent and predictable memory layout for saved contexts.

Example:

```
PUSH {R4, R0, R2, LR}
```

Even if the list appears unordered, values are laid out by register number: R0 (lowest address), then R2, R4, and LR (highest address). Each register occupies 4 bytes, so `SP` decreases by 16 bytes in total. The program counter (`PC`) cannot be pushed by this instruction.



(a) Initial empty stack (b) Stack pointer decremented (c) Registers stored in order

Figure 3.1: Stack operation sequence for `PUSH {R4, R0, R2, LR}`

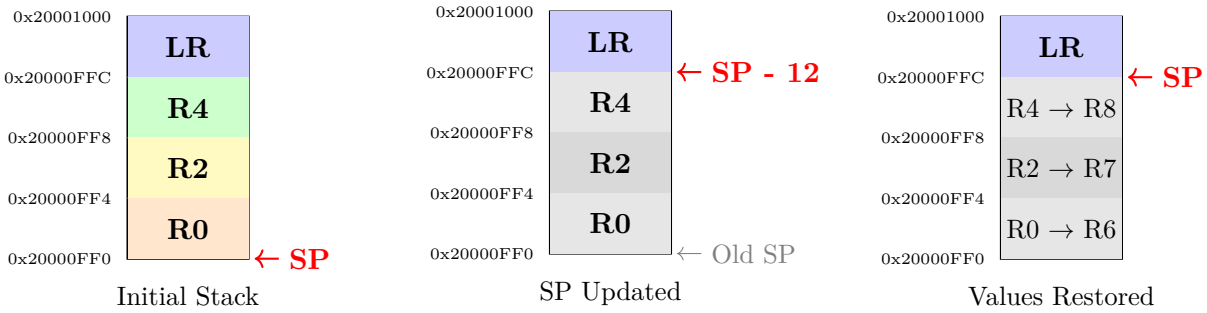
Explanation: Figure 3.1**(a)** shows the empty stack with `SP` at the top (highest address). When `PUSH` executes, the processor **decrements `SP` by 16 bytes** to reserve space for four registers (Figure 3.1**(b)**). Finally, in Figure 3.1**(c)**, values are stored according to the rule: the lowest-numbered register (R0) at the lowest address, then R2 and R4, and the highest-numbered (LR) at the highest address. Because the stack grows downward, `SP` points to the last written word after the operation.

POP The `POP` instruction restores one or more registers from the stack. Conceptually, the processor reads each value from the addresses currently covered by `SP` and then releases that stack space. **Rule:** `POP` loads registers from the stack such that the **lowest-numbered register** is restored from the **lowest memory address**, and the **highest-numbered register** from the **highest memory address**. After all specified registers are restored, `SP` has increased by 4 bytes per register.

Example:

```
POP {R6-R8}
```

This restores R6, then R7, then R8 from successively higher addresses; the stack pointer increases by 12 bytes overall. If `PC` is included in the list, the loaded value becomes the new program counter, returning from the current procedure.



(a) Initial stack after Figure 3.1 (b) Stack pointer incremented (c) Registers restored in order

Figure 3.2: Stack operation sequence for `POP {R6-R8}`

Explanation: In Figure 3.2**(a)**, the stack contains values saved earlier. Executing `POP {R6-R8}` **reclaims 12 bytes of stack space** (three registers) by moving `SP` upward in memory (Figure 3.2**(b)**). Then, as shown in Figure 3.2**(c)**, registers are restored following the rule: the lowest-numbered register (R6) comes from the lowest address, followed by R7 and R8 from higher addresses. After the last load, `SP` points to the top of the reclaimed block, completing the reversal of the earlier push.

3.2 Procedure

3.2.1 Examples

3.2.1.1 Example 1: Array Example — Find Maximum Element

This example demonstrates how to find the maximum element in an array using a standard for loop structure.

```
        AREA    RESET, CODE, READONLY
        EXPORT  __Vectors
__Vectors
        DCD     0x20001000          ; Initial SP
        DCD     Reset_Handler       ; Reset vector
        ALIGN

        AREA    MYCODE, CODE, READONLY
        ENTRY
        EXPORT  Reset_Handler

; Find maximum element in ARR[0..LEN-1]
; Result (max) is written to MAXRES.
Reset_Handler
        LDR     R0, =ARR            ; R0 = &ARR[0]
        MOV     R1, #0              ; R1 = i (index)
        LDR     R2, [R0]            ; R2 = max = ARR[0]
for_start
        CMP     R1, #LEN            ; i >= LEN ?
        BGE     for_end             ; yes -> done

        LDR     R3, [R0, R1, LSL #2]; R3 = ARR[i]
        CMP     R3, R2              ; if ARR[i] > max
        MOVGT   R2, R3              ;    max = ARR[i]
        ADD     R1, R1, #1          ; i++
        B       for_start
for_end
        LDR     R4, =MAXRES         ; store result for easy checking
        STR     R2, [R4]
STOP    B       STOP

        AREA    CONSTANTS, DATA, READONLY
ARR     DCD     10, 20, 30, -5, 11, 0
LEN     EQU     6

        AREA    MYDATA, DATA, READWRITE
MAXRES  DCD     0                   ; expect 30

        END
```

Listing 3.11: Find maximum element in an array

Check: Verify that the maximum element is correctly identified and stored in `MAXRES`.

3.2.1.2 Example 2: String Example — Count Uppercase Letters

This example demonstrates how to process a null-terminated string and count the number of uppercase letters (A-Z) using a while-loop structure.

```
        AREA    RESET, CODE, READONLY
        EXPORT  __Vectors
__Vectors
        DCD     0x20001000          ; Initial SP
        DCD     Reset_Handler       ; Reset vector
        ALIGN

        AREA    MYCODE, CODE, READONLY
        ENTRY
        EXPORT  Reset_Handler

; Count uppercase ASCII letters in the null-terminated string MYSTR.
; Result (count) is written to UPPERCOUNT.
Reset_Handler
        LDR     R0, =MYSTR          ; R0 = ptr to string
        MOV     R1, #0              ; R1 = count

while_next
        LDRB    R2, [R0], #1        ; R2 = *p++; post-increment pointer
        CBZ     R2, while_end       ; if '\0' -> exit

        CMP     R2, #'A'            ; below 'A'?
        BLT     while_next
        CMP     R2, #'Z'            ; above 'Z'?
        BGT     while_next

        ADD     R1, R1, #1          ; count++

        B       while_next

while_end
        LDR     R3, =UPPERCOUNT
        STR     R1, [R3]

STOP    B       STOP

        AREA    CONSTANTS, DATA, READONLY
MYSTR   DCB     "Hello ARM World!", 0

        AREA    MYDATA, DATA, READWRITE
UPPERCOUNT DCD  0                   ; expect 5 ('H','A','R','M','W')

        END
```

Listing 3.12: Count uppercase letters in a string

Check: Verify that the program correctly counts the uppercase letters and stores the result in `UPPERCOUNT`.

3.2.1.3 Example 3: Stack Example — Nested Uppercase Counter

This example demonstrates a nested call: `CountUpperNested(ptr)` scans a null-terminated string and calls `IsUpper(ch)` for each character. It shows saving/restoring `LR` and using a callee-saved register (`R4`) for the running count.

```
        AREA    RESET, CODE, READONLY
        EXPORT  __Vectors
__Vectors
        DCD     0x20001000          ; Initial SP
        DCD     Reset_Handler       ; Reset vector
        ALIGN
        AREA    MYCODE, CODE, READONLY
        ENTRY
        EXPORT  Reset_Handler
IsUpper                             ; IsUpper(R0 = ch) -> R0 = 1 if 'A'..'Z', else 0
        CMP     R0, #'A'
        BLT     not_upper
        CMP     R0, #'Z'
        BGT     not_upper
        MOV     R0, #1
        BX      LR
not_upper
        MOV     R0, #0
        BX      LR
CountUpperNested                    ; CountUpperNested(R0 = ptr) -> R0 = Upper Count
        PUSH    {R4, LR}            ; save callee-saved + return address
        MOV     R1, R0              ; R1 = ptr (keep pointer here)
        MOV     R4, #0              ; R4 = count
cu_next
        LDRB    R0, [R1], #1        ; R0 = *ptr++; post-increment pointer in R1
        CBZ     R0, cu_done         ; if null terminator, finish
        BL      IsUpper             ; R0 = 0/1 based on 'A'..'Z'
        ADD     R4, R4, R0          ; count += result
        B       cu_next
cu_done
        MOV     R0, R4              ; return count in R0
        POP     {R4, PC}
Reset_Handler
        LDR     R0, =mystring
        BL      CountUpperNested
        LDR     R2, =UPPERCOUNT
        STR     R0, [R2]
STOP    B       STOP
        AREA    CONSTANTS, DATA, READONLY
mystring DCB    "Hello ARM World!", 0  ; Uppercase: H, A, R, M, W -> 5
        AREA    MYDATA, DATA, READWRITE
UPPERCOUNT  DCD 0                   ; should be 5
        END
```

Listing 3.13: Nested procedure call to count uppercase letters

Check: Verify that `UPPERCOUNT` contains `5` for the test string.

3.2.2 Tasks

3.2.2.1 Task 1: Count Vowels in a String

Implement procedures to process strings with the following requirements:

- Create a procedure `CountVowels` that takes a string pointer in R0 and returns the number of vowels (a, e, i, o, u) in R0.
- Use nested procedure calls where `CountVowels` calls a helper procedure `IsVowel`.
- Follow AAPCS conventions for parameter passing and register usage.

3.2.2.2 Task 2: Factorial Calculation (Iterative)

Implement a procedure to calculate the factorial of a non-negative integer:

- Create a procedure `Factorial` that takes a non-negative integer in R0 and returns its factorial in R0.
- Use an iterative approach with a loop to compute the factorial.
- Ensure proper handling of edge cases, such as $0! = 1$.
- Follow AAPCS conventions for parameter passing and register usage.

3.2.2.3 Task 3: Factorial Calculation (Recursive)

Implement a recursive version of the factorial calculation:

- Create a procedure `FactorialRec` that takes a non-negative integer in R0 and returns its factorial in R0.
- Use recursion to compute the factorial, ensuring proper base case handling.
- Manage the stack appropriately to save and restore registers as needed.
- Follow AAPCS conventions for parameter passing and register usage.
- Test the procedure with various inputs to verify correctness.

4 Microcontroller Architecture and GPIO Output

Learning Objectives

After completing this experiment, you will be able to:

- Identify the main components and subsystems of the TM4C123 microcontroller and understand their roles within an embedded system.
- Recognize the organization of the ARM Cortex-M4 core, memory map, and peripheral buses.
- Understand the role of core peripherals including NVIC, SysTick, and System Control Block (SCB).
- Configure and control digital output pins using the General-Purpose Input/Output (GPIO) module.
- Write Assembly programs to drive on-board LEDs through direct register access.
- Relate register operations to physical hardware behavior and verify functionality through observation on the LaunchPad board.

Experiment Overview

This experiment introduces the TM4C123 microcontroller architecture and digital I/O interfacing through GPIO ports. You will explore the internal structure including the ARM Cortex-M4 core, memory regions, and peripheral interconnect, then write Assembly programs to control the on-board LEDs connected to PORTF. By the end of this lab, you will understand how peripheral registers are memory-mapped, be able to initialize and configure GPIO ports, and control external hardware through register manipulation—forming the foundation for subsequent experiments with interrupts, timers, and analog peripherals.

4.1 Theoretical Background

4.1.1 TM4C123 Microcontroller Architecture

The TM4C123GH6PM microcontroller is built around the ARM Cortex-M4F processor core, implementing the ARMv7-M architecture. It integrates the CPU core, memory, peripherals, and I/O interfaces on a single chip — a complete system-on-chip (SoC) solution for embedded applications.

4.1.1.1 Core Components

The TM4C123 contains the following major components:

- **ARM Cortex-M4F Processor Core**: 32-bit RISC processor with hardware Floating-Point Unit (FPU)
- **Memory**: 256 KB Flash, 32 KB SRAM, 2 KB EEPROM
- **System Buses**: Advanced High-Performance Bus (AHB) and Advanced Peripheral Bus (APB)
- **Core Peripherals**: NVIC, SysTick Timer, Memory Protection Unit (MPU), System Control Block (SCB)
- **General-Purpose Timers**: Six 16/32-bit and six 32/64-bit timers
- **Communication Interfaces**: UART, I^2C, SSI (SPI), CAN
- **Analog Peripherals**: 12-bit ADC with 12 channels, analog comparators
- **GPIO Ports**: Six ports (A-F) with up to 43 programmable pins

4.1.2 ARM Cortex-M4 Core Peripherals

The ARM Cortex-M4 processor includes several **core peripherals** that are common across all Cortex-M4-based microcontrollers. They are tightly integrated with the CPU and provide essential system functions such as interrupt handling and timing.

4.1.2.1 Nested Vectored Interrupt Controller (NVIC)

The NVIC manages all interrupt and exception handling. It supports up to 240 interrupt sources (138 on the TM4C123), hardware priority levels, and automatic context saving for efficient servicing. Key features include nesting, tail-chaining, and late-arrival handling for minimal interrupt latency.
Important NVIC registers (base address `0xE000E100`):

- `NVIC_ENx` — Set-Enable
- `NVIC_DISx` — Clear-Enable
- `NVIC_PRIx` — Priority Configuration
- `NVIC_ACTIVEx` — Active Status

4.1.2.2 SysTick Timer

The SysTick timer is a simple 24-bit down-counter integrated in the core. It provides a consistent time base for system delays or periodic tasks. Many operating systems use it for time-keeping or scheduling.
SysTick registers (base address `0xE000E010`):

- `STCTRL` — Control and Status
- `STRELOAD` — Reload Value
- `STCURRENT` — Current Counter Value

Other core peripherals such as the System Control Block (SCB) are present but not covered in this course.

4.1.3 Memory Map

The Cortex-M4 processor uses a unified 4 GB address space for all code, data, and peripherals. Every peripheral and memory region occupies a unique address range, allowing direct access through normal load and store instructions.

Region	Address Range	Description
Flash Memory	0x00000000 - 0x0003FFFF	Program storage (256 KB)
SRAM	0x20000000 - 0x20007FFF	On-chip data memory (32 KB)
Peripherals	0x40000000 - 0x400FFFFF	Peripheral registers
GPIO Ports	0x40004000 - 0x40025FFF	GPIO A-F registers
Core Peripherals	0xE0000000 - 0xE00FFFFF	NVIC, SysTick, SCB, etc.

Table 4.1: Simplified TM4C123 Memory Map

Each peripheral's registers are **memory-mapped**, meaning they are accessed just like variables in memory. For example, writing to address `0x400253FC` directly updates the GPIO Port F data register.

4.1.4 General-Purpose Input/Output (GPIO)

GPIO (General-Purpose Input/Output) ports form the primary interface between the microcontroller and external devices such as LEDs, switches, and sensors. Each pin can be configured as either an input or an output.

When a pin is an **output**, software drives it high or low to control external hardware (e.g., turn an LED on/off). When a pin is an **input**, software reads its logic level (e.g., a button press).

4.1.4.1 GPIO Ports Overview

The TM4C123GH6PM provides six GPIO ports (A-F), each with up to eight programmable pins. Not all pins are available on the LaunchPad, and some are reserved for debugging or special functions.

Port	Pins Available	Notes
Port A	PA0-PA7	UART0 (PA0, PA1) shared with USB debug interface
Port B	PB0-PB7	I^2C0, SSI2, ADC channels
Port C	PC0-PC7	PC0-PC3 used for JTAG (avoid modification)
Port D	PD0-PD7	PD7 requires unlock for GPIO use
Port E	PE0-PE5	ADC and UART5 functionality available
Port F	PF0-PF4	On-board LEDs (PF1-PF3) and switches (PF0, PF4)

Table 4.2: GPIO Port Summary — TM4C123GH6PM

Each port exposes its own control and data registers, allowing independent configuration and operation.

4.1.4.2 Memory-Mapped GPIO Registers

GPIO modules are accessed via **memory-mapped registers**: fixed addresses that the CPU reads/writes with standard `LDR`/`STR` instructions.

Every port has a **base address**; key registers sit at fixed offsets from that base.

Port	Base Address
Port A	0x40004000
Port B	0x40005000
Port C	0x40006000
Port D	0x40007000
Port E	0x40024000
Port F	0x40025000

Table 4.3: GPIO Port Base Addresses (TM4C123GH6PM)

In this experiment we use Port F, so:

`GPIODIR` = `0x40025400`, `GPIODEN` = `0x4002551C`, `GPIODATA` = `0x400253FC`.

4.1.4.3 Address Masking in `GPIODATA`

The `GPIODATA` register supports **address masking**: address bits [9:2] form a bit mask that selects which pins are affected.

- **Full access:** Base + `0x3FC` — all 8 pins.
- **Masked access:** Base + (`mask` $\ll$ 2) — only the pins in `mask`.

Example (PF1 only):

`0x40025000 + (0x02 << 2) = 0x40025008`.

This is **address masking** (often confused with "bit-banding"); it enables fast, selective reads/writes to individual pins.

4.1.4.4 Protected and Special-Function Pins

Some pins are locked or reserved at reset due to critical roles:

- **PF0** doubles as the Non-Maskable Interrupt (NMI) input and is locked. To use it as GPIO, write `0x4C4F434B` to `GPIO_LOCK` and set the desired bits in `GPIO_CR`.
- **PC0-PC3** carry the JTAG debug interface and should not be reassigned in normal operation.

Always confirm pin availability in the datasheet before repurposing special-function pins.

4.1.4.5 Port F on the LaunchPad

Port F connects to the on-board RGB LEDs and user push buttons:

Pin	Function	Connected Component	Active Level
PF1	Output	Red LED	Active High
PF2	Output	Blue LED	Active High
PF3	Output	Green LED	Active High
PF0	Input (locked)	SW2 (Push Button)	Active Low (enable pull-up)
PF4	Input	SW1 (Push Button)	Active Low (enable pull-up)

Table 4.4: GPIO Port F Pin Assignments — TM4C123 LaunchPad

LEDs are **active-high** (write 1 to turn on). Push buttons pull to ground when pressed, so inputs read **0 when pressed** and require internal pull-ups.

4.1.4.6 GPIO Configuration Workflow

In practice, GPIO usage has two phases:

- **Initialization (setup, runs once):** enable the port clock, unlock protected pins (if needed), set direction, and enable digital function.
- **Runtime (operation, repeats):** read inputs or write outputs by accessing `GPIODATA` in the main loop or in interrupt service routines (ISRs).

A typical structure is:

```
InitGPIO();                // RCGCGPIO, optional LOCK/CR, DIR, DEN
while (1) {                // Runtime phase
    // Read buttons (inputs) and/or drive LEDs (outputs)
    // Optionally, some I/O is handled inside ISRs
}
```

Listing 4.1: Typical GPIO program structure (setup vs. runtime)

The detailed register-level steps (clock enable, unlock, direction, digital enable, and data access) are demonstrated in the following subsections.

Step 1 — Enable the GPIO Port Clock

Register: `RCGCGPIO` — Run Mode Clock Gating Control for GPIO (`0x400FE608`)
Before accessing any GPIO port, its clock must be enabled. The `RCGCGPIO` register controls the clock to all GPIO modules. Each bit corresponds to a single port (A-F). Writing a '1' to a bit enables the clock to that port, while '0' disables it.

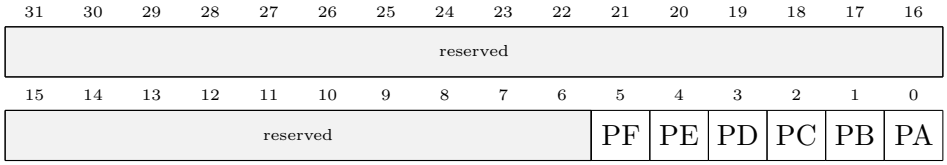


Figure 4.1: RCGCGPIO Register (`0x400FE608`) — GPIO Clock Control

To activate Port F, bit 5 must be set to 1 as shown below.

```
LDR     R1, =0x400FE608     ; RCGCGPIO register
LDR     R0, [R1]
ORR     R0, R0, #0x20       ; Set bit 5 (Port F)
STR     R0, [R1]
```

Listing 4.2: Enable clock for Port F

Once the clock is enabled, a short delay or status polling ensures the peripheral is ready before further configuration.

Step 2 — Unlock Protected Pins

Registers: `GPIO_LOCK` (`0x40025520`), `GPIO_CR` (`0x40025524`)
Certain pins such as PF0 are protected because they share critical alternate functions (e.g., the NMI input). To modify these pins, the port must be unlocked by writing the key value `0x4C4F434B` ("LOCK") into the `GPIO_LOCK` register. The `GPIO_CR` register (Commit Register) then determines which pins can be altered.

Figure 4.2: GPIO_CR Register — Commit Control

The following code unlocks Port F and enables modification of all pins.

```
LDR     R1, =0x40025520     ; GPIO_LOCK
LDR     R0, =0x4C4F434B     ; Unlock key
STR     R0, [R1]
LDR     R1, =0x40025524     ; GPIO_CR
MOV     R0, #0xFF           ; Allow changes to all pins
STR     R0, [R1]
```

Listing 4.3: Unlock Port F

In this experiment, PF0 is not used, so unlocking is optional. After unlocking, configuration registers such as direction and digital enable can be safely modified.

Step 3 — Configure Pin Direction

Register: `GPIODIR` (`0x40025400`) — GPIO Direction Control (Port F)
The `GPIODIR` register determines whether each GPIO pin functions as an input or an output. Writing '0' configures the pin as an input, while '1' configures it as an output.

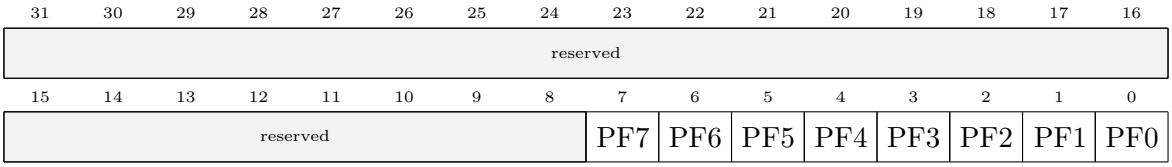


Figure 4.3: GPIODIR Register (Port F) — Direction Control

For the on-board RGB LED, PF1-PF3 must be configured as outputs. Bits 1-3 are therefore set to '1'.

```
LDR     R1, =0x40025400     ; GPIODIR register (Port F)
LDR     R0, [R1]
ORR     R0, R0, #0x0E       ; Set PF1, PF2, PF3 as outputs
STR     R0, [R1]
```

Listing 4.4: Set PF1, PF2, PF3 as outputs

Unmodified bits remain unchanged, allowing input pins (such as PF0 or PF4) to retain their default configuration.

Step 4 — Enable Digital Functionality

Register: `GPIODEN` (`0x4002551C`) — Digital Enable Register (Port F)
Each GPIO pin can serve analog or digital functions. The `GPIODEN` register enables the digital circuitry for selected pins. Pins configured as digital inputs or outputs must have their corresponding bits set to '1'.

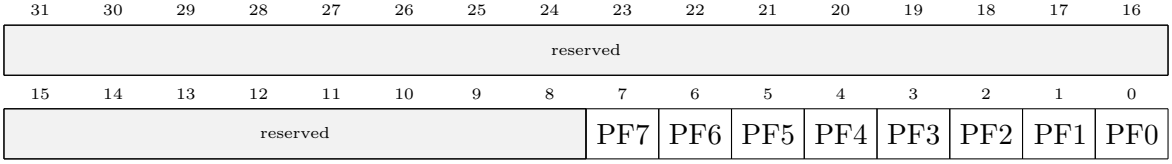


Figure 4.4: GPIODEN Register (Port F) — Digital Enable

Bits 1-3 are set to enable PF1-PF3 as digital outputs for the LED.

```
LDR     R1, =0x4002551C     ; GPIODEN register (Port F)
LDR     R0, [R1]
ORR     R0, R0, #0x0E       ; Enable PF1-PF3 as digital pins
STR     R0, [R1]
```

Listing 4.5: Enable digital function for PF1-PF3

Failing to enable `GPIODEN` leaves the pins electrically inactive even if their direction is set.

Step 5 — Write to the Data Register

Register: `GPIODATA` (`0x400253FC`) — Data Input/Output Register (Port F)
The `GPIODATA` register reflects the current logic levels on all GPIO pins. Writing to a bit drives the corresponding output high ('1') or low ('0'). Bits 1-3 correspond to the RGB LED pins on the LaunchPad: PF1 = Red, PF2 = Blue, PF3 = Green.

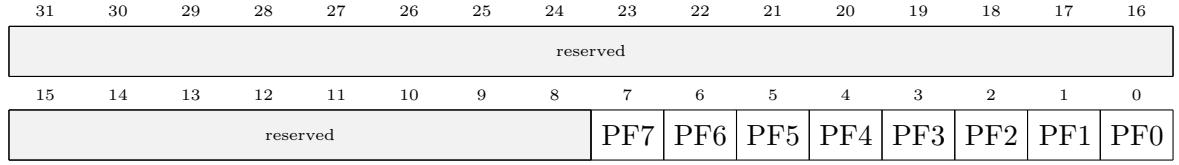


Figure 4.5: GPIODATA Register (Port F) — Data Output/Input

The example below drives PF1 (Red) and PF3 (Green) simultaneously to produce a yellow color.

```
LDR     R1, =0x400253FC     ; GPIODATA register (Port F)
MOV     R0, #0x0A           ; Set PF1 (Red) and PF3 (Green) = Yellow
STR     R0, [R1]
```

Listing 4.6: Turn on Yellow LED (Red + Green = PF1 + PF3)

By writing different bit combinations, various LED colors can be generated:

Color	Red (PF1)	Blue (PF2)	Green (PF3)	Hex	Combination
Red	●	○	○	0x02	Red only
Blue	○	●	○	0x04	Blue only
Green	○	○	●	0x08	Green only
Yellow	●	○	●	0x0A	Red + Green
Cyan	○	●	●	0x0C	Blue + Green
Magenta	●	●	○	0x06	Red + Blue
White	●	●	●	0x0E	All on

Table 4.5: LED Color Combinations on TM4C123 LaunchPad (PF1-PF3)

4.2 Procedure

4.2.1 Examples

The following examples illustrate the complete process of initializing Port F and controlling the on-board LEDs using Assembly language.

4.2.1.1 Example 1 — Simple LED Blink

This example initializes Port F and toggles the Red LED (PF1) on and off using address masking with a software delay. Address Masking for PF1:

```
0x40025000 + (0x02 « 2) = 0x40025008
```

```
        AREA    RESET, CODE, READONLY    ; Code section
        THUMB                            ; Use Thumb instruction set
        EXPORT  __main                   ; Export symbol

__main  PROC
        LDR     R1, =0x400FE608          ; RCGCGPIO address
        LDR     R0, [R1]                 ; Read current value
        ORR     R0, R0, #0x20            ; Enable Port F clock
        STR     R0, [R1]                 ; Write back
        NOP                              ; Small delay
        NOP

        LDR     R1, =0x40025400          ; GPIODIR address
        LDR     R0, [R1]                 ; Read current value
        ORR     R0, R0, #0x02            ; Set PF1 as output
        STR     R0, [R1]                 ; Write back

        LDR     R1, =0x4002551C          ; GPIODEN address
        LDR     R0, [R1]                 ; Read current value
        ORR     R0, R0, #0x02            ; Enable digital function for PF1
        STR     R0, [R1]                 ; Write back

loop    LDR     R1, =0x40025008          ; Masked DATA address for PF1
        MOV     R0, #0x02                ; Turn ON red LED (PF1)
        STR     R0, [R1]                 ; Write to GPIODATA

        LDR     R0, =1000000             ; Load delay count
        BL      DELAY                    ; Call delay

        MOV     R0, #0x00                ; Turn OFF red LED (PF1)
        STR     R0, [R1]                 ; Write to GPIODATA

        LDR     R0, =1000000             ; Load delay count
        BL      DELAY                    ; Call delay

        B       loop                     ; Repeat
        ENDP

DELAY   PROC
        SUBS    R0, R0, #1               ; Decrement counter
        BNE     DELAY                    ; Loop until zero
        BX      LR                       ; Return
        ENDP
        END
```

Listing 4.7: Assembly code to blink the Red LED (PF1) on the TM4C123 LaunchPad

4.2.1.2 Example 2 — Cycle Through RGB Colors

This example extends the previous one by cycling through RGB LED colors (Red, Green, Blue) using a software delay. Address Masking for PF1, PF2, PF3:

```
0x40025000 + ((0x02 | 0x04 | 0x08) « 2) = 0x40025038
```

```
        AREA    RESET, CODE, READONLY    ; Code section
        THUMB                            ; Use Thumb instruction set
        EXPORT  __main                   ; Export symbol
__main  PROC
        BL      PF_Init                  ; Initialize Port F
loop
        LDR     R1, =0x40025038          ; Masked DATA address for PF1-PF3

        MOV     R0, #0x02                ; Turn ON red LED (PF1)
        STR     R0, [R1]                 ; Write to GPIODATA
        LDR     R0, =10000000            ; Load delay count
        BL      DELAY                    ; Call delay

        MOV     R0, #0x04                ; Turn ON blue LED (PF2)
        STR     R0, [R1]
        LDR     R0, =10000000
        BL      DELAY

        MOV     R0, #0x08                ; Turn ON green LED (PF3)
        STR     R0, [R1]
        LDR     R0, =10000000
        BL      DELAY
        B       loop                     ; Repeat
        ENDP

DELAY   PROC
        SUBS    R0, R0, #1               ; Decrement counter
        BNE     DELAY                    ; Loop until zero
        BX      LR                       ; Return
        ENDP

PF_Init PROC
        LDR     R1, =0x400FE608          ; RCGCGPIO address
        LDR     R0, [R1]                 ; Read current value
        ORR     R0, R0, #0x20            ; Enable Port F clock
        STR     R0, [R1]
        NOP                              ; Small delay
        NOP
        LDR     R1, =0x40025400          ; GPIODIR address
        LDR     R0, [R1]
        ORR     R0, R0, #0x0E            ; Set PF1-PF3 as outputs
        STR     R0, [R1]

        LDR     R1, =0x4002551C          ; GPIODEN address
        LDR     R0, [R1]
        ORR     R0, R0, #0x0E            ; Enable digital function for PF1-PF3
        STR     R0, [R1]
        BX      LR                       ; Return
        ENDP
        END
```

Listing 4.8: Assembly code to cycle through RGB colors on the TM4C123 LaunchPad

4.2.2 Tasks

4.2.2.1 Task 1 — Adjust the Blink Rate

Modify the delay routine in **Example 1** to change the blinking speed of the Red LED. Experiment with different delay values until the LED blinks at approximately **1 Hz** (about one second ON and one second OFF).

4.2.2.2 Task 2 — Cycle Through Multiple Colors

Expand **Example 2** to include additional colors by combining the Red, Green, and Blue LEDs. Create a program that automatically cycles through all color combinations listed in Table 4.5.

Hint: Use a loop to step through the color sequence repeatedly instead of writing separate code for each color.

5 GPIO Inputs and Interrupt Handling

Learning Objectives

After completing this experiment, you will be able to:

- Configure GPIO inputs with digital enable and internal pull resistors.
- Read switches via polling with software debouncing.
- Configure edge-triggered GPIO interrupts and write minimal ISRs.
- Enable and manage interrupts in the NVIC (IRQ mapping, ISER).
- Unlock and configure protected pins (PF0/NMI) for general-purpose use.

Experiment Overview

This experiment extends GPIO functionality to inputs and introduces interrupt-driven programming. You will configure GPIO pins to read mechanical switches using both polling and interrupt approaches, implement software debouncing techniques, write interrupt service routines, and enable GPIO interrupts in the NVIC. By the end of this lab, you will understand how to implement both polling and interrupt-driven input handling, and write responsive embedded applications that react to external events in real time.

5.1 Theoretical Background

5.1.1 GPIO Input Configuration

In the previous experiment, GPIO pins were configured as outputs to control LEDs. In this experiment, we configure GPIO pins as **inputs** to read the state of external devices such as switches, buttons, and sensors.

When a GPIO pin is configured as an input, the microcontroller reads the voltage level on the pin and interprets it as a logic '0' (low, typically 0V) or logic '1' (high, typically 3.3V). The pin's digital input buffer must be enabled, and the pin must be connected to a defined voltage level to avoid floating states.

5.1.1.1 Input Pin Requirements

For reliable digital input operation, three conditions must be met:

- **Direction**: The pin must be configured as an input (bit cleared in `GPIODIR`).
- **Digital Enable**: The digital input buffer must be enabled (bit set in `GPIODEN`).
- **Defined Logic Level**: The pin must be connected to a valid logic level (not floating).

If a pin is left floating (not connected to a defined voltage), it can pick up electrical noise and produce random or unstable readings. To prevent this, pins are typically connected to either power (`VCC`) or ground (`GND`) through a resistor, or the microcontroller's internal pull-up or pull-down resistors can be enabled.

5.1.1.2 Pull-Up and Pull-Down Resistors

Digital inputs must not be left floating: an undefined voltage can produce random logic reads. A pull resistor defines the idle (no-press) level of a switch input.

Pull-Up (active-low) A **pull-up** connects the pin weakly to V_{CC}, so the idle level is logic 1. Pressing the switch drives the pin to GND $\rightarrow$ logic 0.

Pull-Down (active-high) A **pull-down** connects the pin weakly to GND, so the idle level is logic 0. Pressing the switch drives the pin to V_{CC} $\rightarrow$ logic 1.

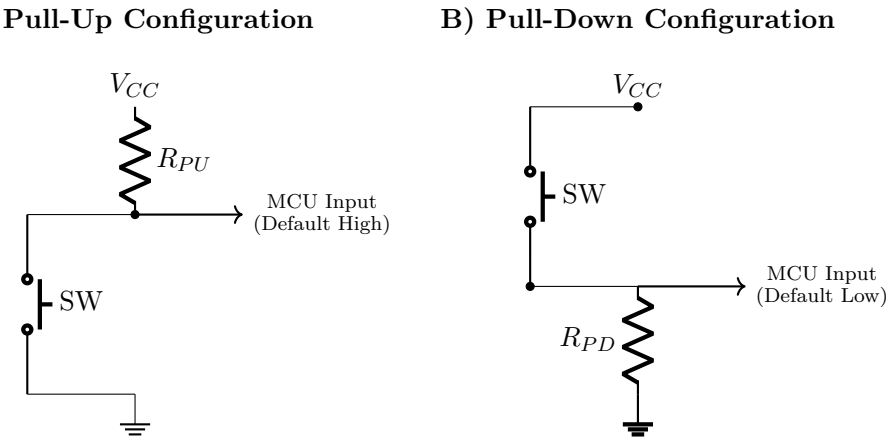


Figure 5.1: Pull-Up and Pull-Down Resistor Configurations for Switch Inputs

Internal Resistors (TM4C123) TM4C123 GPIO pins include configurable **internal** pull resistors:

- `GPIOPUR` — Pull-Up Select (enable bit = pull-up on that pin)
- `GPIOPDR` — Pull-Down Select (enable bit = pull-down on that pin)

Do not enable both on the same pin. On the LaunchPad, SW1 (PF4) and SW2 (PF0) short to GND when pressed, so enable **pull-ups** (`GPIOPUR`) to make them active-low (read 0 when pressed, 1 when released).

5.1.2 Switch Bouncing and Debouncing

When a mechanical switch or button is pressed or released, the metal contacts inside do not make or break contact cleanly. Instead, they **bounce** — rapidly making and breaking contact multiple times before settling into a stable state. This bouncing typically lasts 5-30 milliseconds.

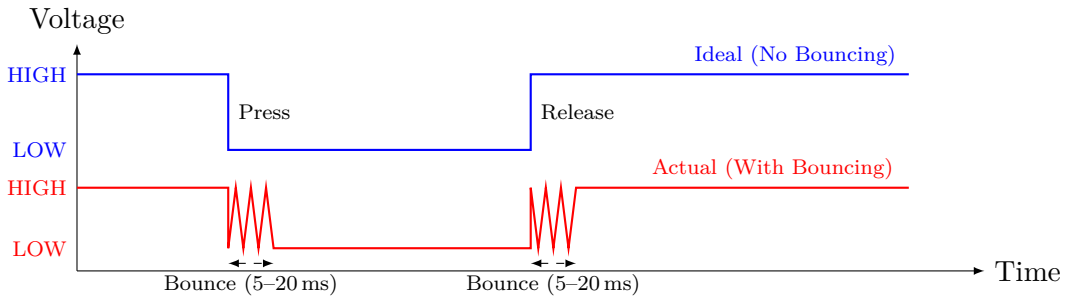


Figure 5.2: Mechanical Switch Bounce — Press and Release Both Exhibit Bouncing

For applications that poll the input at a slow rate or only care about the final state, bouncing may not be a problem. However, for interrupt-driven systems or applications that count button presses, bouncing can cause multiple false triggers.

5.1.2.1 Debouncing Techniques

Two approaches can eliminate or mitigate switch bouncing:

Hardware Debouncing Add an RC (resistor-capacitor) filter circuit or a dedicated debouncing IC to the switch. The capacitor smooths out the voltage transitions, preventing bounces from reaching the microcontroller input.

Software Debouncing After detecting a button press (or release), wait for a short period (typically 10-50 ms) to allow the bouncing to settle, then read the input again to confirm the state. This can be implemented with:

- **Blocking delay debouncing**: Insert a fixed delay (busy-wait loop) after detecting an edge, then verify the input state. This is simple but prevents the CPU from doing other work during the delay period.
- **Timer-based debouncing (non-blocking)**: Use a hardware timer to sample the input periodically without blocking the main program. Only register a press after multiple consistent readings across timer intervals.

For this experiment, we will implement simple delay-based debouncing in our polling examples. More sophisticated timer-based debouncing techniques will be covered in the following experiment.

5.1.3 Reading GPIO Inputs: Polling vs. Interrupts

There are two fundamental approaches to reading digital inputs:

5.1.3.1 Polling (Continuous Checking)

In polling, the CPU continuously reads the input pin in a loop and checks whether the state has changed. This is simple to implement but has drawbacks:

- **CPU Usage**: The CPU is busy checking the input even when nothing is happening.
- **Latency**: The response time depends on how frequently the loop runs.
- **Power Consumption**: The CPU cannot enter low-power sleep modes while polling.

Polling is suitable for simple applications where the CPU has nothing else to do or when inputs change slowly.

```
while (1) {
    if (button is pressed) {
        // Respond to button press
    }
}
```

Listing 5.1: Polling Example (Conceptual)

5.1.3.2 Interrupts (Event-Driven Response)

In interrupt-driven input, the GPIO peripheral notifies the CPU when an input change occurs by generating an **interrupt request** (IRQ). The CPU immediately stops its current task, executes an **Interrupt Service Routine** (ISR), and then resumes normal operation.

- **Efficiency**: The CPU can do other work or sleep until an input event occurs.
- **Responsiveness**: The CPU reacts immediately to input changes.
- **Power Efficiency**: The CPU can remain in low-power modes and wake only when needed.

Interrupts are essential for responsive embedded systems and are the preferred method for handling asynchronous events.

```
// Main loop can perform other tasks
while (1) {
    // Do other work or sleep
}

// ISR is called automatically when button is pressed
void ButtonISR(void) {
    // Respond to button press
    // Clear interrupt flag
}
```

Listing 5.2: Interrupt Example (Conceptual)

5.1.4 How Interrupts Work

Understanding the interrupt mechanism is essential for writing interrupt-driven programs. This section explains the complete lifecycle of an interrupt from trigger to completion.

5.1.4.1 Interrupt Lifecycle

When a GPIO pin configured for interrupts detects the specified event (e.g., a button press), the following sequence occurs:

1. Event Detection The GPIO peripheral continuously monitors the pin according to its configuration (edge/level detection). When the configured condition is met (e.g., falling edge), the peripheral sets an internal flag.

2. Interrupt Request (IRQ) If the interrupt is unmasked (enabled in `GPIOIM`), the GPIO module sends an interrupt request to the NVIC. The NVIC receives interrupt requests from all peripherals and manages their execution.

3. CPU Response The NVIC checks if the interrupt is enabled (`NVIC_ISER`) and compares its priority with currently executing code. If the interrupt should be serviced:

- The CPU finishes executing the current instruction.
- The CPU automatically saves the current execution context (program counter, registers, status flags) onto the stack.
- The CPU loads the address of the Interrupt Service Routine (ISR) from the vector table.
- Execution jumps to the ISR.

4. ISR Execution The ISR (e.g., `GPIOF_Handler()`) executes and must:

- Identify which pin(s) caused the interrupt (read `GPIOMIS`).
- Perform the required response (toggle LED, set flag, etc.).
- Clear the interrupt flag (write to `GPIOICR`) — **critical step**.

5. Return from Interrupt When the ISR completes (returns), the CPU:

- Automatically restores the saved context from the stack.
- Resumes execution from where it was interrupted.

This entire process happens in microseconds, making interrupts extremely responsive.

5.1.4.2 Why Clear the Interrupt Flag?

The interrupt flag remains set until explicitly cleared by software. If the ISR does not clear the flag by writing to `GPIOICR`, the NVIC will immediately re-trigger the interrupt as soon as the ISR returns, creating an infinite loop of interrupts that hangs the system.

```
void GPIOF_Handler(void) {
    // Read which pin caused interrupt
    if (GPIOF->MIS & (1 << 4)) {
        // Handle PF4 interrupt
        // ...

        GPIOF->ICR |= (1 << 4);  // MUST clear flag!
    }
}
```

Listing 5.3: Critical: Always clear the interrupt flag

5.1.4.3 Interrupt Advantages and Considerations

Advantages:

- **Efficient**: CPU can perform other tasks or sleep while waiting for events.
- **Responsive**: Immediate response to external events (microsecond latency).
- **Power-efficient**: CPU can remain in low-power modes between events.
- **Real-time**: Predictable timing for critical events.

Considerations:

- **Complexity**: More complex to implement and debug than polling.
- **Shared data**: ISRs and main code must carefully manage shared variables (use `volatile`).
- **ISR constraints**: ISRs should be short and fast — avoid delays and lengthy operations.
- **Debouncing**: Mechanical switch bouncing can trigger multiple interrupts; software or hardware debouncing is often necessary.

5.1.5 GPIO Input Configuration Registers

Before a GPIO pin can be used as an input, it must first be configured for digital functionality. As discussed in the previous experiment, the following three steps are required for **all GPIO pins**:

1. **Enable the GPIO clock** via `SYSCTL_RCGCGPIO`.
2. **Select the direction** of the pin using `GPIO_DIR` (`0` for input, `1` for output).
3. **Enable digital functionality** using `GPIO_DEN`.

Once a pin is set as an input, the logic level on the pin may be undefined if it is left floating (not driven by an external source). To ensure a stable logic level, internal resistors can be activated: These resistors are controlled by two dedicated registers:

- `GPIO_PUR` — Enable internal pull-up resistor on the selected pin(s).
- `GPIO_PDR` — Enable internal pull-down resistor on the selected pin(s).

GPIOPUR — Pull-Up Resistor Enable

Register: `GPIOPUR` — GPIO Pull-Up Select (Port F: `0x40025510`)
The `GPIOPUR` register enables the internal pull-up resistor for each GPIO pin. When enabled, the pin is weakly pulled to logic '1' (3.3V) through a resistor (typically 13-40 kω). This prevents floating inputs and is essential for switches connected to ground.

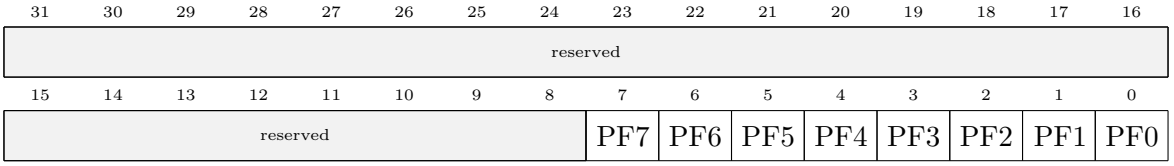


Figure 5.3: GPIOPUR Register — Pull-Up Enable

Bit Values:

- **0**: Pull-up disabled (default)
- **1**: Pull-up enabled

For the LaunchPad switches (SW1 on PF4, SW2 on PF0), pull-ups must be enabled.

GPIOPDR — Pull-Down Resistor Enable

Register: `GPIOPDR` — GPIO Pull-Down Select (Port F: `0x40025514`)
The `GPIOPDR` register enables the internal pull-down resistor for each GPIO pin. When enabled, the pin is weakly pulled to logic '0' (0V) through a resistor. This is used for switches connected to power (`VCC`).

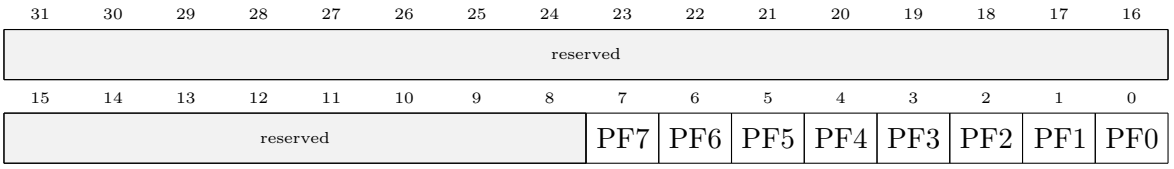


Figure 5.4: GPIOPDR Register — Pull-Down Enable

Bit Values:

- **0**: Pull-down disabled (default)
- **1**: Pull-down enabled

Note: Pull-up and pull-down resistors should not be enabled simultaneously on the same pin.

5.1.6 Unlocking Protected GPIO Pins

Some GPIO pins on the TM4C123 are **locked** by default to prevent accidental reconfiguration of critical functions such as JTAG/SWD (debug interface) and NMI (Non-Maskable Interrupt).

On Port F, pin **PF0** is locked because it is the NMI input by default. To use PF0 as a general-purpose GPIO input (for SW2 on the LaunchPad), it must be unlocked.

GPIOLOCK — Lock Register

Register: `GPIOLOCK` — GPIO Lock (Port F: `0x40025520`)
The `GPIOLOCK` register controls write access to the `GPIOCR` register. Writing the magic value `0x4C4F434B` ("LOCK" in ASCII) unlocks the port and allows modifications to the commit register.
Values:

- Write `0x4C4F434B`: Unlock the port (allow changes to `GPIOCR`)
- Write any other value: Lock the port (prevent changes to `GPIOCR`)
- Read: Returns `0x00000001` if locked, `0x00000000` if unlocked

GPIOCR — Commit Register

Register: `GPIOCR` — GPIO Commit (Port F: `0x40025524`)
The `GPIOCR` register controls which pins can be reconfigured. After unlocking with `GPIOLOCK`, set the corresponding bit in `GPIOCR` to allow changes to that pin's configuration registers.

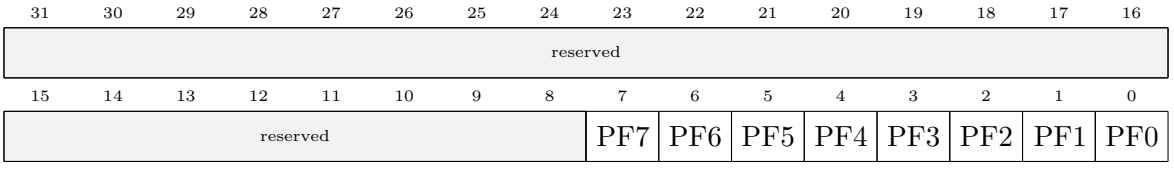


Figure 5.5: GPIOCR Register — Commit Control

Bit Values:

- **0**: Pin configuration is locked (cannot be modified)
- **1**: Pin configuration is unlocked (can be modified)

Unlocking Procedure

To unlock a protected pin (such as PF0):

1. Write the unlock key `0x4C4F434B` to the `GPIOLOCK` register.
2. Set the corresponding bit in the `GPIOCR` (Commit Register) to allow changes.
3. Configure the pin normally (`GPIODIR`, `GPIODEN`, etc.).

```
GPIOF->LOCK = 0x4C4F434B;     // Unlock Port F
GPIOF->CR   |= (1 << 0);      // Allow changes to PF0
```

Listing 5.4: Unlocking PF0 for GPIO use

After unlocking, PF0 can be configured like any other GPIO pin.

5.1.7 GPIO Interrupt Configuration Registers

To configure a GPIO pin to generate interrupts, several registers must be configured. The process involves selecting the trigger condition (edge or level, rising or falling) and enabling the interrupt at both the GPIO module and the NVIC.

GPIOIS — Interrupt Sense Register

Register: `GPIOIS` — GPIO Interrupt Sense (Port F: `0x40025404`)
The `GPIOIS` register determines whether interrupts are triggered by signal **edges** (transitions) or **levels** (steady states).

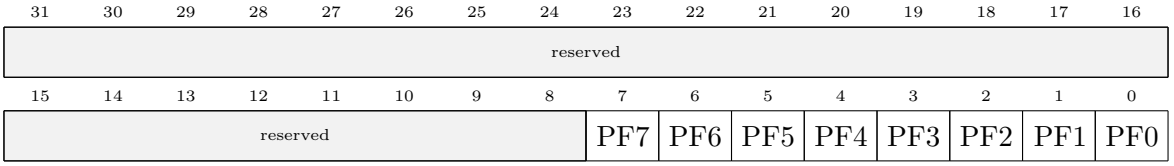


Figure 5.6: GPIOIS Register — Interrupt Sense

Bit Values:

- **0**: Edge-sensitive (detects transitions) — typical for buttons
- **1**: Level-sensitive (detects steady state)

GPIOIBE — Interrupt Both Edges Register

Register: `GPIOIBE` — GPIO Interrupt Both Edges (Port F: `0x40025408`)
When edge-sensitive mode is selected (bit cleared in `GPIOIS`), this register determines whether interrupts occur on a single edge or both edges.

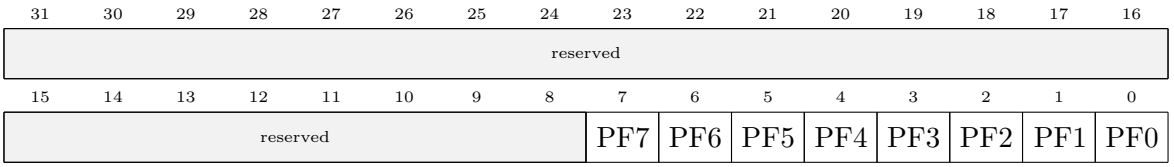


Figure 5.7: GPIOIBE Register — Interrupt Both Edges

Bit Values:

- **0**: Single edge (rising or falling, defined by `GPIOIEV`)

- **1**: Both rising and falling edges trigger interrupts

GPIOIEV — Interrupt Event Register

Register: `GPIOIEV` — GPIO Interrupt Event (Port F: `0x4002540C`)
For single-edge interrupts (when the corresponding bit in `GPIOIBE` is 0), this register selects which edge triggers the interrupt.

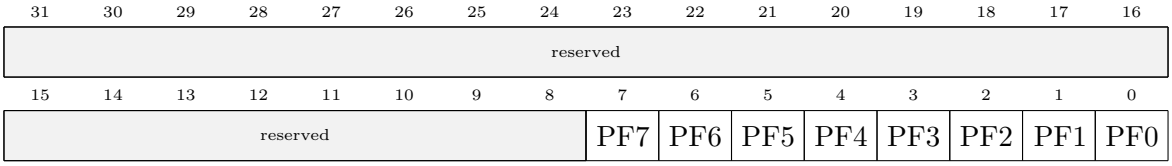


Figure 5.8: GPIOIEV Register — Interrupt Event

Bit Values:
- **0**: Falling edge (high-to-low transition)
- **1**: Rising edge (low-to-high transition)

For switches with pull-ups (LaunchPad SW1/SW2), falling edge interrupts detect button presses.

GPIOIM — Interrupt Mask Register

Register: `GPIOIM` — GPIO Interrupt Mask (Port F: `0x40025410`)
The `GPIOIM` register enables or disables (masks/unmasks) interrupt generation for each pin. Setting a bit to '1' allows that pin to generate interrupts.

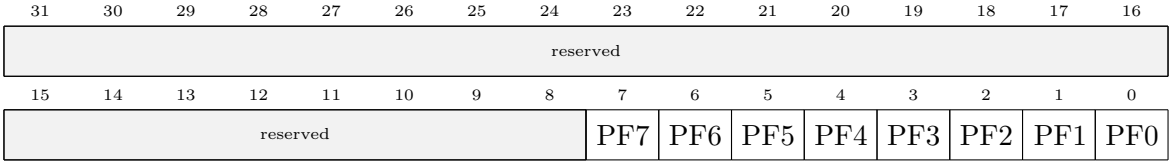


Figure 5.9: GPIOIM Register — Interrupt Mask

Bit Values:
- **0**: Interrupt disabled (masked) — default
- **1**: Interrupt enabled (unmasked)

GPIOMIS — Masked Interrupt Status Register

Register: `GPIOMIS` — GPIO Masked Interrupt Status (Port F: `0x40025418`)
This read-only register shows which pins have pending interrupts after masking. The ISR reads this register to determine which pin(s) caused the interrupt.

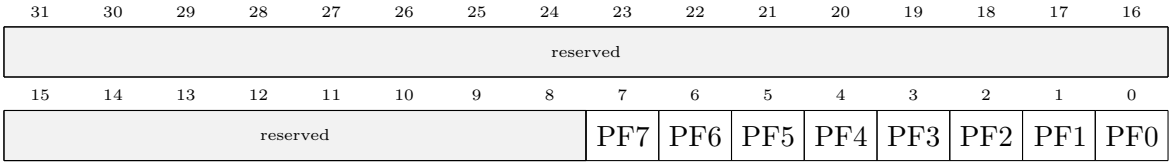


Figure 5.10: GPIOMIS Register — Masked Interrupt Status (Read-Only)

Bit Values:

- **0**: No interrupt pending for this pin
- **1**: Interrupt pending for this pin

GPIOICR — Interrupt Clear Register

Register: `GPIOICR` — GPIO Interrupt Clear (Port F: `0x4002541C`)
Writing '1' to a bit in this register clears the corresponding interrupt flag. This is **critical**: the ISR must clear the flag, or the interrupt will continuously re-trigger.

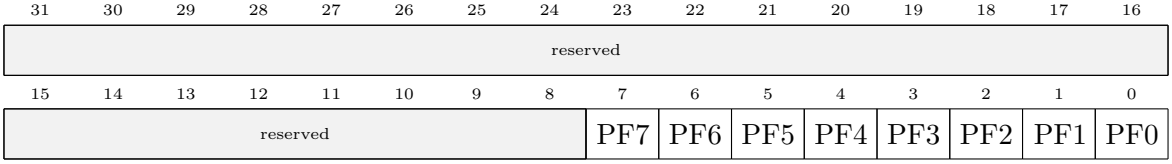


Figure 5.11: GPIOICR Register — Interrupt Clear

Bit Values:

- Write **1** to clear the interrupt flag for that pin
- Write **0** has no effect

5.1.8 Configuration Workflow

GPIO interrupt configuration is divided into two parts, configuring the pin as an input and configuring the interrupt settings. The following steps summarize the process:

Step 1: Pin Configuration Configure the GPIO pin as an input with proper electrical characteristics:

1. Enable the GPIO port clock (`RCGCGPIO`).
2. Unlock protected pins if needed (`GPIOLOCK`, `GPIOCR`).
3. Configure the pin as input (`GPIODIR` = 0).
4. Enable the digital function (`GPIODEN` = 1).
5. Enable pull-up or pull-down resistor (`GPIOPUR` or `GPIOPDR`).

Step 2: Interrupt Configuration

1. Configure interrupt sense (`GPIOIS` = 0 for edge-sensitive).
2. Select edge(s) (`GPIOIBE`/`GPIOIEV`).
3. **Clear any prior flags** (`GPIOICR` = 1 for the pin).
4. Unmask the pin in the GPIO (`GPIOIM` = 1).
5. Enable the IRQ in the NVIC.

5.1.9 NVIC — Nested Vectored Interrupt Controller

The NVIC manages all peripheral interrupts on the Cortex-M4, deciding which ISR to execute based on enable bits, masking, and priority.

NVIC_ISER — Interrupt Set-Enable Registers

Registers: `NVIC_ISER[0..3]` (`0xE000E100-0xE000E10C`)
Each 32-bit `ISER` register enables up to 32 interrupt sources. Writing a '1' to a bit enables that IRQ; writing '0' has no effect.

Register	IRQ Range	Address
`ISER[0]`	0-31	0xE000E100
`ISER[1]`	32-63	0xE000E104
`ISER[2]`	64-95	0xE000E108
`ISER[3]`	96-127	0xE000E10C

Table 5.1: Interrupt ranges for NVIC_ISER registers

For the TM4C123, all GPIO interrupts (IRQ0-IRQ4) for Ports A-E, and Port F (IRQ30) are in `ISER[0]`.

Example — Enabling Port F Manually

```
NVIC->ISER[0] |= (1 << 30);   // IRQ30 - NVIC_EN0 bit 30
```

Listing 5.5: Enable GPIO Port F interrupt manually

CMSIS Alternative (Recommended):

```
NVIC_EnableIRQ(GPIOF_IRQn);
```

This function automatically computes the correct register and bit:

```
void NVIC_EnableIRQ(IRQn_Type IRQn) {
    if (IRQn >= 0)
        NVIC->ISER[IRQn >> 5] |= (1u << (IRQn & 0x1F)); // /32 and %32
}
```

Explanation:

$$\text{Register index} = \left\lfloor \frac{\text{IRQn}}{32} \right\rfloor, \qquad \text{Bit position} = \text{IRQn} \bmod 32.$$

Each `ISER` register enables 32 interrupts. Dividing the IRQ number by 32 selects the correct register index, and the remainder (`IRQn % 32`) identifies the bit position within that 32-bit register.

5.1.10 Interrupt Service Routine (ISR)

When a GPIO interrupt occurs, the NVIC calls the corresponding ISR. For GPIO Port F, the ISR is named `GPIOF_Handler()`. The ISR must:

1. Determine which pin caused the interrupt (read `GPIOMIS`).
2. Perform the required action (e.g., toggle an LED).
3. Clear the interrupt flag (write to `GPIOICR`).

```
void GPIOF_Handler(void) {
    if (GPIOF->MIS & (1 << 4)) {     // Check if PF4 caused interrupt
        // Respond to SW1 press
        GPIOF->ICR |= (1 << 4);      // Clear PF4 interrupt flag
    }
    if (GPIOF->MIS & (1 << 0)) {     // Check if PF0 caused interrupt
```

```
        // Respond to SW2 press
        GPIOF->ICR |= (1 << 0);      // Clear PF0 interrupt flag
    }
}
```

Listing 5.6: GPIO Port F ISR Template

5.2 Procedure

5.2.1 Examples

The following examples demonstrate GPIO input configuration using polling and interrupt-based approaches.

5.2.1.1 Example 1 — Reading Switch Input Using Polling

This example continuously polls the state of SW1 (PF4) and turns on the red LED (PF1) when the button is pressed.

```
#include "TM4C123.h"

#define RED_LED 0x02
#define SWITCH 0x10

int main(void) {
    unsigned int state;
    SYSCTL->RCGCGPIO |= (1<<5);       // Enable Port F
    GPIOF->PUR |= SWITCH;                   // Enable pull-up resistor on PF4
    GPIOF->DIR |= RED_LED;                  // Set PF1 as an output pin and PF4 as
    an input pin
    GPIOF->DEN |= (RED_LED | SWITCH);       // Enable PF1 and PF4 as a digital GPIO
    pins
    while(1) {
        state = GPIOF->DATA & SWITCH;       // Read the state of the switch
        if (state == 0) {                   // If the switch is pressed (since it's
    Pull-up)
            GPIOF->DATA |= RED_LED;         // Turn on the LED
        } else {                            // If the switch is not pressed
            GPIOF->DATA &= ~RED_LED;        // Turn off the LED
        }
    }
}
```

Listing 5.7: Polling-based switch input reading

Explanation:

- `GPIOF->PUR |= SWITCH;` enables the internal pull-up resistor on PF4.
- The `while(1)` loop continuously reads the switch state.
- When `state == 0`, the button is pressed (pull-up makes it active-low).
- The LED is controlled directly in the main loop without interrupts.

5.2.1.2 Example 2 — Interrupt-Driven Switch Input

This example configures SW1 and SW2 to generate interrupts on button presses and toggles the green LED in the ISR.

```c
#include "TM4C123.h"

#define GREEN_LED 0x08
#define SW1 0x10
#define SW2 0x01

int main(void)
{
    SYSCTL->RCGCGPIO |= (1<<5);             // Enable clock to GPIOF

    GPIOF->LOCK = 0x4C4F434B;               // unlock commit register
    GPIOF->CR = 0x01;                       // make PORTF0 configurable


    // PINS Configurations
    GPIOF->DIR &= ~(SW1 | SW2);             // Set SW1, SW2 as input pins
    GPIOF->DIR |= GREEN_LED;                // Set GREEN_LED as output pin

    GPIOF->DEN |= (GREEN_LED | SW1 | SW2);  // Enable digital function for
    GREEN_LED, SW1, SW2
    GPIOF->PUR |= (SW1 | SW2);              // Enable pull-up resistors on SW1, SW2

    // Interrupt Configurations
    GPIOF->IS  &= ~(SW1 | SW2);             // SW1, SW2 are edge-sensitive
    GPIOF->IBE &= ~(SW1 | SW2);             // SW1, SW2 are not both edges
    GPIOF->IEV &= ~(SW1 | SW2);             // falling edge trigger
    GPIOF->ICR |= (SW1 | SW2);              // clear any prior interrupt
    GPIOF->IM  |= (SW1 | SW2);              // unmask interrupt

    // Enable Interrupts
    NVIC->ISER[0] |= (1<<30);               // enable interrupt 30 in NVIC
    // NVIC_EnableIRQ(GPIOF_IRQn);          // Alternative way using CMSIS function
    while(1)
    {
    }
}

void GPIOF_Handler(void) {
    if (GPIOF->MIS & SW1) {                 // check if interrupt causes by PF4/SW1
        GPIOF->DATA |= GREEN_LED;           // turn on green LED
        GPIOF->ICR |= SW1;                  // clear the interrupt flag
    }
    else if (GPIOF->MIS & SW2) {            // check if interrupt causes by PF0/SW2
        GPIOF->DATA &= ~GREEN_LED;          // turn off green LED
        GPIOF->ICR |= SW2;                  // clear the interrupt flag
    }
}
```

Listing 5.8: Interrupt-driven switch input handling

Explanation:

- `GPIOF->LOCK = 0x4C4F434B;` unlocks Port F for PF0 configuration.

- `GPIOF->CR = 0x01;` allows changes to PF0.
- `GPIOF->IS &= ~(SW1 | SW2);` configures edge-sensitive interrupts.
- `GPIOF->IEV &= ~(SW1 | SW2);` selects falling-edge trigger (button press).
- `GPIOF->ICR |= (SW1 | SW2);` clears any prior interrupt flags before enabling.
- `GPIOF->IM |= (SW1 | SW2);` unmasks (enables) interrupts for SW1 and SW2.
- `NVIC->ISER[0] |= (1<<30);` enables GPIO Port F interrupt in NVIC.
- The ISR checks `GPIOF->MIS` to identify which switch caused the interrupt.
- Each interrupt flag must be cleared with `GPIOF->ICR` to prevent re-triggering.

5.2.2 Tasks

5.2.2.1 Task 1 — Toggle LED Using Polling

Modify Example 1 to toggle the green LED (PF3) with each press of SW1 (PF4). The LED should change state (ON→OFF or OFF→ON) every time the button is pressed and released.

Requirements:

- Use polling to detect button presses.
- Implement software debouncing by adding a 20 ms delay after detecting a press.
- Toggle the LED state instead of simply turning it on or off.
- Ensure the LED changes state only once per button press.

5.2.2.2 Task 2 — LED Sequence Using Interrupts

Modify Example 2 to cycle through LED colors using interrupts:

- Pressing SW1 (PF4) cycles through: **Red → Blue → Green → Red...**
- Pressing SW2 (PF0) cycles through: **Yellow → Magenta → Cyan → Yellow...**

Requirements:

- Use interrupt-driven input handling.
- Maintain separate state variables for each button's LED sequence.
- Update the LED color in the ISR based on the current state.
- Clear interrupt flags properly to avoid repeated triggering.

Hint: Use a global variable (e.g., `sw1_state`) to track the current position in the sequence. In the ISR, increment the state and use a switch-case or modulo operation to cycle through colors. Refer to Table 4.5 in Experiment 4 for LED color codes.

6 Hardware Timers and Timing Control

Learning Objectives

After completing this experiment, you will be able to:

- Understand the role and limitations of the SysTick timer in the ARM Cortex-M4 core.
- Configure the SysTick timer for periodic interrupts and precise timing.
- Understand the architecture and capabilities of General-Purpose Timer Modules (GPTM).
- Configure GPTM in 16-bit and 32-bit modes for periodic and one-shot operation.
- Calculate timer periods using prescalers and reload values.
- Implement timer-based debouncing for mechanical switches.
- Use multiple timers to control independent timing tasks.

Experiment Overview

This experiment introduces precise timing control using the TM4C123's SysTick timer and General-Purpose Timer Modules (GPTM). You will configure SysTick for periodic interrupts and millisecond timing, explore GPTM's 16-bit and 32-bit modes with prescalers, and implement multiple independent timing tasks. By the end of this lab, you will understand how to select and configure timers for different timing requirements and implement timer-based interrupt-driven applications.

6.1 Theoretical Background

6.1.1 Introduction to Timers

Hardware timers are specialized peripherals that count clock cycles and generate events at precise intervals. Unlike software-based delays (which block the CPU and waste power), hardware timers run independently in the background and generate interrupts when they reach a predetermined count.

Why Use Hardware Timers?

- **Precise timing**: Timers operate at the system clock frequency (50 MHz), providing microsecond-level accuracy
- **Non-blocking**: The CPU can perform other tasks while the timer runs independently
- **Power efficient**: Timers can wake the MCU from sleep mode
- **Multiple independent channels**: Several timers can run simultaneously with different periods

Feature	SysTick	GPTM
Bit width	24-bit	16-bit or 32-bit
Max period @ 50MHz	335 ms	32-bit: 86 seconds
Use case	OS ticks, simple delays	Complex timing, PWM, capture
Availability	Built into ARM Cortex-M core	6-12 on TM4C123

Table 6.1: Choosing Between SysTick and GPTM

When to Use Each Timer For short delays (< 300 ms), SysTick is simpler. For longer delays or advanced features, use GPTM.

6.1.2 Timer Fundamentals

Understanding the basic operation of timers will help you configure them correctly and debug timing issues.

6.1.2.1 Down-Counting Operation

Both SysTick and GPTM timers operate as **down-counters** in their most common configuration (periodic mode):

1. **Load**: A reload value is written to the timer's load register (`LOAD` for SysTick, `TAILR` for GPTM)
2. **Count**: The timer counts down from the reload value to zero, decrementing once per clock cycle (or prescaled cycle)
3. **Timeout**: When the counter reaches zero:
   - The timeout flag is set
   - An interrupt is generated (if enabled and unmasked)
   - The counter automatically reloads from the load register (periodic mode) or stops (one-shot mode)
4. **Clear**: The interrupt handler must clear the timeout flag by writing to the interrupt clear register

6.1.2.2 Timer Modes

Timers can operate in different modes depending on the application:

- **Periodic Mode**: The timer automatically reloads and continues counting. Ideal for fixed-rate tasks like LED blinking, sampling sensors, or OS ticks.
- **One-Shot Mode**: The timer counts down once and stops. Useful for timeouts, delays, or single events.
- **Capture Mode**: (GPTM only) Captures the counter value when an external event occurs on a GPIO pin. Used for measuring pulse widths or frequencies.
- **PWM Mode**: (GPTM only) Generates pulse-width modulated output signals for motor control, dimming LEDs, etc.

6.1.2.3 Prescaler (16-bit Mode Only)

In 16-bit mode, GPTM provides an 8-bit prescaler (`TAPR`) that extends the timer range by dividing the input clock:

$$T = \frac{(\text{TAILR} + 1) \times (\text{TAPR} + 1)}{f_{\text{SysClk}}}$$

The prescaler is ignored in 32-bit mode. For example, with `TAPR = 255` (divisor = 256), the effective clock frequency is reduced to $50\,\text{MHz}/256 \approx 195.3\,\text{kHz}$.

6.1.2.4 Interrupt Handling

Proper interrupt handling is critical for timer-based applications:

1. **Enable the interrupt**: Set the appropriate bit in the timer's interrupt mask register (`IMR`)
2. **Enable in NVIC**: Set the corresponding bit in the NVIC's `ISER` register
3. **Implement the ISR**: Write an interrupt service routine with the correct name (e.g., `SysTick_Handler()`, `TIMER1A_Handler()`)
4. **Clear the flag**: Write 1 to the appropriate bit in the interrupt clear register (`ICR`) to acknowledge the interrupt

Best Practice: Clear the interrupt flag early in the ISR to prevent missing subsequent interrupts:

```
void TIMER1A_Handler(void) {
    TIMER1->ICR = (1 << 0);  // Clear TATOCINT flag immediately
    // Perform minimal work: toggle GPIO, update counters, etc.
}
```

6.1.3 SysTick Timer

The SysTick timer is a 24-bit down-counter that is part of the ARM Cortex-M4 core. It is present in all Cortex-M processors and provides a standard, simple timing mechanism for operating systems and applications.

6.1.3.1 SysTick Features

The SysTick timer provides:

- **24-bit down-counter**: Counts from a reload value down to zero.
- **Automatic reload**: When the counter reaches zero, it automatically reloads from the `LOAD` register.
- **Optional interrupt**: Can generate an interrupt when the counter reaches zero.
- **Clock source selection**: Can use the processor clock or an external reference clock.

- **CMSIS support**: The CMSIS library provides a simple `SysTick_Config()` function for easy setup.

The SysTick timer is ideal for creating system ticks (e.g., 1 ms intervals for RTOS scheduling), implementing delays, and measuring time intervals. However, its 24-bit width limits the maximum period at high clock frequencies.

6.1.3.2 SysTick Registers

The SysTick timer is controlled through three main registers:

SysTick Control and Status Register (STCTRL)

Register: `SysTick->CTRL` — SysTick Control and Status (`0xE000E010`)
The `CTRL` register controls the SysTick timer operation and provides status information.

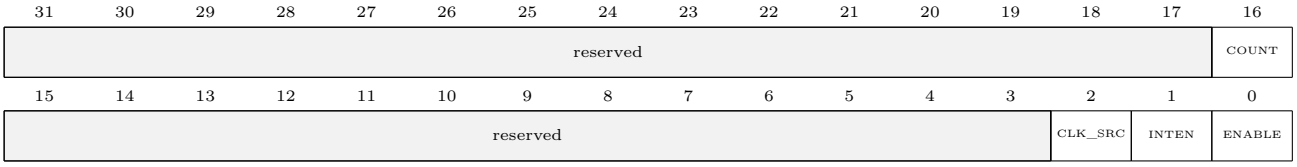


Figure 6.1: SysTick CTRL Register

Key Bits:

- **Bit 0 (ENABLE)**: Enable/disable the SysTick timer
  - 0 = Timer disabled
  - 1 = Timer enabled
- **Bit 1 (INTEN)**: Enable SysTick interrupt
  - 0 = No interrupt when counter reaches zero
  - 1 = Generate interrupt when counter reaches zero
- **Bit 2 (CLK_SRC)**: Clock source selection
  - 0 = External reference clock
  - 1 = Processor clock (typical)
- **Bit 16 (COUNT)**: Counter reached zero since last read (read-only)
  - 0 = Has not counted to zero
  - 1 = Has counted to zero (cleared on read)

SysTick Reload Value Register (STLOAD)

Register: `SysTick->LOAD` — SysTick Reload Value (`0xE000E014`)
The `LOAD` register holds the value that is loaded into the counter when it reaches zero or when the timer is enabled.

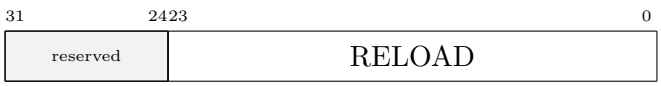


Figure 6.2: SysTick LOAD Register — 24-bit Reload Value

RELOAD Field (Bits 23:0): The value to load into the counter. Valid range: `0x000001` to `0xFFFFFF`. Writing zero disables the counter.

SysTick Current Value Register (STCURRENT)

Register: `SysTick->VAL` — SysTick Current Value (`0xE000E018`)
The `VAL` register contains the current value of the SysTick counter.

Figure 6.3: SysTick VAL Register — Current Counter Value

CURRENT Field (Bits 23:0): Current counter value. Writing any value clears the counter to zero and clears the `COUNTFLAG` in `CTRL`.

6.1.3.3 SysTick Timing Calculation

The SysTick timer counts down from the `LOAD` value to zero. The period is determined by:

$$T = \frac{\text{LOAD} + 1}{f_{\text{clock}}}$$

where f_{clock} is the processor clock frequency (typically 50 MHz on the TM4C123).
Maximum Period: With a 24-bit counter at 50 MHz:

$$T_{\max} = \frac{2^{24}}{50{,}000{,}000} = \frac{16{,}777{,}216}{50{,}000{,}000} \approx 0.335 \text{ seconds}$$

For longer periods, accumulate ticks in software or use the GPTM.

6.1.3.4 SysTick Configuration with CMSIS

The CMSIS library provides a convenient function to configure the SysTick timer:

```
// Configure SysTick for 1 ms interrupts at SystemCoreClock
SysTick_Config(SystemCoreClock / 1000);  // 50,000 ticks = 1 ms period
```

Listing 6.1: SysTick configuration using CMSIS

This function:

- Sets the `LOAD` register to the specified value minus 1
- Clears the `VAL` register
- Enables the SysTick interrupt in the NVIC
- Enables the SysTick timer with the processor clock source

The SysTick interrupt handler is named `SysTick_Handler()`:

```
void SysTick_Handler(void) {
    // Called every 1 ms
    // Increment global counter, toggle LED, etc.
}
```

6.1.4 General-Purpose Timer Module (GPTM)

The TM4C123 includes twelve General-Purpose Timer Modules (GPTM): six 16/32-bit timers (TIMER0-TIMER5) and six 32/64-bit wide timers (WTIMER0-WTIMER5). Each module can operate as two independent 16-bit timers or one 32-bit timer.

6.1.4.1 GPTM Features

GPTM modules provide:

- **16-bit or 32-bit operation**: Configurable timer width
- **Two independent timers**: Timer A and Timer B in each module
- **Multiple operating modes**:
  - One-Shot mode: Timer runs once and stops
  - Periodic mode: Timer reloads automatically
  - Real-Time Clock (RTC) mode: Precise time-keeping
  - PWM mode: Pulse Width Modulation output
  - Input Capture mode: Measure input signal timing
- **Prescaler**: Extends timer range in 16-bit mode (8-bit prescaler for standard timers)
- **Interrupt generation**: Timeout, match, and capture interrupts
- **Trigger and synchronization**: Can trigger ADC conversions or synchronize with other timers

6.1.4.2 GPTM Architecture

Each GPTM module contains:

- **Timer A**: Independent timer with its own registers
- **Timer B**: Independent timer with its own registers
- **Configuration Register (CFG)**: Selects 16-bit or 32-bit mode
- **Mode Registers (TAMR, TBMR)**: Configure operating mode for each timer
- **Load Registers (TAILR, TBILR)**: Set the reload/start value
- **Prescaler Registers (TAPR, TBPR)**: Extend timer range (16-bit mode only)
- **Control Register (CTL)**: Enable/disable timers and configure behavior
- **Interrupt Registers (IMR, RIS, MIS, ICR)**: Manage interrupts

In 32-bit mode, Timer A operates as a full 32-bit timer, and Timer B is not available. In 16-bit mode, both Timer A and Timer B operate independently as 16-bit timers.

6.1.4.3 GPTM Configuration Registers

GPTMCFG — Timer Configuration Register

Register: `TIMERx->CFG` — GPTM Configuration (`Base + 0x000`)
The `CFG` register selects the timer width (16-bit or 32-bit mode).

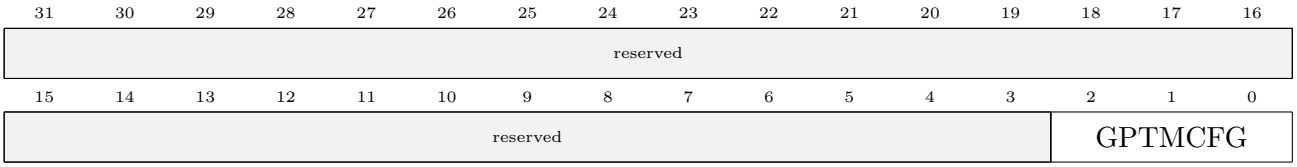


Figure 6.4: GPTMCFG Register — Timer Width Configuration

GPTMCFG Field (Bits 2:0):

- `0x0`: For a 16/32-bit timer, this value selects the 32-bit timer configuration. For a 32/64-bit wide timer, this value selects the 64-bit timer configuration.
- `0x1`: For a 16/32-bit timer, this value selects the 32-bit real-time clock (RTC) counter configuration. For a 32/64-bit wide timer, this value selects the 64-bit real-time clock (RTC) counter configuration.
- `0x2-0x3`: Reserved

- `0x4`: For a 16/32-bit timer, this value selects the 16-bit timer configuration. For a 32/64-bit wide timer, this value selects the 32-bit timer configuration. The function is controlled by bits 1:0 of **GPTMTAMR** and **GPTMTBMR**.
- `0x5-0x7`: Reserved

GPTMTAMR — Timer A Mode Register

Register: `TIMERx->TAMR` — GPTM Timer A Mode (`Base + 0x004`)
The `TAMR` register configures the operating mode for Timer A.

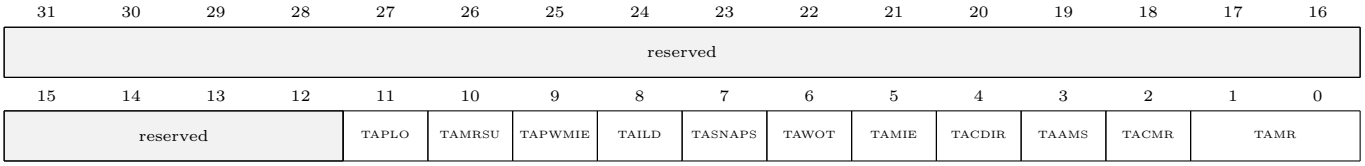


Figure 6.5: GPTMTAMR Register — Timer A Mode Configuration

Key Fields:
- **Bits 1:0 (TAMR)**: Timer A Mode
  - `0x1`: One-Shot mode
  - `0x2`: Periodic mode
  - `0x3`: Capture mode
- **Bit 4 (TACDIR)**: Timer A Count Direction (0 = down, 1 = up)

For this experiment, we will only be using One-Shot and Periodic modes. For more details on other fields, refer to the datasheet.

GPTMTAILR — Timer A Interval Load Register

Register: `TIMERx->TAILR` — GPTM Timer A Interval Load (`Base + 0x028`)
The `TAILR` register sets the start/reload value for Timer A.

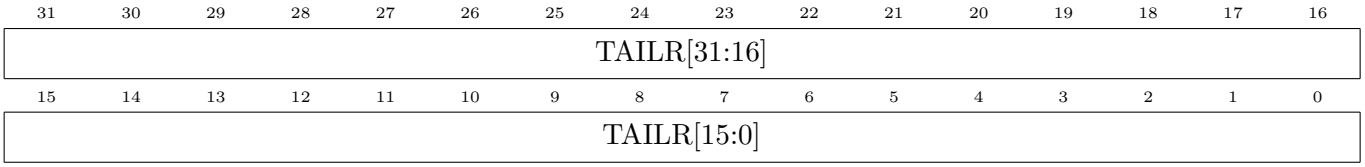


Figure 6.6: GPTMTAILR Register — Timer A Interval Load Value

TAILR Field:
- **Count-down mode**: Specifies the starting count value loaded into the timer
- **Count-up mode**: Sets the upper bound for the timeout event
- **32-bit mode**: Full 32-bit register (upper 16 bits correspond to GPTMTBILR contents)
- **16-bit mode**: Only bits [15:0] are used (upper 16 bits read as 0, no effect on GPTMTBILR)
- **64-bit Wide Timer mode**: Contains bits [31:0] of the 64-bit count (GPTMTBILR contains bits [63:32])

GPTMTAPR — Timer A Prescaler Register

Register: `TIMERx->TAPR` — GPTM Timer A Prescaler (`Base + 0x038`)
The `TAPR` register extends the timer range in 16-bit mode by providing an 8-bit prescaler.

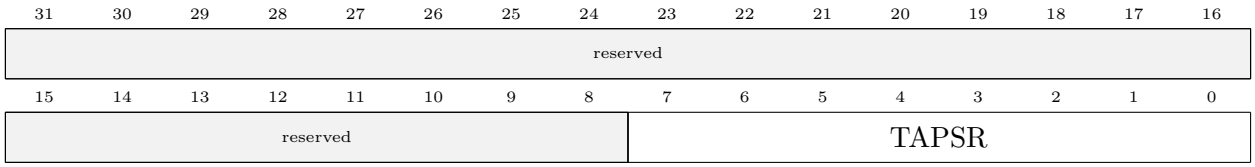


Figure 6.7: GPTMTAPR Register — Timer A Prescaler (16-bit mode only)

TAPSR Field (Bits 7:0):

- Prescaler value: `0x00` to `0xFF` (0 to 255)
- Only used in 16-bit mode; ignored in 32-bit mode
- Effective divisor: `TAPSR + 1`

GPTMCTL — Timer Control Register

Register: `TIMERx->CTL` — GPTM Control (`Base + 0x00C`)
The `CTL` register enables/disables timers and configures their behavior.

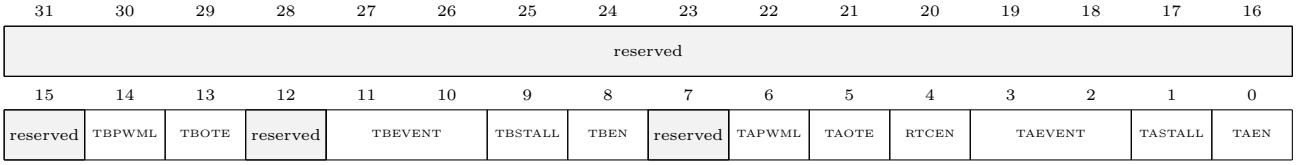


Figure 6.8: GPTMCTL Register — Timer Enable and Control

Key Control Functions:

- **Timer Enable Control**: TAEN (bit 0) and TBEN (bit 8) independently enable/disable Timer A and Timer B
- **Output Trigger Control**: TAOTE (bit 5) and TBOTE (bit 13) enable timers to trigger external peripherals (ADC, other timers, etc.)
- **Independent Operation**: Each timer can be controlled separately, allowing flexible dual-timer configurations

GPTMIMR — Interrupt Mask Register

Register: `TIMERx->IMR` — GPTM Interrupt Mask (`Base + 0x018`)
The `IMR` register enables or disables (masks/unmasks) timer interrupts.

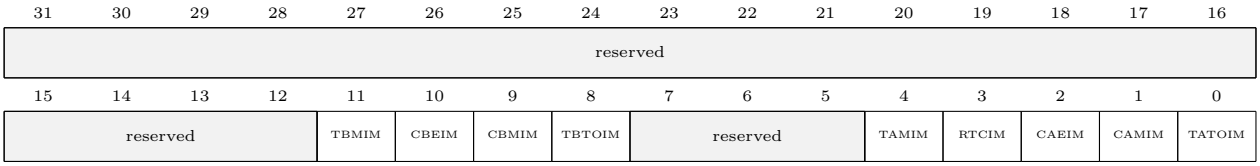


Figure 6.9: GPTMIMR Register — Interrupt Mask

Key Bits:

- **Bit 0 (TATOIM)**: Timer A Timeout Interrupt Mask (0 = masked, 1 = enabled)
- **Bit 8 (TBTOIM)**: Timer B Timeout Interrupt Mask (0 = masked, 1 = enabled)

GPTMICR — Interrupt Clear Register

Register: `TIMERx->ICR` — GPTM Interrupt Clear (`Base + 0x024`)
Writing '1' to a bit in this register clears the corresponding interrupt flag.

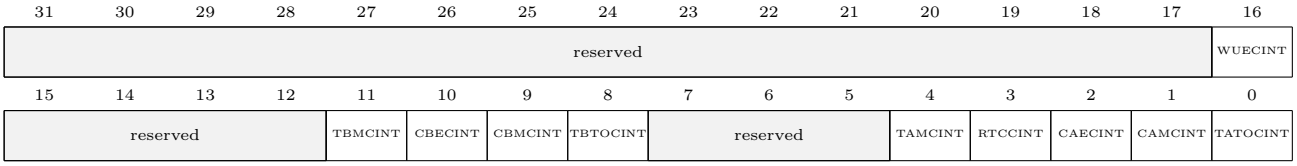


Figure 6.10: GPTMICR Register — Interrupt Clear

6.1.4.4 GPTM Timing Calculations

The timer period depends on the operating mode and configuration:

32-bit Mode (No Prescaler)

In 32-bit mode, the period is:

$$T = \frac{\text{TAILR} + 1}{f_{\text{clock}}}$$

At 50 MHz, the maximum period is:

$$T_{\max} = \frac{2^{32}}{50{,}000{,}000} = \frac{4{,}294{,}967{,}296}{50{,}000{,}000} \approx 85.9 \text{ seconds}$$

16-bit Mode (With Prescaler)

In 16-bit mode, the prescaler extends the timer range:

$$T = \frac{(\text{TAILR} + 1) \times (\text{TAPR} + 1)}{f_{\text{clock}}}$$

At 50 MHz, with maximum values (`TAILR = 0xFFFF`, `TAPR = 0xFF`):

$$T_{\max} = \frac{65{,}536 \times 256}{50{,}000{,}000} = \frac{16{,}777{,}216}{50{,}000{,}000} \approx 0.335 \text{ seconds}$$

6.1.4.5 GPTM Interrupt Numbers

Each GPTM module has a unique interrupt number for Timer A and Timer B; you can find these in the TM4C123 datasheet or in the device headers. The CMSIS-compliant header `tm4c123gh6pm.h` defines these as `IRQn_Type` enumerations (e.g., `TIMER1A_IRQn`), which work directly with the NVIC helper functions.

Enabling an interrupt (CMSIS).

Use `NVIC_EnableIRQ(IRQn_Type irqn)`; CMSIS handles the correct ISER register and bit index for you.

```
#include "TM4C123.h"

// Enable TIMER1A interrupt:
NVIC_EnableIRQ(TIMER1A_IRQn);

// Enable multiple GPTM interrupts:
NVIC_EnableIRQ(TIMER0A_IRQn);
NVIC_EnableIRQ(TIMER1A_IRQn);
NVIC_EnableIRQ(TIMER2A_IRQn);
```

Setting interrupt priority.

Use `NVIC_SetPriority(IRQn_Type irqn, uint32_t priority)` before enabling the interrupt. On Cortex-M4 (TM4C123), priorities are typically 0-7 (0 = highest urgency).

```
// Assign priorities (lower number = higher priority)
NVIC_SetPriority(TIMER0A_IRQn, 3);
NVIC_SetPriority(TIMER1A_IRQn, 4);
NVIC_SetPriority(TIMER2A_IRQn, 5);

// Then enable them
NVIC_EnableIRQ(TIMER0A_IRQn);
NVIC_EnableIRQ(TIMER1A_IRQn);
NVIC_EnableIRQ(TIMER2A_IRQn);
```

6.1.5 Configuration Workflow

6.1.5.1 SysTick Configuration Steps

To configure the SysTick timer:

1. Calculate the reload value: `LOAD = (Period x SystemCoreClock) - 1`
2. Load the value into `SysTick->LOAD`
3. Clear the current value: `SysTick->VAL = 0`
4. Configure `SysTick->CTRL`:
   - Set bit 0 (ENABLE) to enable the timer
   - Set bit 1 (TICKINT) to enable interrupts
   - Set bit 2 (CLKSOURCE) to use processor clock
5. Implement `SysTick_Handler()` to handle interrupts

Or use the CMSIS function:

```
SysTick_Config(SystemCoreClock / 1000);  // 1 ms period
```

6.1.5.2 GPTM Configuration Steps (Periodic Mode)

To configure a GPTM timer in periodic mode:

1. Enable the timer clock in `SYSCTL->RCGCTIMER`
2. Wait for clock stabilization (3 NOP instructions or check ready bit)
3. Disable the timer: `TIMERx->CTL = 0`

4. Select timer width: `TIMERx->CFG` (`0x00` for 32-bit, `0x04` for 16-bit)
5. Configure mode: `TIMERx->TAMR = 0x02` (periodic mode, count down)
6. Set prescaler (16-bit mode only): `TIMERx->TAPR = value`
7. Set reload value: `TIMERx->TAILR = value`
8. Clear interrupt flag: `TIMERx->ICR = 0x01`
9. Enable timeout interrupt: `TIMERx->IMR |= 0x01`
10. Enable interrupt in NVIC: `NVIC->ISER[n] |= (1 « bit)`
11. Enable the timer: `TIMERx->CTL |= 0x01`
12. Implement the ISR (e.g., `TIMER1A_Handler()`)

Register	**Purpose**	**Typical Value**
`RCGCTIMER`	Enable timer clock	`Set bit for TIMERx`
`TIMERx->CFG`	Select width (16-bit vs 32-bit)	`0x00 (32-bit) or 0x04 (16-bit)`
`TIMERx->TAMR`	Set mode	`0x02 (periodic, count down)`
`TIMERx->TAILR`	Set reload value	`From desired period`
`TIMERx->TAPR`	Prescaler (extends range in 16-bit mode)	`0-255 (16-bit only)`
`TIMERx->CTL`	Enable timer	`TAEN = 1 (bit 0)`
`TIMERx->IMR`	Unmask interrupt	`TATOIM = 1 (bit 0)`
`TIMERx->ICR`	Clear interrupt flag (write-1-to-clear)	`TATOCINT = 1 (bit 0)`

Table 6.2: GPTM Configuration Register Summary

6.2 Procedure

6.2.1 Examples

The following examples demonstrate SysTick and GPTM configuration and usage.

6.2.1.1 Example 1 — Millisecond Counter with SysTick Timer

This example configures the SysTick timer to generate interrupts every 100 ms and toggles the green LED (PF3) in the interrupt handler.

```
#include "TM4C123.h"
#define GREEN_LED 0x08          // PF3 - Green LED

volatile uint32_t systick_counter = 0;  // Global counter for SysTick interrupts

int main(void)
{
  SYSCTL->RCGCGPIO |= (1<<5);     // Enable clock to GPIOF

  __asm__("NOP");
  __asm__("NOP");
  __asm__("NOP");

  GPIOF->DIR |= GREEN_LED;       // Set green LED as output pin
  GPIOF->DEN |= GREEN_LED;       // Enable digital function for green LED


  SysTick->LOAD = 5000000 - 1;       // Set reload value for 100ms
  SysTick->VAL = 0;            // Clear current value
  SysTick->CTRL = 0x07;          // Enable SysTick with processor clock and interrupt

  while(1)                 // Main loop
  {
  }
}

void SysTick_Handler(void)
{
  systick_counter+=100;        // Increment counter by 100ms
  GPIOF->DATA ^= GREEN_LED;      // Toggle green LED
}
```

Listing 6.2: SysTick timer example — LED blink every 100ms

Explanation:

- `SysTick->LOAD = 5000000 - 1;` sets the reload value for 100 ms at 50 MHz.
- `SysTick->VAL = 0;` clears the current counter value.
- `SysTick->CTRL = 0x07;` enables the timer with processor clock and interrupt.
- `SysTick_Handler()` is called every 100 ms and toggles the LED.
- `systick_counter` tracks the number of interrupts (can be used for longer timing).

6.2.1.2 Example 2 — Maximum 16-bit Delay with GPTM

This example configures TIMER1 in 16-bit periodic mode with maximum prescaler to achieve the longest possible delay and toggles the blue LED (PF2).

```
#include "TM4C123.h"
#define BLUE_LED 0x04


int main(void)
{
  SYSCTL->RCGCGPIO |= (1<<5);     // Enable clock to GPIOF
  SYSCTL->RCGCTIMER |= (1<<1);    // Enable clock to Timer1

  __asm__("NOP");           // Wait for clock stabilization
  __asm__("NOP");           // Wait for clock stabilization
  __asm__("NOP");           // Wait for clock stabilization


  GPIOF->DIR |= BLUE_LED;       // Set blue LED as output pin
  GPIOF->DEN |= BLUE_LED;       // Enable digital function for blue LED

  TIMER1->CTL = 0;              // Disable the timer
  TIMER1->CFG = 0x4;            // Choose 16-bit mode
  TIMER1->TAMR = 0x02;            // Periodic mode
  TIMER1->TAPR = 256  - 1;      // Set prescaler value
  TIMER1->TAILR = 65536 - 1;      // Set initial reload value
  TIMER1->ICR = 0x1;              // Clear any prior interrupts
  TIMER1->IMR |=(1<<0);       // Enable timeout interrupt
  NVIC->ISER[0] |= (1<<21);     // Enable Timer1A interrupt in NVIC
  TIMER1->CTL |= 0x01;              // Enable the timer

    while(1)              // Main loop
    {
    }
}


void TIMER1A_Handler()
{
  if(TIMER1->MIS & 0x1) {       // Check if timer timeout interrupt occurred
    GPIOF->DATA  ^= BLUE_LED;   // Toggle blue LED
    TIMER1->ICR = 0x1;        // Clear timer interrupt flag
  }
}
```

Listing 6.3: GPTM Timer1A example — 16-bit mode with prescaler

Explanation:

- `SYSCTL->RCGCTIMER |= (1<<1);` enables clock to TIMER1.
- `TIMER1->CFG = 0x4;` selects 16-bit mode.
- `TIMER1->TAMR = 0x02;` configures periodic mode (count down).
- `TIMER1->TAPR = 256 - 1;` sets prescaler to maximum (divisor = 256).
- `TIMER1->TAILR = 65536 - 1;` sets interval load to maximum (65536).
- Period: $T = \frac{65536 \times 256}{50{,}000{,}000} \approx 0.335$ seconds.
- `NVIC->ISER[0] |= (1<<21);` enables TIMER1A interrupt (IRQ 21).

- `TIMER1A_Handler()` toggles the blue LED and clears the interrupt flag.

6.2.2 Tasks

6.2.2.1 Task 1 — Debouncing a Push Button with SysTick

Modify the SysTick example so that the RED LED (PF1) toggles only when SW1 (PF4) is pressed and properly debounced using the SysTick timer.

Requirements:

- Use `SysTick_Config(SystemCoreClock/1000)` to generate a 1 ms tick.
- Implement interrupt-driven GPIO input for SW1 (PF4) as in Experiment 5.
- In the GPIO ISR, use a global millisecond counter to implement 150 ms debouncing.
- Only toggle the RED LED if at least 150 ms have elapsed since the last press.

Hint: Declare global variables:

```
volatile uint32_t global_ms = 0;
volatile uint32_t last_press_ms = 0;
```

In `SysTick_Handler()`, increment `global_ms`. In `GPIOF_Handler()`, check:

```
if ((global_ms - last_press_ms) >= 150) {
    GPIOF->DATA ^= (1 << 1);  // Toggle RED LED
    last_press_ms = global_ms;
}
```

6.2.2.2 Task 2 — Multiple Blinking LEDs with GPTM

Use three GPTM timers (TIMER0A, TIMER1A, TIMER2A) to blink the three LEDs at different rates:

- **RED LED (PF1)**: Blink every 250 ms (use TIMER0A)
- **BLUE LED (PF2)**: Blink every 500 ms (use TIMER1A)
- **GREEN LED (PF3)**: Blink every 1000 ms (use TIMER2A)

Requirements:

- Configure each timer in 32-bit periodic mode.
- Calculate the `TAILR` values for each period at 50 MHz.
- Enable interrupts for each timer in the NVIC.
- Implement separate ISRs (`TIMER0A_Handler()`, `TIMER1A_Handler()`, `TIMER2A_Handler()`).
- Each ISR should toggle its corresponding LED and clear the interrupt flag.

7 Character LCD Display Interface

Learning Objectives

After completing this experiment, you will be able to:

- Understand the architecture and operation of the HD44780-based 16x2 LCD module.
- Differentiate between command and data registers and their respective control signals.
- Configure the LCD in 4-bit mode to minimize GPIO pin usage.
- Implement the initialization sequence for 4-bit LCD operation.
- Send commands and data to the LCD using proper timing and enable pulse sequences.
- Display text strings at specific cursor positions on the LCD.
- Implement scrolling and dynamic display effects using LCD commands.
- Integrate LCD interfacing with GPIO and interrupt-driven input from push buttons.

Experiment Overview

This experiment introduces LCD interfacing using the HD44780-based 16x2 character display module. You will learn the controller's architecture, implement 4-bit communication mode to minimize GPIO usage, and create a reusable LCD driver library with functions for initialization, command transmission, and data display. By the end of this lab, you will be able to interface an LCD with the TM4C123 microcontroller and create interactive display applications combining LCD output with GPIO input and interrupts.

7.1 Theoretical Background

7.1.1 HD44780 LCD Controller Architecture

The HD44780 is an LCD controller/driver developed by Hitachi and widely adopted as an industry standard for character-based LCD modules. It provides a simple microprocessor interface for controlling alphanumeric displays with minimal external components.

7.1.1.1 LCD Module Overview

The 16x2 LCD module consists of:

- **HD44780 Controller**: Manages display operations, character generation, and timing
- **LCD Panel**: 2 rows x 16 columns of character positions
- **Character Generator ROM (CGROM)**: Contains 208 predefined character patterns (alphanumeric, symbols, Japanese kana)
- **Character Generator RAM (CGRAM)**: 64 bytes for up to 8 user-defined custom characters (5x8 pixels each)
- **Display Data RAM (DDRAM)**: 80 bytes storing characters currently displayed (40 bytes per line)
- **Backlight**: Optional LED backlight for improved visibility

The controller handles all low-level display refresh, character rendering, and cursor management automatically. The microcontroller simply writes characters to DDRAM and issues commands for cursor positioning, display control, and special effects.

7.1.1.2 LCD Registers

The HD44780 has two main registers accessible by the microcontroller:

Instruction Register (IR) The Instruction Register receives commands that control display operations:

- Clear display and return cursor to home
- Set cursor position in DDRAM
- Control display on/off, cursor visibility, and blinking
- Set entry mode (cursor direction, display shift)
- Shift cursor or entire display left/right
- Configure interface width (4-bit or 8-bit), number of lines, and font size

Data Register (DR) The Data Register receives character codes (ASCII) to be displayed:

- Writing to DR displays a character at the current cursor position
- The cursor automatically advances after each write (direction set by entry mode)
- Reading from DR retrieves the character at the current cursor position (rarely used)

7.1.2 LCD Pin Configuration

The HD44780 interface consists of 16 pins (some modules have 18 pins with additional backlight control):

Pin	Name	Description
1	VSS	Ground (0V)
2	VDD	Power supply (+5V or +3.3V)
3	V0	Contrast adjustment (connect to potentiometer)
4	RS	Register Select: 0 = Instruction (command), 1 = Data (character)
5	RW	Read/Write: 0 = Write to LCD, 1 = Read from LCD (usually grounded for write-only)
6	E	Enable: Falling edge latches data/command into LCD
714	D0-D7	8-bit data bus (D0-D3 unused in 4-bit mode)
15	A	Backlight anode (+5V, typically with series resistor)
16	K	Backlight cathode (Ground)

Table 7.1: HD44780 LCD Pin Definitions

7.1.3 4-Bit vs. 8-Bit Communication Mode

The HD44780 supports two communication modes:

7.1.3.1 8-Bit Mode

- Uses all 8 data pins (D0-D7)
- Each command or character requires one write cycle
- Faster communication (single byte transfer)
- Requires 11 GPIO pins total (3 control + 8 data)

7.1.3.2 4-Bit Mode

- Uses only upper 4 data pins (D4-D7), D0-D3 are left unconnected
- Each command or character requires two write cycles (upper nibble, then lower nibble)
- Slightly slower due to two transfers per byte
- Requires only 7 GPIO pins total (3 control + 4 data)
- **Preferred mode** for GPIO-constrained systems

4-Bit Communication Protocol:

1. Set RS and RW to appropriate values
2. Place upper 4 bits (bits 7-4) on D7-D4
3. Generate enable pulse: E high → delay → E low
4. Place lower 4 bits (bits 3-0) on D7-D4
5. Generate another enable pulse: E high → delay → E low

In this experiment, we use 4-bit mode with the following connections:

- **PB0** → RS (Register Select)
- **PB2** → E (Enable)
- **PB4-PB7** → D4-D7 (Data pins)
- **RW** → Ground (write-only operation)

7.1.4 LCD Initialization Sequence

Proper initialization is critical for reliable LCD operation. The HD44780 requires a specific sequence of commands with precise timing delays, especially during the transition to 4-bit mode.

7.1.4.1 Initialization Steps (4-Bit Mode)

The initialization sequence must account for the fact that the LCD powers up in 8-bit mode and must be switched to 4-bit mode:

1. **Wait for power-on stabilization**: Delay at least 40 ms after VDD rises to 4.5V
2. **Initial function set (8-bit interface)**: Send `0x30` (upper nibble only) three times:
   - First time: Wait $>$ 4.1 ms
   - Second time: Wait $>$ 100 μs
   - Third time: Wait $>$ 100 μs
3. **Switch to 4-bit mode**: Send `0x20` (upper nibble only), wait $>$ 100 μs
4. **Configure display parameters**: Now send full 8-bit commands in two nibbles:
   - `0x28`: Function set — 4-bit mode, 2 lines, 5x8 font
   - `0x0C`: Display control — Display ON, cursor OFF, blink OFF
   - `0x06`: Entry mode set — Increment cursor, no display shift
   - `0x01`: Clear display
   - `0x02`: Return home

Critical Note: Steps 2-3 send only the *upper nibble* because the LCD is still in 8-bit mode. After step 3 completes, the LCD switches to 4-bit mode, and all subsequent commands must be sent as two nibbles (upper first, then lower).

7.1.5 LCD Command Set

The HD44780 supports a comprehensive set of commands for display control, cursor manipulation, and special effects.

Command	Hex Code	Description
Clear Display	`0x01`	Clears entire display, sets DDRAM address 0, cursor to home
Return Home	`0x02`	Sets DDRAM address 0, cursor to home (display content unchanged)
Entry Mode Set	`0x04-0x07`	Controls cursor direction and display shift
	`0x04`	Cursor moves left, no shift
	`0x06`	Cursor moves right, no shift (default)
	`0x05`	Cursor moves left, display shifts right
	`0x07`	Cursor moves right, display shifts left
Display Control	`0x08-0x0F`	Controls display, cursor, and blink on/off
	`0x08`	Display OFF
	`0x0C`	Display ON, cursor OFF, blink OFF
	`0x0E`	Display ON, cursor ON, blink OFF
	`0x0F`	Display ON, cursor ON, blink ON
Cursor/Display Shift	`0x10-0x1F`	Shifts cursor or display left/right
	`0x10`	Shift cursor left
	`0x14`	Shift cursor right
	`0x18`	Shift display left
	`0x1C`	Shift display right
Function Set	`0x20-0x3F`	Sets interface length, line number, font
	`0x28`	4-bit mode, 2 lines, 5x8 font
	`0x38`	8-bit mode, 2 lines, 5x8 font
Set DDRAM Address	`0x80 + addr`	Positions cursor at DDRAM address
	`0x80`	Beginning of line 1 (address 0x00)
	`0xC0`	Beginning of line 2 (address 0x40)

Table 7.2: Common HD44780 LCD Commands

7.1.6 DDRAM Address Mapping

The Display Data RAM (DDRAM) is 80 bytes, but only 32 bytes (16 per line) are visible at any time. Understanding the address mapping is essential for cursor positioning:

Display Position	DDRAM Address	Command
Line 1, Column 0	`0x00`	`0x80`
Line 1, Column 1	`0x01`	`0x81`
Line 1, Column 15	`0x0F`	`0x8F`
Line 2, Column 0	`0x40`	`0xC0`
Line 2, Column 1	`0x41`	`0xC1`
Line 2, Column 15	`0x4F`	`0xCF`

Table 7.3: DDRAM Address Mapping for 16x2 LCD

Formula for cursor positioning:

$$\text{Command} = \begin{cases} 0x80 + \text{column} & \text{for line 1 (row 0)} \\ 0xC0 + \text{column} & \text{for line 2 (row 1)} \end{cases}$$

7.1.7 Timing Requirements

The HD44780 requires specific timing for reliable operation:

- **Enable pulse width (PW_EH)**: Minimum 450 ns (typically use 1-2 μs for safety)
- **Enable cycle time (t_cycE)**: Minimum 1 μs (1 MHz max frequency)
- **Setup time (t_AS, t_DSW)**: RS and data must be stable 60 ns before E rises
- **Hold time (t_AH, t_DHW)**: RS and data must remain stable 20 ns after E falls
- **Command execution time**:
  - Normal commands: $\sim$40 μs (use 1-2 ms for safety)
  - Clear display (`0x01`): $\sim$1.6 ms (use 2-3 ms)
  - Return home (`0x02`): $\sim$1.6 ms (use 2-3 ms)

At 50 MHz system clock:

- 1 μs = 50 clock cycles
- 1 ms = 50,000 clock cycles

We implement microsecond delays using the SysTick timer to meet these timing requirements precisely.

7.1.8 LCD Driver Implementation Strategy

A well-structured LCD driver separates concerns into distinct layers:

Low-Level Functions

- `LCD_EnablePulse()`: Generates enable pulse (E high $\rightarrow$ delay $\rightarrow$ E low)
- `LCD_SendNibble()`: Sends 4 bits on D7-D4 with enable pulse
- `delay_us()`, `delay_ms()`: Precise timing using SysTick

Mid-Level Functions

- `LCD_Command()`: Sends 8-bit command (RS=0) as two nibbles
- `LCD_Data()`: Sends 8-bit character (RS=1) as two nibbles

High-Level API

- `LCD_Init()`: Complete initialization sequence
- `LCD_Clear()`: Clears display

- `LCD_SetCursor(row, col)`: Positions cursor
- `LCD_Print(string)`: Displays string at current cursor position

7.2 Procedure

7.2.1 Example: Basic LCD Driver Implementation

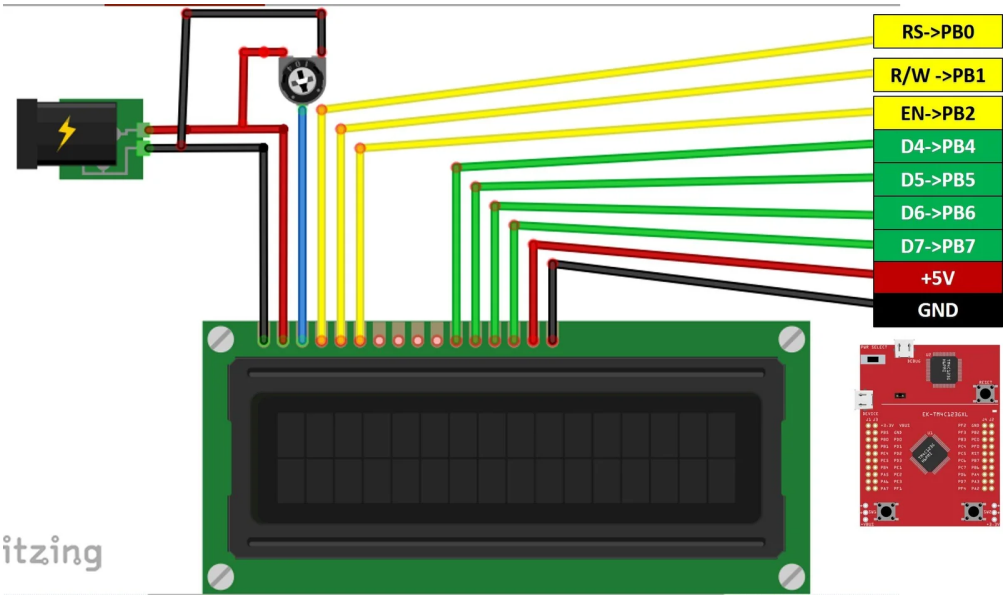


Figure 7.1: LCD Connection Schematic - TM4C123 to 16×2 HD44780 LCD Module[1]

This figure shows the complete wiring diagram for connecting the 16x2 LCD module to the TM4C123 microcontroller using 4-bit mode. The connections are:

- **Power**: VDD to VBus, VSS to Ground, V0 to contrast potentiometer
- **Control**: RS to PB0, E to PB2, RW to Ground (write-only)
- **Data**: D4-D7 to PB4-PB7 respectively
- **Backlight**: A to VBus, K to Ground

The contrast potentiometer (typically 10kΩ) allows adjustment of the display visibility - rotating it changes the voltage on V0 pin between 0V and VDD. The following code demonstrates a complete LCD driver in 4-bit mode with initialization, command/data transmission, and text display functions.

[1] Source: https://microcontrollerslab.com/16x2-lcd-interfacing-with-tm4c123-tiva-launchpad-keil-uvision/

7.2.1.1 LCD Header File

```c
#ifndef LCD_H
#define LCD_H

#include "TM4C123.h"

// LCD pin definitions (connected to PORTB)
#define RS         (1 << 0)  // PB0
#define RW         (1 << 1)  // PB1
#define EN         (1 << 2)  // PB2
#define DATA_MASK  0xF0      // PB4-PB7

// Function prototypes
void LCD_Init(void);
void LCD_Command(unsigned char cmd);
void LCD_Data(unsigned char data);
void LCD_Clear(void);
void LCD_SetCursor(unsigned char row, unsigned char col);
void LCD_Print(char *str);
void delay_us(int us);
void delay_ms(int ms);

#endif
```

Listing 7.1: LCD driver header file (`lcd.h`)

7.2.1.2 LCD Implementation File

```c
#include "lcd.h"

#define CYCLES_PER_US   (SystemCoreClock / 1000000u)



//====================[ SysTick Delay Functions ]====================
void SysTick_Init(void)
{
    SysTick->CTRL = 0;
    SysTick->LOAD = CYCLES_PER_US - 1;  // 1us delay at 50MHz
    SysTick->VAL = 0;
    SysTick->CTRL = 0x5;     // Enable with system clock
}

void delay_us(int us)
{
    SysTick->LOAD = (CYCLES_PER_US * us) - 1;
    SysTick->VAL = 0;
    SysTick->CTRL = 0x5; // Enable with system clock
    while ((SysTick->CTRL & 0x10000) == 0);
    SysTick->CTRL = 0;
}

void delay_ms(int ms)
{
```

```
    while (ms--)
        delay_us(1000);
}

//====================[ LCD Helper Functions ]====================
void LCD_EnablePulse(void)
{
    delay_us(1);
    GPIOB->DATA |= EN;
    delay_us(1);
    GPIOB->DATA &= ~EN;
    delay_us(1);
}

void LCD_SendNibble(unsigned char nibble)
{
    // Send nibble to PB4-PB7
    GPIOB->DATA = (GPIOB->DATA & ~DATA_MASK) | ((nibble << 4) & DATA_MASK);
    LCD_EnablePulse();
}

//====================[ LCD Initialization ]====================
void LCD_Init(void)
{
    // Enable clock to PORTB
    SYSCTL->RCGCGPIO |= (1 << 1);
    while ((SYSCTL->PRGPIO & (1 << 1)) == 0)
        ;

    // Configure PB0 (RS), PB1 (EN), PB4-PB7 (data) as output
    GPIOB->DIR |= RS | RW | EN | DATA_MASK;
    GPIOB->DEN |= RS | RW | EN | DATA_MASK;
    GPIOB->DATA &= ~(RS | RW | EN | DATA_MASK);  // Clear all

    SysTick_Init();

    delay_ms(50);  // Wait for LCD to power up

    // Initialization sequence (8-bit interface mode to start)
    LCD_SendNibble(0x03);
    delay_ms(5);

    LCD_SendNibble(0x03);
    delay_us(150);

    LCD_SendNibble(0x03);
    delay_us(150);

    LCD_SendNibble(0x02);  // Set 4-bit mode
    delay_us(150);

    // Now in 4-bit mode: use full commands
    LCD_Command(0x28); // Function set: 4-bit, 2 lines, 5x8 dots
    LCD_Command(0x0C); // Display ON, Cursor OFF
    LCD_Command(0x06); // Entry mode: increment cursor
    LCD_Command(0x01); // Clear display
```

```
    delay_ms(2);
}

//====================[ LCD Command/Data API ]====================
void LCD_Command(unsigned char command)
{
    GPIOB->DATA &= ~RS; // RS = 0 for command
    delay_us(1);
    LCD_SendNibble(command >> 4);      // Upper nibble
    LCD_SendNibble(command & 0x0F);    // Lower nibble
    delay_ms(2);
}

void LCD_Data(unsigned char data)
{
    GPIOB->DATA |= RS; // RS = 1 for data
    delay_us(1);
    LCD_SendNibble(data >> 4);
    LCD_SendNibble(data & 0x0F);
    delay_ms(1);
}

void LCD_Clear(void)
{
    LCD_Command(0x01);
    delay_ms(2);
}

void LCD_SetCursor(unsigned char row, unsigned char col)
{
    unsigned char address = (row == 0) ? 0x80 + col : 0xC0 + col;
    LCD_Command(address);
    delay_ms(1);
}

void LCD_Print(char *str)
{
    while (*str)
    {
        LCD_Data(*str++);
    }
}
```

Listing 7.2: LCD driver implementation (`lcd.c`)

7.2.1.3 Main Application

```
#include "TM4C123.h"
#include "lcd.h"



int main(void)
```

```
{

    LCD_Init();
    LCD_Clear();        // Ensure display is clear
    LCD_SetCursor(0,0); // Set cursor to beginning
    LCD_Print("ENCS4110 Lab");

    while(1)
    {
    }
}
```

Listing 7.3: Main application using LCD driver (`main.c`)

7.2.2 Code Explanation

Initialization Sequence The `LCD_Init()` function implements the complete 4-bit initialization:

1. Enables GPIO PORTB clock and configures pins as outputs
2. Waits 50 ms for LCD power-on stabilization
3. Sends `0x03` (upper nibble) three times with delays (8-bit mode reset)
4. Sends `0x02` (upper nibble) to switch to 4-bit mode
5. Sends configuration commands: `0x28` (4-bit, 2 lines), `0x0C` (display on), `0x06` (entry mode), `0x01` (clear)

Nibble Transmission The `LCD_SendNibble()` function:

- Masks out current data bits (PB4-PB7)
- Places the 4-bit nibble on PB4-PB7 (shifted left by 4)
- Generates enable pulse: delay → E high → delay → E low → delay

Command vs. Data

- `LCD_Command()`: Sets RS=0, sends upper nibble, sends lower nibble
- `LCD_Data()`: Sets RS=1, sends upper nibble, sends lower nibble

Cursor Positioning The `LCD_SetCursor(row, col)` function calculates the DDRAM address:

```
address = (row == 0) ? 0x80 + col : 0xC0 + col;
```

Then sends the address as a command.

String Printing The `LCD_Print(str)` function iterates through the string and sends each character using `LCD_Data()`.

7.2.3 Tasks

7.2.3.1 Task 1: Display Your Name and ID

Update the main program to display your name on the first line and your student ID on the second line of the LCD.

Requirements:

- Clear the display
- Set cursor to line 1, column 0
- Print your name (up to 16 characters)
- Set cursor to line 2, column 0
- Print your student ID

Hint:

```
LCD_Clear();
LCD_SetCursor(0, 0);  // Line 1
LCD_Print("Your Name");
LCD_SetCursor(1, 0);  // Line 2
LCD_Print("ID: 1234567");
```

7.2.3.2 Task 2: Button-Controlled Name Scrolling

Write a program that displays your name on the LCD and allows the user to scroll the text left or right using the two on-board push buttons (SW1 and SW2).

Requirements:

- Display your name on line 1
- Configure SW1 (PF4) and SW2 (PF0) with GPIO interrupts (falling edge, internal pull-up)
- When SW1 is pressed: Shift display left (command `0x18`)
- When SW2 is pressed: Shift display right (command `0x1C`)
- The display should not scroll automatically; only respond to button presses

7.2.3.3 Task 3: Bidirectional Continuous Scrolling

Write a program that displays your name on line 1 and your student ID on line 2, with continuous scrolling in opposite directions after a button press.

Requirements:

- Display your name on line 1 and ID on line 2
- Initially, the display is static (no scrolling)
- When SW1 is pressed, start continuous scrolling:
  - Line 1 scrolls right
  - Line 2 scrolls left
- Pressing SW1 again stops the scrolling
- Use a timer interrupt to handle the scrolling at a fixed interval (e.g., every 500 ms)

8 Analog-to-Digital Converter (ADC)

Learning Objectives

After completing this experiment, you will be able to:

- Understand analog-to-digital conversion principles including sampling, quantization, and resolution.
- Recognize the architecture of the TM4C123 ADC modules including dual 12-bit converters and shared input channels.
- Configure ADC sample sequencers (SS0-SS3) with different depths and trigger sources.
- Configure GPIO pins for analog input using AFSEL, AMSEL, and DEN registers.
- Implement software-triggered and hardware-triggered ADC conversions.
- Read and interpret ADC conversion results from FIFO registers.
- Convert digital ADC values to physical quantities such as voltage and temperature.
- Implement both polling and interrupt-driven ADC operation modes.
- Utilize hardware averaging and multi-channel sampling for improved measurement accuracy.

Experiment Overview

This experiment introduces analog-to-digital conversion using the TM4C123's dual 12-bit ADC modules. You will configure sample sequencers for automatic multi-channel data acquisition, implement both polling and interrupt-driven sampling modes, and interface with analog sensors such as potentiometers and the internal temperature sensor. By the end of this lab, you will understand how to convert continuous analog signals to discrete digital values, configure trigger sources for periodic sampling, and implement sensor interfacing applications that bridge the analog and digital domains.

8.1 Theoretical Background

8.1.1 Introduction to Analog-to-Digital Conversion

Analog-to-Digital Converters (ADCs) are essential components in embedded systems, enabling microcontrollers to interface with the analog world by converting continuous analog signals (such as voltage from sensors) into discrete digital values for processing. They are widely used in applications like temperature sensing, light intensity measurement, and audio signal processing.

8.1.1.1 ADC Resolution and Sampling

ADCs are characterized by their **resolution**, which indicates the number of discrete levels they can represent, typically expressed in bits. For instance, a 12-bit ADC can represent $2^{12} = 4096$ discrete levels. The resolution determines the smallest detectable change in the analog input, known as the **least significant bit (LSB)**. The voltage corresponding to one LSB is calculated as:

$$\text{LSB Voltage} = \frac{V_{\text{REFP}} - V_{\text{REFN}}}{2^{\text{Resolution}}} \tag{8.1}$$

where V_{REFP} is the positive reference voltage, V_{REFN} is the negative reference voltage, and Resolution is the number of bits.

8.1.2 TM4C123 ADC Architecture

The TM4C123GH6PM microcontroller features **two identical 12-bit ADC modules** (ADC0 and ADC1) that share 12 analog input channels. These modules provide high-precision conversion of continuous analog voltages to discrete digital numbers.

8.1.2.1 Key Features

The ADC modules offer the following capabilities:

- **Resolution**: 12-bit precision (0 to 4095 discrete levels)
- **Shared Input Channels**: 12 analog input channels (AIN0-AIN11) shared between both modules
- **Internal Temperature Sensor**: On-chip temperature measurement capability
- **Maximum Sample Rate**: 1 million samples per second (1 MSPS)
- **Reference Voltage**: VDDA = 3.3V (separate analog power supply)
- **Input Voltage Range**: 0V to 3.3V for single-ended inputs
- **Sample Sequencers**: Four programmable sequencers per module (SS0-SS3) with depths of 8, 4, 4, and 1 samples
- **Hardware Averaging**: Automatic averaging of up to 64 samples for noise reduction
- **Digital Comparators**: Eight digital comparators per module for threshold detection
- **DMA Support**: Dedicated μDMA channels for efficient data transfer with burst requests
- **Isolated Power**: Separate analog power and ground pins for improved signal integrity

8.1.2.2 ADC Input Channels

The TM4C123 provides 12 analog input channels (AIN0 to AIN11) that can be connected to various GPIO pins, shared between both ADC modules. The mapping is as follows:

Channel	AIN0	AIN1	AIN2	AIN3	AIN4	AIN5	AIN6	AIN7	AIN8	AIN9	AIN10	AIN11
Pin	PE3	PE2	PE1	PE0	PD3	PD2	PD1	PD0	PE5	PE4	PB4	PB5

Table 8.1: ADC Input Channel Pin Mapping

8.1.2.3 Sample Sequencers

Each ADC module has **four independent sample sequencers** that manage sampling control and data capture autonomously. They differ only in the number of samples and FIFO depth.

Sequencer	Number of Samples	FIFO Depth	Priority
SS0	8	8 entries	Highest
SS1	4	4 entries	High
SS2	4	4 entries	Low
SS3	1	1 entry	Lowest

Table 8.2: Sample Sequencer Capabilities and FIFO Depths

8.1.2.4 Trigger Sources

ADC conversions can be initiated by various trigger sources for flexibility:

- **Processor (Software)**: Manual trigger via ADCPSSI register
- **Analog Comparators**: Trigger on comparator output events
- **GPIO**: External signal on GPIO pins
- **General-Purpose Timers**: Periodic sampling synchronized with timer events
- **PWM Generators**: Synchronize sampling with PWM signals for motor control
- **Continuous**: Free-running continuous sampling mode

Both ADC modules can use independent or shared triggers. When shared, a programmable phase shifter allows delaying sampling by a phase angle for time-interleaved sampling to achieve higher effective rates.

8.1.3 Internal Temperature Sensor

The internal temperature sensor can be sampled by setting the TSn bit in the ADCSSCTLn register. Temperature (TEMP in °C) is calculated from the ADC reading (ADCCODE, 0 to 4095) as:

$$\text{TEMP} = 147.5 - \left(\frac{75 \times (V_{\text{REFP}} - V_{\text{REFN}}) \times \text{ADCCODE}}{4096} \right) \tag{8.2}$$

Note: This measures the microcontroller's die temperature, not ambient, and is useful for thermal management but may require calibration.

8.1.4 ADC Registers

The TM4C123 ADC modules are configured via registers. Refer to the datasheet for details. Key registers include:

RCGCADC — ADC Run Mode Clock Gating Control

Enables/disables clock for ADC modules. Bits correspond to modules (1 = enable, 0 = disable).

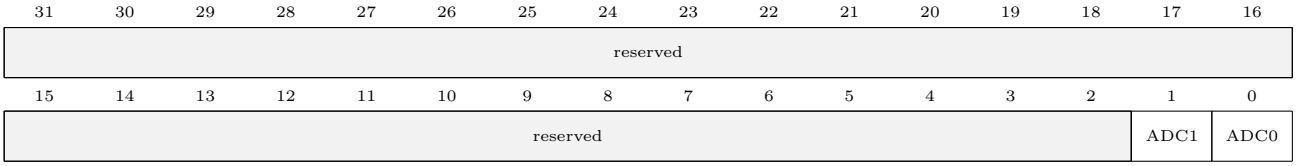


Figure 8.1: RCGCADC Register — ADC Run Mode Clock Gating Control

ADCACTSS — ADC Active Sample Sequencer

Enables/disables sample sequencers. Bits correspond to sequencers (1 = enable, 0 = disable). BUSY indicates ADC status (1 = busy, 0 = idle).

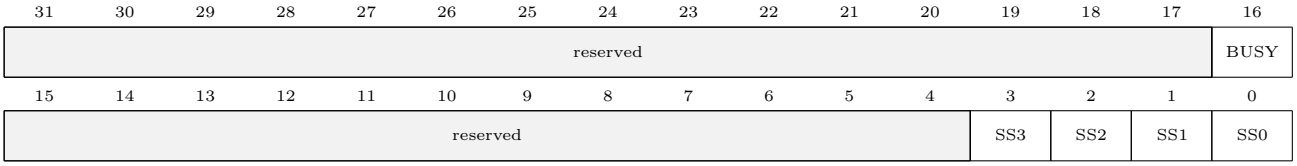


Figure 8.2: ADCACTSS Register — ADC Active Sample Sequencer

ADCSSMUXn — ADC Sample Sequence Input Multiplexer Select

Selects input channels for each sample in a sequencer. Each 4-bit field (MUX0-MUX7) specifies the channel (0-11 for AIN0-AIN11).

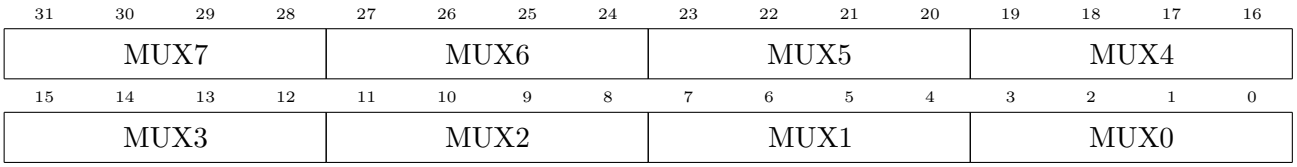


Figure 8.3: ADCSSMUXn Register — ADC Sample Sequence Input Multiplexer Select

ADCSSCTLn — ADC Sample Sequence Control

Configures each sample in a sequence. The END bit must be set for the final sample. This 32-bit register handles up to eight samples.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
TS7	IE7	END7	D7	TS6	IE6	END6	D6	TS5	IE5	END5	D5	TS4	IE4	END4	D4
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TS3	IE3	END3	D3	TS2	IE2	END2	D2	TS1	IE1	END1	D1	TS0	IE0	END0	D0

Figure 8.4: ADCSSCTLn Register — ADC Sample Sequence Control

Where:
- TSx: Temperature sensor select (1 = sample temperature sensor)
- IEx: Interrupt enable (1 = enable interrupt on sample completion)
- ENDx: End of sequence (1 = last sample in sequence)
- Dx: Differential select (1 = differential input, 0 = single-ended)
- x: Sample index (0-7)

ADCEMUX — ADC Event Multiplexer Select

Selects trigger source for each sample sequencer. Each 4-bit field (EM0-EM3) specifies the trigger source.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EM3				EM2				EM1				EM0			

Figure 8.5: ADCEMUX Register — ADC Event Multiplexer Select

The EMx field specifies the trigger source for sample sequencer x:

Value	Trigger Source
0x0	Processor (software trigger via ADCPSSI register)
0x1	Analog Comparator 0
0x2	Analog Comparator 1
0x3	Reserved
0x4	External GPIO (connected to GPIO interrupt)
0x5	Timer (requires TnOTE bit set in GPTMCTL register)
0x6	PWM Generator 0
0x7	PWM Generator 1
0x8	PWM Generator 2
0x9	PWM Generator 3
0xA–0xE	Reserved
0xF	Continuous (always sampling)

Table 8.3: ADC Event Multiplexer Trigger Sources

ADCPSSI — ADC Processor Sample Sequence Initiate

Initiates sampling for the specified sample sequencer when triggered by the processor.

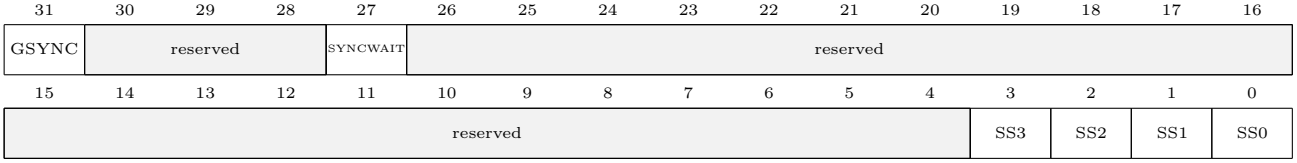


Figure 8.6: ADCPSSI Register — ADC Processor Sample Sequence Initiate

Where:

- **SSx**: Sample Sequencer x Initiate (write 1 to trigger conversion, self-clearing)
- **SYNCWAIT**: Synchronize Wait (1 = wait for synchronization signal before initiating)
- **GSYNC**: Global Synchronize (1 = generate synchronization signal for multiple ADC modules)

Note: The SSx bits are write-only and automatically clear after initiating the conversion. For synchronized sampling across modules, set GSYNC in one module to trigger all modules with SYNCWAIT enabled.

ADCSSFIFOn — ADC Sample Sequence FIFO

Holds the conversion results for the specified sample sequencer. Each read retrieves the next sample from the FIFO.

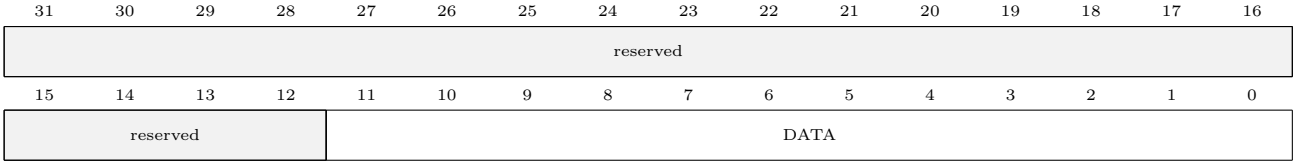


Figure 8.7: ADCSSFIFOn Register — ADC Sample Sequence FIFO

Where:

- DATA: 12-bit conversion result (0 to 4095)
- reserved: Unused bits

You can read from the ADCSSFIFOn register to retrieve conversion results. Each read operation removes the oldest sample from the FIFO. Ensure that the FIFO is not empty before reading to avoid invalid data. You can check the FIFO status using the ADCSSSTAT register (refer to the datasheet for details).

ADCRIS — ADC Raw Interrupt Status

Holds the raw interrupt status for each sample sequencer.

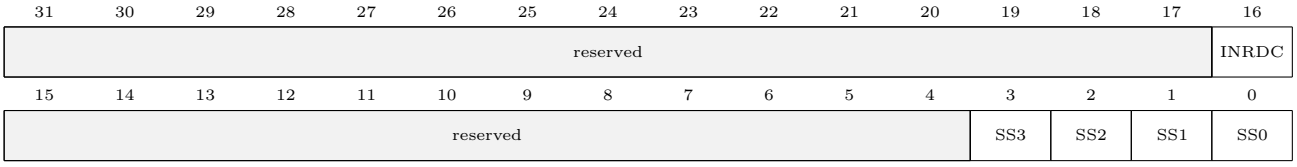


Figure 8.8: ADCRIS Register — ADC Raw Interrupt Status

Where:
- SSx: Sample Sequencer x Interrupt Status (1 = interrupt pending, 0 = no interrupt)
- INRDC: Digital Comparator Raw Interrupt Status
- reserved: Unused bits

ADCIM — ADC Interrupt Mask

Masks/unmasks interrupts for each sample sequencer.

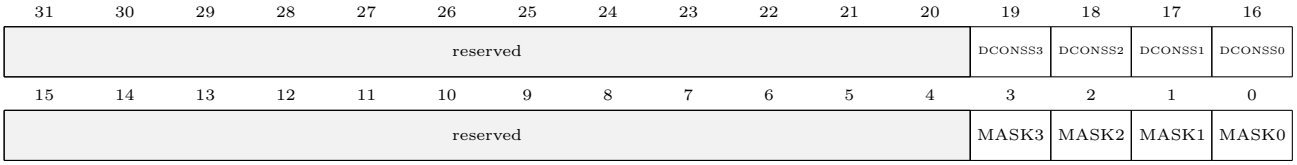


Figure 8.9: ADCIM Register — ADC Interrupt Mask

Where:
- MASKx: Sample Sequencer x Interrupt Mask (1 = unmask interrupt, 0 = mask interrupt)
- DCONSSx: Digital Comparator Interrupt Mask for Sample Sequencer x (1 = unmask comparator interrupt, 0 = mask comparator interrupt)
- x: Sample sequencer index (0-3)
- reserved: Unused bits

ADCISC — ADC Interrupt Status and Clear

Holds and clears the interrupt status for each sample sequencer.

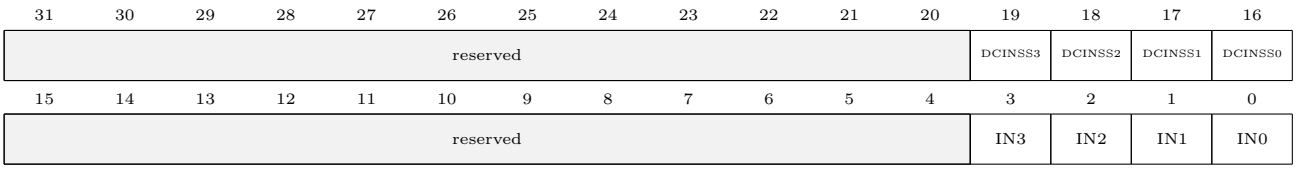


Figure 8.10: ADCISC Register — ADC Interrupt Status and Clear

Where:
- SSx: Sample Sequencer x Interrupt Clear (write 1 to clear interrupt)
- DCINSSx: Digital Comparator Interrupt Clear for Sample Sequencer x (write 1 to clear comparator interrupt)
- reserved: Unused bits

8.1.5 GPIO Registers

To configure GPIO pins for ADC input, you will need to set the appropriate GPIO registers. Below are the key registers involved in configuring GPIO pins for analog input:

GPIODEN — GPIO Digital Enable

Enables/disables digital functionality for GPIO pins. To use a pin as an analog input, its corresponding bit in this register must be cleared (0 = disable digital function).

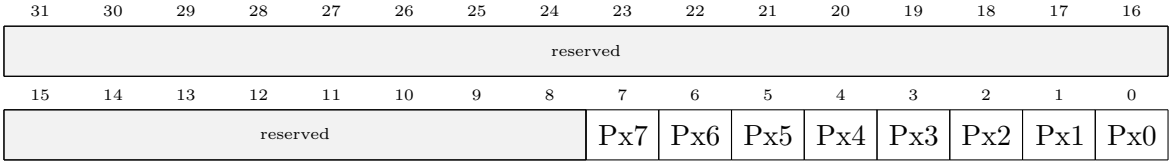


Figure 8.11: GPIODEN Register (Port x) — Digital Enable

Where:

- Pxn: Digital Enable for Pin n of Port x (1 = enable digital function, 0 = disable digital function)
- reserved: Unused bits
- x: Port letter (A-F)
- n: Pin number (0-7)

Note: To configure a pin for analog input, clear its corresponding bit in the GPIODEN register.

GPIOAFSEL — GPIO Alternate Function Select

Selects alternate functions for GPIO pins. To use a pin for ADC input, its corresponding bit in this register must be set (1 = alternate function enabled).

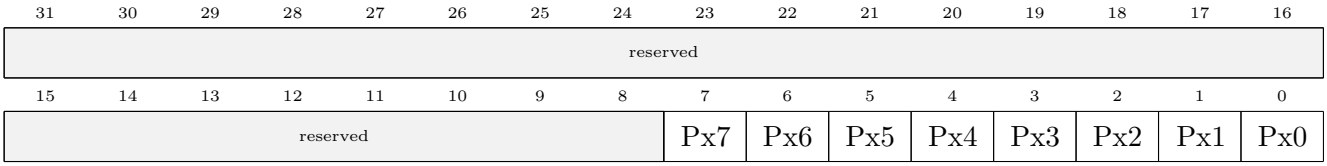


Figure 8.12: GPIOAFSEL Register (Port x) — Alternate Function Select

Where:

- Pxn: Alternate Function Select for Pin n of Port x (1 = enable alternate function, 0 = GPIO function)
- reserved: Unused bits
- x: Port letter (A-F)
- n: Pin number (0-7)

Note: To configure a pin for ADC input, set its corresponding bit in the GPIOAFSEL register.

GPIOAMSEL — GPIO Analog Mode Select

Enables/disables analog functionality for GPIO pins. To use a pin as an analog input, its corresponding bit in this register must be set (1 = enable analog function).

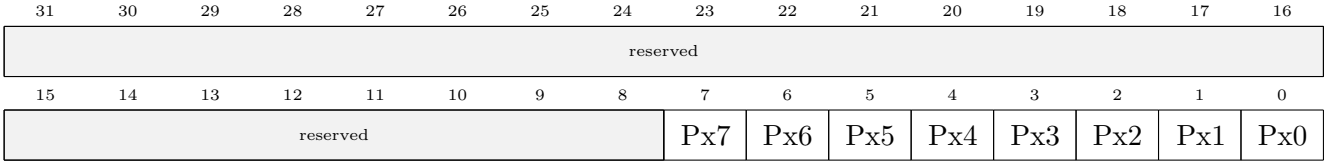


Figure 8.13: GPIOAMSEL Register (Port x) — Analog Mode Select

Where:

- Pxn: Analog Mode Select for Pin n of Port x (1 = enable analog function, 0 = disable analog function)
- reserved: Unused bits
- x: Port letter (A-F)
- n: Pin number (0-7)

Note: To configure a pin for ADC input, set its corresponding bit in the GPIOAMSEL register.

8.1.6 Configuration Steps

Configuring a GPIO pin for ADC input on the TM4C123 microcontroller involves three stages:

8.1.6.1 GPIO Pin Configuration

To prepare a GPIO pin for analog input:

1. Enable the clock for the corresponding GPIO port by setting the appropriate bit in the `RCGCGPIO` register.
2. Enable the pin's alternate function using the `GPIOAFSEL` register.
3. Disable its digital function by clearing the pin's bit in the `GPIODEN` register.
4. Enable the analog function by setting the pin's bit in the `GPIOAMSEL` register.
5. Set the pin direction as input by clearing its bit in the `GPIODIR` register.

8.1.6.2 ADC Sampling Sequencer Setup

After the pin is configured, the ADC module and sampling sequencer must be set up:

1. Enable the ADC clock by setting the correct bit in the `RCGCADC` register.
2. Disable the chosen sample sequencer by clearing its bit in the `ADCACTSS` register.
3. Select the input channel(s) by configuring the `ADCSSMUXn` register.
4. Set the control options for each sample (e.g., END, IE) in the `ADCSSCTLn` register, ensuring the END bit is set for the final sample.
5. Choose the trigger source by configuring the `ADCEMUX` register.
6. Re-enable the sequencer by setting its bit in `ADCACTSS`.
7. If processor-triggered sampling is used, start a conversion using the `ADCPSSI` register.

8.1.6.3 Interrupt Configuration

If interrupts are required:

1. Enable interrupts for the selected sequencer by setting its bit in the `ADCIM` register.
2. Enable the corresponding interrupt in the NVIC by setting the appropriate bit in the `NVIC_ISER` register.
3. In the ISR, read the conversion result from the `ADCSSFIFO` register.
4. Clear the interrupt flag by writing to the appropriate bit in the `ADCISC` register.

8.2 Procedure

This section demonstrates how to configure and use the ADC module on the TM4C123 microcontroller using both polling and interrupt-driven sampling methods. A potentiometer is used as a controllable analog input source by wiring its terminals to 3.3 V and ground, with the adjustable rotor connected to an ADC input such as AIN0 (PE3). Rotating the potentiometer varies the rotor voltage between 0 V and 3.3 V, allowing you to generate different analog levels and observe the corresponding ADC conversion results.

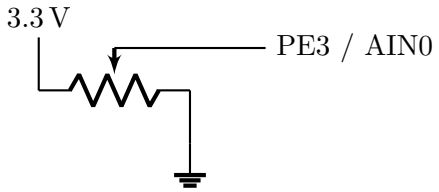


Figure 8.14: Potentiometer Voltage Divider for ADC Input

8.2.1 Example 1: Single ADC Channel Reading (Polling Method)

In this example, you will configure the ADC to read a single analog input channel (AIN0) using sample sequencer 3 (SS3) and turn on an LED if the voltage exceeds a certain threshold.

```
#include "TM4C123.h"
#define GREEN_LED 1 << 3
volatile unsigned int adc_value;
int main(void)
{
    SYSCTL->RCGCGPIO |= (1 << 4); // Enable Clock to GPIOE or PE3/AN0
    SYSCTL->RCGCADC |= (1 << 0);  // AD0 clock enable

    GPIOE->AFSEL |= (1 << 3);  // enable alternate function
    GPIOE->DEN &= ~(1 << 3);   // disable digital function
    GPIOE->AMSEL |= (1 << 3);  // enable analog function

    ADC0->ACTSS &= ~(1 << 3);            // disable SS3 during configuration
    ADC0->EMUX &= ~0xF000;                // Software Trigger
    ADC0->SSMUX3 = 0;                    // get input from channel 0
    ADC0->SSCTL3 |= (1 << 1) | (1 << 2); // take one sample at a time, set flag at
    1st sample
    ADC0->ACTSS |= (1 << 3);             // enable ADC0 sequencer 3

    SYSCTL->RCGCGPIO |= 0x20; // turn on bus clock for GPIOF
    GPIOF->DIR |= GREEN_LED;       // set GREEN pin as a digital output pin
    GPIOF->DEN |= GREEN_LED;       // Enable PF3 pin as a digital pin

    while (1)
    {
        ADC0->PSSI |= (1 << 3);            // initiate SS3 conversion
        while ((ADC0->RIS & 8) == 0);      // Wait untill sample conversion completed
        adc_value = ADC0->SSFIFO3;         // read adc coversion result from SS3 FIFO
        ADC0->ISC = 8;                     // clear coversion clear flag bit

        if (adc_value >= 2048)
            GPIOF->DATA |= GREEN_LED; // turn on green LED
        else if (adc_value < 2048)
            GPIOF->DATA &= ~GREEN_LED; // turn off green LED
    }
}
```

Listing 8.1: Single ADC Channel Reading Example

8.2.2 Example 2: Reading Internal Temperature Sensor

In this example, you will configure the ADC to read the internal temperature sensor using sample sequencer 3 (SS3) and display the temperature value on the debugger using the Watch window.

1. **Start a debug session:** After building the program successfully, start a debug session in Keil uVision IDE (Ctrl + F5).
2. **Add variables to the Watch window:** by selecting the variable name in the code, right-clicking, and choosing "Add to Watch" or manually entering the variable name in the Watch window.
3. **Run the program:** Use the "Run" button (F5) to execute the program. The Watch window will update with the current values of the monitored variables.

```c
#include "TM4C123.h"

volatile unsigned int adc_value;
volatile float temp;

void ADC0SS3_Handler(void)
{
    adc_value = ADC0->SSFIFO3 & 0xFFF; // Read 12-bit ADC value from FIFO
    temp = 147.5 - (75 * 3.3f * adc_value)/ 4096; // Convert to temperature in
    Celsius
    ADC0->ISC = 8; // Clear SS3 interrupt flag
}

int main(void)
{
    SYSCTL->RCGCGPIO |= (1 << 4); // Enable clock to Port E
    SYSCTL->RCGCADC |= (1 << 0);  // Enable clock to ADC0

    GPIOE->AFSEL |= (1 << 3);  // Enable alternate function on PE3
    GPIOE->DEN &= ~(1 << 3);   // Disable digital I/O on PE3
    GPIOE->AMSEL |= (1 << 3);  // Enable analog input on PE3

    ADC0->ACTSS &= ~(1 << 3);             // Disable SS3 during configuration
    ADC0->EMUX |= 0xF000;                 // Set SS3 to continuous sampling mode
    ADC0->SSMUX3 = 0;                     // Configure SS3 to sample AIN0 (PE3)
    ADC0->SSCTL3 |= (1 << 1) | (1 << 2) | (1 << 3);   // Set end of sequence, enable
    interrupt, enable temperature sensor

    ADC0->IM |= (1 << 3);        // Enable SS3 interrupt mask
    NVIC_EnableIRQ(ADC0SS3_IRQn); // Enable ADC0 SS3 interrupt in NVIC

    ADC0->ACTSS |= (1 << 3);  // Enable SS3

    while (1) {} // Wait for interrupts
}
```

Listing 8.2: Single ADC Channel Reading with Interrupts Example

8.2.3 Tasks

8.2.3.1 Task 1: Timer-Based Temperature Measurement

Modify Example 2 so that the ADC sampling is triggered by a timer every second, instead of using continuous sampling mode. Connect an analog temperature sensor (e.g., TMP36) to an ADC input channel (such as AIN0 on PE3). Configure the ADC to read the sensor voltage on each timer trigger and calculate the corresponding temperature. Monitor the temperature values in real time by adding the variable `temp` to the Watch window in the debugger.

8.2.3.2 Task 2: Comparing Two ADC Channels Using Dual Modules

Extend Example 2 to read two different analog inputs using separate ADC modules: use **ADC0 module** to sample AIN0 (PE3) and **ADC1 module** to sample AIN1 (PE2), both with sample sequencer 3 (SS3). Configure both ADC modules to use interrupts so that each conversion triggers its respective ISR. In the interrupt handlers, compare the two ADC values and use the result to control LEDs on GPIOF: for example, turn on the green LED if ADC0's value is higher, or turn on the red LED if ADC1's value is higher. This demonstrates simultaneous sampling using both ADC modules. Display both ADC values in the debugger Watch window to observe the comparison in real time.

9 Universal Asynchronous Receiver-Transmitter (UART)

Learning Objectives

After completing this experiment, you will be able to:

- Understand the principles of asynchronous serial communication including start bits, stop bits, and data frames.
- Configure UART baud rate using integer and fractional divisors for precise timing.
- Configure GPIO pins for UART alternate function using AFSEL, PCTL, and DEN registers.
- Implement basic UART transmission and reception functions in C.
- Send and receive strings via serial communication using polling methods.
- Interface with a PC terminal emulator for debugging and data visualization.
- Calculate and verify baud rate divisors for different system clock frequencies.
- Understand UART FIFO buffering and status flag operations.

Experiment Overview

This experiment introduces serial communication using the TM4C123's UART modules. You will configure UART0 for 115200 baud operation on pins PA0 (RX) and PA1 (TX), implement character and string transmission functions, and create an echo server that receives data from a PC terminal and echoes it back. By the end of this lab, you will understand how asynchronous serial communication enables data exchange between embedded systems and external devices, forming the foundation for debugging, sensor interfacing, and wireless communication modules.

9.1 Theoretical Background

9.1.1 Introduction to Serial Communication

Serial communication is a method of transmitting data one bit at a time over a single communication line, as opposed to parallel communication which sends multiple bits simultaneously. It is widely used in embedded systems due to its simplicity, low pin count, and ability to communicate over long distances.

9.1.1.1 Asynchronous vs Synchronous Communication

Serial communication can be classified into two categories:

- **Asynchronous (UART)**: No shared clock signal; timing is established by agreed-upon baud rate. Uses start and stop bits for synchronization.
- **Synchronous (SPI, I^2C)**: Dedicated clock line synchronizes transmitter and receiver. More efficient but requires additional pin.

UART (Universal Asynchronous Receiver-Transmitter) is an asynchronous protocol that requires only two wires: TX (transmit) and RX (receive). It is commonly used for debugging, GPS modules, Bluetooth communication, and PC interfacing.

9.1.2 UART Frame Structure

A UART transmission consists of a **data frame** that includes:

1. **Start Bit**: Logic low (0) that signals the beginning of transmission
2. **Data Bits**: 5 to 9 bits of actual data (typically 8 bits)
3. **Parity Bit**: Optional error-checking bit (even, odd, or none)
4. **Stop Bits**: Logic high (1) that signals the end of transmission (1, 1.5, or 2 bits)

The most common configuration is **8N1**: 8 data bits, No parity, 1 stop bit.

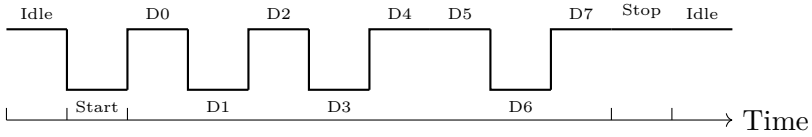


Figure 9.1: UART Frame Structure (8N1 Format)

9.1.3 Baud Rate Generation

The **baud rate** defines the number of bits transmitted per second, measured in bits per second (bps). Common baud rates include 9600, 19200, 38400, 57600, and 115200 bps. Both transmitter and receiver must use the same baud rate for successful communication.

9.1.3.1 Baud Rate Divisor Calculation

The TM4C123 UART uses a **22-bit baud rate divisor (BRD)** consisting of a 16-bit integer part (BRDI) and a 6-bit fractional part (BRDF). This fractional divisor allows the UART to generate all standard baud rates with high precision.

The baud rate divisor is calculated from the system clock frequency as:

$$\text{BRD} = \text{BRDI} + \text{BRDF} = \frac{\text{UARTSysClk}}{\text{ClkDiv} \times \text{BaudRate}} \tag{9.1}$$

where:

- **UARTSysClk**: System clock connected to UART (typically 50 MHz)
- **ClkDiv**: Clock divider (16 if HSE = 0 in UARTCTL, or 8 if HSE = 1)
- **BaudRate**: Desired baud rate (e.g., 115200 bps)

The integer part (BRDI) is loaded into the `UARTIBRD` register, and the fractional part is calculated and loaded into the `UARTFBRD` register using:

$$\text{UARTFBRD[DIVFRAC]} = \text{integer}(\text{BRDF} \times 64 + 0.5) \tag{9.2}$$

The multiplication by 64 and addition of 0.5 accounts for the 6-bit fractional representation and rounding errors.

9.1.3.2 Baud Rate Generation Process

The UART generates an internal baud rate reference clock at either 8x or 16x the target baud rate (called Baud8 or Baud16), depending on the HSE bit setting. This reference clock is then divided by 8 or 16 to generate the actual transmit clock and is used for error detection during receive operations.

9.1.3.3 Calculation Example

Example: For 115200 baud with 50 MHz system clock and ClkDiv = 16 (default, HSE = 0):

$$\begin{aligned}
\text{BRD} &= \frac{50{,}000{,}000}{16 \times 115{,}200} = \frac{50{,}000{,}000}{1{,}843{,}200} = 27.1267361 \\
\text{BRDI} &= 27 \\
\text{BRDF} &= 0.1267361 \\
\text{UARTFBRD[DIVFRAC]} &= \text{integer}(0.1267361 \times 64 + 0.5) \\
&= \text{integer}(8.111 + 0.5) = \text{integer}(8.611) = 8
\end{aligned}$$

Therefore: `UARTIBRD = 27` and `UARTFBRD = 8`
The actual baud rate achieved can be verified:

$$\text{Actual Baud Rate} = \frac{50{,}000{,}000}{16 \times (27 + 8/64)} = \frac{50{,}000{,}000}{16 \times 27.125} = 115{,}207.4 \text{ bps} \tag{9.3}$$

This gives an error of only 0.0064%, well within acceptable tolerance for reliable communication.

9.1.4 TM4C123 UART Modules

The TM4C123GH6PM provides **eight UART modules** (UART0-UART7) with the following features:

- **Programmable Baud Rate**: Generated from system clock with 16-bit divisor
- **Data Format**: 5 to 9 data bits, optional parity (even, odd, stick), 1 or 2 stop bits
- **FIFO Buffers**: 16-byte transmit and receive FIFOs for reduced interrupt overhead
- **Full-Duplex Operation**: Simultaneous transmission and reception
- **Error Detection**: Framing, parity, overrun, and break detection
- **Interrupt Generation**: Configurable interrupts for TX, RX, and error conditions
- **DMA Support**: μDMA channels for high-speed data transfer
- **IrDA and ISO 7816**: Optional infrared and smart card support
- **9-Bit Mode**: For RS-485 multi-drop networks

9.1.4.1 FIFO Operation

Each UART module contains two 16x8 FIFOs: one for transmit and one for receive. Both FIFOs are accessed through the `UARTDR` register. Read operations return a 12-bit value consisting of 8 data bits and 4 error flags, while write operations place 8-bit data into the transmit FIFO.

FIFO Status Monitoring:

FIFO status can be monitored using two registers:

1. **UARTFR (Flag Register)**: Contains empty and full flags
   - **TXFE** (bit 7): Transmit FIFO Empty
   - **TXFF** (bit 5): Transmit FIFO Full
   - **RXFE** (bit 4): Receive FIFO Empty
   - **RXFF** (bit 6): Receive FIFO Full

9.1.4.2 UART Pin Mapping

Each UART module is connected to specific GPIO pins. The table below shows the default pin assignments:

UART Module	TX Pin	RX Pin
UART0	PA1	PA0
UART1	PB1	PB0
UART2	PD7	PD6
UART3	PC7	PC6
UART4	PC5	PC4
UART5	PE5	PE4
UART6	PD5	PD4
UART7	PE1	PE0

Table 9.1: UART Pin Assignments

Note: UART0 on PA0/PA1 is commonly used because it connects to the LaunchPad's on-board USB-to-Serial converter, allowing direct PC communication via USB cable.

9.1.5 UART Registers

The TM4C123 UART modules are configured through memory-mapped registers. Key registers include:

RCGCUART — UART Run Mode Clock Gating Control

Enables clock for UART modules. Each bit corresponds to a UART module (1 = enable, 0 = disable).

Figure 9.2: RCGCUART Register — UART Run Mode Clock Gating Control

UARTCTL — UART Control

Configures UART operation including enable, transmit enable, and receive enable.

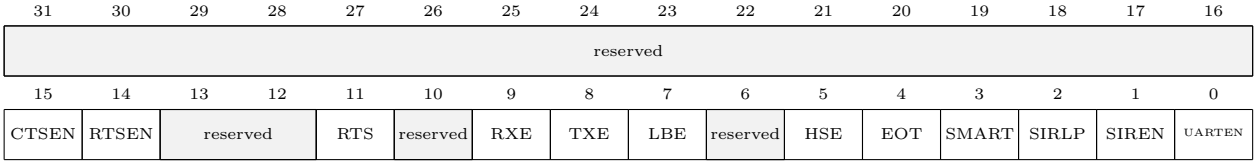


Figure 9.3: UARTCTL Register — UART Control

Where:

- **UARTEN** (bit 0): UART Enable (1 = enable, 0 = disable)
- **TXE** (bit 8): Transmit Enable (1 = enable transmitter)
- **RXE** (bit 9): Receive Enable (1 = enable receiver)
- **HSE** (bit 5): High-Speed Enable (1 = baud rate clock = SysClk/8, 0 = SysClk/16)

UARTIBRD — UART Integer Baud-Rate Divisor

Contains the integer part of the baud rate divisor.

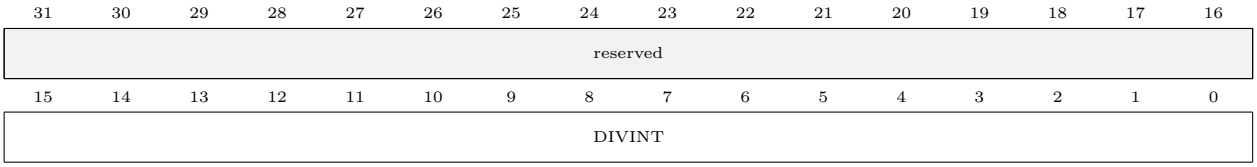


Figure 9.4: UARTIBRD Register — UART Integer Baud-Rate Divisor

Where:

- **DIVINT**: Integer baud rate divisor (0-65535)

UARTFBRD — UART Fractional Baud-Rate Divisor

Contains the fractional part of the baud rate divisor.

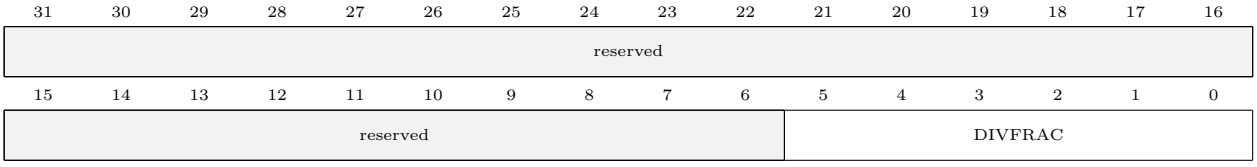


Figure 9.5: UARTFBRD Register — UART Fractional Baud-Rate Divisor

Where:

- **DIVFRAC**: Fractional baud rate divisor (0-63)

UARTLCRH — UART Line Control

Configures data format including word length, FIFO enable, parity, and stop bits.

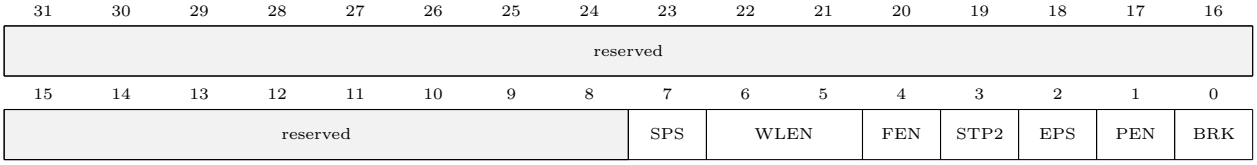


Figure 9.6: UARTLCRH Register — UART Line Control

Where:

- **WLEN** (bits 6-5): Word Length (0x3 = 8 bits, 0x2 = 7 bits, 0x1 = 6 bits, 0x0 = 5 bits)
- **FEN** (bit 4): FIFO Enable (1 = enable 16-byte FIFOs, 0 = disable)
- **STP2** (bit 3): Two Stop Bits (1 = two stop bits, 0 = one stop bit)
- **EPS** (bit 2): Even Parity Select (1 = even parity, 0 = odd parity)
- **PEN** (bit 1): Parity Enable (1 = enable parity, 0 = no parity)
- **BRK** (bit 0): Send Break (1 = send break condition)

Common Configuration (8N1 with FIFO):

- WLEN = 0x3 (8 bits)
- FEN = 1 (FIFO enabled)
- STP2 = 0 (one stop bit)
- PEN = 0 (no parity)

Result: `UARTLCRH = 0x60` (0b01100000)

UARTCC — UART Clock Configuration

Selects the clock source for UART baud rate generation.

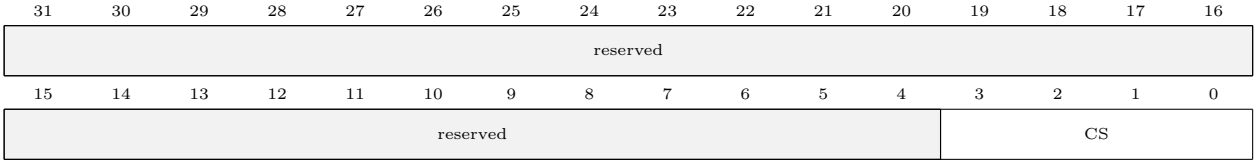


Figure 9.7: UARTCC Register — UART Clock Configuration

Where:

- **CS** (bits 3-0): Clock Source
  - 0x0 = System clock (default, recommended)
  - 0x5 = PIOSC (16 MHz internal oscillator)

UARTFR — UART Flag Register

Contains status flags for transmit/receive operations.

Figure 9.8: UARTFR Register — UART Flag Register

Where:

- **TXFE** (bit 7): Transmit FIFO Empty (1 = empty, 0 = not empty)
- **RXFF** (bit 6): Receive FIFO Full (1 = full, 0 = not full)
- **TXFF** (bit 5): Transmit FIFO Full (1 = full, 0 = not full)
- **RXFE** (bit 4): Receive FIFO Empty (1 = empty, 0 = not empty)
- **BUSY** (bit 3): UART Busy (1 = transmitting, 0 = idle)

Usage:

- Before transmitting: Wait while `TXFF = 1` (transmit FIFO full)
- Before receiving: Wait while `RXFE = 1` (receive FIFO empty)

UARTDR — UART Data Register

Contains the data to transmit or received data. Reading removes data from RX FIFO; writing adds to TX FIFO.

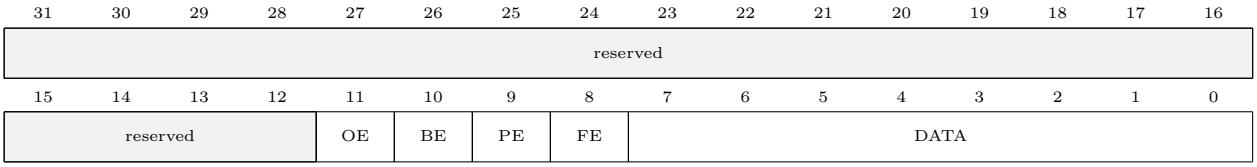


Figure 9.9: UARTDR Register — UART Data Register

Where: Data is contained in the lower 8 bits. The upper 4 bits indicate error flags:

- **FE** (bit 3): Framing Error
- **PE** (bit 2): Parity Error
- **BE** (bit 1): Break Error
- **OE** (bit 0): Overrun Error
- **DATA** (bits 7-0): Received or Transmit Data

9.2 Procedure

9.2.1 Configuration Steps

Configuring UART0 on the TM4C123 microcontroller involves two stages:

9.2.1.1 GPIO Pin Configuration

To prepare GPIO pins for UART function:

1. Enable the clock for UART0 module and GPIO Port A by setting the appropriate bits in the `RCGCUART` and `RCGCGPIO` registers.
2. Enable alternate function on PA0 (RX) and PA1 (TX) using the `GPIOAFSEL` register.
3. Configure the UART function encoding in the `GPIOPCTL` register (encoding = 1 for UART).
4. Enable digital function for PA0 and PA1 pins by setting bits in the `GPIODEN` register.

9.2.1.2 UART Module Setup

After the pins are configured, the UART module must be initialized:

1. Disable the UART by clearing all bits in the `UARTCTL` register to prevent undefined behavior during configuration.
2. Calculate the baud rate divisor values. For 115200 baud at 50 MHz: BRD = 50,000,000 / (16 x 115,200) = 27.1267361, giving IBRD = 27 and FBRD = 8.
3. Write the integer baud rate divisor to the `UARTIBRD` register.
4. Write the fractional baud rate divisor to the `UARTFBRD` register.
5. Configure the line control register (`UARTLCRH`) for 8 data bits, no parity, one stop bit, and FIFO enable (value = 0x60). Writing to this register latches the baud rate configuration.
6. Select the clock source by writing to the `UARTCC` register (0 = system clock).
7. Enable the UART, transmitter, and receiver by setting UARTEN, TXE, and RXE bits in the `UARTCTL` register.

9.2.2 UART0 Initialization and Code Implementation

The provided code includes three files demonstrating complete UART functionality:

- `uart.h`: Function prototypes and pin definitions
- `uart.c`: Implementation of UART initialization, read, and write functions
- `main.c`: Echo server application

UART Header File (`uart.h`)

```c
#ifndef UART_H
#define UART_H

#include "TM4C123.h"  // Or your MCU's main header

#define U0_TX ( 1 << 1 )
#define U0_RX ( 1 << 0 )

#define MAX_STR_LEN 50

void UART0_WriteChar(char c);
void UART0_WriteString(char *str);

char UART0_ReadChar();
void UART0_ReadString(char *buffer, int maxLen);

void UART0_Init();



#endif // UART_H
```

Listing 9.1: UART Header File

UART Implementation File (`uart.c`)

```c
#include "uart.h"

void UART0_WriteChar(char c) {
    while ((UART0->FR & (1 << 5)) != 0); // Wait until TX FIFO is not full (TXFF bit
    5)
    UART0->DR = c; // Write character to Data Register
}

void UART0_WriteString(char *str) {
    while (*str) { // Loop through each character until null terminator
        UART0_WriteChar(*(str++)); // Send each character
    }
}

void UART0_Init() {
    SYSCTL->RCGCUART |= (1 << 0); // Enable clock to UART0 module
    SYSCTL->RCGCGPIO |= (1 << 0); // Enable clock to GPIO Port A

    GPIOA->AFSEL |= U0_RX | U0_TX; // Enable alternate function on PA0 (RX) and PA1
    (TX)
```

```
    GPIOA->PCTL |= (1 << 0) | (1 << 4); // Configure PA0 and PA1 for UART (U0RX/U0TX)
    GPIOA->DEN |= U0_RX | U0_TX; // Enable digital function for PA0 and PA1

    UART0->CTL = 0; // Disable UART0 while configuring
    UART0->IBRD = 27; // Set integer baud rate divisor for 115200 baud @ 50MHz
    UART0->FBRD = 8; // Set fractional baud rate divisor
    UART0->LCRH = 0x60; // 8 data bits, no parity, one stop bit, FIFOs enabled
    UART0->CC = 0; // Use system clock for UART
    UART0->CTL = (1 << 0) | (1 << 8) | (1 << 9); // Enable UART0, TX and RX
}

char UART0_ReadChar(void) {
    while (UART0->FR & (1 << 4)); // Wait while RX FIFO empty (RXFE bit 4)
    return (char)(UART0->DR & 0xFF); // Return received character
}

void UART0_ReadString(char *buffer, int maxLen) {
    int i = 0;
    char c;

    while (i < (maxLen - 1)) { // Leave space for null terminator
        c = UART0_ReadChar();
        if (c == '\r' || c == '\n') { // End of input
            break;
        }
        buffer[i++] = c;
    }

    buffer[i] = '\0'; // Null terminate the string
}
```

Listing 9.2: UART Implementation

Echo Server Main Program (`main.c`)

```
#include <stdio.h>
#include "TM4C123.h"
#include "uart.h"

int main(void) {
    UART0_Init();
    UART0_WriteString("Hello World!\r\n");  // Send greeting with newline

    while (1) {
    char buff[MAX_STR_LEN];
    UART0_ReadString(buff, MAX_STR_LEN);

    UART0_WriteString("Received: ");
    UART0_WriteString(buff);
    UART0_WriteString("\r\n");
    }
}
```

Listing 9.3: Echo Server Main Program