NXP Kinetis K64F
• Cortex M4 CPU
• 32-bit architecture.
• Harvard based architecture
• Little endian
• 256KB RAM
• 1024KB Flash ROM

Power
• DC power to run CPU and I/O circuit. 3.3V or 5V DC
• 3.3V for digital and analog pins.
• Use a voltage divider if you have 5V input (i.e. Input from Arduino)
• 25mA maximum per pin drive current

Oscillator (Clock)
• Periodic square wave for clocking the CPU.
• Has an internal low-quality oscillator.
• Permits use of an external high-quality oscillator
• 32kHz and 4MHz on internal
• 32kHz to 48MHz external
• With PLL (phase-locked loop), CPU clock up to 120MHz

Why "slow" clock speed?
• Keeps costs down
• Saves power; reduces need for cooling
• Fast enough for many purposes
• Reduces emitted radio interference

32-Bit Architecture
• "x-bit" architecture
Datapath, memory address, registers, address buses, and integer size are all 32-bits.
• K64F has a 32-bit architecture
- Integer size: 32bit -> 4 bytes
- Memory address width: 32 bit -> 2^32
- Max addressable memory space: 4GB

Memory Types
• Registers
• RAM (Random Access Memory)
- Volatile
- Stores runtime program and/or data
• ROM (Read Only Memory)
- Nonvolatile
- Stores program code and static data
• EEPROM
- Electrically Erasable Programmable ROM
- Yet, Needs a dedicated external circuit (often built in) to write or erase data
• Flash ROM
Erasable directly in the system

Data & Address
Memory can be visualized as a bunch of sequential spaces
• Each space has a unique address that is used to refer the location.

BUS Architecture
• System Bus: Address, data, command lines.
• 32-bit architecture: Data line width = address line width = 32 bit

Read and Write Operations
Two main memory operations: read and write.

Write Example
Write data from a CPU register to a memory location.
1. The address is placed on the address bus
2. Data is placed on the data bus
3. A write command is issued

Read Example
Read data from memory to a CPU register.
*Data bus is a "two-way" path; data moves from memory during a read operation.
1. The address is placed on the address bus
2. A read command is issued
3. A copy of the data is placed in the data bus and shifted into the data register.

Memory Model
• Von Neuman Architecture
• Harvard Architecture
• Modified Harvard Architecture

Von Neuman Architecture
A single memory space for both program code and data.
Simple; Flexible in address space allocation

Harvard Architecture
Separate data and program memory spaces.
Allowing simultaneous data and code fetches; Better performance

Modified Harvard Architecture
Most modern MCUs (Arm Cortex-M4, NXP K64F, AVR)
Hybrid: Keep the two buses, but unifies the memory

NXP K64F Memory Map
• Modified Harvard Architecture
• Memory space size = 4GB
• Code and data are mapped in a single address space
• Actual RAM/Flash size
• RAM (data): 256KB
• Flash (code): 1MB

Discussion Question 1
>> Which of the following best describes the modified Harvard architecture used in many MCUs?
- A. Uses a single memory space and a single bus for both instructions and data
- B. Maintains separate instruction and data buses for parallel access, while allowing unified memory mapping.
- C. Enforces strict physical separation of instruction and data memory, with no

shared access between them.

- D. Does not support program execution from flash memory.

>>Answer: B

Endianness

Endianness determines which byte is put first in the memory.

• Big-endian: most significant byte first

• Little-endian: least significant byte first

CPU Registers

CPU registers: Used solely to perform general purpose operations such as arithmetic logic, and program flow control

---

I/O Registers

Mainly used to configure the operations of peripheral functions, to hold data transferred in and out of the peripheral subsystem, and to record the status of I/O operations

I/O registers in MCU can further be classified into

• Data

• Data direction

• Control and status registers

Memory Mapped

I/O registers (like GPIO, UART, ADC, interrupts) are treated as memory locations.

• Access them just like reading/writing variables in memory

• Every peripheral has a pre-defined, fixed memory address - assigned by the chip manufacturer

• e.g., *(volatile unit8_t*)0x4000 = 0x01; // Write 0x01 to the GPIO register at address 0x4000 to turn on the LED connected to pin 0.

K64F Program Execution

When the system is powered up (or reset)

• Fetches the code address stored at 0x0000_0004 into PC (Program Counter)

• Starts program execution from the fetched address

Summary

K64F MCU and Cortex-M4 architecture

• 32-bit architecture

• BUS architecture and address & data line widths

• Register size

• Addressable memory space

• Modified Harvard Architecture

• Configuration endianness -> “little endian” by default

Program execution on K64F

• Execution form Flash by default

General-Purpose Input/Output
GPIO PINS
• Signal pins that can be configured to either input or output
• No predefined purpose
• Controllable by the user program at runtime
• Most signal pins serve multiple functions -> Needs configuration
• K64F: All GPIO pins are initially disabled by hardware to conserve power and need to be enabled in order to work

---

GPIO Ports (or parallel I/O Ports)
Each port consists of group of pins (each pin: "0" or "1") is associated with an I/O controller.
Has one or more registers to control its operations.

K64F MCU Pin Assignment
Each port is 32 bits wide, but only certain bits are associated with GPIO pins.
• GPIO Ports A, B, C, D, E have physical connections to the development board.
• Each port is 32 bits wide, but not all 32 bits are associated with a physical pin.

Port Controllers
• Port control/data/status registers are mapped to MCU memory
• Port controllers manage GPIO operations via memory mapped registers.
• Writing data to predefined memory locations automatically set up the port controller's operating mode.
• I/O port controller obtains its mode and operating data for datapath from pre-assigned memory locations.

Port Input
Input Mode: Port controller detects voltage of an input pin terminal -> receiving it as logic 1 or 0
• Configurable with Data Direction Registers

Port Output
Output Mode: Port controller writes logic 1 or 0 to an output pin -> sending the voltage accordingly
• Configurable with Data Direction Registers

I/O Voltage Levels
• V_{IH}: Input high voltage – the minimum voltage the MCU recognizes as 1 when receiving
• V_{IL}: Input low voltage – the maximum voltage the MCU recognizes as 0 when receiving
• V_{OH}: Output high voltage – the minimum voltage the MCU drives for 1 when sending
• V_{OL}: Output low voltage – the maximum voltage the MCU outputs for 0 when sending

---

Noise Margin
Real chips have four voltages that matter:
• V_{OH}: High enough for the receiving input to detect as a logic "1" --> $V_{OH} > V_{IH}$
• V_{OL}: Low enough for the receiving input to detect as logic "0" --> $V_{OL} < V_{IL}$

Current Levels
Output drive currents:
• I_{OH}: Sourcing current limit if output is high
• I_{OL}: Sinking current limit if output is low

GPIOx_PDDR

Port Data Direction Register

- GPIOx_PDDR: Port x's direction
- GPIOx_PDDR = 1 = output
- GPIOx_PDDR = 0 = input

GPIOx_PDOR

Port Data Output Register

- Output mode -> Writing a binary value (0 or 1) to a bit in PORTx
- e.g. GPIOA_PDOR = 0xFFFFFFFF: driving all pins of Port A high
- Read and writable

Recall: Bit Operations

• Setting the n-th bit of PORT A -> GPIOA_PDOR |= (1UL << n); // Logical OR

• Clearing the n-th bit of PORT A -> GPIOA_PDOR &= ~(1UL << n); // Invert + AND

• Toggling the n-th bit of PORT A -> GPIOA_PDOR ^= (1UL << n); // XOR

GPIOx_PDIR

Port Data Input Register

• Input mode -> Reading a binary value from a bit in PORT

• This register is read only

GPIOx_PSOR

Port x's Set Output Register

• Assigning 1 to a bit causes the associated pin to have a HIGH logic level

• This register can only set and maintain HIGH logic levels

GPIOx_PCOR

Port x's Clear Output Register

• Assigning 1 to a bit causes the associated pin to have a low logical level.

• The register can ONLY set and maintain LOW logic levels.

GPIOx_PTOR

Port x's Toggle Output Register

• Assigning 1 to a bit causes the associated pin **INVERT** its current logic level

• This register can only TOGGLE the logic level on a pin.

Toggling means flipping the logic level

Memory Mapped

You can manually write data to the memory address associated with the register without using the register access macros

Configuring Pins for GPIO

• Although you can use GPIOx_PDDR to configure a port to be input or output, that will not enable the pins for GPIO

• All pins are automatically disabled by hardware to save power, so in order to use them as GPIO, you need to configure them.

PORTx_PCRn: Configures port x's n-th pin

PORTx_GPCHR: Configurs port x's upper 16 pins

PORTx_GPCLR: Configures port x's lower 16 pins

PORTx_PCRn

Port x's n-th pin Control Register

• There are 32 pin control registers per port

• Each register is 32 bits wide

• Bits 32-25, 23-20, and 14-11, are reserved, you don't need to write to these

• Bit 24 and bit 19-16 are important for interrupts

• For the purpose of this course, bit 15 will always be 0. Setting it to 1 makes that register read-only until system reset.

• Bits 10-8 are the most important for our purposes. For GPIO, we will select them to be 001. All bits remaining in this register can stay 0's.

• Configuring 32 pins for 5 ports (A-E) is too tedious to code, so you won't write to these registers to configure pins for GPIO in your labs.

• Instead, you will use PORTx_GPCHR and PORTx_GPCLR

PORTx_GPCHR

Port x's Global Pin Control High Register

• The name isn't helpful for anything, but essentially this register can configure Port x's pins 31-16 with one statement.

• Upper 16 bits: select which pins (from 31-16) to configure

• Lower 16 bits: Writing configuration the PCRs [15:0] -> 0x0100 for GPIO

Ex: PORTC_GPCHR = 0x1BF0100;

• 01BF is Pin

• 0100; is Configuration

01BF = b0000_0001_1011_1111

(pin 31)----------(pin16)

Configures Pins 16-21 and 23-24 on Port C to be GPIO (=0100) and everything else is not affected

PORTx_GPCLR

Port x's Global Pin Control Low Register

• Similar to PORTx_GPCHR, but configures lower 16 bits (from 15 to 0).

Ex: PORTC_GPCLR = 0x10BF0100;

01BF = b0000_0001_1011_1111

(pin 15)----------(pin0)

Configures pins 0 to 5 and 7-8 on Port C to be GPIO and everything else is not affected.

Cons of Global Pin Control Registers

• You cannot configure pins for other features (e.g. interrupts or DMA requests) with Global Pin Control Registers
• To enable interrupts, you can manually configure it in the **PORTx_PCRn** register

Clock Gating

K64F MCU is designed to be extremely low power; even after configuring the data direction registers and the pin configuration registers, we still can't use our GPIO pins

• We must enable the **gate clocking** on each of the GPIO ports.

• System Clock Gating Control Register 5 bits [13:9]

Port E: SIM_SCGC5 [13]
Port D: SIM_SCGC5 [12]
Port C: SIM_SCGC5 [11]
Port B: SIM_SCGC5 [10]
Port A: SIM_SCGC5 [9]

In an MCU, each hardware module (like GPIO, timers, UART, etc) needs a clock signal to function – just like how flip-flops in digital circuits need a clock to change state

To save power, the system disables the clock to modules that aren't in use – this is called **clock gating**

Before using a module in your code, you must enable its clock via a specific register, or it simply won't work – even if everything else is correctly set up

It reduces the flip-flop power consumption to just leakage current (virtually nothing)

• When the clock is gated (turned off), it stops switching

SIM_SCGC5

0 Clock Disabled
1 Clock Enabled

Macros Available:
e.g.,
SIM_SCGC5_PORTD_MASK

Steps to use K64F GPIO pins

1. Clock gate control
2. Pin configurations as GPIO
3. GPIO direction (input/output)
4. Reaad or write to port data registers

Interfacing with LEDs

LED (ligh emitting diode)

• Illuminates when a sufficient current is supplied

Current Limiting Resistors

Why? To limit the current flow

• Recall the maximum current per GPIO pin on the K64F MCU is 25mA

C code example

```
#include "fsl_device_registes.h"

unsinged int i;

int main(void) {
SIM_SCGC5 |=
SIM_SCGC5_PORTD_MASK;
// Enable Port D; Clock Gating
PORTD_GPCLR = 0x00FF0100;
// Pin 0-7 on port D to be GPIO
GPIOD_PDDR = 0x000000FF;
// [7:0] Output Mode
GPIOD_PDOR = 0xAA;
// Init as 1010 1010

while(1) {
for (i=0; i <100000; i++); // SW
delay
GPIOD_PTOR = 0xFF; // Invert
PORTD [7:0]
// Switches between b10101010
and // b01010101
}
}
```

This program blinks the even and odd LEDs alternately on Port D pins 0-7 (i.e., 8 LEDS), using a software delay loop

7 Segment Display
A simple device to display decimal numbers

Two types
Common cathode: all cathodes (-) are connected together
• Lights up when high is applied to a segment pin

Common anode: all anodes (+) are connected together
• Lights up when low is applied to a segment pin

Display digits 0 to 9 on 7SD
C code Ex

```
#include "fsl_device_registers.h"
long i, val;
void main(void) {
SIM_SCGC5 |=
SIM_SCGC5_PORTD_MASK;
// Enable clock Gating
PORTD_GPCLR = 0x00FF0100;
// GPIO Config
GPIOD_PDDR = 0x000000FF;
// Output Mode
val = 0;
while(1) {
if (val == 0) GPIOD_PDOR =
0x7E;
else if (val == 1) GPIOD_PDOR =
0x30;
else if (val == 2) GPIOD_PDOR =
0x60;
...
val = (val + 1) % 10;
for (i=0; i < 100000; i++);
// SW Delay
```

Code Ex2. Improve ver

```
#include "fsl_device_registers.h"

unsigned char decoder[10] =
{0x7E, 0x30, 0x6D, 0x79, 0x33,
0x5B, 0x5F, 0x70, 0x7F, 0x7B};

unsigned int i, val;
void main(void) {
SIM_SCGC5 |=
SIM_SCGC5_PORTD_MASK;
// Clock gating

PORTD_GPCLR = 0x00FF0100;
// Configures pin 0-7 on Port D to
be GPIO

GPIO_PDDR = 0x000000FF;
// Output Mode
val = 0;

while(1) {
GPIOD_PCOR = 0xFF;
// Clears output on PortD[0:7]

GPIOD_PSOR = (unsigned
int)decoder[val];
// sets output to converted value

val = (val + 1) % 10;
for (i=0; i < 100000; i++);
// SW Delay
}
}
```

Switch Bounce
Switches have mechanical contacts
• They take time to move positions
• They take time to mechanically stabilize when opening and closing
• They create sparks, especially when opening at high current

Software Debouncing
Idea: Wait until switch value is steady for some time
-> Saves external hardware components

Professor's Comments
You will not be able to succeed in this class without knowing how to use datasheet and reference manual
-> Read K64F MCU Reference Manual

Resources for this lecture and GPIO labs are available in chapter 10, 11, 12, and chapter 55

Read these sections and have a thorough understanding of each register's function and purpose

It might be useful to create a 'cheat sheet' from those sections to help with the labs and exams.