

# k60 Control - libbase

Author: Peter Tse ([mcreng](#)), Dipsy Wong ([dipsywong98](#))

## k60 Control - libbase

[Introduction](#)

[Pin Configuration](#)

[GPIO](#)

[GPI](#)

[GPO](#)

[PWM](#)

[ADC](#)

[UART](#)

[PIT](#)

[SPI](#)

[I2C](#)

[Flash](#)

[NVIC](#)

## Introduction

This sections include controls to basic peripherals provided by the MCU. In your SmartCar development, you may seldom use these configurations, but these can help you a lot since you have a greater freedom using libbase than libsc. Understanding these could also allow you to develop the project with a more appropriate mindset.

## Pin Configuration

There are 144 pins in a k60 chip and each pin has a specific role. Consider the following excerpt on pin assignments of k60 chips.

144 LQFP	144 MAP BGA	Pin Name	Default	ALT0	ALT1	ALT2	ALT3	ALT4	ALT5	ALT6	ALT7	EzPort
—	L5	RTC_WAKEUP_B	RTC_WAKEUP_B	RTC_WAKEUP_B								
—	M5	NC	NC	NC								
—	A10	NC	NC	NC								
—	B10	NC	NC	NC								
—	C10	NC	NC	NC								
1	D3	PTE0	ADC1_SE4a	ADC1_SE4a	PTE0	SPI1_PCS1	UART1_TX	SDHC0_D1		I2C1_SDA	RTC_CLKOUT	
2	D2	PTE1/LLWU_P0	ADC1_SE5a	ADC1_SE5a	PTE1/LLWU_P0	SPI1_SOUT	UART1_RX	SDHC0_D0		I2C1_SCL	SPI1_SIN	

Each pin has a default functionality, but one can alter its functionality through pin configuration. Consider pin #1, its default is a pin for ADC (Analog-Digital Converter), but you can alter it by changing it to a GPIO pin (PTE0), a SPI pin and a UART pin, etc.

The pin configurations are defined in `libscccc/inc/libsc/k60/config/<group name>.h`. For example for `2017_inno.h`, we have the following excerpt.

```
#define LIBSC_ST7735R_RST libbase::k60::Pin::Name::kPte3
#define LIBSC_ST7735R_DC libbase::k60::Pin::Name::kPte0
#define LIBSC_ST7735R_CS libbase::k60::Pin::Name::kPte4
#define LIBSC_ST7735R_SDAT libbase::k60::Pin::Name::kPte1
#define LIBSC_ST7735R_SCLK libbase::k60::Pin::Name::kPte2
```

Here it defines the five pins of `ST7735R` to correspond to the five pins (according to the pin names) of the chip. Since the module uses SPI (you will learn it in later sections), we will find a `SpiMaster` definition in `libscccc/inc/libsc/st7735r.h`.

```
SpiMaster m_spi;
```

When the class `St7735r` is being constructed, the following member initialization is triggered.

```
St7735r::St7735r(const Config &config)
    : m_spi(GetSpiConfig()) /* ... */ { /* ... */ }
```

The function `GetSpiConfig()` is defined in `libscccc/src/libsc/st7735r.cpp`

```
St7735r::SpiMaster::Config GetSpiConfig()
{
    St7735r::SpiMaster::Config config;
    config.sout_pin = LIBSC_ST7735R_SDAT;
    config.sck_pin = LIBSC_ST7735R_SCLK;
    /* OMITTED */
    return config;
}
```

We can see the pin is being used to configure the module by passing it to SPI configurations.

As a `SpiMaster` class is created, its constructor located at `libscccc/src/libbase/k60/spi_master.cpp` is called.

```
SpiMaster::SpiMaster(const Config &config)
    : m_sin(nullptr),
      m_sout(nullptr),
      m_sck(nullptr),
      m_is_init(false)
{
    /* OMITTED */
    InitPin(config);
    /* OMITTED */
}
```

Inside `InitPin(const Config)`, we have

```

void SpiMaster::InitPin(const Config &config)
{
    if (config.sin_pin != Pin::Name::kDisable)
    {
        Pin::Config sin_config;
        sin_config.pin = config.sin_pin; // specifies Pin
        sin_config.mux = PINOUT::GetSpiSinMux(config.sin_pin); // specifies Alt
        m_sin = Pin(sin_config);
    }

    /* OMITTED */
}

```

And finally, the function `PINOUT::GetSpiSinMux(Pin::Name)` in `libsccl/src/libbase/k60/pinout/<mcu_name>.cpp` allows one to alter the functionalities of the pins.

```

Pin::Config::MuxControl Mk60f15Lqfp144::GetSpiSinMux(const Pin::Name pin)
{
    switch (pin)
    {
        default:
            assert(false);
            // no break

        case Pin::Name::kPta17:
        case Pin::Name::kPtb17:
        case Pin::Name::kPtb23:
        case Pin::Name::kPtc7:
        case Pin::Name::kPtd3:
        case Pin::Name::kPtd14:
        case Pin::Name::kPte3:
            return Pin::Config::MuxControl::kAlt2;

        case Pin::Name::kPte1:
            return Pin::Config::MuxControl::kAlt7;
    }
}

```

In `Pin::Config::MuxControl::kAlt#`, the pins are configured to have the functionalities of `ALT#` in the pin assignment.

The pin configurations are similar for other protocols, just that the functions for them are located in different places (located at `InitPin(Pin::Name)` function of respective library).

## GPIO

GPIO, which stands for **General-purpose Input/Output**, is a generic pin which allow either high (1) or low (0) state. In GPIO, pins can be further divided into an input pin (**GPI**) and an output pin (**GPO**).

Location: `libbase/k60/gpio.h`

## GPI

A GPI (**General-purpose Input**) pin allows to be read with either high or low state.

Config	Datatype	Description
pin	Pin::Name	Pin name
interrupt	Pin::Config::Interrupt	Could be <code>kDisable</code> , <code>kRising</code> , <code>kFalling</code> and <code>kBoth</code> for no interrupt, interrupt at rising edge, interrupt at falling edge and interrupt at both rising and falling edge
config	std::bitset<6>	Specify the configuration of the pin, either allow <code>kOpenDrain</code> , <code>kPassiveFilter</code> , <code>kPullEnable</code> and <code>kPullUp</code> for open drain, low pass filter and pull-up/pull-down.
isr	void(Gpi*)	Gpi listener

Sample code:

```
void GPIListener(Gpi *gpi) {
    if (gpi->Get()) { // get state of GPI
        // if high
    } else {
        // if low
    }
}

Gpi::Config ConfigGPI;
ConfigGPI.pin = Pin::Name::kPtb0;
ConfigGPI.interrupt = Pin::Config::Interrupt::kBoth;
ConfigGPI.config.set(Pin::Config::kPassiveFilter);
ConfigGPI.isr = GPIListener;
Gpi gpi(ConfigGPI);
```

## GPO

A GPO (**General-Purpose Output**) pin allows to be written with either high or low state.

Config	Datatype	Description
pin	Pin::Name	Pin name
config	std::bitset<6>	Specify the configuration of the pin, either allow <code>kHighDriveStrength</code> and <code>kSlowSlewRate</code> for low/high drive strength and slow/fast slew rate
is_high	bool	Default output state of the pin

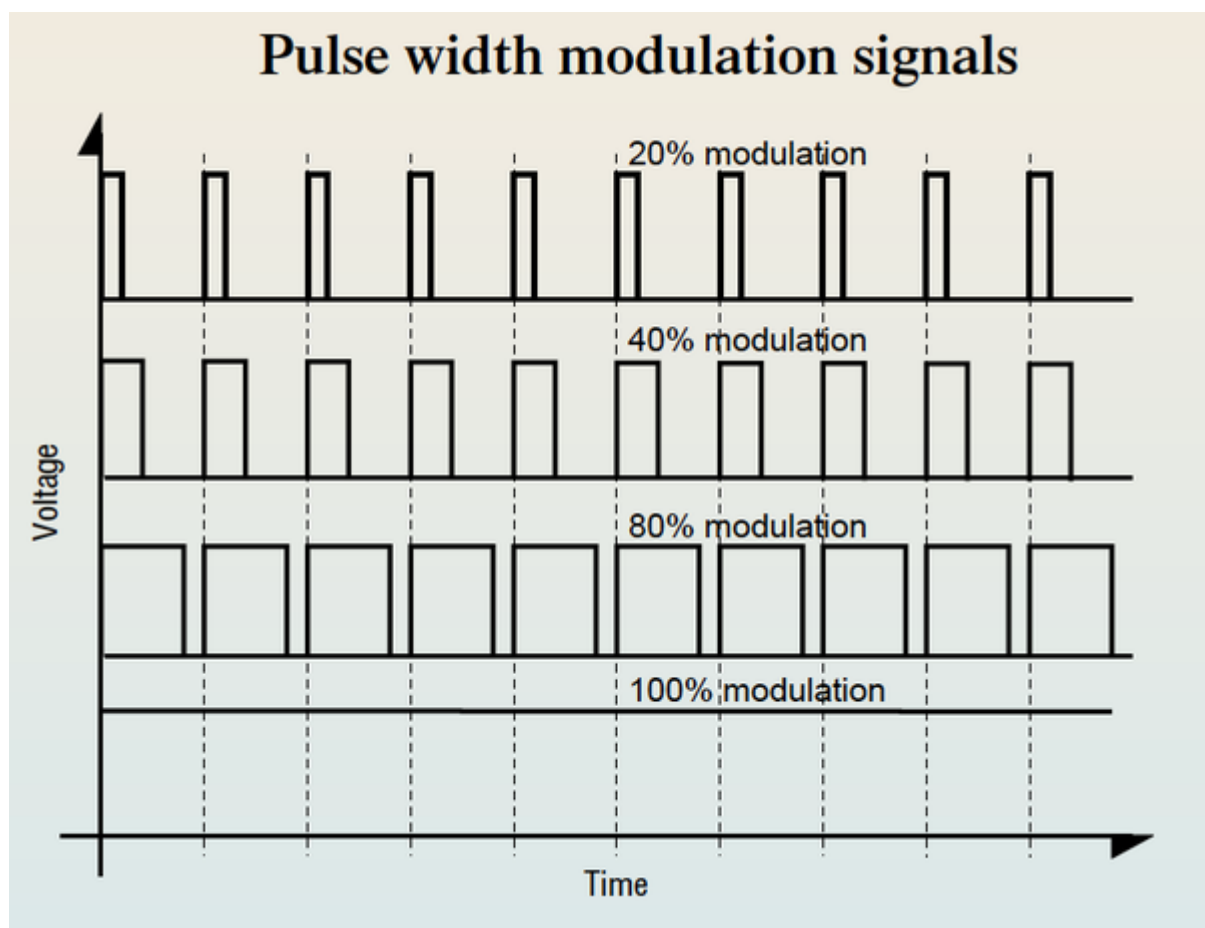
Sample code:

```
Gpo::Config ConfigGPO;
ConfigGPO.pin = Pin::Name::kPtB0;
ConfigGPO.is_high = false
Gpo gpo(ConfigGPO);

gpo.Set(true); // set output to 1
gpo.Set(false); // set output to 0
gpo.Turn(); // toggle output to 1
gpo.Turn(); // toggle output to 0
```

## PWM

PWM, which stands for **Pulse-Width Modulation**, is essentially a signal of square wave, which components like motors and servos use it to input the percentage power/angle.



Location: `libbase/k60/ftm_pwm.h`

Config	Datatype	Description
pin	Pin::Name	Pin name
period	uint32_t	Period of a cycle
pos_width	uint32_t	Pulse width in a cycle
precision	Precision	Unit for period and pulse width, either in us (default) or in ns
alignment	Alignment	Alignment of PWM signals, either with edge of period or center of period

Sample code:

```
FtmPwm::Config ConfigPWM;
ConfigPWM.pin = Pin::Name::kPta0; // need Pin with Ftm functionality
ConfigPWM.period = 10;
ConfigPWM.pos_width = 2;
ConfigPWM.alignment = FtmPwm::Config::Alignment::kEdge;
FtmPwm pwm(ConfigPWM);
// the pin PTA0 is now in 20% modulation with 10us cycle
```

## ADC

ADC, which means **Analog Digital Converter**, converts analog signals into digital signals.

Config	Datatype	Description
pin	Pin::Name	Pin name (override <code>adc</code> )
adc	Adc::Name	Adc name (can specify certain ADC for pins that support multiple ADCs)
is_diff_mode	bool	Determine whether the ADC uses differential conversion
resolution	Resolution	Resolution of the ADC, determines the output type, either 8, 10, 12 or 16 bit
speed	Speed	Speed of the ADC
is_continuous_mode	bool	Allow continuous conversion (?)
avg_pass	AveragePass	Number of sample average used for output
conversion_isr	void(Adc*, uint16_t)	Listener when a conversion is finished

Sample code:

```

Adc::Config ConfigADC;
ConfigADC.adc = Adc::Name::kAdc3Ad6A; // ADC3 AD6A (ADC3_SE6a in Pta6)
ConfigADC.speed = Adc::Config::SpeedMode::kExSlow;
ConfigADC.is_continuous_mode = true;
ConfigADC.avg_pass = Adc::Config::AveragePass::k32;
Adc adc(ConfigADC);

adc.StartConvert(); // start ADC
adc.GetResult(); // get result in int
adc.GetResultF(); // get result in float [0, 3.3]
adc.StopConvert(); // stop ADC

```

## UART

UART, which means **Universal Asynchronous Receiver-Transmitter**, is a protocol for serial communication between modules, usually used in Bluetooth for the case of SmartCar.

Location: `libsc/k60/uart_device.h`

Config	Datatype	Description
<code>id</code>	<code>uint8_t</code>	UART id
<code>baud_rate</code>	<code>Uart::Config::BaudRate</code>	Baud rate of UART communication, usually use <code>k115200</code>
<code>rx_isr</code>	<code>bool(const Byte*, const size_t)</code>	UART listener, return true if the data is consumed

Sample code:

```

bool UARTListener(const Byte *data, const size_t size) { // for RX
    /* ... */
    return true;
}

UartDevice::Config ConfigUART;
ConfigUART.id = 0;
ConfigUART.baud_rate = Uart::Config::BaudRate::k115200;
ConfigUART.isr = UARTListener;
UartDevice uart(ConfigUART);

// for TX
uart.SendStr("Hello World");
Byte byte = 20;
uart.SendBuffer(&Byte, 1);

```

## PIT

PIT, which means **Periodic Interrupt Timer**, is a listener function will be triggered periodically when the timer count a specific time, during the program run time. Be careful, when PIT is triggered, the normal program rundown will pause. Therefore, if PIT is called too frequently, your program cannot run.

Location: `libbase/k60/pit.h`

Config	Datatype	Description
<code>channel</code>	<code>uint8_t</code>	0~3, different channel (timer)
<code>count</code>	<code>uint32_t</code>	1 count = timer oscillate once 75000 counts = 1ms
<code>isr</code>	<code>void(Pit*)</code>	pit listener

Sample code:

```
void Job(Pit*){
    /*code which will execute periodically*/
}

int main(){
    Pit::Config pitConfig;
    pitConfig.channel = 0;
    pitConfig.count = 75000*250;    //job executed once per 250ms
    pitConfig.isr = Job;
    Pit pit(pitConfig);
    while(1);
    return 0;
}
```

## SPI

The Serial Peripheral Interface bus (SPI) is a synchronous serial communication interface specification used for short distance communication. One example of module which uses SPI is St7735r (tft-Lcd)

Location: `libbase/k60/spi_master.h` `libbase/k60/soft_spi_master.h` (soft\_ means software emulated)

(There is number of inheritance relationship inside the implementation of i2c)



Config	Datatype	Description
sout_pin	Pin::Name	At least one among SIN/SOUT have to be set
sin_pin	Pin::Name	At least one among SIN/SOUT have to be set
sck_pin	Pin::Name	clock pin
baud_rate	uint32_t	*Baud rate hint, the closest possible value will be used
frame_size	uint8_t	Number of bits transfered per frame, [4, 16]
is_sck_idle_low	bool	Set the clock polarity
is_sck_capture_first	bool	Set the clock phase, if true, data is captured on the leading edge of SCK and changed on the following edge
is_msb_first	bool	Specifies whether the LSB or MSB of the frame is transferred first
slaves	Config::Slave	Slaves connected to this SPI instance, a maximum of 6 slaves are supported. The array should be filled in sequence (from [0] to [5]) The position will serve as the slave id used in Exchange()
pcs_to_sck_delay_ns	uint32_t	*
after_sck_delay_ns	uint32_t	*
after_transfer_delay_ns	uint32_t	*
is_modified_timing	bool	*
tx_isr	OnTxFillListener	*
rx_isr	OnRxDrainListener	*

\*for normal spi only

Sample Code

```

SpiMaster::Config config;
config.sout_pin = LIBSC_ST7735R_SDAT;
config.sck_pin = LIBSC_ST7735R_SCLK;
config.baud_rate_khz = 15000;
config.frame_size = 8;
config.is_sck_idle_low = true;
config.is_sck_capture_first = true;
config.is_msb_firt = true;
SpiMaster spi(config);
uint16_t feedback = spi.ExchangeData(0,data);//slave id, exchange data buffer, const uint16_t

```

Here is only some parts of the spi methods, you may look deeper yourself.

## I2C

I2C (pronounced as i-square-c) is a protocol widely used for attaching lower-speed peripheral ICs to processors and microcontrollers in short-distance, intra-board communication. It is multi-master multi-slave. One example of module which uses I2C is MPU6050 (Gyro+Accelerometer)



Location : `libbase/k60/i2c_master.h` `libbase/k60/soft_i2c_master.h` (soft\_ means software emulated)  
 (There is number of inheritance relationship inside the implementation of i2c)

Config	Datatype	Description
scl_pin	Pin::Name	clock pin
sda_pin	Pin::Name	arbitrated pin
baud_rate_khz	uint16_t	Set the baud rate. The standard i2c frequency is 100kHz, many later devices can go up to 400kHz (such as MPU6050). Misbehavior if too high
scl_low_timeout	uint16_t	click low timeout
is_use_repeated_start	bool	Generate a START signal followed by a calling command without generating a STOP signal first. This saves the time needed for the STOP signal
min_sda_hold_time_ns	uint16_t	*The SDA hold time is the delay from the falling edge of SCL to the changing of SDA
min_scl_start_hold_time_ns	uint16_t	*The SCL start hold time is the delay from the falling edge of SDA while SCL is high (start condition) to the falling edge of SCL
min_scl_stop_hold_time_ns	uint16_t	*The SCL stop hold time is the delay from the rising edge of SCL to the rising edge of SDA while SCL is high (stop condition)
is_high_drive_mode	bool	*
freq_khz	uint32_t	^software simulated baud rate

\*for normal i2c only, ^for soft i2c only

Sample Code:

```

I2cMaster::Config config;
config.scl_pin = libbase::k60::Pin::Name::kPtb0;
config.sda_pin = libbase::k60::Pin::Name::kPtb1;
config.baud_rate_khz = 400;
config.scl_low_timeout = 1000;

I2cMaster i2c(config);

//address of slave (for example MPU6050), address of the interesting register, byte
if(i2c.SendByte(0x68, 0x72, 0x03)){
    //successfully sent
}else{
    //fail
}

Byte byte[10]={1,2,3,4,5,6,7,8,9,10};

```

```

//address of slave, address of the interesting register, byte array (pointer), size
if(i2c.SendBytes(0x68, 0x72, byte, 10)){
    //successfully sent
}else{
    //fail
}

Byte result;
//address of slave, address of the interesting register, address for result
if(i2c.GetByte(0x68, 0x72, &result)){
    //successfully get
    deal_with_result(result);
}else{
    //fail
}

//address of slave, Address of the interesting register, size
vector<Byte> results = i2c.GetBytes(0x68, 0x72, 10);

```

Note: you may use `assert` function which accepts a value that `assert(0)` will terminate the program. You need `#include <cassert>` for it.

## Flash

Flash is the permanent memory in the MCU, which means you can get back the variable value when your smart car is turned off. Be careful, memory may crash if you turn off your smartcar while saving value to flash.

Location : `libbase/k60/flash.h`

Config	Datatype	Description
<code>sectorStartIndex</code>	<code>uint8_t</code>	start sector, change only if you are sure what are you doing
<code>size</code>	<code>size_t</code>	memory size, default is 4096 Byte, don't waste memory

### Read/Write to Flash

If you have only one type of variables to save, you can directly pass an array pointer to the function.

```

#include "libbase/k60/flash.h"
namespace libbase {
namespace k60 {
Mcg::Config Mcg::GetMcgConfig() {
    Mcg::Config config;
    config.external_oscillator_khz = 50000;
    config.core_clock_khz = 150000;
    return config;
}
} // namespace k60
} // namespace libbase

using libbase::k60::Flash;

```

```

int main(){
    Flash flash(Flash::Config);
    int myArray[10] = {};
    memset(myArray, 0, sizeof(myArray));
    flash.Read(myArray,sizeof(myArray));
    //first time: destroyed datas, all nan
    //second time onwards: {0,1,2,3,4,5,6,7,8,9}
    for(int i=0;i<10;i++) myArray[i]=i;
    flash.Write(myArray,sizeof(myArray));
    return 0;
}

```

If you need to save more than one type of variables, I will suggest you to build a new byte array and use `memcpy()` to copy values.

```

Byte byte[15];
int v1 = 0;
float v2 = 0;
bool v3 = true;
uint16_t v4 = 0;
char v5 = '';
char* v6 = "hi";

//retriving stored data here
flash.Read(byte,15);
memcpy(v1,byte,4);
memcpy(v2,byte+4,4);
memcpy(v3,byte+8,1);
memcpy(v4,byte+9,2);
memcpy(v5,byte+11,1);
memcpy(v6,byte+12,3);

//saving data
memcpy(byte,v1,4);
memcpy(byte+4,v2,4);
memcpy(byte+8,v3,1);
memcpy(byte+9,v4,2);
memcpy(byte+11,v5,1);
memcpy(byte+12,v6,3);
flash.Write(byte,15);

```

To ensure the data is not `nan`, you can use `==` operator.

```

if(variable==variable){
    //this variable is a number
}
else{
    //this varialbe is not a number
}

```

NVIC