ARM Cortex-M programming

While reading this tutorial I assume that:

1. You know what is a microcontroller
2. program in C language & ability to read a datasheet
3. Electronics an mathematics background
4. Write a **makefile**(its very important for baremetal projects)

Although I will be taking some extra steps in explaining most of concepts
I will be using K66F , TinyK20, TinyK22 microcontrollers to show some examples with explanations

CHAPTER 1: C FOR EMBEDDED SYSTEMS

C Data Types For Embedded Systems

char-*it is byte*[1]*size data whose bits are designated as D7-DO. It can be signed or unsigned. In signed format the D7 bit is used for the +or – sign takes values between -128 to +127. In unsigned char we have values between 0x00 to 0xFF in hex or 0 to 255 in decimal*

Short int-*its 2 byte size whose bits are designated as D15-D0. It can be signed or unsigned. In signed format the D15 bit is used for the + or -sign takes value between -32,768 to +32,767. In unsigned format we have values between 0x0000 to 0xFFFF in hex or 0 to 65,535 in decimal since there is no sign and entire 16 bits are used for magnitude*

Long int -*it is 4 byte size data whose bits are designated as D31-D0. It can be signed or unsigned. In signed format the D31 bit is used for + or – sign and takes values between* $–2^{31}$ *to* $+2^{31}–1$*. in unsigned format we have values between 0x00000000 to 0xFFFFFFFF in hex*

Long long- *it is 8 byte size data whose bits are designated as D63-D0. It can be signed or unsigned. In signed format the D63 bit is used for + or – sign and takes values between* $–2^{63}$ *to* $+2^{63}–1$*. in unsigned format we have values between 0x00000000 to 0xFFFFFFFFFFFFFFFF in hex*

Data type	Size	Range
char	*1 byte*	$–2^{8}$ to $+2^{8}–1$
Unsigned char	*1 bytes*	0 to 2^{8}
Short int	*2 bytes*	$–2^{16}$ to $+2^{16}–1$
Unsigned short int	*2 bytes*	0 to 2^{16}
Long	*4 bytes*	$–2^{31}$ to $+2^{31}–1$
Unsigned long	*4 bytes*	0 to 2^{31}
Long long	*8 bytes*	$–2^{63}$ to $+2^{63}–1$
Unsigned long long	*8 bytes*	0 to 2^{63}

1 1 byte = 8 bits

Data Types in ISO C99 Standard

In ANSI C (ISO C89), The sizes of integer data types were not defined and are up to the compilers to decide. By conventions, Char is one byte and short is two byte size. But int and long varies greatly among compilers.
In ISO C99 standard, a set of data types were defined with number of bits and sign clearly defined in data type names. This standard is extensively used in RTOS[2].
The range is same as ANSI C standard except that it uses more descriptive syntax
This data types are defined in header file called *stdint.h*[3] ***You need to include this header file in order to use these data types.***

Data type	Size	Range
int8_t	*1 byte*	*$–2^{8}$ to $+2^{8}–1$*
uint8_t	*1 bytes*	*0 to 2^{8}*
int16_t	*2 bytes*	*$–2^{16}$ to $+2^{16}–1$*
uint16_t	*2 bytes*	*0 to 2^{16}*
int32_t	*4 bytes*	*$–2^{31}$ to $+2^{31}–1$*
uint32_t	*4 bytes*	*0 to 2^{31}*
int64_t	*8 bytes*	*$–2^{63}$ to $+2^{63}–1$*
uint64_t	*8 bytes*	*0 to 2^{63}*

Bit-Wise Operations in C

One of most important and powerful features of the C language is its ability to to perform bit manipulation.
While every C programmer is familiar with the logical operators
AND (&&), OR(||), and NOT(!) .
Many are less familiar with the bit-wise operations
AND(&),OR(|),EX-OR[4](^),INVERTER(~),SHIFT RIGHT(>>),SHIFT LEFT(<<).
These bit-wise operators are widely used in software engineering for embedded systems and control.
Their understanding and mastery are critical in microprocessor based system design and interfacing

A	B	AND (A&B)	OR(A|B)	EX-OR(A^B)	Invert (B)
0	0	0	0	0	1
0	1	0	1	1	0
1	0	0	1	1	1
1	**1**	**1**	**1**	**0**	**0**

2 RTOS - Real time operating system
3 #include<stdint.h>
4 EX-OR non-equivalence =true, equivalence =false (Revisit Digital Electronics)

Examples using C Bit-Wise operations

```
0x35 & 0x0F=

      00110101
      00001111
    &------------
      00000101    in hex  0x05
```

```
0x04|0x68=

      00000100
      01101000
     |-------------
      01101100    in hex  0x6C
```

```
0x54^0x78=

      01010100
      01111000
     ^-------------
      00101100    in hex 0x2C
```

```
~0x55

      01010101
    ~--------------
      10101010
                  in hex 0xAA
```

Example 1
Write a C program to perform above bit-wise operations

Answers in the GitHub repository--

Prepared by K. Kipruto

Setting and Clearing (Masking) bits

OR can be used to set a bit. Anything ORed with a 1 results in a 1; anything ORed with a 0 results
in no change.
AND can be used to clear a bit. Anything ANDed with a 1 results in no change; anything ANDed with a 0 results in a zero.
Anything EX-ORed with a 1 results in the complement; anything EX-ORed with a 0 results in no change.

Example 2

a) write a C program that toggles only bit 4 of var1 continously without disturbing rest of the bits
b) Write a C program to monitor bit 5 of var1. If it is HIGH, change value of var2to 0x55; otherwise change value of var2 to 0xAA.

Answers in the GitHub repository--

Bit-wise Shift operation in C

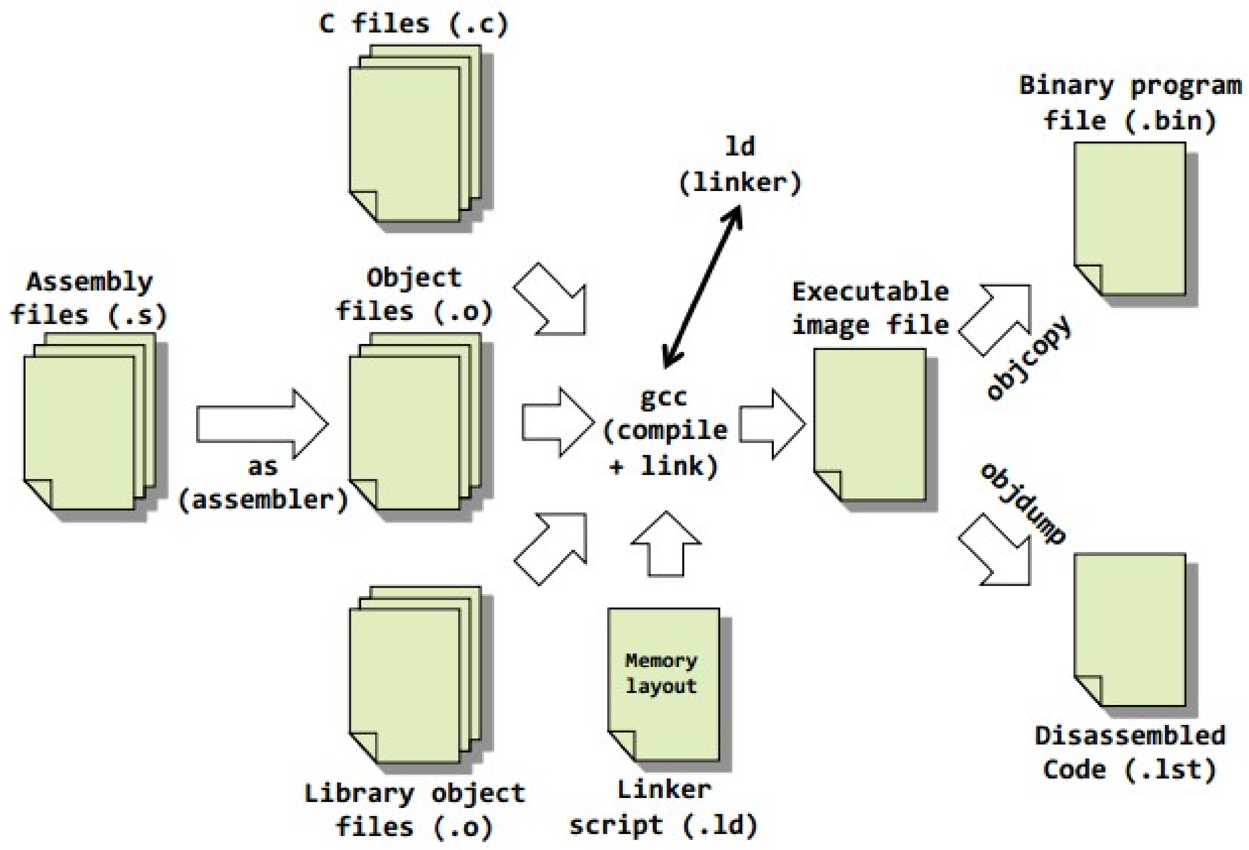
There are two bit-wise shift operations in C

Operation	Symbol	Format of Shift Operation
Shift Right	>>	data>>number of bits-positions to be shifted right
Shift Left	<<	data<<number of bits-positions to be shifted left

0b00010000 >> 3 it equals 00000010. Shifting right 3 times
0b00010000 << 3 it equals 10000000. Shifting left 3 times
0b00000001 << 3 1 << 3 it equals 00001000. Shifting left 3 times

CHAPTER 2: ARM I/O Bare-Metal Programming

First for complete C/C++ development system we need the following components:
1. GNU ARM Bare Metal Tool Chain (arm-none-eabi)
2. GDB Server
3. Integrated Development Environment (not always necessary as we can use makefile)
We had covered installation of Tool Chain for Recap visit here (will be writing notes on tool chain)



Tool Chain

Tool chain has a name convention:

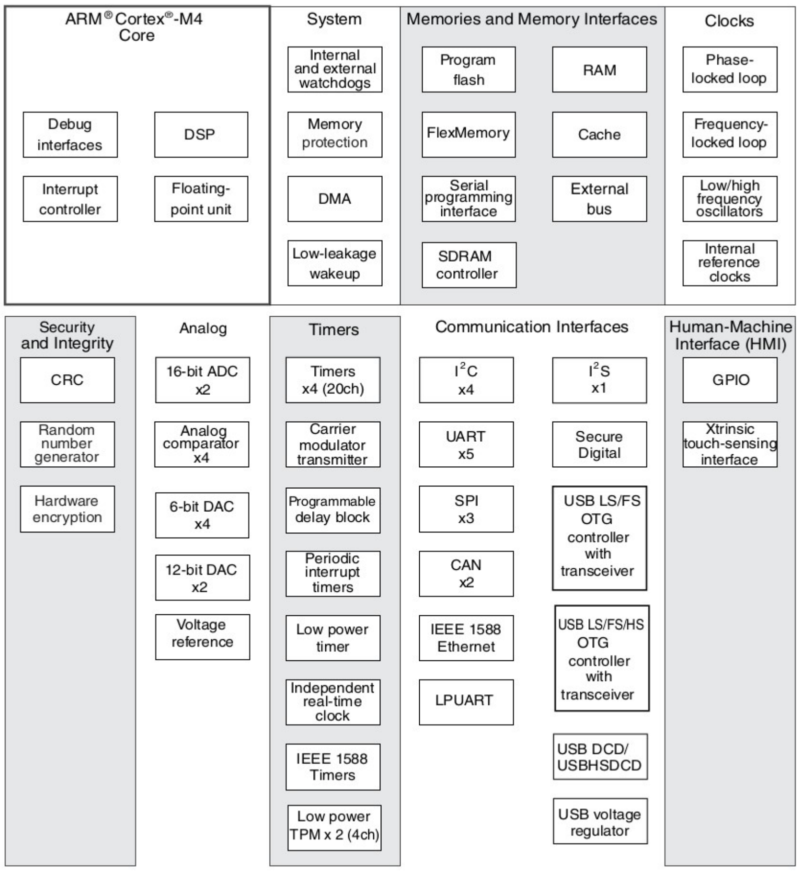
arch[-vendor][-OS]-abi

1. **arch** – architecture arm, MIPS, x86, i686, etc.
2. **vendor** - tool chain supplier
3. **OS** - operating system Linux, none (bare metal)
4. **abi** - application binary interface eabi, gnueabi, gnueabihf

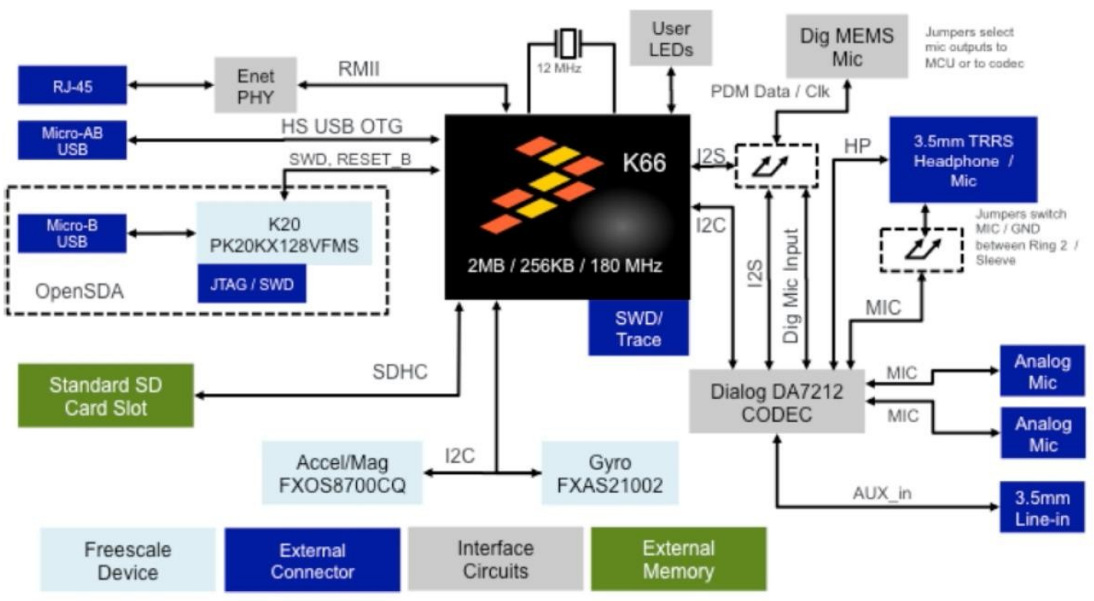
In microcontroller we use the general purpose input output (GPIO) pins to interface with led, Switch,LCD etc. We will be examining Peripheral I/O addresses and ho they are accessed and used in ARM processors. As stated earlier we will be using NXP K66F,K22F,K20xx and LPC804 microcontrollers in examples each covered individually and extensively. Will examine also their memory and I/O maps
One important note before we start is when programming an ARM chip, you have two choices
1.*Use the functions written by the vendor to access the peripheral*
s. The vast majority of the vendors/companies making the ARM chip provide a proprietary device library of functions allowing access to their peripherals. These device library functions are copyrighted and cannot be used with another vendor's ARM chip. For students and
developers, the problem with this approach is you have no control over the functions and it is very hard to customize them for your project.
2. *The second approach is to access the peripheral's special function registers directly using C language and create your own custom library since you have total control over each function. Much of these functions can be modified and used with another vendor if you decide to change the ARM chip vendor. This approach is difficult and tedious, but the rewards are great*

I would always not recommend using vendors development environment is that its quite complicated and usually consisted of code up cycled through different generations of microcontrollers and you will likely won't know why things are done the particular way.
And probably starting on anything else we require some of these:
1. ***The microprocessor data sheet****-shows all available registers, the memory map of the micro-controller.*
2. ***The board's schematics****-show the routing of the micro-controller pins in the printed circuit board, this is especially useful because it will help you know which pin is connected to what external circuitry.*
3. The board's user guide-gives the general overview of the kit, the features what can be used and for what purpose.
4.Quick reference guide-gives a summary of some configurations possible on the board andsome examples as code snippets, just to show how one may program the board.

NXP FRDM-K66F Microcontroller

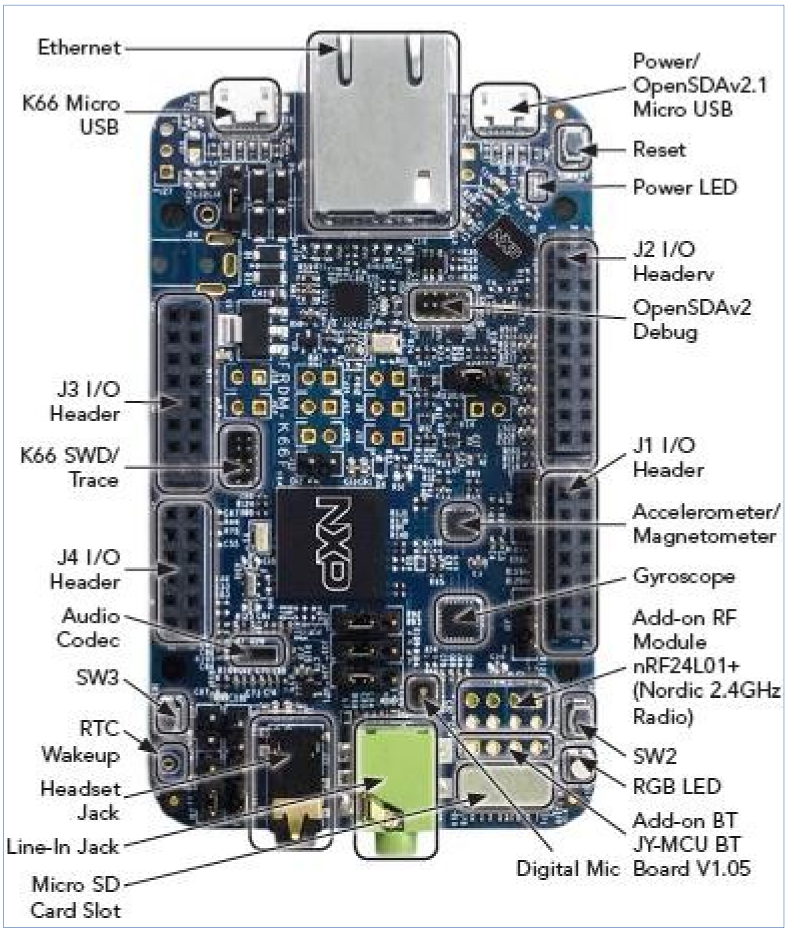


The NXP Freedom K66F hardware, FRDM-K66F, is a simple, yet sophisticated design featuring a Kinetis K series microcontroller, built on the ARM© Cortex®-M4 core. It features a MK66FN2M0VMD18, with maximum operation frequency of 180MHz, 2MB of flash, 256KB RAM, a high-speed USB controller, Ethernet controller, Secure Digital Host controller, etc



Prepared by K. Kipruto

K66F Block Diagram



FRDM-K66F Trainer Board

It looks complex but I'll break down everything making easier to understand to baremetal

32 bit ARM has 4GB of memory space. It uses memory mapped I/O meaning I/O peripheral ports are mapped into 4GB memory space

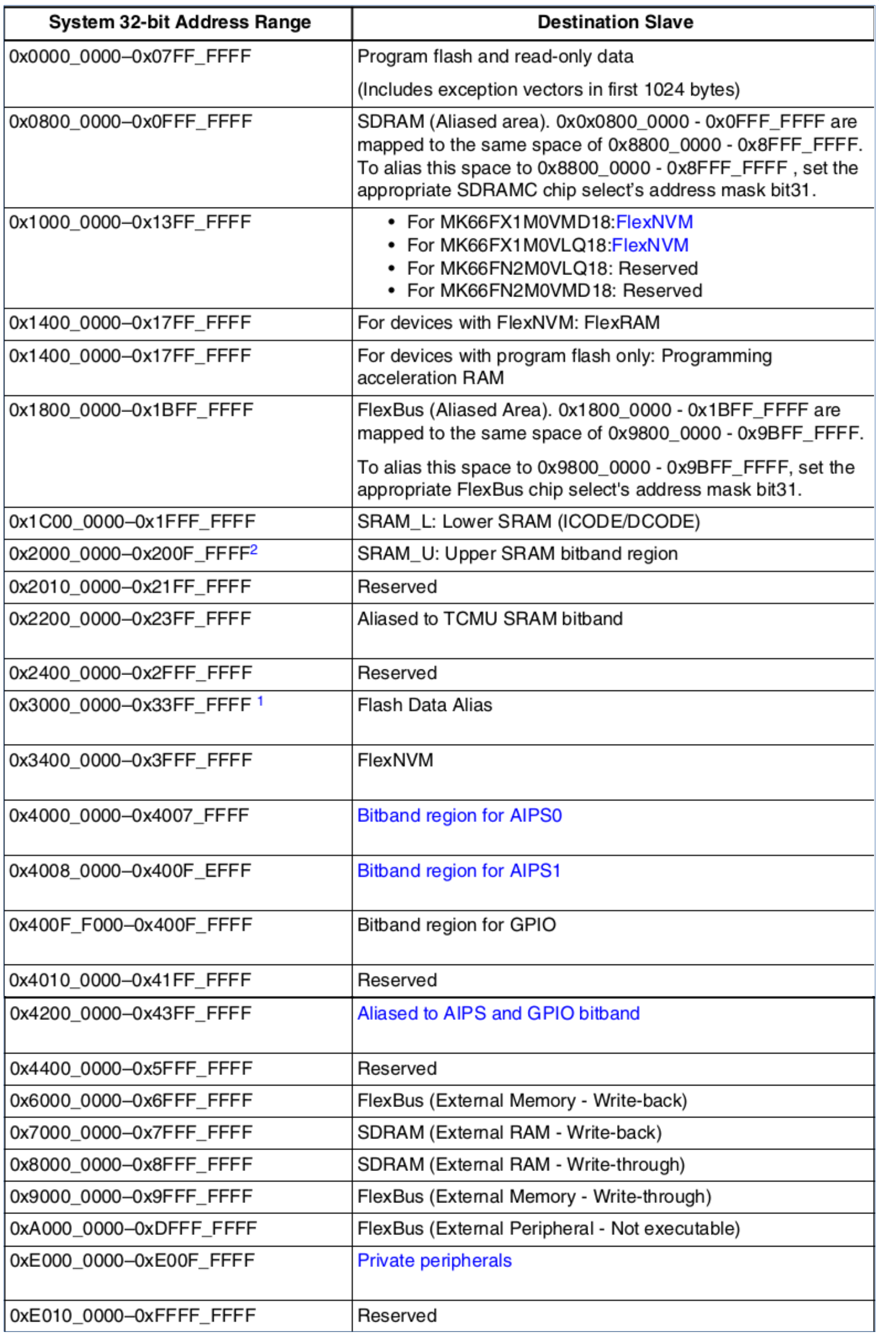
Memory [5]Map in MK66FN2M0VMD18

	Allocated Size	32-bit Address Range	
Flash	2048	0x0000_0000–0x07FF_FFFF	Program flash and read-only data (Includes exception vectors in first 1024 bytes)
SRAM	512 KB	0x1C00_0000–0x1FFF_FFFF	SRAM_L: Lower SRAM (ICODE/DCODE)
		0x2000_0000–0x200F_FFFF	SRAM_U: Upper SRAM bit-band region
I/O	All peripherals	0x400F_F000–0x400F_FFFF	Bit-band region for GPIO

5 *Additional information available on chapter 5 system memory map page 103 of K66 Sub-family Reference manual.*

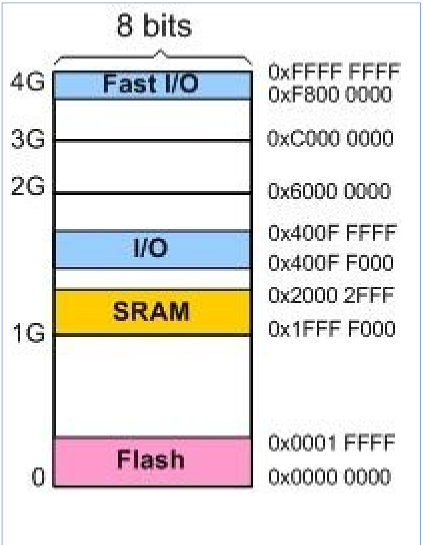
Prepared by K. Kipruto

1. 2048KB of flash memory is used for program code. One can also store in flash ROM constant (fixed) data such as look-up table if needed. The flash memory is organized into 4 blocks (512 KB each) of program flash consisting of 4 KB sector
2. the 16kb SRAM is for various variables and stack it starts at address 0x2000_0000
3.The peripherals such as I/Os, Timers, ADCs are mapped to addresses starting at 0x400F_F000



System 32-bit Address Range	Destination Slave
0x0000_0000–0x07FF_FFFF	Program flash and read-only data (Includes exception vectors in first 1024 bytes)
0x0800_0000–0x0FFF_FFFF	SDRAM (Aliased area). 0x0800_0000 - 0x0FFF_FFFF are mapped to the same space of 0x8800_0000 - 0x8FFF_FFFF. To alias this space to 0x8800_0000 - 0x8FFF_FFFF , set the appropriate SDRAMC chip select's address mask bit31.
0x1000_0000–0x13FF_FFFF	• For MK66FX1M0VMD18:FlexNVM • For MK66FX1M0VLQ18:FlexNVM • For MK66FN2M0VLQ18: Reserved • For MK66FN2M0VMD18: Reserved
0x1400_0000–0x17FF_FFFF	For devices with FlexNVM: FlexRAM
0x1400_0000–0x17FF_FFFF	For devices with program flash only: Programming acceleration RAM
0x1800_0000–0x1BFF_FFFF	FlexBus (Aliased Area). 0x1800_0000 - 0x1BFF_FFFF are mapped to the same space of 0x9800_0000 - 0x9BFF_FFFF. To alias this space to 0x9800_0000 - 0x9BFF_FFFF, set the appropriate FlexBus chip select's address mask bit31.
0x1C00_0000–0x1FFF_FFFF	SRAM_L: Lower SRAM (ICODE/DCODE)
0x2000_0000–0x200F_FFFF[2]	SRAM_U: Upper SRAM bitband region
0x2010_0000–0x21FF_FFFF	Reserved
0x2200_0000–0x23FF_FFFF	Aliased to TCMU SRAM bitband
0x2400_0000–0x2FFF_FFFF	Reserved
0x3000_0000–0x33FF_FFFF [1]	Flash Data Alias
0x3400_0000–0x3FFF_FFFF	FlexNVM
0x4000_0000–0x4007_FFFF	Bitband region for AIPS0
0x4008_0000–0x400F_EFFF	Bitband region for AIPS1
0x400F_F000–0x400F_FFFF	Bitband region for GPIO
0x4010_0000–0x41FF_FFFF	Reserved
0x4200_0000–0x43FF_FFFF	Aliased to AIPS and GPIO bitband
0x4400_0000–0x5FFF_FFFF	Reserved
0x6000_0000–0x6FFF_FFFF	FlexBus (External Memory - Write-back)
0x7000_0000–0x7FFF_FFFF	SDRAM (External RAM - Write-back)
0x8000_0000–0x8FFF_FFFF	SDRAM (External RAM - Write-through)
0x9000_0000–0x9FFF_FFFF	FlexBus (External Memory - Write-through)
0xA000_0000–0xDFFF_FFFF	FlexBus (External Peripheral - Not executable)
0xE000_0000–0xE00F_FFFF	Private peripherals
0xE010_0000–0xFFFF_FFFF	Reserved

K66F Memory Map



Memory Map example

GPIO

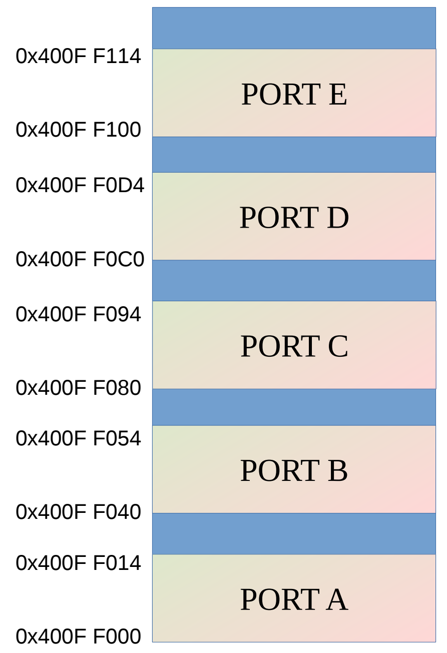
Prepared by K. Kipruto

General purpose I/O ports in **MK66FN2M0VMD18** are designed from port A to port E

GPIO signal descriptions	Description	I/O
PORTA31–PORTA0	General-purpose input/output	I/O
PORTB31–PORTB0	General-purpose input/output	I/O
PORTC31–PORTC0	General-purpose input/output	I/O
PORTD31–PORTD0	General-purpose input/output	I/O
PORTE31–PORTE0	General-purpose input/output	I/O

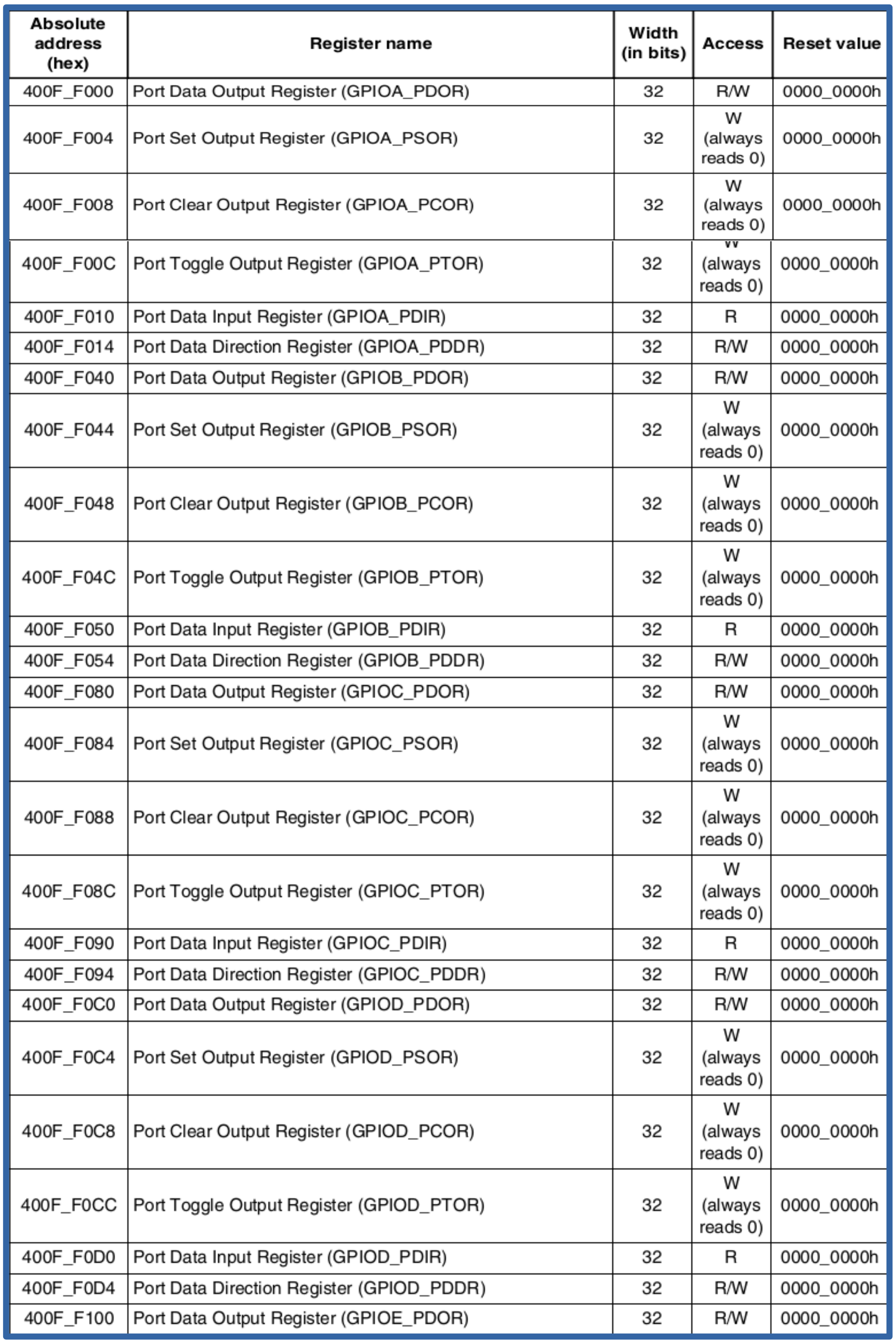
The following shows the address range assigned to each GPIO[6] port

- GPIO Port A : 0x400F F000 to 0x400F F014
- GPIO Port B : 0x400F F040 to 0x400F F054
- GPIO Port C : 0x400F F080 to 0x400F F094
- GPIO Port D : 0x400F F0C0 to 0x400F F0D4
- GPIO Port E : 0x400F F100 to 0x400F F114



GPIO Memory Map for FRDM K66F

6 Additional information is available Chapter 63 page 2185-2187 of K66 Sub-Family Reference Manual, Rev. 4, August 2018 General-Purpose Input/Output (GPIO)



Absolute address (hex)	Register name	Width (in bits)	Access	Reset value
400F_F000	Port Data Output Register (GPIOA_PDOR)	32	R/W	0000_0000h
400F_F004	Port Set Output Register (GPIOA_PSOR)	32	W (always reads 0)	0000_0000h
400F_F008	Port Clear Output Register (GPIOA_PCOR)	32	W (always reads 0)	0000_0000h
400F_F00C	Port Toggle Output Register (GPIOA_PTOR)	32	W (always reads 0)	0000_0000h
400F_F010	Port Data Input Register (GPIOA_PDIR)	32	R	0000_0000h
400F_F014	Port Data Direction Register (GPIOA_PDDR)	32	R/W	0000_0000h
400F_F040	Port Data Output Register (GPIOB_PDOR)	32	R/W	0000_0000h
400F_F044	Port Set Output Register (GPIOB_PSOR)	32	W (always reads 0)	0000_0000h
400F_F048	Port Clear Output Register (GPIOB_PCOR)	32	W (always reads 0)	0000_0000h
400F_F04C	Port Toggle Output Register (GPIOB_PTOR)	32	W (always reads 0)	0000_0000h
400F_F050	Port Data Input Register (GPIOB_PDIR)	32	R	0000_0000h
400F_F054	Port Data Direction Register (GPIOB_PDDR)	32	R/W	0000_0000h
400F_F080	Port Data Output Register (GPIOC_PDOR)	32	R/W	0000_0000h
400F_F084	Port Set Output Register (GPIOC_PSOR)	32	W (always reads 0)	0000_0000h
400F_F088	Port Clear Output Register (GPIOC_PCOR)	32	W (always reads 0)	0000_0000h
400F_F08C	Port Toggle Output Register (GPIOC_PTOR)	32	W (always reads 0)	0000_0000h
400F_F090	Port Data Input Register (GPIOC_PDIR)	32	R	0000_0000h
400F_F094	Port Data Direction Register (GPIOC_PDDR)	32	R/W	0000_0000h
400F_F0C0	Port Data Output Register (GPIOD_PDOR)	32	R/W	0000_0000h
400F_F0C4	Port Set Output Register (GPIOD_PSOR)	32	W (always reads 0)	0000_0000h
400F_F0C8	Port Clear Output Register (GPIOD_PCOR)	32	W (always reads 0)	0000_0000h
400F_F0CC	Port Toggle Output Register (GPIOD_PTOR)	32	W (always reads 0)	0000_0000h
400F_F0D0	Port Data Input Register (GPIOD_PDIR)	32	R	0000_0000h
400F_F0D4	Port Data Direction Register (GPIOD_PDDR)	32	R/W	0000_0000h
400F_F100	Port Data Output Register (GPIOE_PDOR)	32	R/W	0000_0000h

GPIO Memory Map For K66F

GPIO (General Purpose I/O) Programming and Interfacing

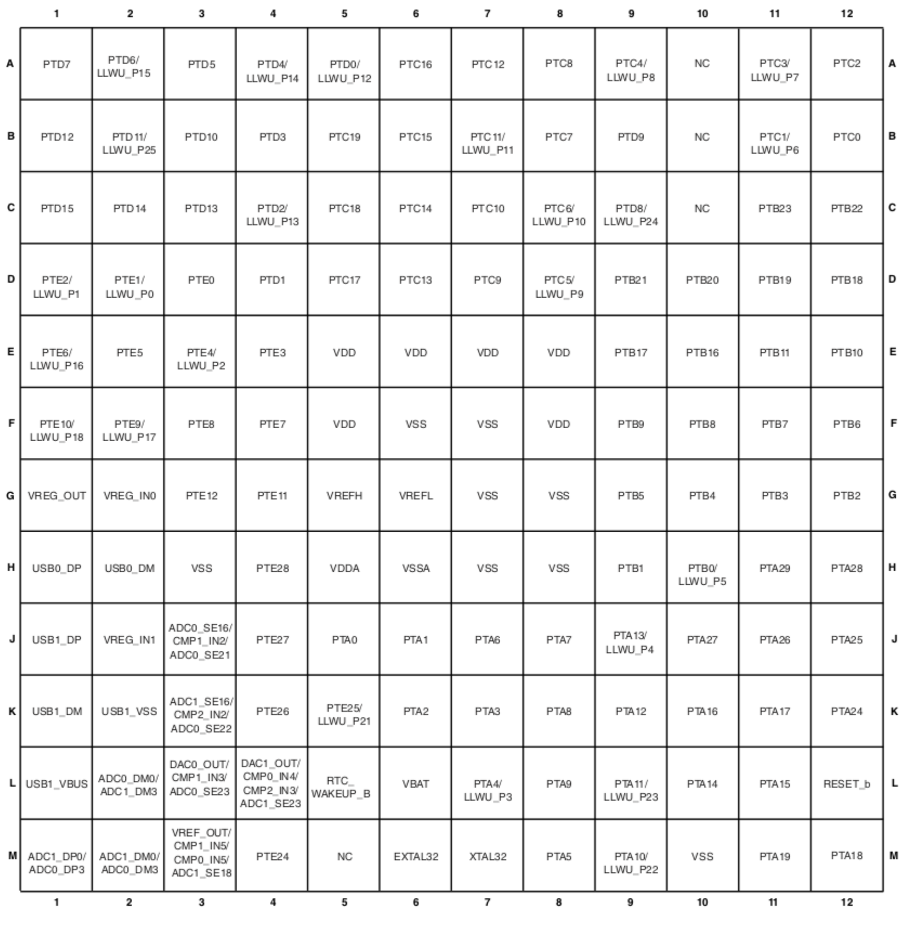
While memory holds code and data for CPU to process, the I/O ports are used by the CPU to access input and output devices. We have two types of of I/O:

1. General Purpose I/O (GPIO): The GPIO ports are used for interfacing devices such as LED's, switches, LCD, keypad etc.
2. Special Purpose I/O -These I/O ports have designated function such as ADC (Analog-to-Digital), Timer, UART (universal asynchronous receiver transmitter) etc.

In this section we examine the GPIO and its interfacing and show how to access them using C programs.

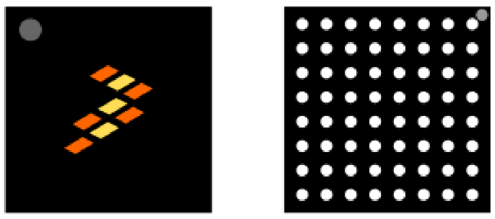
I/O Pins in NXP FRDM board

In NXP ARM chips, I/O ports are named with alphabets A,B,C,D&E. Each port have up to 32 pins and they are designated as PTA0-PTA31,PTB0-PTB31,PTC0-PTC31,PTD0-PTD31 and PTE0-PTE31. Also must be noted that not all 32 pins of each port are implemented. The figure below is MK66FN2M0VMD18 pin-out its



	1	2	3	4	5	6	7	8	9	10	11	12	
A	PTD7	PTD6/ LLWU_P15	PTD5	PTD4/ LLWU_P14	PTD0/ LLWU_P12	PTC16	PTC12	PTC8	PTC4/ LLWU_P8	NC	PTC3/ LLWU_P7	PTC2	A
B	PTD12	PTD11/ LLWU_P25	PTD10	PTD3	PTC19	PTC15	PTC11/ LLWU_P11	PTC7	PTD9	NC	PTC1/ LLWU_P6	PTC0	B
C	PTD15	PTD14	PTD13	PTD2/ LLWU_P13	PTC18	PTC14	PTC10	PTC6/ LLWU_P10	PTD8/ LLWU_P24	NC	PTB23	PTB22	C
D	PTE2/ LLWU_P1	PTE1/ LLWU_P0	PTE0	PTD1	PTC17	PTC13	PTC9	PTC5/ LLWU_P9	PTB21	PTB20	PTB19	PTB18	D
E	PTE6/ LLWU_P16	PTE5	PTE4/ LLWU_P2	PTE3	VDD	VDD	VDD	VDD	PTB17	PTB16	PTB11	PTB10	E
F	PTE10/ LLWU_P18	PTE9/ LLWU_P17	PTE8	PTE7	VDD	VSS	VSS	VDD	PTB9	PTB8	PTB7	PTB6	F
G	VREG_OUT	VREG_IN0	PTE12	PTE11	VREFH	VREFL	VSS	VSS	PTB5	PTB4	PTB3	PTB2	G
H	USB0_DP	USB0_DM	VSS	PTE28	VDDA	VSSA	VSS	VSS	PTB1	PTB0/ LLWU_P5	PTA29	PTA28	H
J	USB1_DP	VREG_IN1	ADC0_SE16/ CMP1_IN2/ ADC0_SE21	PTE27	PTA0	PTA1	PTA6	PTA7	PTA13/ LLWU_P4	PTA27	PTA26	PTA25	J
K	USB1_DM	USB1_VSS	ADC1_SE16/ CMP2_IN2/ ADC0_SE22	PTE26	PTE25/ LLWU_P21	PTA2	PTA3	PTA8	PTA12	PTA16	PTA17	PTA24	K
L	USB1_VBUS	ADC0_DM0/ ADC1_DM3	DAC0_OUT/ CMP1_IN3/ ADC0_SE23	DAC1_OUT/ CMP0_IN4/ CMP2_IN3/ ADC1_SE23	RTC_ WAKEUP_B	VBAT	PTA4/ LLWU_P3	PTA9	PTA11/ LLWU_P23	PTA14	PTA15	RESET_b	L
M	ADC1_DP0/ ADC0_DP3	ADC1_DM0/ ADC0_DM3	VREF_OUT/ CMP1_IN5/ CMP0_IN5/ ADC1_SE18	PTE24	NC	EXTAL32	XTAL32	PTA5	PTA10/ LLWU_P22	VSS	PTA19	PTA18	M
	1	2	3	4	5	6	7	8	9	10	11	12	

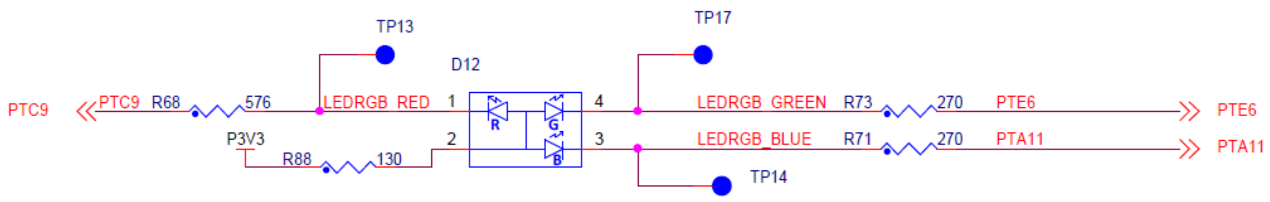
K66 144 MAPBGA Pin-out Diagram



From figure above, each port limited number of pins are implemented example on port C we have only PTC0 to PTC19 implemented

Its tradition to test the microcontroller by blinking an LED it is similar to “hello world” to embedded developers. In this section we will be writing our first program to blink led in FRDM-K66F development board(it can be also a useful debugging tool as we will see later)
FRDM-K66F has an RGB LED connected as follows

LED	K66F Connection
RED	**PTC9**/ADC1_SE5B/CMP0_IN3/FTM3_CH5/I2S0_RX_BCLK/FB_AD6/SDRAM_A14/FTM_FLT0
GREEN	**PTE6**/LLWU_P16/SPI1_PCS3/UART3_CTS/I2S0_MCLK/FTM3_CH1/USB0_SOF_OUT
BLUE	**PTA11**/LLWU_P23/FTM2_CH1/MII0_RXCLK/I2C2_SDA/FTM2_QD_PHB/TPM2_CH1

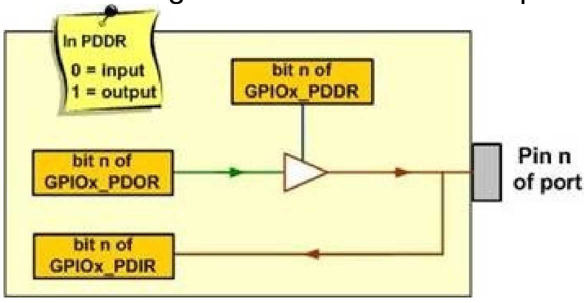


before starting to write code we have to understand that they are many registers associated with above GPIO and they have designated addresses[7] in memory map

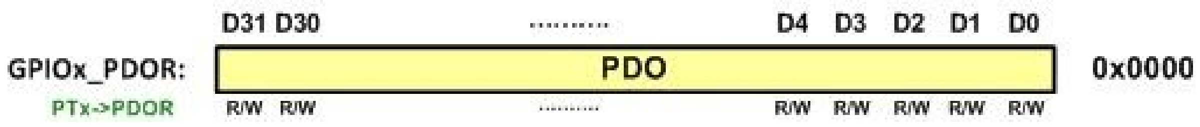
7 Base addresses -within that base address we have many registers associated with that port

Direction And Data Registers

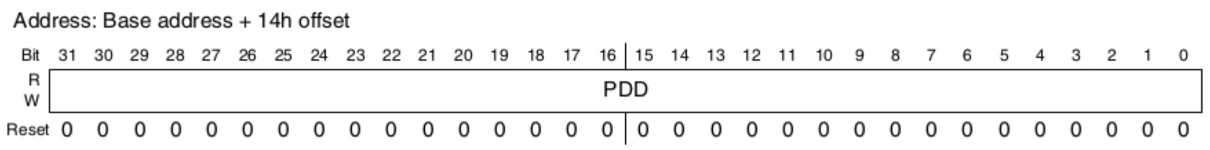
The Direction register is used to make the pin either input or output. After the Direction register is properly configured, then we use the Data register to actually write to the pin or read data from the pin. It is the Direction register (when configured as output) that allows the information written to the Data register to be driven to the pins of the device



The Port Data Output Register (GPIOx_PDOR) is located at the offset address of 0x0000 from the Base address of that port. This is shown below.


D31 D30 .... D4 D3 D2 D1 D0
GPIOx_PDOR:
PTx->PDOR
PDO
0x0000
R/W R/W .... R/W R/W R/W R/W R/W

Direction Register in NXP FRDM-K66F



Address: Base address + 14h offset

Bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R W	PDD																															
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

GPIOx_PDDR field descriptions

each of the direction register needs to be configured as

0 pin is configured as general purpose input
1 pin is configured as general purpose output

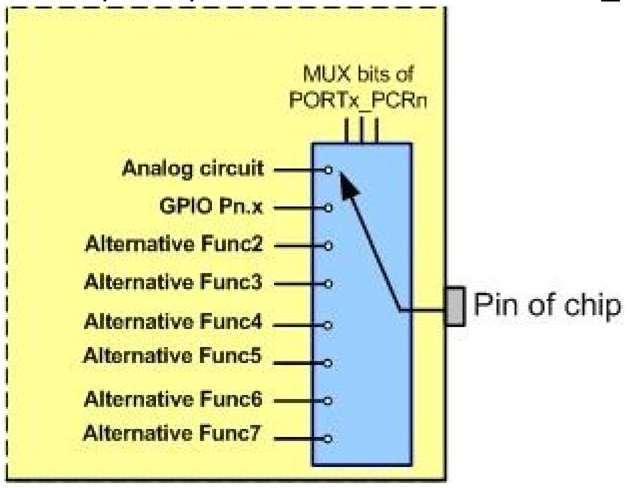
for example writing 0x03(00000011) into GPIOC_PDDR, pins 0,1 become outputs while all other become inputs
For our case PTC9 is in GPIO C pin 9

400F_F080	Port Data Output Register (GPIOC_PDOR)	32	R/W	0000_0000h	63.3.1/2189
400F_F084	Port Set Output Register (GPIOC_PSOR)	32	W (always reads 0)	0000_0000h	63.3.2/2190
400F_F088	Port Clear Output Register (GPIOC_PCOR)	32	W (always reads 0)	0000_0000h	63.3.3/2190
400F_F08C	Port Toggle Output Register (GPIOC_PTOR)	32	W (always reads 0)	0000_0000h	63.3.4/2191
400F_F090	Port Data Input Register (GPIOC_PDIR)	32	R	0000_0000h	63.3.5/2191
400F_F094	Port Data Direction Register (GPIOC_PDDR)	32	R/W	0000_0000h	63.3.6/2192

So the physical address of location of GPIO direction for PORT C is 0x400F F094

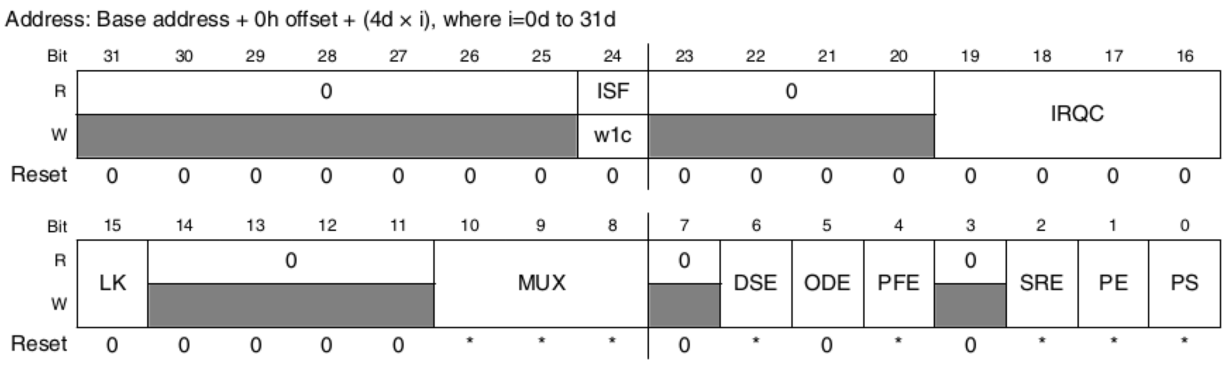
Pin Multiplexing

Each pin can be used for one of several functions including GPIO. For example one pin ca be used as simple digital I/O,analog input or I2C pin of course not all at the same time we must make sure that a pin is assigned to only one peripheral function at a time.
Portx pin control(PORTx_PCRn) special function register allows us to program pin n to be used for a given alternate function.
It must be noted that each pin of ports A-E has its own PORTx_PCRn register.



Pin Multiplexing

With the PORTx_PCRn register, not only we select the alternate I/O function of a given pin, we can also control its Drive Strength and its internal Pull-up (or Pull-down) resistor.



Address: Base address + 0h offset + (4d × i), where i=0d to 31d
Bit 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16
R 0 ISF 0
W w1c
IRQC
Reset 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Bit 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
R LK 0 MUX 0 DSE ODE PFE 0 SRE PE PS
W
Reset 0 0 0 0 0 * * * 0 * 0 * 0 * * *

The D1 (PE,Pull enable) bit of the PORTx_PCRn is used to enable the internal Pull resistor option. If PE=1, then we use the 0 bit (PS, pull Select) to enable the pull-up (or pull-down) option. We can also control the drive capability (fan-out and fan-in) of a digital I/O pin with D6 (Drive Strength Enable) bit. These options are widely used when connecting a pin to a switch or LED.
More information on page 222 of reference manual

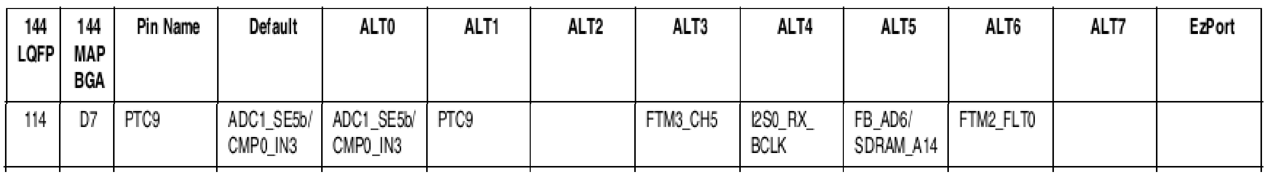
PORTx_PCRn field descriptions

Field	Description
31–25 Reserved	This field is reserved. This read-only field is reserved and always has the value 0.
24 ISF	Interrupt Status Flag The pin interrupt configuration is valid in all digital pin muxing modes. 0 Configured interrupt is not detected. 1 Configured interrupt is detected.
23–20 Reserved	This field is reserved. This read-only field is reserved and always has the value 0.
19–16 IRQC	Interrupt Configuration The pin interrupt configuration is valid in all digital pin muxing modes. The corresponding pin is configuredto generate interrupt/DMA request as follows: 0000 Interrupt Status Flag (ISF) is disabled. 0001 ISF flag and DMA request on rising edge. 0010 ISF flag and DMA request on falling edge. 0011 ISF flag and DMA request on either edge. 0100 Reserved. 0101 Reserved. 0110 Reserved. 0111 Reserved. 1000 ISF flag and Interrupt when logic 0. 1001 ISF flag and Interrupt on rising-edge. 1010 ISF flag and Interrupt on falling-edge. 1011 ISF flag and Interrupt on either edge. 1100 ISF flag and Interrupt when logic 1. 1101 Reserved. 1110 Reserved.

	1111 Reserved.
15 LK	Lock Register 0 Pin Control Register fields [15:0] are not locked. 1 Pin Control Register fields [15:0] are locked and cannot be updated until the next system reset.
14–11 Reserved	This field is reserved. This read-only field is reserved and always has the value 0.
10–8 MUX	Pin Mux Control Not all pins support all pin muxing slots. Unimplemented pin muxing slots are reserved and may result inconfiguring the pin for a different pin muxing slot. The corresponding pin is configured in the following pin muxing slot as follows: 000 Pin disabled (analog). 001 Alternative 1 (GPIO). 010 Alternative 2 (chip-specific). 011 Alternative 3 (chip-specific). 100 Alternative 4 (chip-specific). 101 Alternative 5 (chip-specific). 110 Alternative 6 (chip-specific). 111 Alternative 7 (chip-specific).
7 Reserved	This field is reserved. This read-only field is reserved and always has the value 0.
6 DSE	Drive Strength Enable This field is read-only for pins that do not support a configurable drive strength. Drive strength configuration is valid in all digital pin muxing modes. 0 Low drive strength is configured on the corresponding pin, if pin is configured as a digital output. 1 High drive strength is configured on the corresponding pin, if pin is configured as a digital output.
5 ODE	Open Drain Enable This field is read-only for pins that do not support a configurable open drain output. Open drain configuration is valid in all digital pin muxing modes. 0 Open drain output is disabled on the corresponding pin. 1 Open drain output is enabled on the corresponding pin, if the pin is configured as a digital output.
4 PFE	Passive Filter Enable This field is read-only for pins that do not support a configurable passive input filter. Passive filter configuration is valid in all digital pin muxing modes. 0 Passive input filter is disabled on the corresponding pin. 1 Passive input filter is enabled on the corresponding pin, if the pin is configured as a digital input.
3 Reserved	This field is reserved. This read-only field is reserved and always has the value 0.
2 SRE	Slew Rate Enable This field is read-only for pins that do not support a configurable slew rate. Slew rate configuration is valid in all digital pin muxing modes 0 Fast slew rate is configured on the corresponding pin, if the pin is 1 configured as a digital output. 1 Slow slew rate is configured on the corresponding pin, if the pin is configured as a digital output.
1 PE	Pull Enable This field is read-only for pins that do not support a configurable pull resistor. Pull configuration is valid in all digital pin muxing modes. 0 Internal pullup or pulldown resistor is not enabled on the corresponding pin.

	1 Internal pullup or pulldown resistor is enabled on the corresponding pin, if the pin is configured as a digital input.
0 PS	Pull Select This bit is read only for pins that do not support a configurable pull resistor direction. Pull configuration is valid in all digital pin muxing modes. 0 Internal pulldown resistor is enabled on the corresponding pin, if the corresponding PE field is set. 1 Internal pullup resistor is enabled on the corresponding pin, if the corresponding PE field is set.

K66 Signal Multiplexing and Pin Assignments



144 LQFP	144 MAP BGA	Pin Name	Default	ALT0	ALT1	ALT2	ALT3	ALT4	ALT5	ALT6	ALT7	EzPort
114	D7	PTC9	ADC1_SE5b/ CMP0_IN3	ADC1_SE5b/ CMP0_IN3	PTC9		FTM3_CH5	I2S0_RX_ BCLK	FB_AD6/ SDRAM_A14	FTM2_FLT0		

Blinky Time

As tradition we ill write code that will toggle the red led in K66F.
To be able to blink Red led the following steps will be followed

1) enable the clock to PORTC, since access is denied to the port registers until the clock is enabled,
2) configure PORTC_PCR9 (Pin Control Register) to select GPIO function for PTC9,
3) set the Direction register bit 9 of PTC as output,
4) write HIGH to PTC9 in data register,
5) call a delay function,
6) write LOW to PTC9 in data register,
7) call a delay function,
8) Repeat steps 4 to 7.

Explanation with some code

//uses header file from MCUXpresso which is compliant with CMSIS,each PORT is defied as pointer to struct with the registers as members of struct

1.Clock gating enables us to activate ONLY the peripherals
of the micro-controller that we need to use, leaving all the other peripherals inactive which reduces overall power consumption. From the data sheet, the clock gating for port A is done under the System Integration Module (SIM), in the System Clock Gating Control register 5 (SIM_SCGC5) bit 11. this can be done by;

(page 263 ref manual)

```
 SIM->SCGC5|=0x800;
```

2 Each pin of the micro-controller can be configured for a number of alternate functions. From the data sheet, we can see that we can configure a port pin to any supported alternate function, to configure PORT C pin 9 as GPIO this done by pin mux control bits 001-alternative 1 GPIO(pg 223)

```
PORTC->PCR[9]=0x100;
```

3 And then finally, we need to configure the directionality of the GPIO pin.
Since we have already configured our pin to GPIO, and GPIO is an entity on its own, we then go to the gpio C Port Data Direction Register, GPIOC_PDDR. Directionality simply means determining whether the pin is an output or an input pin. Obviously we need to see some output through our LED, so we only only need to set the pin to output data. And can be done by bit 9 (pg 2192)

```
PTC->PDDR|=0x200;
```

4 finally writing high to PTC 9 remember bit masking. We will write high and low to PTC 9 by using port data output register(PDOR) this can be done by; (pg 2189)

```
PTC->PDOR &=~0x200;
//call delay function
PTC->PDOR |=0x200;
//call delay function
```

5 delay function- this can be implemented using nested for loop

```
delay_ms(int n){
  int j;
  int i;
  for (i = 0; i < n; i++) {
    for (j = 0; j < 7000; j++) {}
  }
```

Putting all the code together

we can define the physical address of these special function registers belonging to I/O ports. It can be tedious and prone to errors we will be using vendors header file for now then latter approach later

MK66F18.h
Text

For our code to compile successfully we need the tool-chain as we had explained before.
Linker script to combine these object files and, in the process, to resolve all of the unresolved symbols.

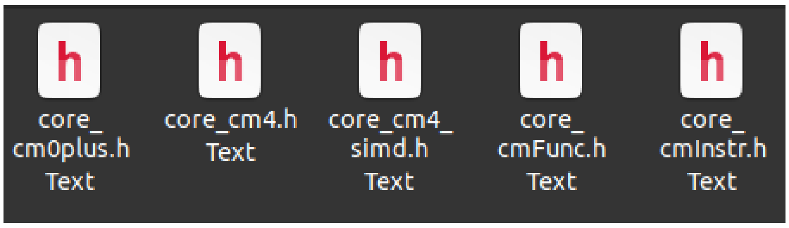
MK66FN2M0xxx18.ld
Text

Startup code which is a small block of assembly language code that prepares the way for the execution of software written in a high-level language.

startup_MK66F18.S
Text

Startup code for C programs usually consists of the following series of actions:
1. Disable all interrupts.
2. Copy any initialized data from ROM to RAM.
3. Zero the uninitialized data area.
4. Allocate space for and initialize the stack.
5. Initialize the processor’s stack pointer.
6. Call main .

CMSIS library for Cortex-M (Cortex-M Core Peripheral Access Layer)



I will be discussing these in detail about the linker script and startup code(future work)

```
#include "MK66F18.h"
//delay function
delay_ms(int n){
  int j;
  int i;
  for (i = 0; i < n; i++) {
    for (j = 0; j < 7000; j++) {}
  }
}
int main(void) {
  //enable clock to port C
  SIM->SCGC5|=0x800;
  //make PTC9 pin as GPIO
  PORTC->PCR[9]=0x100;
  //make ptc 9 as output pin
  PTC->PDDR|=0x200;
  while (1) {
    //turn on red led on PTC 9
    PTC->PDOR &=~0x200;
    delay_ms(100);//delay
    //turn off red led on PTC 9
    PTC->PDOR |=0x200;
    delay_ms(100);//delay
  }

}
```

then run the compile the code and upload to microcontroller and RED led will start blinking(will be covering about PYOCD/OPENOCD and JLINK) for onboard debuggers with daplink will be easy because of drag and drop but know how of how to use pyocd/openocd Is very important
Next I will be covering tinyK22