Understanding Ports: The KL43Z General purpose I/O Ports

John Gilmer October 15, 2019

Background:

Input/Output devices, even the simplest ones originally called "parallel ports," have gotten progressively more complicated over the microprocessor era. Early microprocessors used discrete "port" integrated circuits that included, basically, a latch for output, and a facility to read input signals. The Motorola 6820 was an example. The "registers" of the port were memory mapped, and provided a capability to use any given pin as either input or output, possibly even changing that use dynamically. Early microcontrollers brought such peripherals into the microcontroller. The 68HC11 series included B and C parallel ports of 8 bits, as well as some pins of an E Port that could be used as inputs to an analog to digital converter instead. This need to "choose what to use the pin for" issue resulted from progress. It was possible to build more capability into the device than a user could simultaneously use due to a limitation on the number of pins. The trend has only gone further. In the S08QG8 microcontrollers (used in EGR222), the same pin can usually serve several different purposes. The designer must choose whether to use that pin, for example, for a serial port, and A/D converter channel, or as a bit of a general-purpose I/O port. Figure 1, a detail from the device block diagram, shows how the pins of "Port B" of that device may be used for multiple purposes. Do you want an external crystal clock? It costs you the use of Port B pins 6 and 7. The serial port costs pins 0 and 1. The unused resources within the device are not a concern; the silicon is cheaper than the package.

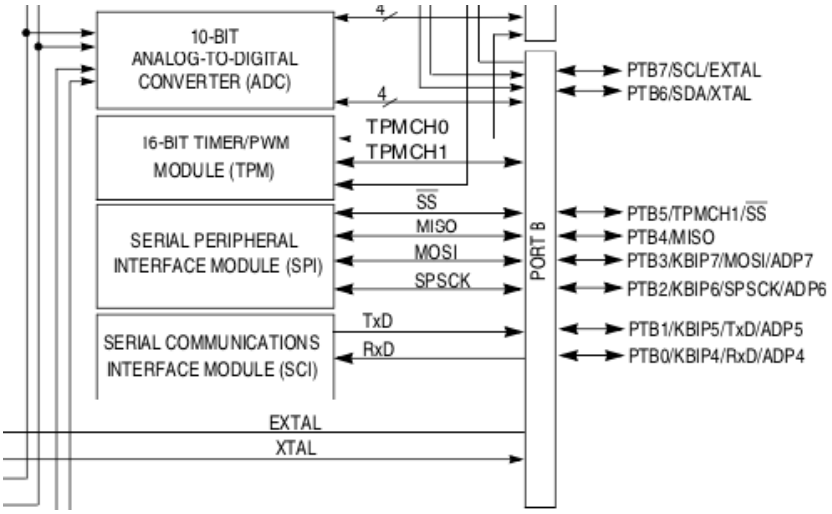

Figure 1 Port B Use for the MC9S08QG8 (From Freescale, MC9S08QG8.pdf, p20)

The way this worked, a given pin was default general purpose I/O. But if an alternate module of the device was enables, that device would take over the pin. So, enabling the serial port disabled the general-purpose capability of pins 0 and 1 as shown.

The Kinetis KL43Z processor is a more modern product that is being used for EE342. The trend toward multiple uses of pins has only gotten more complex. Because of this, the whole approach of mapping functionality to pins has been changed, even for general purpose I/O. This, I believe, reflects that most functionality of modern microcontrollers is exercised though the more specialized modules such as A/D and D/A channels, specialized series

communications protocols, and hosting particular peripherals, such as LCD displays for the KL43Z. Increasingly, the microcontroller is absorbing functionality that earlier would have resided in multiple external devices. The designer chooses which modules to use, in the process making others that exist unavailable if they require the use of the same pin. If the designer needs both, then the choice is typically to use a different microcontroller family or member that has more modules and functionality (and more cost).

What has changed significantly, though, is the way modules, and general purpose I/O ports, are mapped to pins. With the increased importance of low power usage, it is necessary to shut down parts of the device that are not needed or being used, even (and perhaps especially) unused port pins. So, by default, the KL43Z port pins connect to nothing and their electronics is unpowered. The designed must now explicitly choose which pins to use and exactly how to use them. This is expressed by a "Pin Multiplex" layer in the hardware, which must be managed using corresponding software. This is a level of complexity beyond what designers using the S08 family will be familiar with. That's why this document is provided.

The KL43Z Ports:

The S08QG8 (and S08SH8) ports each included several "words" of 8 bits memory mapped as registers for data (in or out), data direction, and (in less accessible parts of memory) pull-up enables, drive strengths and slew rates. In each of these registers, bit 0 corresponded to what would control Bit 0 of the particular port. So, for example, if the Port B data direction register was 0b00001111, the upper 4 bits would be inputs and the lower 4 bits would be outputs.

The KL43Z organizes things in a manner completely orthogonal to this. Each pin is controlled by its own 32 bit word in the part of memory mapped for I/O. Figure 2 is a diagram of the pin control register. Or, rather, two separate registers, with the upper 16-bit word controlling interrupts and the bottom one the other pin properties. There is one of these for every pin (up to 32 per port) and every port (Ports A through E). This uses a lot more memory addresses than the methods used for the S08 family, but with a 32 bit address space this "waste" of address space has negligible impact.

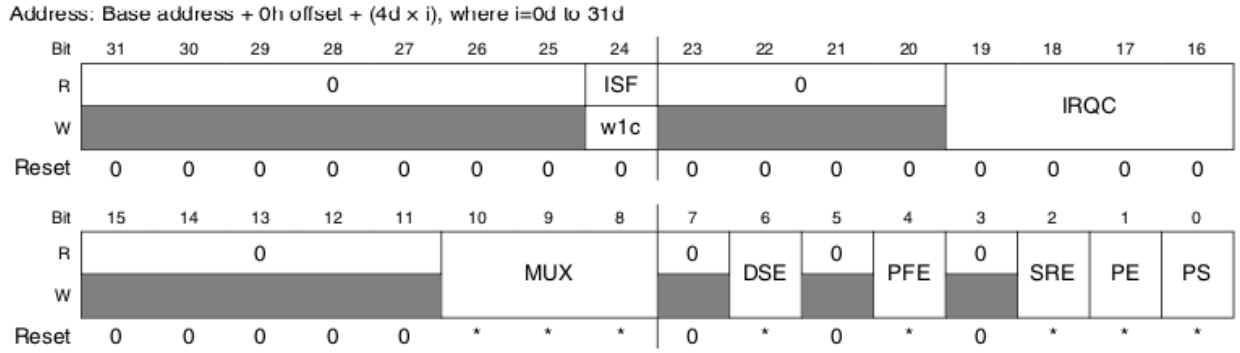

Figure 2 KL43Z Pin Control Register (From KL43ReferenceManual.pdf, page 139)

The various fields of the pin control register control the same kinds of things as the simpler single port registers did for the S08. For example, bit 6 controls drive strength (only a few port pins can be given high drive strength). Bit 4 is a filter enable (not applicable to most pins), bit 2 controls slew rate, bit 1 is the pull-up enable, and bit 0 selects whether the pull-up (if enables) is pull-up or pull-down. What is new is the three bit "MUX" field, bits 8 to 10. That

controls what is connected to the pin. A zero leaves the pin unconnected. A 1 connects the pin to its General Purpose I/O (GPIO) Port bit, which is a separate entity from the actual pins. Values of 2 to 7 can connect a given pin to "alternative" module signals which vary with the pin, just as the S08 Port B pins had a variety of different alternate uses.

Notice what is missing: the data! Data direction is missing as well. That's because they are logically part of the "Port" rather than electrical aspects of the pin (such as slew rate, strength, etc.) What had been simply a "Port" in the S08 has been pulled apart into separate abstraction levels of "Pin Multiplexing" and "General Purpose I/O."

There are some additional registers for pin control that are important, but that will not be detailed here. One must ensure that the clock for the port is turned on. (That's in "System Options Register 5", just as there are clock controls for other modules.) Unclocked pins won't do anything, and won't dissipate power. There are also provisions for simultaneously writing the same control data into multiple pin control registers that we won't need to get into. Interrupt control and status can be important, but will not be further addressed in this document.

The actual control of pin direction, and putting data into the pins (or reading data out) is "General Purpose I/O" and is considered a separate module from the pin control described above. The GPIO registers are organized more like those of the S08, with 32 bit registers, each bit corresponding to a pin of a given port. But, of course, it's more complicated than that! Figure 3 shows an excerpt from the GPIO Memory Map showing the registers associated with GPIO Port A.

GPIO memory map

Absolute address (hex)	Register name	Width (in bits)	Access	Reset value	Section/ page
400F_F000	Port Data Output Register (GPIOA_PDOR)	32	R/W	0000_0000h	41.3.1/831
400F_F004	Port Set Output Register (GPIOA_PSOR)	32	W (always reads 0)	0000_0000h	41.3.2/832
400F_F008	Port Clear Output Register (GPIOA_PCOR)	32	W (always reads 0)	0000_0000h	41.3.3/832
400F_F00C	Port Toggle Output Register (GPIOA_PTOR)	32	W (always reads 0)	0000_0000h	41.3.4/833
400F_F010	Port Data Input Register (GPIOA_PDIR)	32	R	0000_0000h	41.3.5/833
400F_F014	Port Data Direction Register (GPIOA_PDDR)	32	R/W	0000_0000h	41.3.6/834
400F_F040	Port Data Output Register (GPIOB_PDOR)	32	R/W	0000_0000h	41.3.1/831

(Note: registers for other ports follow according to the pattern for port A.)

Figure 3 Memory Map Excerpt Showing Port A General Purpose I/O Registers

There is a data direction register much like that in the S08 family which controls the directions of port pins (those actually connected to physical pins). The Port Data Output Register can be written to simultaneously change all of the bits of a particular port. But it is also possible to specifically choose particular bits to be set, cleared, or toggled. (Perhaps this makes up for the lack of bit manipulation in the instruction set.) The Port Data Input Register allows

reading all of the port pins. (If you want to know the status of just one pin, a mask and shift operation is needed.)

Access to the Ports: Hardware vs Software

Here's a big difference between programming for the S08 family (or other earlier microcontrollers) and programming the KZ43Z and other more modern microcontrollers: In the KL43Z, the hardware is "hidden". With the S08, if you wanted to configure PTB7 and PTB6 as outputs having the value "10", you would simply write C code as follows:

```
PTBDD = 0b11000000; /* Set PTB bits 6 and 7 to be outputs */
PTBD  = 0b10000000; /* Make top two bits = 10 */
```

In this way of programming, the programmer has direct access to the hardware facilities of the machine. This is contrary to trends in programming and software engineering. On a small project, direct manipulation of input and output may work out because there is likely just one programmer, and the scope of the project is so limited that the opportunity for confusion is minimum. But, on a large project with multiple programmers, each responsible for a small subset of the overall project, it is difficult if not impossible for each programmer to "know about everything". Modern software practice is to subdivide the software into modules, for which interfaces define everything someone outside the module needs to know in order to use the module. The use of global variables is discouraged. Direct interaction with hardware, except for creatures assigned to write "drivers", is especially frowned upon. So, the idea is that each programmer can only see the external appearance of other people's modules, and is the only one with access to the internals of how his own modules work.

In C, this idea is implemented by using "header" (.h) files for the external interfaces of a module. A module will usually include a header file "module.h" (for whatever "module" it is) that contains constants (generally defined symbols) others need to know about, and the declarations of function named that users of the module call in order to do what they need it to do. The .h file will often include considerable comments about how to use the functions declared (to exist). The actual code for the functions are in a "module.c" file. Conceptually, nobody else should need to look at the "module.c" file, just the person doing the development.

This is the approach that is used for programming the KL43Z. The programmer is not expected to know any of the details about how the hardware for the pin multiplexing and port control works, or the addresses in memory to which the port pins' registers are mapped. All the programmer needs to know is the set of functions needed to initialize and put data out to or read the pins. But, it is interesting to see how this all works: how is this software model implemented. The way general purpose I/O is done is described in some detail here as an example of similar modules that are used for other units within the microcontroller.

KL43Z Pin Multiplexing (Pin Control)

This is a separate "module" of the KL43Z software, with a "fsl_port.h" header file. If you wanted to initialize pins for doing input and output in, say, a module "board.c" that initializes the FRDM-KL43Z product, you need to do include the "fsl_port.h" file. The heart of the "fsl_port.h" file is the definition of a structure (C "struct") that corresponds exactly to the hardware register (bottom 16 bits) that controls the pin. The structure is defined as being a "port_pin_config_t". In a program, then, you can create one of these by defining a

port_pin_config_t and filling it in. But, the way you do that is by calling a function PORT_SetPinConfig and passing it the structure with all the settings you have selected for that pin.

```
/*! @brief PORT pin config structure */
typedef struct _port_pin_config
{
    uint16_t pullSelect : 2; /*!< no-pull/pull-down/pull-up select */
    uint16_t slewRate : 1;   /*!< fast/slow slew rate Configure */
    uint16_t : 1;
    uint16_t passiveFilterEnable : 1; /*!< passive filter enable/disable */
#if defined(FSL_FEATURE_PORT_HAS_OPEN_DRAIN) && FSL_FEATURE_PORT_HAS_OPEN_DRAIN
    uint16_t openDrainEnable : 1; /*!< open drain enable/disable */
#else
    uint16_t : 1;
#endif                           /* FSL_FEATURE_PORT_HAS_OPEN_DRAIN */
    uint16_t driveStrength : 1; /*!< fast/slow drive strength configure */
    uint16_t : 1;
    uint16_t mux : 3; /*!< pin mux Configure */
    uint16_t : 4;
#if defined(FSL_FEATURE_PORT_HAS_PIN_CONTROL_LOCK) &&
FSL_FEATURE_PORT_HAS_PIN_CONTROL_LOCK
    uint16_t lockRegister : 1; /*!< lock/unlock the pcr field[15:0] */
#else
    uint16_t : 1;
#endif /* FSL_FEATURE_PORT_HAS_PIN_CONTROL_LOCK */
} port_pin_config_t;
```

This structure is more easily understood (but without the details) in a structured comment in the file giving an example of its use for initializing a pin:

```
/*!
 * @brief Sets the port PCR register.
 *
 * This is an example to define an input pin or output pin PCR configuration:
 * @code
 * // Define a digital input pin PCR configuration
 * port_pin_config_t config = {
 *      kPORT_PullUp,
 *      kPORT_FastSlewRate,
 *      kPORT_PassiveFilterDisable,
 *      kPORT_OpenDrainDisable,
 *      kPORT_LowDriveStrength,
 *      kPORT_MuxAsGpio,
 *      kPORT_UnLockRegister,
 * };
 * @endcode
 *
 * @param base  PORT peripheral base pointer.
 * @param pin   PORT pin number.
 * @param config PORT PCR register configure structure.
 */
```

The way this structure would be created and filled in in a C program would be something like the code fragment below. “GREEN_LED” is the name given to this particular pin, since it is physically connected to a green LED that is to be controlled. Once the struct LED_GREEN is defined, a call to `PORT_SetPinConfig` is made using the constants `BOARD_LED_GREEN_GPIO_PORT` (Which is the base address for Port E, the constant PORTE), `BOARD_LED_GREEN_GPIO_PIN` (which is defined elsewhere as 32, the pin of port E that we want), and LED_GREEN, the structure that

is loaded up with the configuration we want. The (all caps) constants would be defined with #include statements in one of the user's "module.h" files, which would be included in the file containing the sample code shown here. Device specific constants are in other include files, "MKL43Z4.h" in the case of "PORTE", which are included in "fsl_port.h"

```
    const port_pin_config_t LED_GREEN
                   ={kPORT_PullDisable,kPORT_SlowSlewRate,kPORT_PassiveFilterDisable,
                    kPORT_LowDriveStrength, kPORT_MuxAsGpio};
    PORT_SetPinConfig(BOARD_LED_GREEN_GPIO_PORT,BOARD_LED_GREEN_GPIO_PIN, &LED_GREEN);
```

The various constants such as "`kPORT_SlowSlewRate`" are defined in enumerations earlier in "fsl_port.h". For example, the pull-up enumeration for the 2 bits of pull up options are defined as follows. So, rather than having to remember that no pullup is 0, the programmer only needs to know that the constant to put into the structure is "`kPORT_PullDisable`".

```
/*! @brief  Internal resistor pull feature selection */
enum _port_pull
{
    kPORT_PullDisable = 0U, /*!< internal pull-up/down resistor is disabled. */
    kPORT_PullDown = 2U,    /*!< internal pull-down resistor is enabled. */
    kPORT_PullUp = 3U,      /*!< internal pull-up resistor is enabled. */
};
```

The function PORT_SetPinConfig is not just declared in "ports.h". It is also defined there. That is, we have the code for it (rather than the code being in a separate "fsl_port.c" file. The function definition is:

```
static inline void PORT_SetPinConfig(PORT_Type *base, uint32_t pin, const
port_pin_config_t *config)
{
    assert(config);
    uint32_t addr = (uint32_t)&base->PCR[pin];
    *(volatile uint16_t *)(addr) = *((const uint16_t *)config);
}
```

This is a bit hard to follow because of the pointer structure for finding the correct address in memory, and the casts to make it work. Essentially, that 16 bit structure is turned into a 16 bit word and plopped down in the appropriate address for the port and pin given, setting all of the pin control fields simultaneously. Because this is an "inline" function, it is placed directly in the code whenever the source code shows a call, rather than have the source code call a subroutine, as would be the case for most functions. That is why the function definition needs to be here in the header file rather than in a separate .c source file.

The details won't be presented here, but somewhere around here in the code the clock for the port also needs to be turned on:

```
    CLOCK_EnableClock(kCLOCK_PortE);
```

So, after all that, the pin has been connected to the Port E GPIO module, and the electronics of the pin have been configured (no pull-ups, etc.). But it has not yet been made either an input nor an output, and does not have any but the default value of 0 (should it become an output).

KL43Z GPIO Pin Use

Before the pin can be used, it must be configured. The appropriate include file is "fsl_gpio.h". The amount of information needed is just the data direction and (if an output) the initial value. Naturally, that is implemented as a structure. So, a structure is defined and then an initialization function is called to actually set the registers. As in the case of the port setup, various constants are defined in enumerations for the options that can be used. The structure definition in the "fsl_gpio.h" file is:

```
/*!
 * @brief The GPIO pin configuration structure.
 *
 * Every pin can only be configured as either output pin or input pin at a time.
 * If configured as a input pin, then leave the outputConfig unused
 * Note : In some cases, the corresponding port property should be configured in advance
 *          with the PORT_SetPinConfig()
 */
typedef struct _gpio_pin_config
{
    gpio_pin_direction_t pinDirection; /*!< gpio direction, input or output */
    /* Output configurations, please ignore if configured as a input one */
    uint8_t outputLogic; /*!< Set default output logic, no use in input */
} gpio_pin_config_t;
```

Here is the enumeration used for pin direction:

```
/*! @brief GPIO direction definition*/
typedef enum _gpio_pin_direction
{
    kGPIO_DigitalInput = 0U,  /*!< Set current pin as digital input*/
    kGPIO_DigitalOutput = 1U, /*!< Set current pin as digital output*/
} gpio_pin_direction_t;
```

The declaration of a function to actually configure the pin (as defined in the structure) is:

```
void GPIO_PinInit(GPIO_Type *base, uint32_t pin, const gpio_pin_config_t *config);
```

Unlike the pin control register function, this one is not "inline". The function definition is in the "fsl_gpio.c" file. In theory, you don't need to look at this file. But, in the interest of seeing it, here is the function. It is pretty straightforward.

```
void GPIO_PinInit(GPIO_Type *base, uint32_t pin, const gpio_pin_config_t *config)
{
    assert(config);

    if (config->pinDirection == kGPIO_DigitalInput)
    {
        base->PDDR &= ~(1U << pin);
    }
    else
    {
        GPIO_WritePinOutput(base, pin, config->outputLogic);
        base->PDDR |= (1U << pin);
    }
}
```

For the green LED example cited earlier, the code in one's program would include the "fsl_gpio.h" file, and would have where the setup of hardware is being done the following code fragment. After executing this, the LED will see an output of "1", which will leave it off. (On the KL43Z board, the LED's are driven active-low, so a 0 output turns it on.)

```
    gpio_pin_config_t LED_GREEN_config ={
                    .pinDirection =kGPIO_DigitalOutput,
                    .outputLogic = 1U};
    GPIO_PinInit(BOARD_LED_GREEN_GPIO, BOARD_LED_GREEN_GPIO_PIN, &LED_GREEN_config);
```

Now that the Port multiplexing and GPIO configuration is complete, the pin driving the green LED is available for use. One can turn it on and off with calls to the GPIO_WritePinOutput function, which is an inline function defined in "fsl.gpio.h":

```
/*! @name GPIO Output Operations */
/*@{*/

/*!
 * @brief Sets the output level of the multiple GPIO pins to the logic 1 or 0.
 *
 * @param base   GPIO peripheral base pointer(GPIOA, GPIOB, GPIOC, and so on.)
 * @param pin    GPIO pin's number
 * @param output GPIO pin output logic level.
 *         - 0: corresponding pin output low logic level.
 *         - 1: corresponding pin output high logic level.
 */
static inline void GPIO_WritePinOutput(GPIO_Type *base, uint32_t pin, uint8_t output)
{
    if (output == 0U)
    {
        base->PCOR = 1 << pin;
    }
    else
    {
        base->PSOR = 1 << pin;
    }
}
```

This is fairly straightforward. For the green LED example, we could turn it on thus:

```
output=0;
GPIO_PinInit(BOARD_LED_GREEN_GPIO, BOARD_LED_GREEN_GPIO_PIN, \
          &(gpio_pin_config_t){kGPIO_DigitalOutput, (output)})
```

Or, alternatively (and more simply):

```
GPIO_ClearPinsOutput(BOARD_LED_GREEN_GPIO, 1U << BOARD_LED_GREEN_GPIO_PIN);
```

Turning the LED off again would be:

```
GPIO_SetPinsOutput(BOARD_LED_GREEN_GPIO, 1U << BOARD_LED_GREEN_GPIO_PIN)
```

Conclusion:

Get to know and understand C structures; we will be using them to drive the hardware. What we are doing here to get individual port bits to work is similar to how we do other stuff.