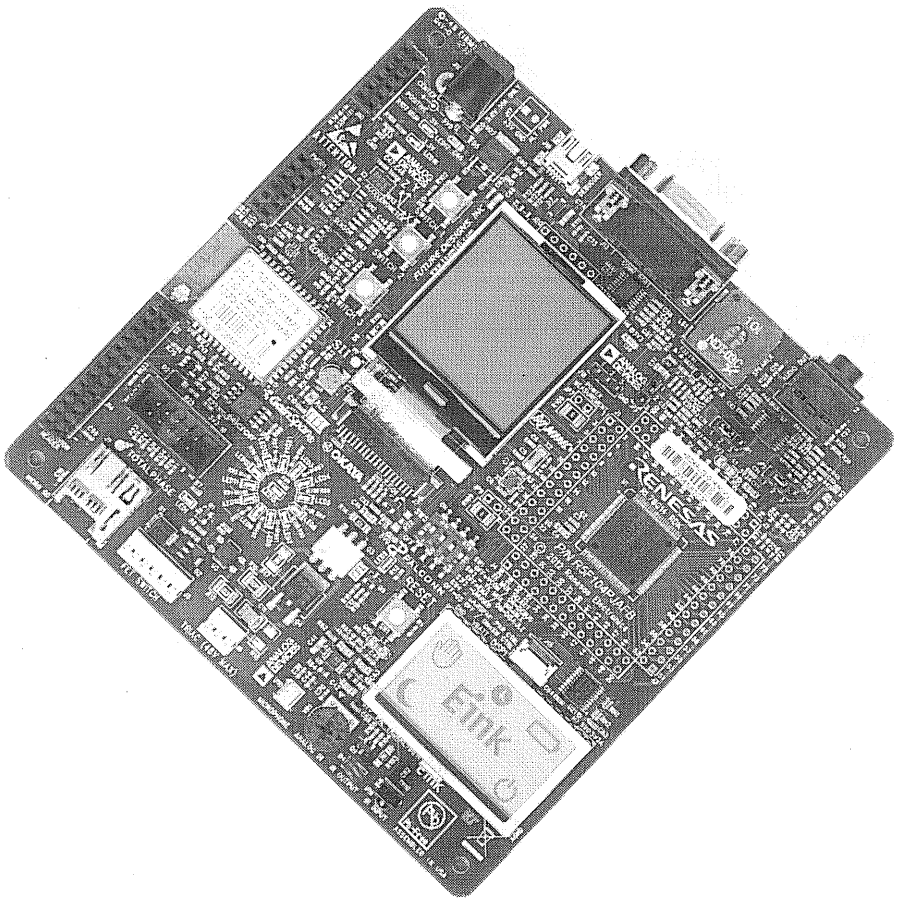
RENESAS

Embedded Software

Lecture notes (Rev. 1.0)

A training course for new engineers



Renesas Electronics Corp.
Renesas Design Vietnam
Engineering Software Department

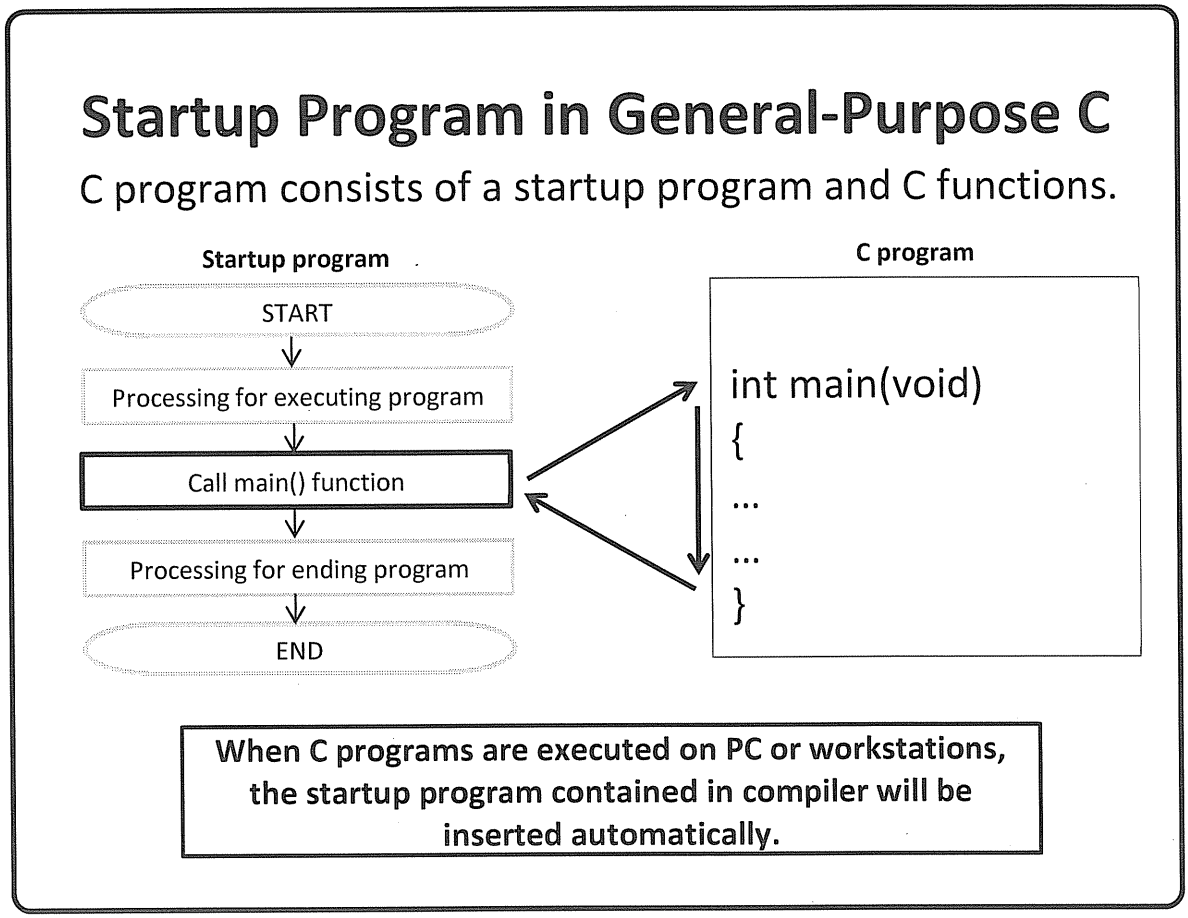
Table of contents

Chapter 1.

Introduction to Embedded C

In this chapter:

- Startup Program
- Memory Partition
- Hardware Access



Startup program

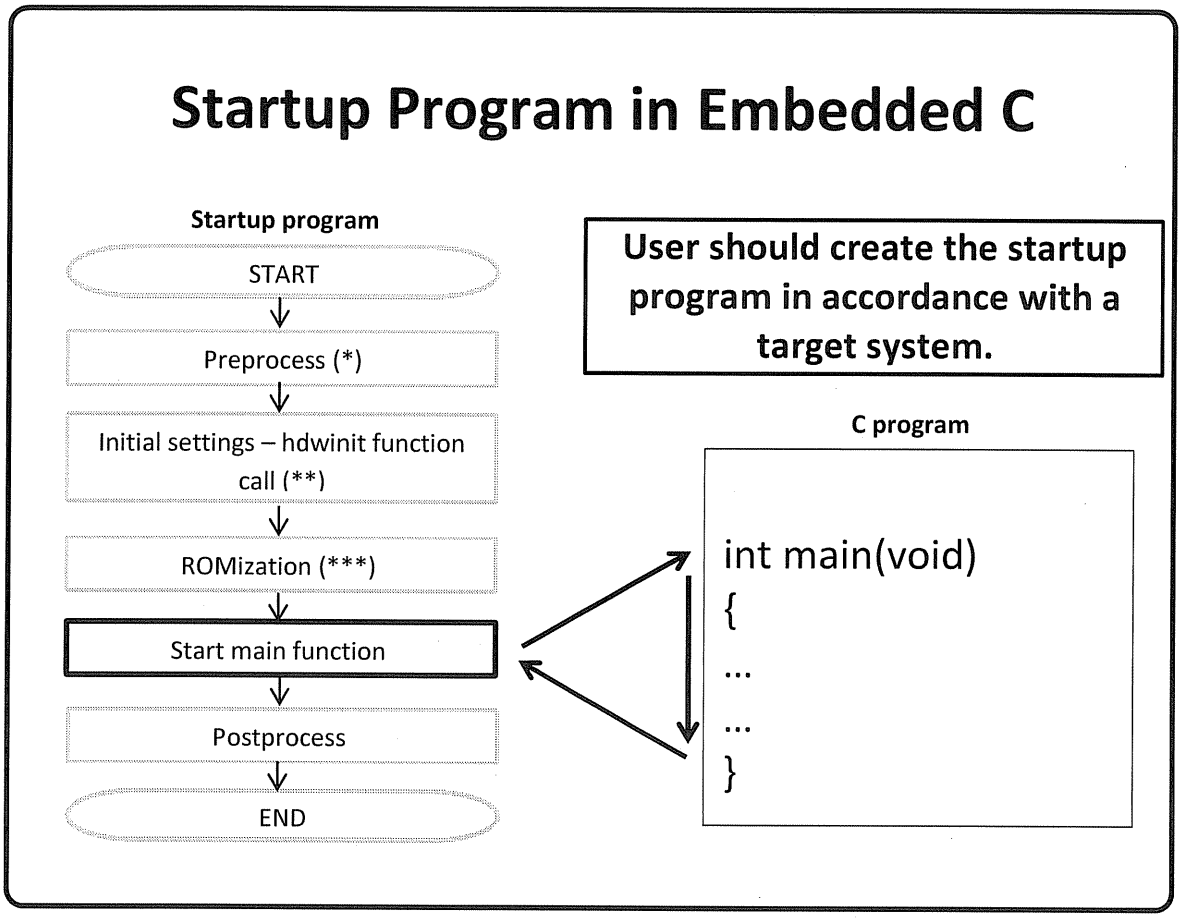
This is a program that performs the initialization of a microcomputer and peripheral devices, as well as to call main() function.

Startup program in general-purpose C

C programs executed on personal computers or workstations are called "general-purpose". In personal computers or workstations which are equipped with a general-purpose OS such as Windows or Unix, hardware initialization is done by the OS. Startup program is designed in such a way that it is complied with OS requirements, thus, it is unnecessary for user to write it. Usually, it is automatically created by a compiler.

OS

OS is fundamental software that provides a variety of services according to the various requests from a system. OS is classified into two categories; one is *general-purpose OS* which provides the environments for both application execution and development, the other is *embedded OS* which is specialized for a target system and aimed to bring out the utmost performance of the system.



Embedded C programs

Unlike PC already equipped with an application execution environment, we must equip a microcomputer with some program to control devices along with its hardware. A microcomputer embedded with such a system is called "embedded system" and the C language used for developing embedded system is called "embedded C language", and the object to be embedded is called "target system".

Startup program in embedded system

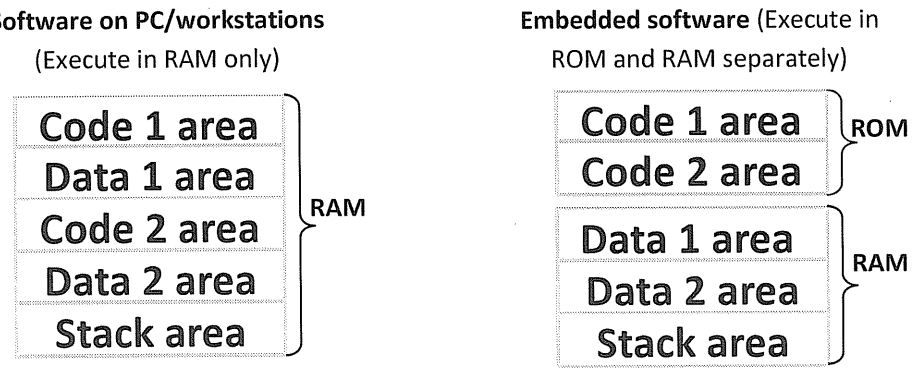
This program is necessary for an embedded system to initialize hardware and variables based on a target system. Therefore, users need to write it individually for each target system. In general, a sample of startup program is provided by a compiler that supports a microcomputer. Users can modify this sample program directly according to the target system.

Notes

- (*) If standard library is used, the processing related to the library is performed first.
- (**) hdwinit is a function created by user to initialize peripheral devices.
- (***) ROMization is a process of copying values located in ROM to variables in RAM.

Partitioning Memory to Use

Memory area is divided into ROM and RAM for running embedded application software.

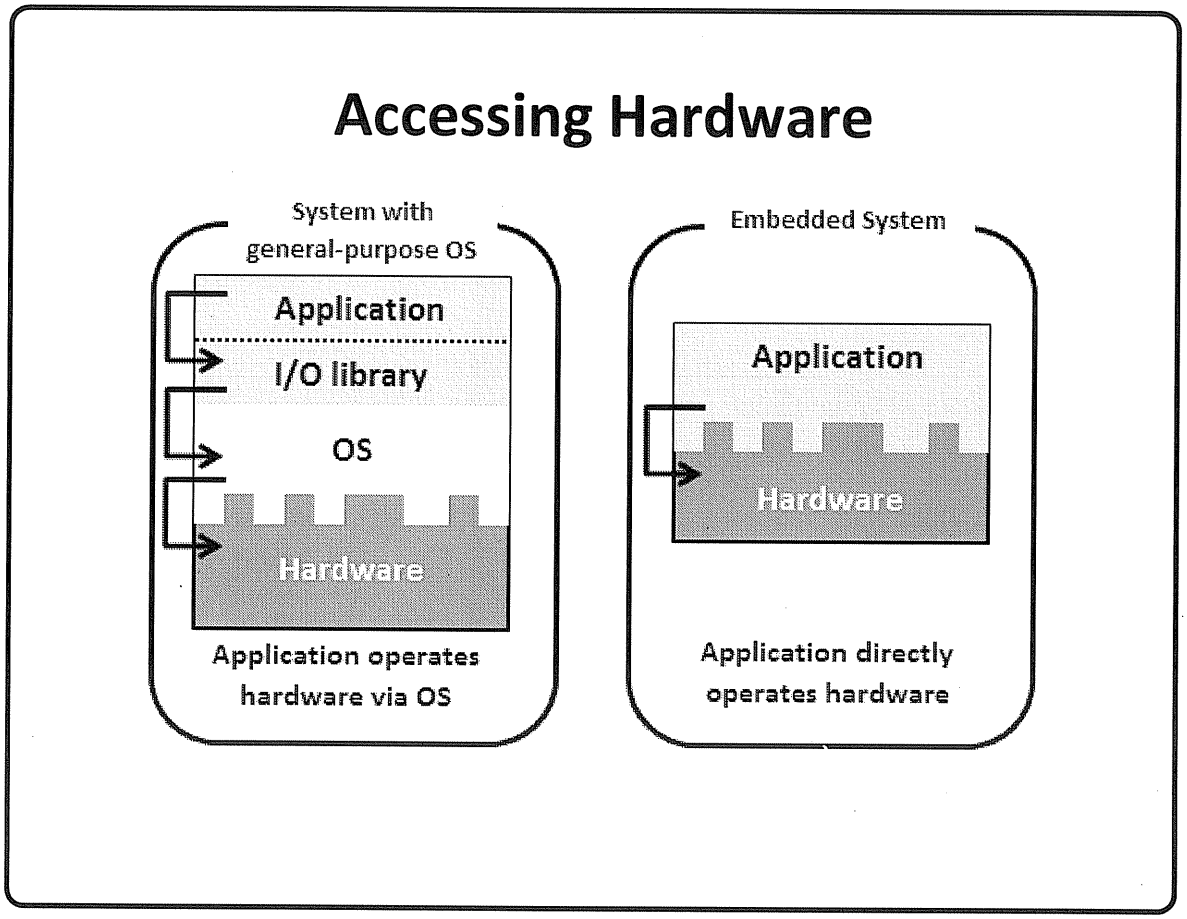


Memory of PC/workstation and embedded system

On PC or workstation, source code and data of a general-purpose C program are all loaded to main memory (RAM) by OS via external storage devices and executed in RAM. In other words, code/data/stack areas are all in main memory (RAM). Therefore, when programming with general-purpose C language users do not need to mind the difference between ROM and RAM. On the contrary, with embedded system, source code and data/stack must be allocated into two different areas ROM and RAM respectively.

ROM and RAM

ROM (Read Only Memory) is memory that can be read only. It includes microcomputer built-in ROM, external masked ROM, electrically erasable programmable EEPROM, PROM, etc... The data stored in ROM is retained even when there is no power supplied to the system. RAM (Random Access Memory) is memory that can be read from and written to. It includes built-in RAM of microcomputer, external SRAM, DRAM, etc... The data is only retained in RAM if there is a power supplied to the system. Recently, readable/rewritable flash memory that can hold stored contents without power supply is also used as ROM.



System with general-purpose OS

A general-purpose C program accesses hardware through the prepared I/O library.

Embedded system

When accessing hardware in embedded C language, we must write programs to directly control the hardware.

I/O library

The set of I/O functions prepared by compiler for utilizing standard keyboard and monitor attached to PC/workstation is called standard I/O library. An application program can access the standard hardware by the use of the library functions. Sometimes, a dedicated I/O library for writing programs to control a device mounted on the system is also provided by device manufacturer. Here, both of them are called I/O library.

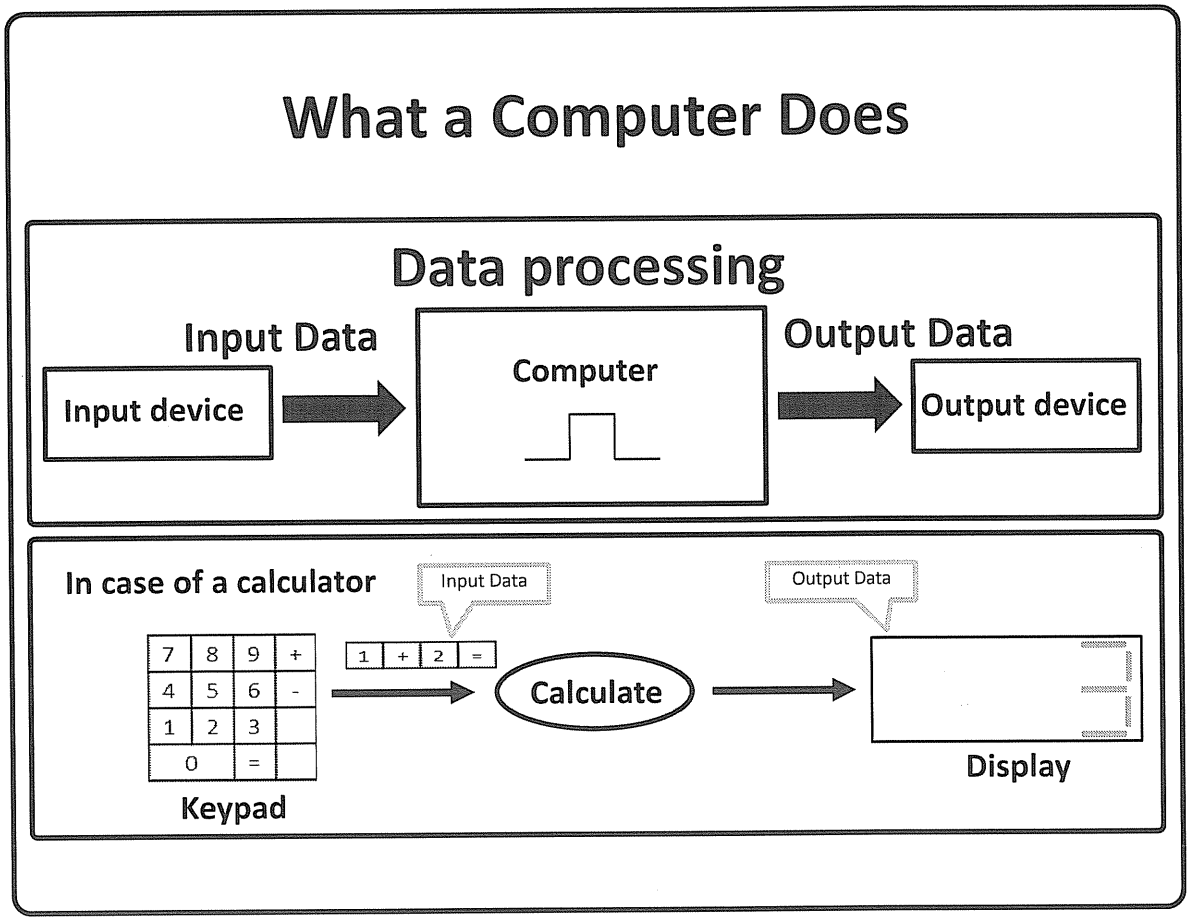
(This page intentionally left blank)

Chapter 2.

Introduction to Microcomputer

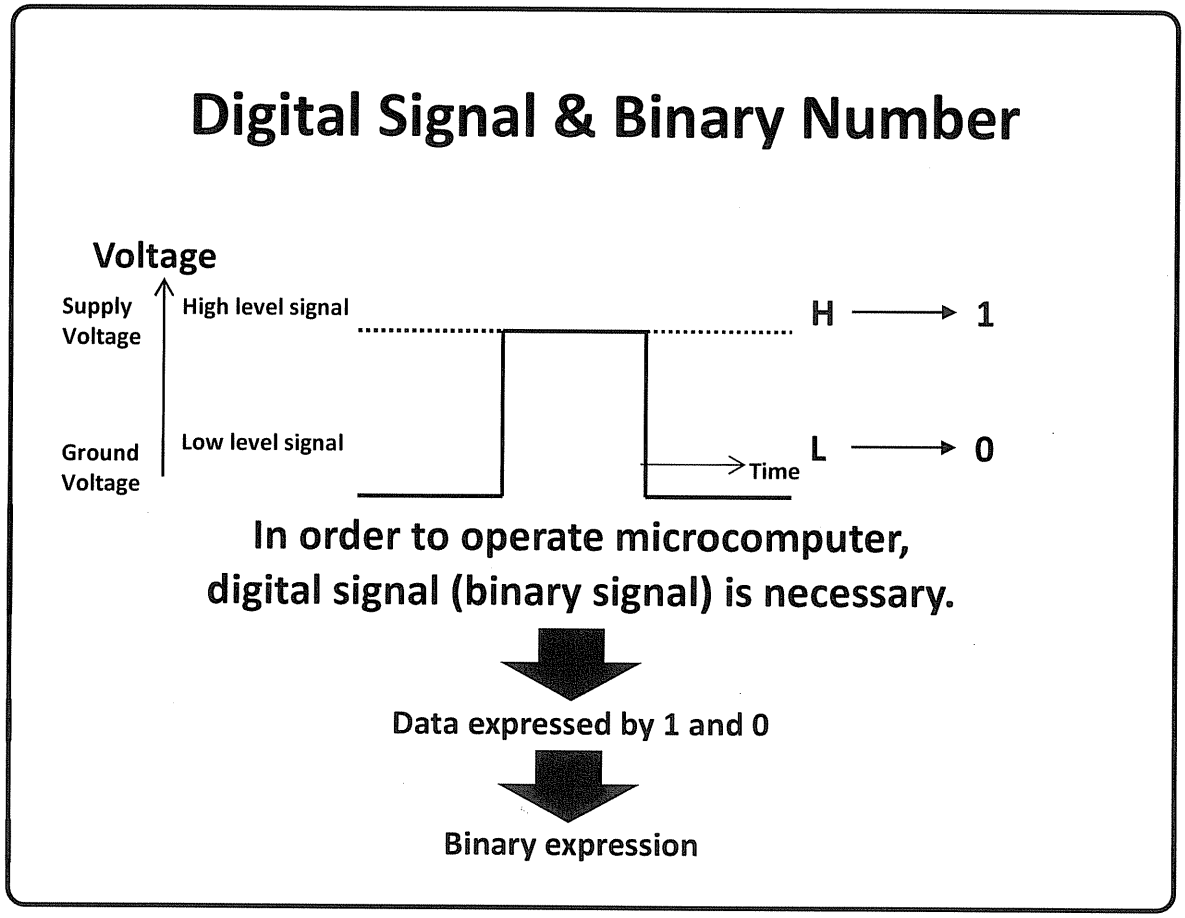
In this chapter:

- Digital Signal and Numeric Value
- Basic Structure and Operation of Microcomputer
- RL78 Architecture and Core Overview



What a computer does?

Computer's job is to read data (signal) from input device, then process and convert this data into a signal that can be identified by output device. Finally, the processed data will be sent to output device for further purposes e.g., displaying it on LCD device...



Information Used by Microcomputers

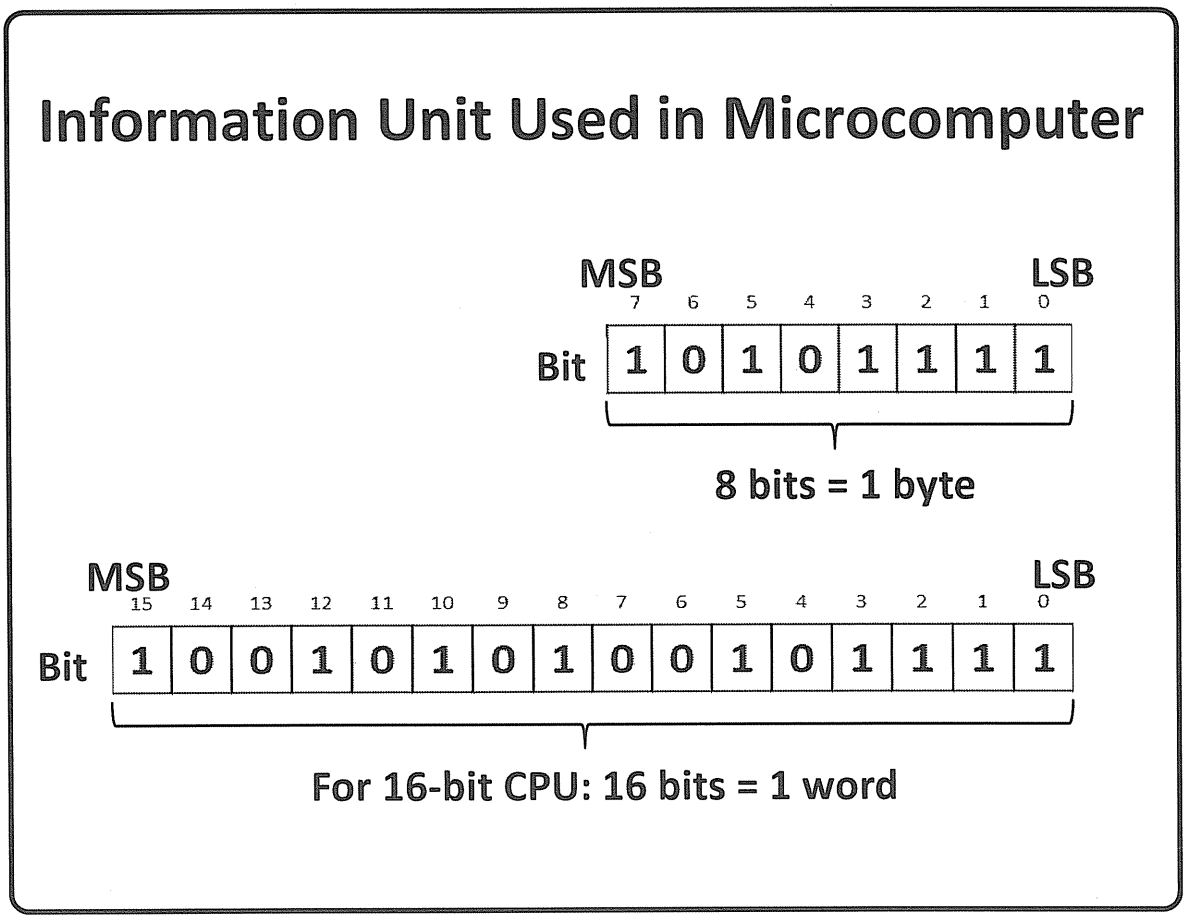
All information in microcomputers is in combination of "1"s and "0"s.

Digital and analog signals

The information which is expressed by using only 2 digits, namely "1" and "0" is called "digital". Conversely, a signal whose values spread on a continuous range is called "analog". Analog signal is the most natural kind of signal, and it presents everywhere, for example:

- ambient temperature
- air pressure
- speed of a vehicle

As microcomputer only understands and operates with digital signal, we normally need to convert analog data into digital data for microcomputer to do processing.



Bit

Bit is an abbreviation of "Binary Digit", and is the smallest unit which microcomputer can deal with. There are two kinds of value that are binary digital "0" and "1" for one bit.

Byte

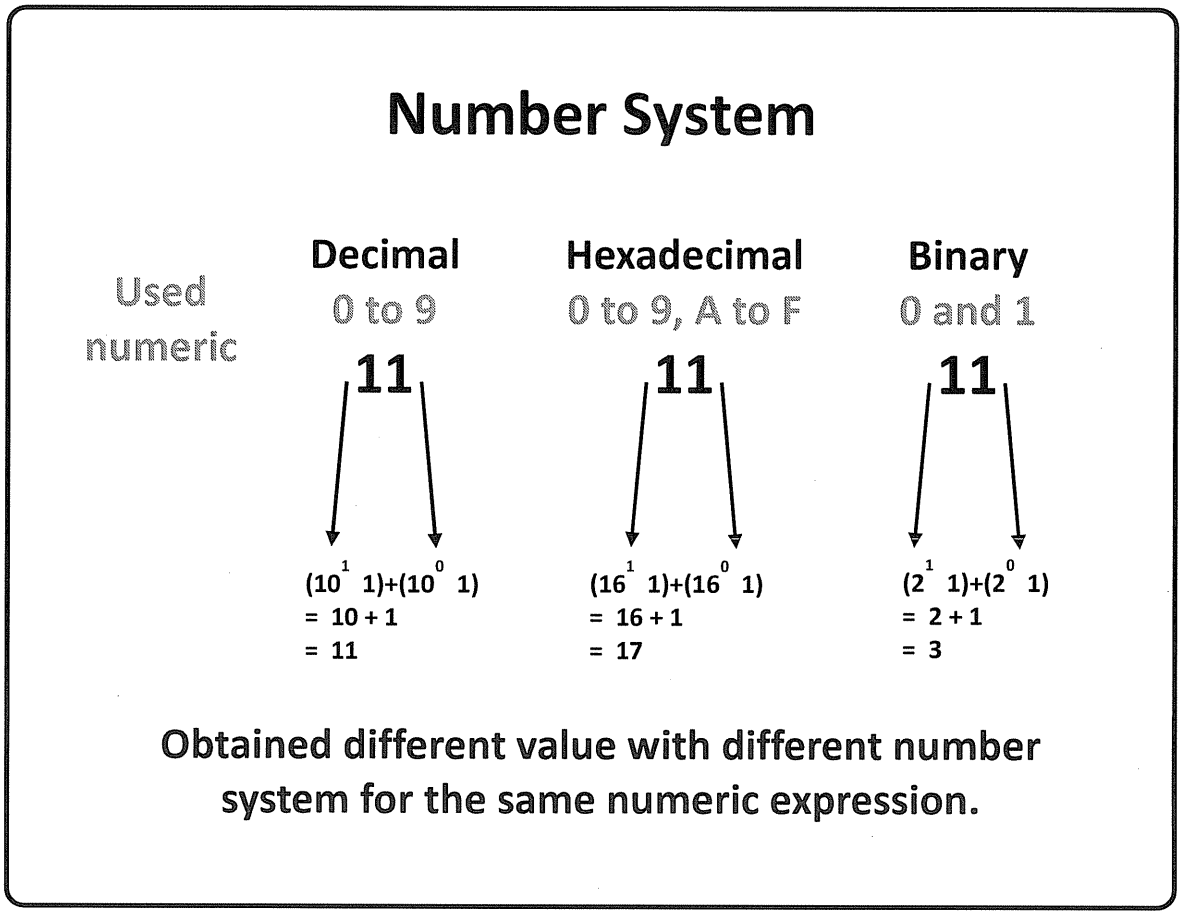
It consists of 8 bits and is the unit for data processing. One byte allows to represent 256 (2^8) different values, e.g., from 0 to 255 or -128 to 127.

Word

How many bits make up one word depends on the used computer. Normally, the length of a word is equal to the most basic-data length that a computer can handle during operating. We can also say a word size matches with the width of data bus.

LSB and MSB

When a value is expressed in binary, the lowest bit is called LSB (Least Significant Bit), and the highest bit is MSB (Most Significant Bit). Here, least significant and most significant represent the lowest and highest digit. For binary, the digit of 2^0 is the lowest one.



Number representation

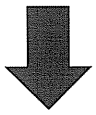
To represent number in different systems, we need to use symbols and suffixes. Furthermore, different expression methods may be applied according to high-level programming language and assembly specifications. In this training course, the following expressions are applicable:

	Example	Assembly language	C language	User's manual
Decimal	11	11	11	11
Hexadecimal	BH	0B H	0xB	B16
Binary	1011 B	1011 B	N/A	10112

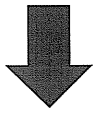
Example: Number representation in different systems

Binary, Decimal and Hexadecimal Conversion

0101B



Four-digit binary can be expressed in one-digit hexadecimal.



5H

Correspondence of binary, decimal and hexadecimal

D	B	H	D	B	H
0	0000	0	9	1001	9
1	0001	1	10	1010	A
2	0010	2	11	1011	B
3	0011	3	12	1100	C
4	0100	4	13	1101	D
5	0101	5	14	1110	E
6	0110	6	15	1111	F
7	0111	7	16	10000	10
8	1000	8	17	10001	11

D: decimal **B**: binary **H**: hexadecimal

How to express negative binary data

Each half of the numeric range expressed in bit width is allocated respectively to express the positive and negative data. For 8-bit data, the expressible range is 0 to 127 and -1 to -128. But for 16-bit data, it is 0 to 32767 and -1 to -32768. However, how to process the number "FFH"? It should be either 255 or -1 depending on our program. Usually, the most significant bit is used to represent the sign of the number (MSB = 1 corresponds to negative number).

Hexadecimal	Binary	Decimal
7FH	0111 1111	127
	:	
01H	0000 0001	1
00H	0000 0000	0
FFH	1111 1111	-1
FEH	1111 1110	-2
	:	
80H	1000 0000	-128

Expressible range of negative data with 8 bits

Standard Code

This code is used to exchange information between different devices. ASCII is a kind of standard codes, where alpha-numeric and control codes are expressed in 7-bit.

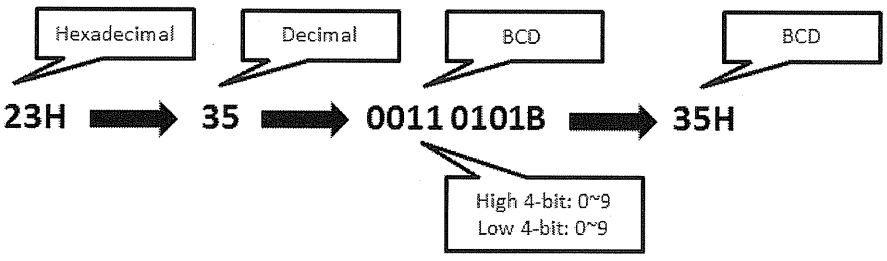
USASCII code chart

b_4	b_3	b_2	b_1	Row ↓ / Column →	0 (000)	1 (001)	2 (010)	3 (011)	4 (100)	5 (101)	6 (110)	7 (111)
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	|
1	1	0	1	13	CR	GS	-	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

(Source: http://en.wikipedia.org/wiki/ASCII)

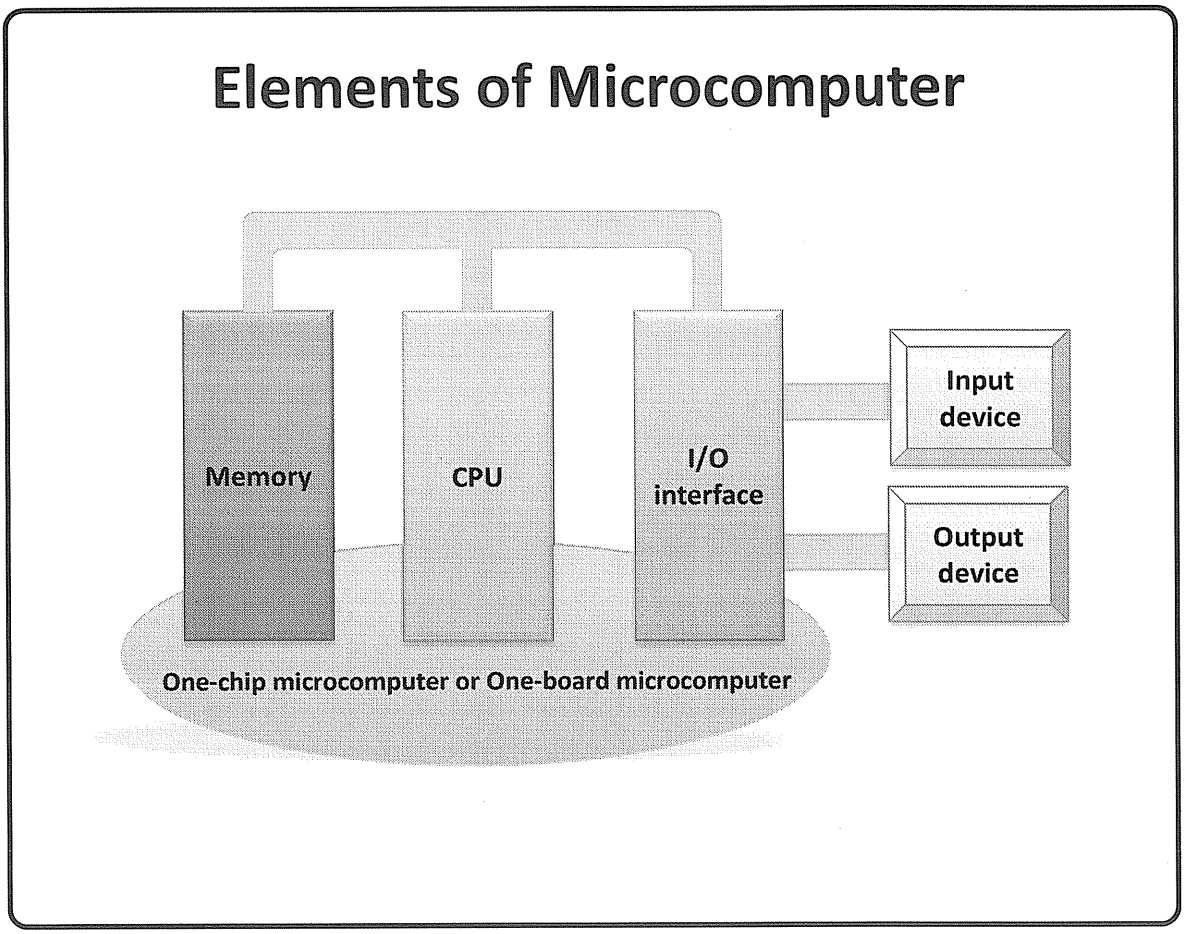
Binary Coded Decimal notation (BCD)

Since decimal is used in daily life, it is more convenient to input and output data in decimal when operating microcomputer. Therefore, we often convert decimal to binary at input operation and convert binary back to decimal at output operation. BCD is a method to complete such conversion in microcomputer operation. BCD is to express 4 binary bits in 1 decimal digit (0 to 9).



Encoding and decoding

Converting a character to a value is called "encoding". The converse operation is called "decoding".



Single-chip Microcomputer (one-chip microcomputer)

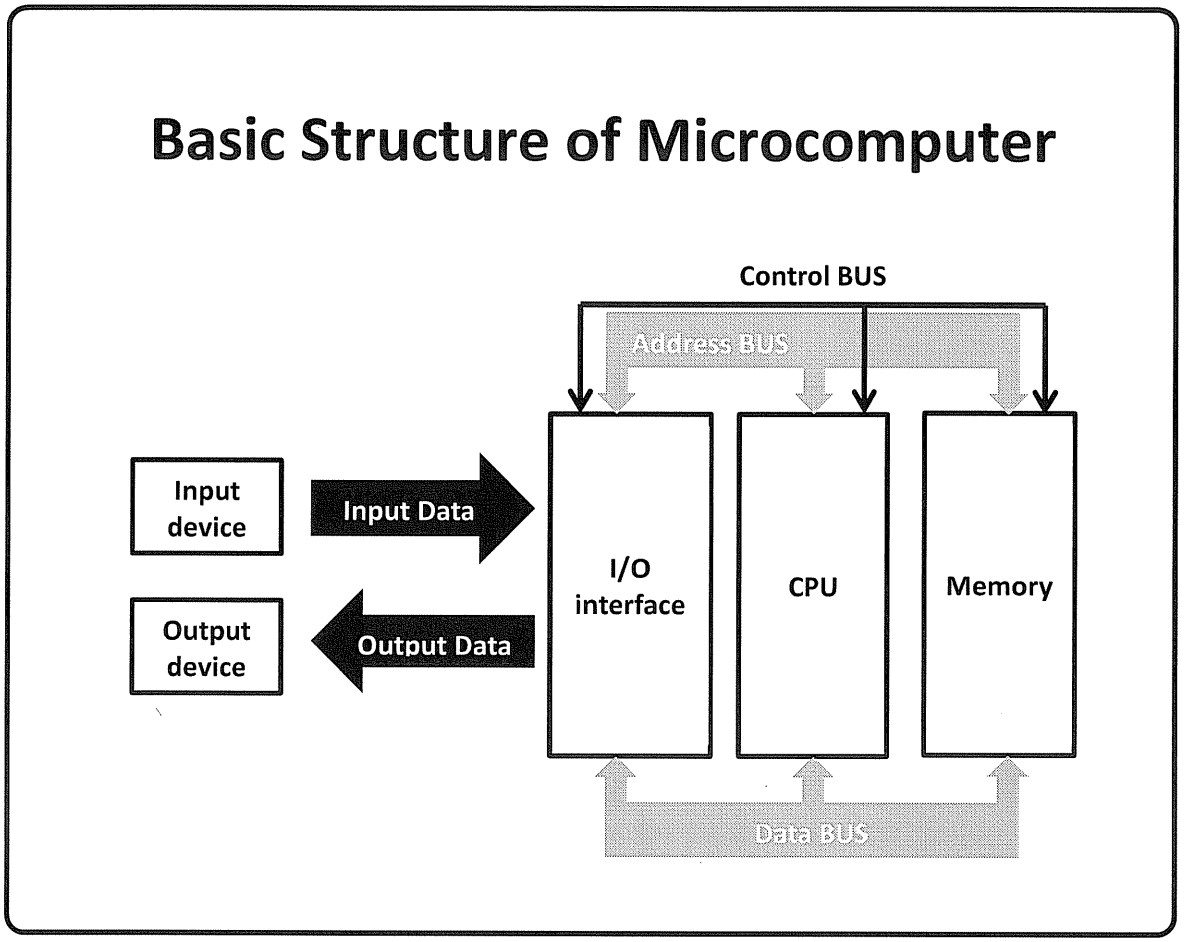
This is a packaged microcomputer on which CPU, memory and I/O interface are designed on a chip. By selecting such a microcomputer as target system, we can develop a low-cost and compact product and reduce the number of external hardware.

General-purpose Microcomputer

This is a microcomputer mounted with CPU and minimal required capacity memory. And it can also connect to a large capacity memory and LSI for having more I/O interfaces. The board equipped with a basic LSI structure is called one-board microcomputer.

LSI (Large Scale Integration)

Large-scale integration (LSI) is the process of integrating or embedding thousands of transistors on a single silicon semiconductor microchip.



CPU (Central Processing Unit)

It performs three basic tasks:

- Control: Read and execute instructions saved in memory, control timing to transfer data between memory and I/O interface, perform appropriate control to external signal.
- Computation: Perform numeric and logical computation.
- Storage: Temporarily save necessary information and operating result in order to execute instructions in a given sequence.

Memory

It stores data such as computed results, programs and so on. The data stored in memory can be identified by its "address".

I/O interface (Input/Output interface)

It provides means for CPU to communicate with external devices.

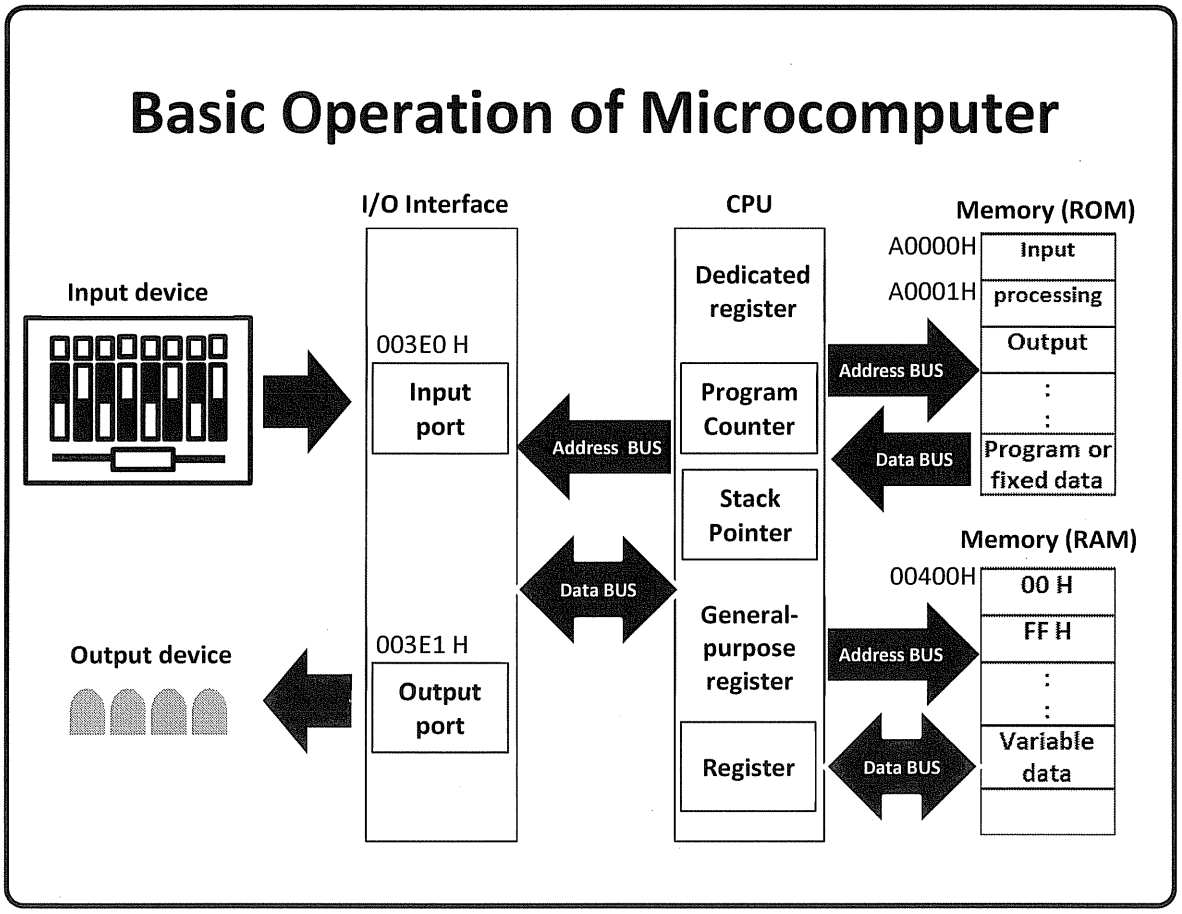
BUS

There are three kinds of bus described as below:

- Address BUS: This is a signal transfer channel used to carry address signal from CPU to memory or I/O interface.
- Data BUS: This is a signal transfer channel used to exchange data among CPU and memory, I/O interface.
- Control BUS: This is a signal transfer channel used to control timing to operate data among CPU and memory, I/O interface.

About "BUS"

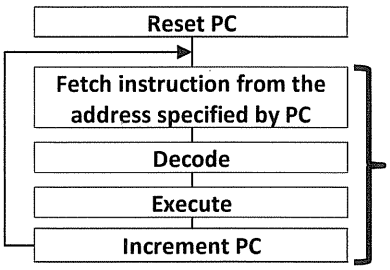
The origin meaning of this word is stagecoach for carrying passengers. In our context, we should take the following meaning: "send signal in a group to multiple destinations".



CPU register

Register is a specific memory area where data used in CPU is stored. This special memory area is called register so as to distinguish from common memory area. CPU has two kinds of registers. One is called general-purpose register used to store multi-purpose data, the other is dedicated register that has special functions such as Program Counter (PC) and Stack Pointer (SP). PC is a dedicated register indicating the address of an instruction to be executed. SP is a dedicated register used to store return address and indicate the current address of memory. Register, PC and SP will be described in more details in the next chapters.

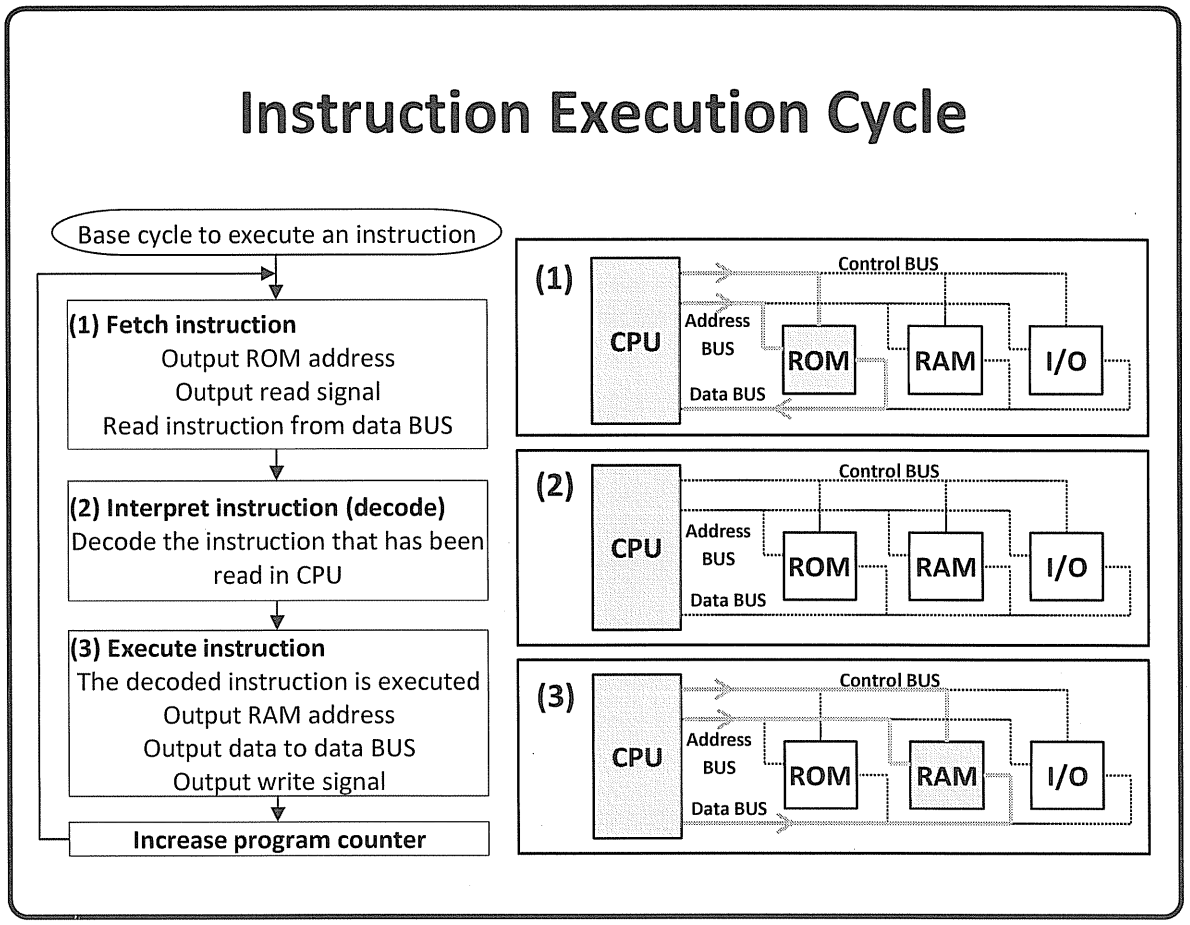
Basic operation of microcomputer



As soon as an external reset signal is applied, the inner microcomputer will be reset. PC is also reset, and the instruction would be executed from the address.

Basic cycle of instruction execution

Reading an instruction code is called fetch cycle. Computing and reading to/writing from memory/port/internal register will be performed in the execution cycle.

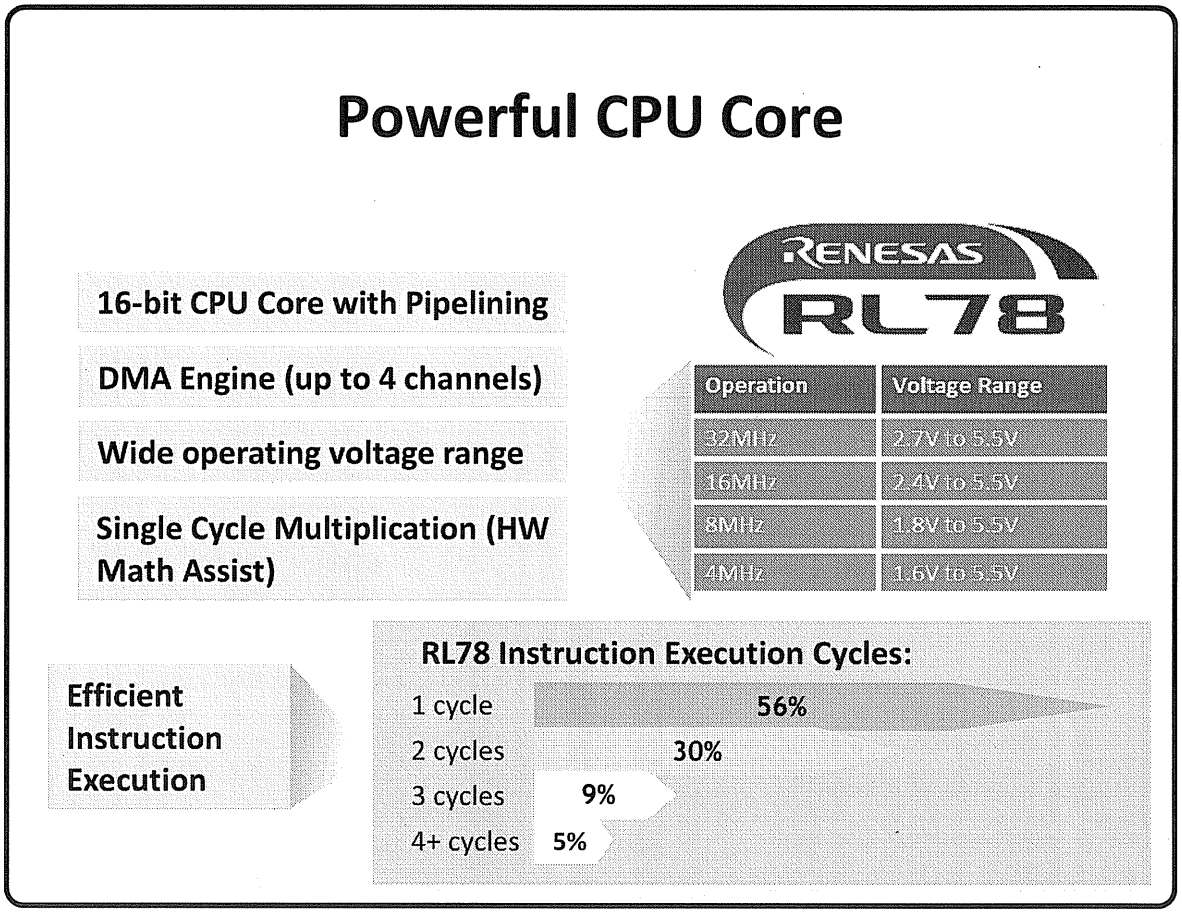


Instruction execution cycle

Operations to execute an instruction of CPU can be divided into:

- Read instruction (fetch)
- Interpret instruction (decode)
- Run instruction (execute)

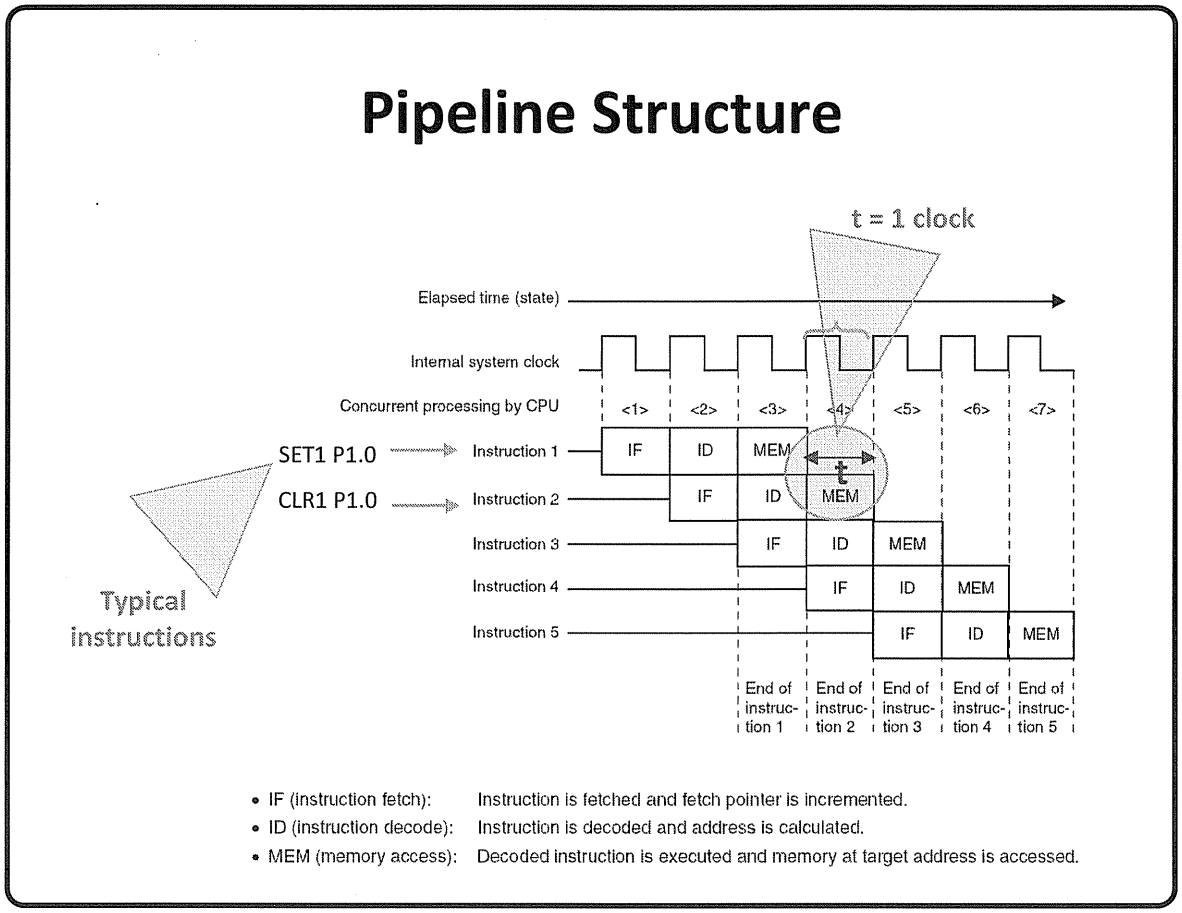
This set of operations is repeated completing the read/write action of memory, port and internal register. This operation set is called "instruction execution cycle".



A quick overview of RL78 CPU core

- It is a 16-bit CPU core with a three-stage pipeline.
- All the above different speed modes and voltage ranges are supported, which means there are no special device grades or versions required, allowing the RL78 to be implemented in a wide variety of applications.
- The RL78 also features a DMA engine with up to 4 channels. That enables direct transfer of data from peripherals to RAM area and vice versa.
- The RL78 core also features a very efficient instruction set where 86% of the instructions are done in one or two cycles. More specifically, 56% are done within one cycle, 30% in two cycles, and 9% in 3 cycles, which leaves only 5% of the overall instruction set executed in four cycles or more.
- In addition to these very efficient instructions, there are also some hardware assist functions implemented in the core.

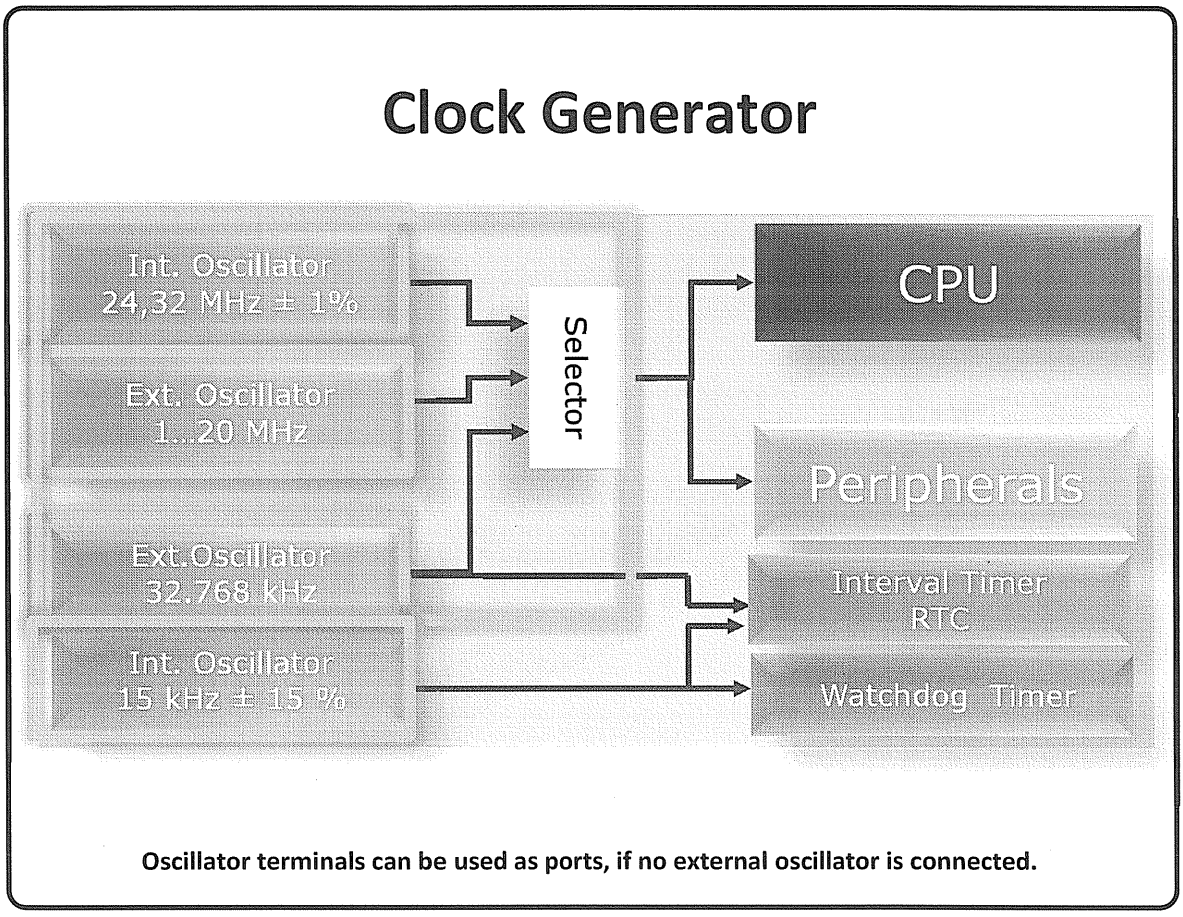
HW Assist for Math	Operation	Clock Cycles
16-bit Barrel Shifter for Shift and Rotate	16-bit n Shift/Rotate (n = 1 to 15)	1
Multiply Signed & Unsigned	16 x 16 = 32-bit result	1
Multiply/Accumulate Signed & Unsigned	16 x 16 + 32 = 32-bit result	2



Features

Three-stage pipeline control is used to enable single-cycle execution of almost all instructions. Instructions are executed in three stages: Instruction fetch (IF), instruction decode (ID), and memory access (MEM).

This three-stage pipeline enables the RL78 to execute most instructions in just one clock cycle. The actual time required to execute an instruction is dependent upon the main system clock. For example, if the device is running at 20MHz operation speed, one cycle is equal to 50 nano-seconds. This means that typical instructions like "SET1 P1.0", which is an access to a special function register (see chapter 4) for setting the port bit, are executed in only 50 nano-seconds.

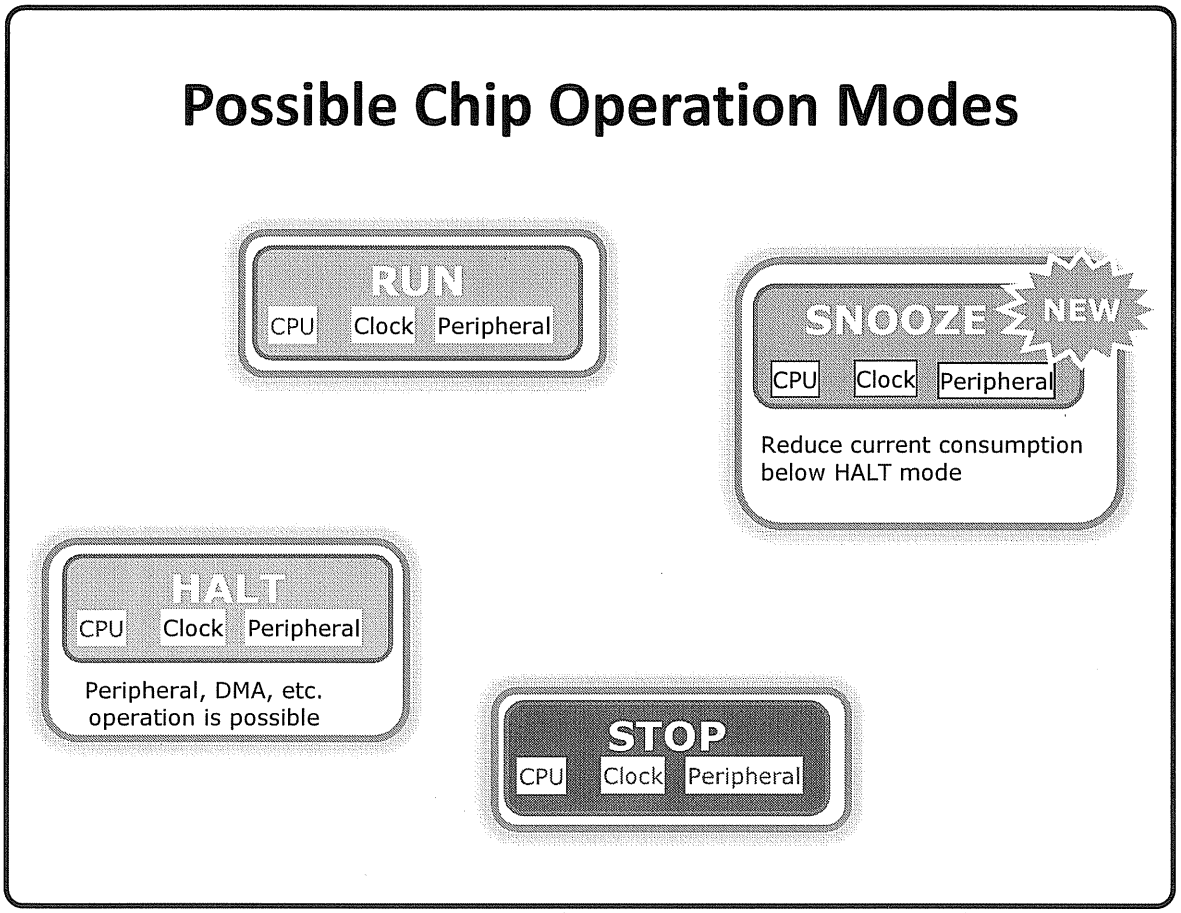


Up to four different oscillators

On the left side you see the internal oscillator operating at 24 or 32MHz with an accuracy of 1% over the whole temperature range. Next you will find the external oscillator circuit which is able to support external crystals or resonators from 1 to 20 MHz, or alternatively the external 32 KHz oscillator.

All these 3 clocks can be fed into a selector and after the selector these clocks feed into the CPU core and into the peripherals. The 32 KHz sub oscillator can be fed into the interval timer and the Real Time Clock.

The internal low speed oscillator running at only 15 KHz, does not need any additional external components and is able to supply the watchdog timer and the interval timer.



Main system clock and high-speed on-chip oscillator

- **Halt mode**
  - Oscillator is operating, CPU core is stopped, and peripherals are active.
  - Halt mode release possible with interrupts or reset.
- **Stop mode**
  - Oscillator is stopped, CPU core is stopped, and peripherals are not active.
  - Stop mode release possible with external interrupts or reset or internal interrupts by internal low speed oscillator. Or sub-system-clock operated peripherals.
- **Snooze mode (internal high-speed oscillator)**
  - CSI00 or UART0 data reception is possible (see chapter 9 for CSI and UART).
  - Timer triggered A-D conversion (see chapter 9 for A-D conversion).

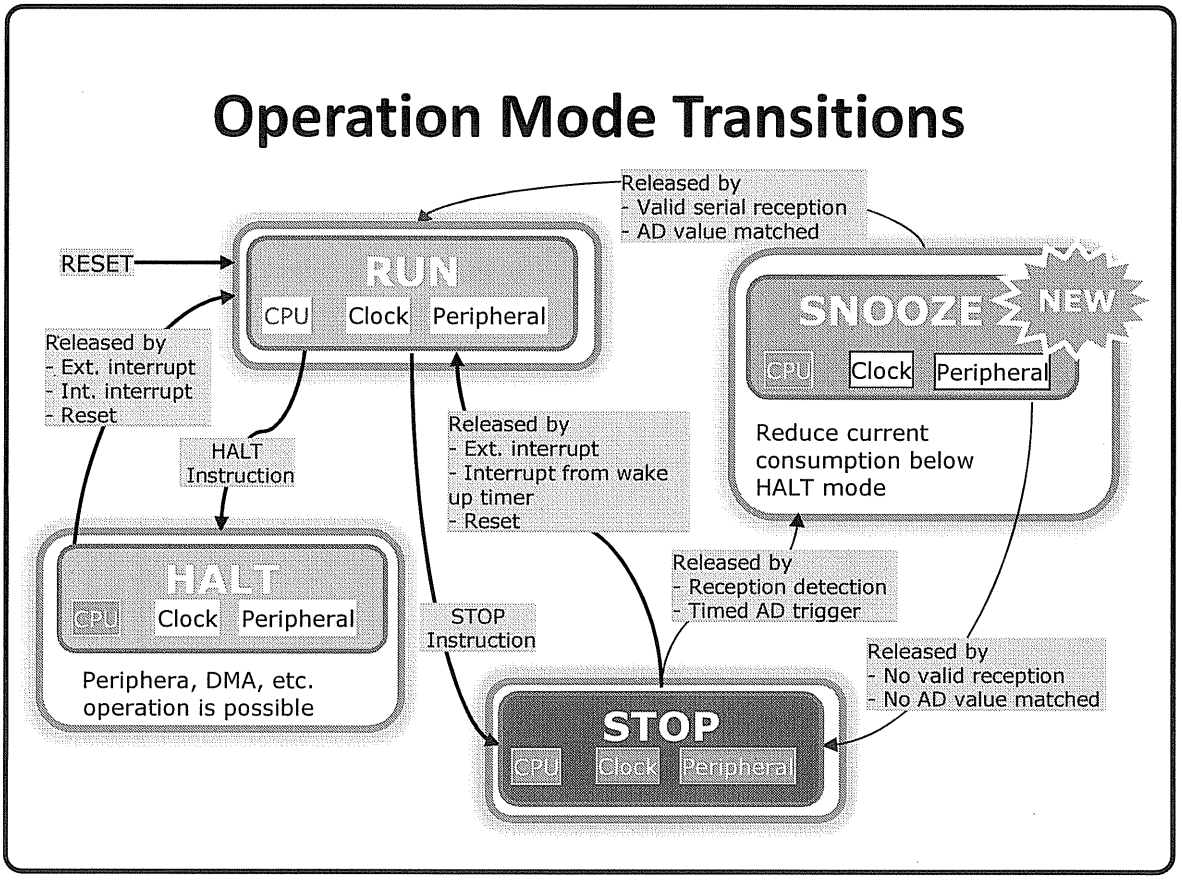
15 kHz on-chip oscillator

Only the watchdog timer (see chapter 9 for watchdog timer), interval timer and real-time clock (see chapter 6 for timer) can operate with the low-speed on-chip oscillator.

Subsystem clock

- **Operation mode:** CPU core and peripherals are using subsystem clock.

- **Halt mode:** CPU core is stopped, peripherals are using subsystem clock. Halt mode release possible with external interrupts, reset, peripherals supplied by subsystem clock or watchdog timer.



Transitions between different operation modes

The main mode is Run mode. In run mode, the CPU, clock and all the peripherals are running. Exiting the Reset state the device will directly enter into Run mode. By using the Halt command, we can switch from Run mode into Halt mode.

In Halt mode the clock is still running and all the peripherals are running. For example, timers are still running using the main system clock, with just the CPU switched off. To exit Halt mode and go back into Run mode, an external or internal interrupt, which are generated by a peripheral or a Reset signal, can be used.

The next mode is Stop mode, to enter Stop mode from Run mode, the Stop instruction is used. Returning from Stop mode can also be done via an external interrupt, a Reset signal or an internal interrupt from any internal peripherals which are running in Stop mode. Because the main system clock, the CPU and typically all the standard peripherals are stopped in Stop mode, this interrupt must come from the internal low speed timer, typically the 15 kHz or the 32 kHz subsystem clock. These two peripherals are able to drive the interval timer and the real-time clock. Switching from Stop mode to Run mode can be done directly using these timers.

And now move to the new Snooze mode. Snooze mode is somewhere between Stop and Run mode. If Stop mode is called with special conditions, the CPU will not return directly back into Run mode, but instead first goes into the Snooze mode. In Snooze mode, the CPU is still stopped with just the clock enabled and the peripherals supported by Snooze mode running. This could be the A-D converter or a serial interface, where without using the CP serial data can be received or an A-D conversion can be done.

Memory Organization (1)

Two boot clusters with 4 KB each

- Support of secure self programming by boot swapping

Vector table area

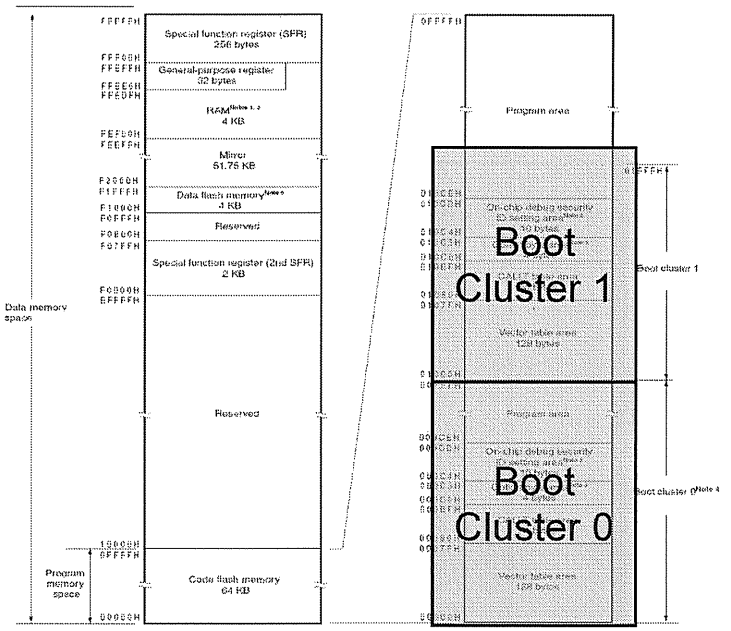
- 64 entries

Option Bytes - Control of:

- On chip oscillator
- Watchdog timer
- Low voltage detector
- On chip debug (OCD) interface

Security ID for OCD

3.8 KB user code



Code flash area

It contains two 4 KB boot clusters. This configuration allows easy implementation of the self-programming functions using the included boot loader, which provides 3.8 KB of space per cluster for user code. By using the boot-swap feature, a new boot function including new interrupt vector tables, reset vectors and option byte settings, can be written to boot cluster which can then be selected using the boot-swap command.

Each boot cluster contains a vector table that accommodates up to 64 entries. The first entry, address at zero, contains the reset vectors. This is followed by the interrupt vectors for each interrupt slot. When creating an interrupt, care must be taken as the 16-bit addresses in this range must be located in the lower 64 KB page.

Also included in each boot cluster are option bytes which can be used to control many of the RL78's features. These include control over the on-chip oscillator frequency – selectable as either 24 or 32 MHz - watchdog timer settings, low voltage settings and the debug interface. When controlling the debug interface, for example, the option bytes determine if the debug interface is opened or closed.

Memory Organization (2)

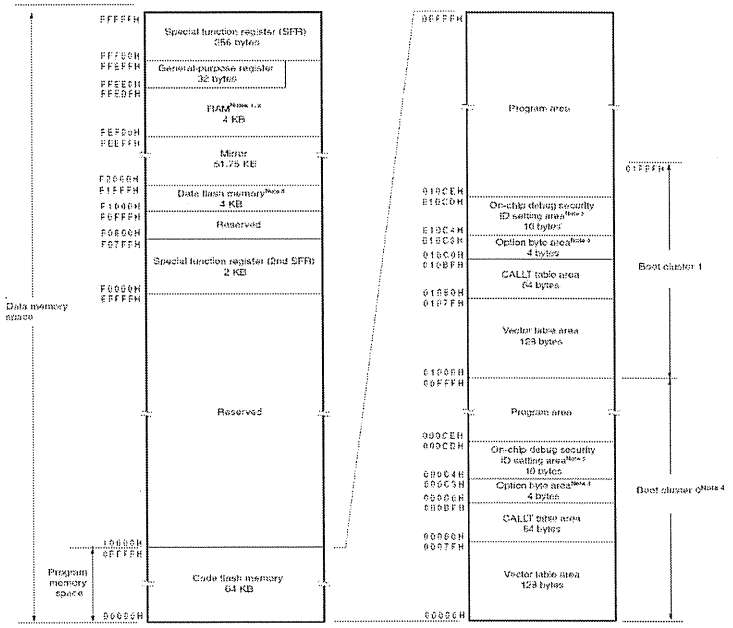
Program Memory
- Up to 256 KB

1st and 2nd SFR areas

RAM Memory
- Including CPU registers

Mirror area
- Depending on RAM size

Data Flash area
- Up to 8 KB



RL78's normal program area

The area located above the two boot clusters is where the normal program area starts. The second boot cluster can also be used as program memory if only one boot cluster is utilized. The program memory itself ranges in size from 16 KB up to 512 KB on RL78 devices.

RL78's SFR and RAM memory areas

Above the normal program area is the special functional register (SFR) and RAM memory areas. As shown in the above schema, there are actually two SFR areas. In the older MCU architectures, only one SFR is necessary; this 1st SFR area allows up to 256 special function registers. However, due to the complexity of the newer devices, with more timer and security functions provided, the 2nd SFR area, which is 2 KB of size, is inserted into the RAM area.

The RAM memory area itself is up to 30 KB including the CPU registers. The CPU registers that are not used by the application can also be used as standard RAM memory. Below the RAM area is the mirror area, which allows us to use shorter instructions to handle data.

And finally we have the data flash area which can be up to 8 KB of size.

Note 1: Use of the area FE900H to FED09H, that is used as a work area for the library, is prohibited when using the self-programming and data flash functions.

Note 2: Instructions can be executed from the RAM area excluding the general-purpose register area.

Note 3: When boot swap is not used: Set the option bytes to 000C0H to 000C3H, and the on-chip debug security IDs to 000C4H to 000CDH.
When boot swap is used: Set the option bytes to 000C0H to 000C3H and 010C0H to 010C3H, and the on-chip debug security IDs to 000C4H to 000CDH and 010C4H to 010CDH.

Note 4: Writing boot cluster 0 can be prohibited depending on the setting of security.

Note 5: The flash memory to which a program can be written, erased, and overwritten while mounted on the board. The flash memory includes the "code flash memory", in which programs can be executed, and the "data flash memory", an area for storing data.

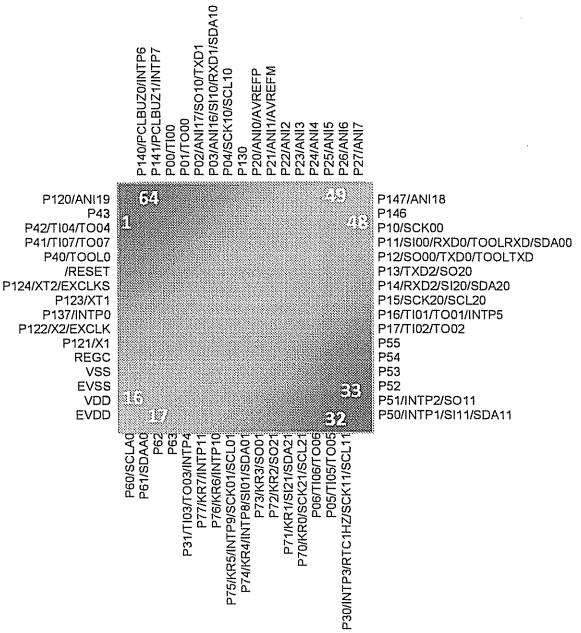
Ports

What is Port?

- Port is a special memory area
- Its function is to exchange data between CPU and external device

RL78 I/O ports

- Provide many special features, giving them maximum flexibility with regards to system design, configuration and implementation



Example of 64-pin chip, VDD=4.0V~5.5V

RL78 port architecture

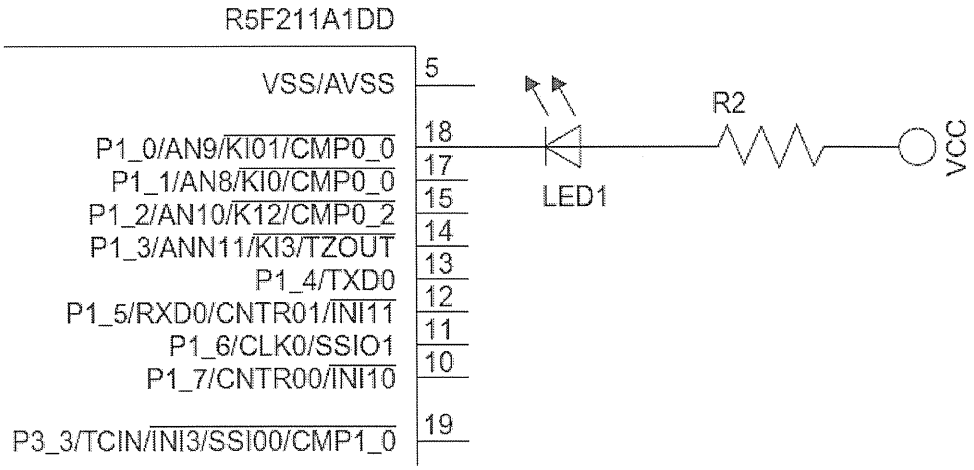
The RL78 has multiple I/O ports, numbered starting at P0. Each port typically has eight bits, with each bit connected to a specific pin on the MCU package. A port pin may serve several purposes, for example, one might be used as a general purpose I/O pin, as an A-D converter input, or as an interrupt input based on how it is configured. Depending on the purposes these pins serve, they might have extra registers.

Controlling registers

- The Port Mode register (PM) controls whether a particular port bit is an input.
- The Port register (P) holds the data value for the port.
- The Pull-Up option register (PU) controls whether an internal pull-up resistor is connected.
- The Port Input Mode register (PIM) controls which type of voltage threshold (TTL or CMOS) is used to determine whether an input is a 1 or a 0.
- The Port Output Mode register (POM) controls whether an output can be driven up or down (push-pull mode, selected with 0), or just down (N-channel open-drain, selected with 1).

- The Port Mode Control register (PMC) controls whether an input is used for analog.
- The Peripheral I/O Redirection register (PIOR) allows certain peripheral signals to be routed to one of two possible port pins, to simplify circuit design.

Using LEDs as Outputs



The microprocessor is sinking current from the LED. Turning on the LED requires.

Using the general-purpose I/O ports to interface with LEDs device

One common output device is a light-emitting diode (LED). The above figure shows the way of connecting a LED to a microprocessor. Most microcontrollers I/O pins can sink more current than they can supply, so LEDs are typically connected with cathode to the microcontroller pin. This requires writing a zero to the port pin to turn on the LED.

Example code

Let's look at the code to drive an LED with its cathode connected to port 1 bit 0. First, we configure the port as an output by writing 0 to the port mode register bit:

```
PM1_bit.no0 = 0;
```

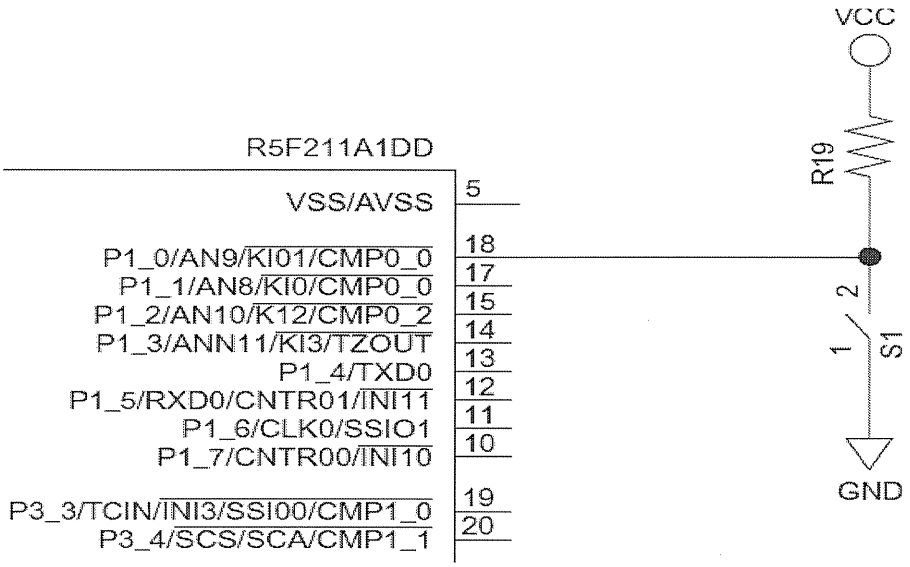
Now, we can write to the port bit to turn the LED on:

```
P1_bit.no0 = 0;
```

We can also turn it off:

```
P1_bit.no0 = 1;
```

Using Switches as Inputs



Basic switch connection to a microprocessor input pin.

Using the general-purpose I/O ports to interface with switches device

One common input device is a single-pole single-throw (SPST) switch. The resistor will pull the voltage on that pin to logic HIGH until the button connects to ground. When the button is pressed, the voltage at that pin will drop to zero (logic LOW).

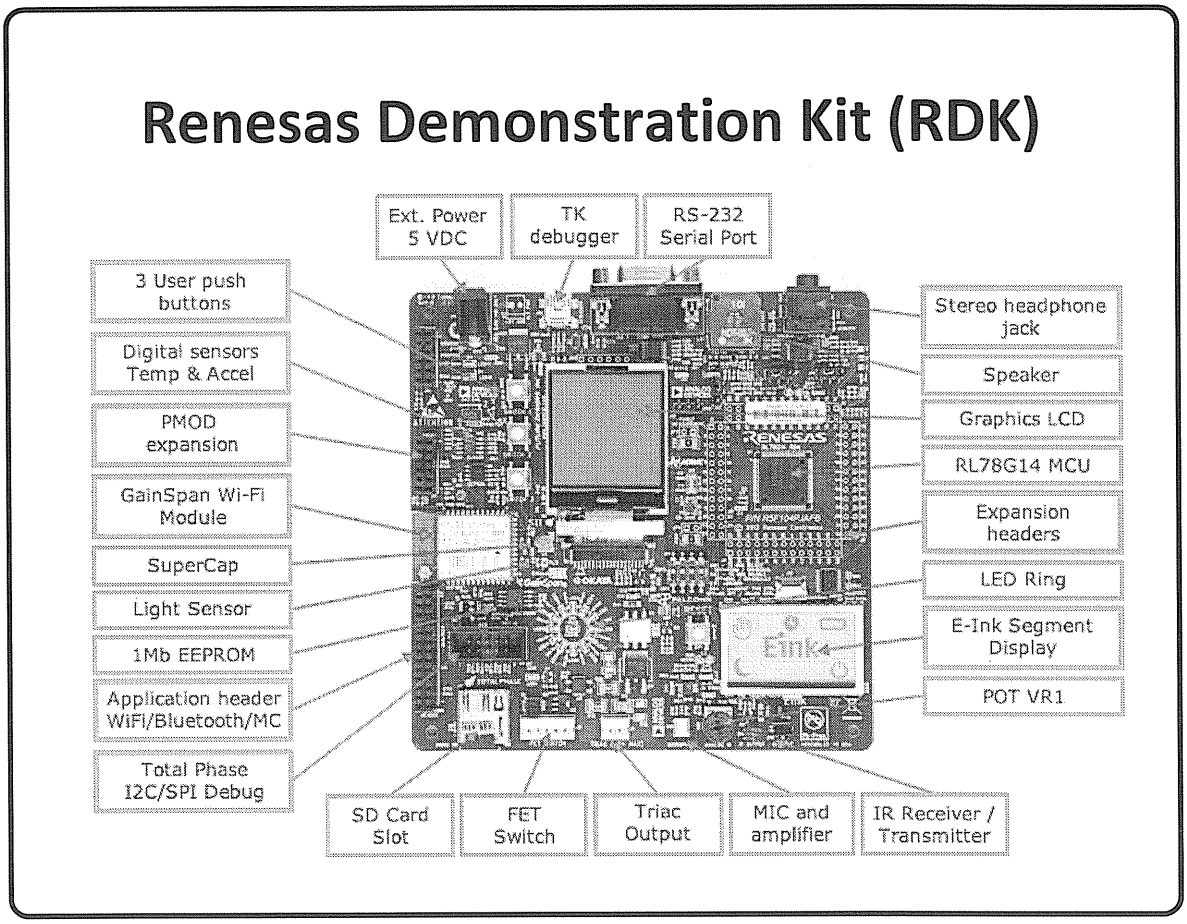
Example code

Let's look at the code to use port 1 bit 1 for this input. First, we configure the port as an input by writing 1 to the port mode register bit.

```
PM1_bit.no1 = 1;
```

Now, we can read from the pin, keeping in mind that a "0" indicates the switch is pressed, and a "1" indicates it is released. In this case we will turn off an LED whenever the switch is pressed, and turn it on when the switch is released.

```
if (P1_bit.no1 == 0) {
    P1_bit.no0 = 1;
} else {
    P1_bit.no0 = 0;
}
```



Overview

The Renesas Demonstration Kit (RDK) for RL78/G14 is an evaluation and demonstration tool for Renesas RL78/G14 microcontrollers and its partners. The goal is to provide the user with a powerful debug and demonstration platform targeted at common applications.

To aid in the evaluation of the RL78 family of microcontrollers, we include two development environments:

- e2studio: Renesas Eclipse embedded studio. It is a complete development and debug environment based on the popular Eclipse CDT project. Essentially open source, the Eclipse CDT covers build (editor, compiler and linker control) as well as debug phase based on an extended GDB interface. It supports GCC and IAR compiler.
- IAR's Embedded Workbench Kickstart Edition for RL78: This full features tool suite allows for up to 16 KB of application code to be implemented on RL78/G14 RDK.

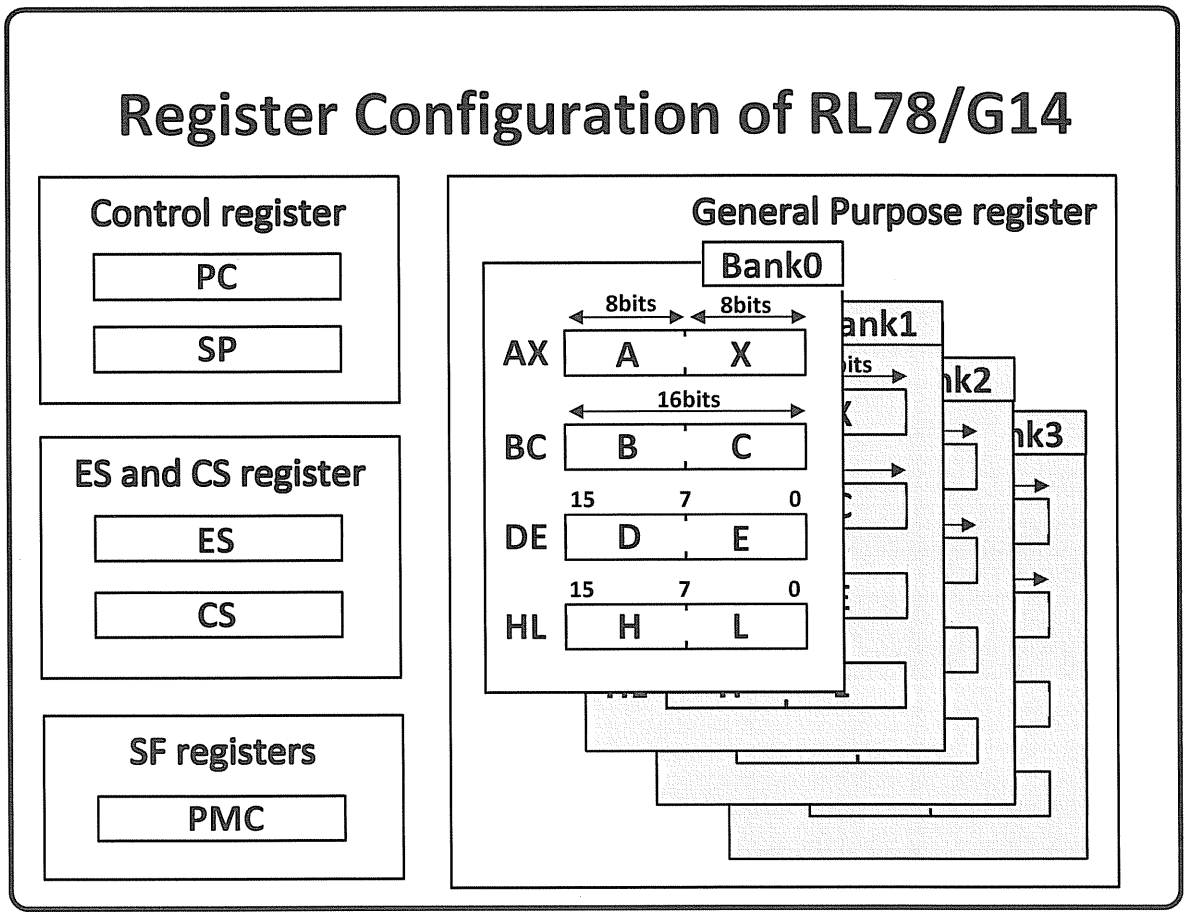
To facilitate programming the RDK, both tools support the Renesas TK Debugger which is on-board the RL78/G14 RDK.

(This page intentionally left blank)

(This page intentionally left blank)

Chapter 3

Assembly Language



Control Registers

Control registers control the program sequence, statuses and stack memory.

General-purpose Registers

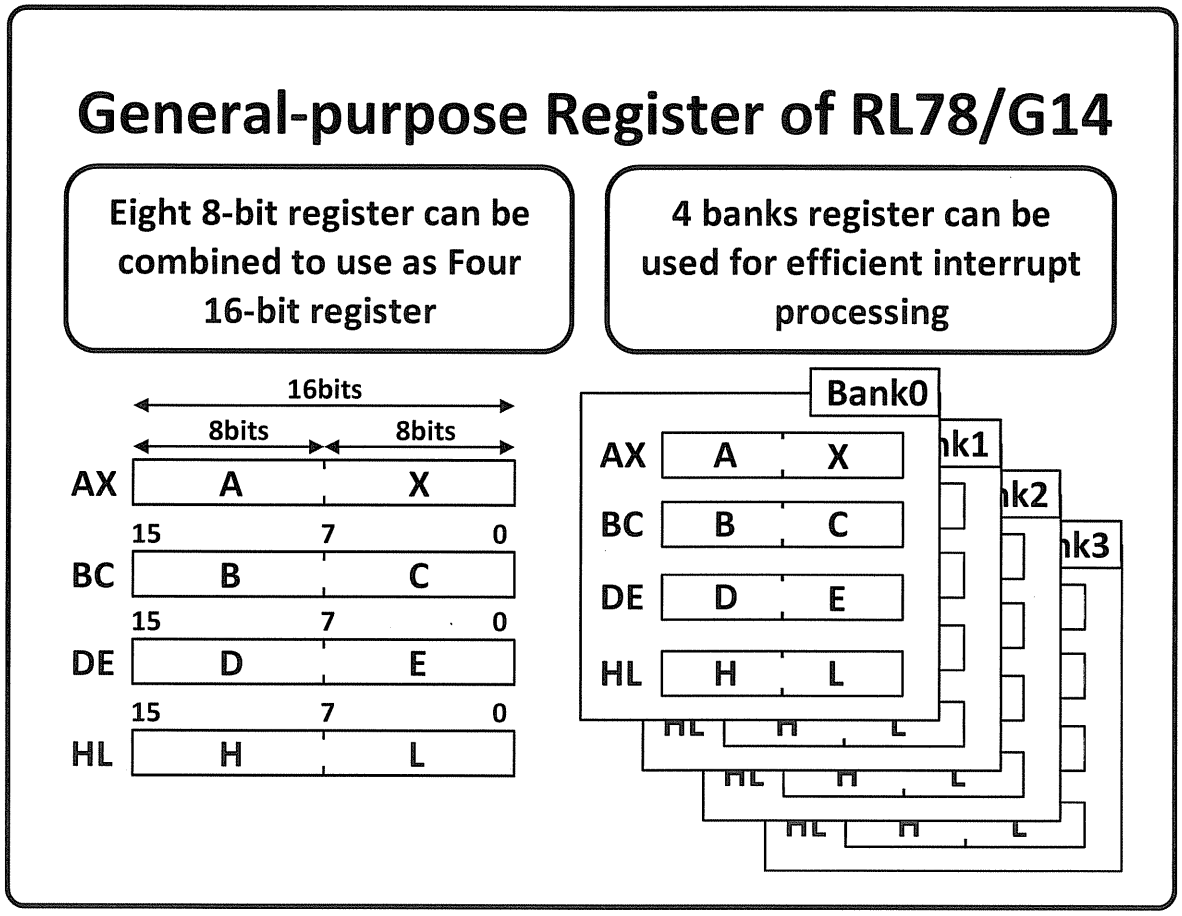
General-purpose registers consist of 4 banks, each bank consists of eight 8-bit registers (X, A, C, B, E, D, L, H). General-purpose registers can be used to store both data and addresses in the program.

ES and CS Registers

ES register supports higher addresses for data access. CS register supports higher addresses for execution branch instructions.

Special Function Registers

Special function registers control the special functions.



General-purpose Register

General-purpose registers are used primarily in data transfer, arithmetic and logical instruction

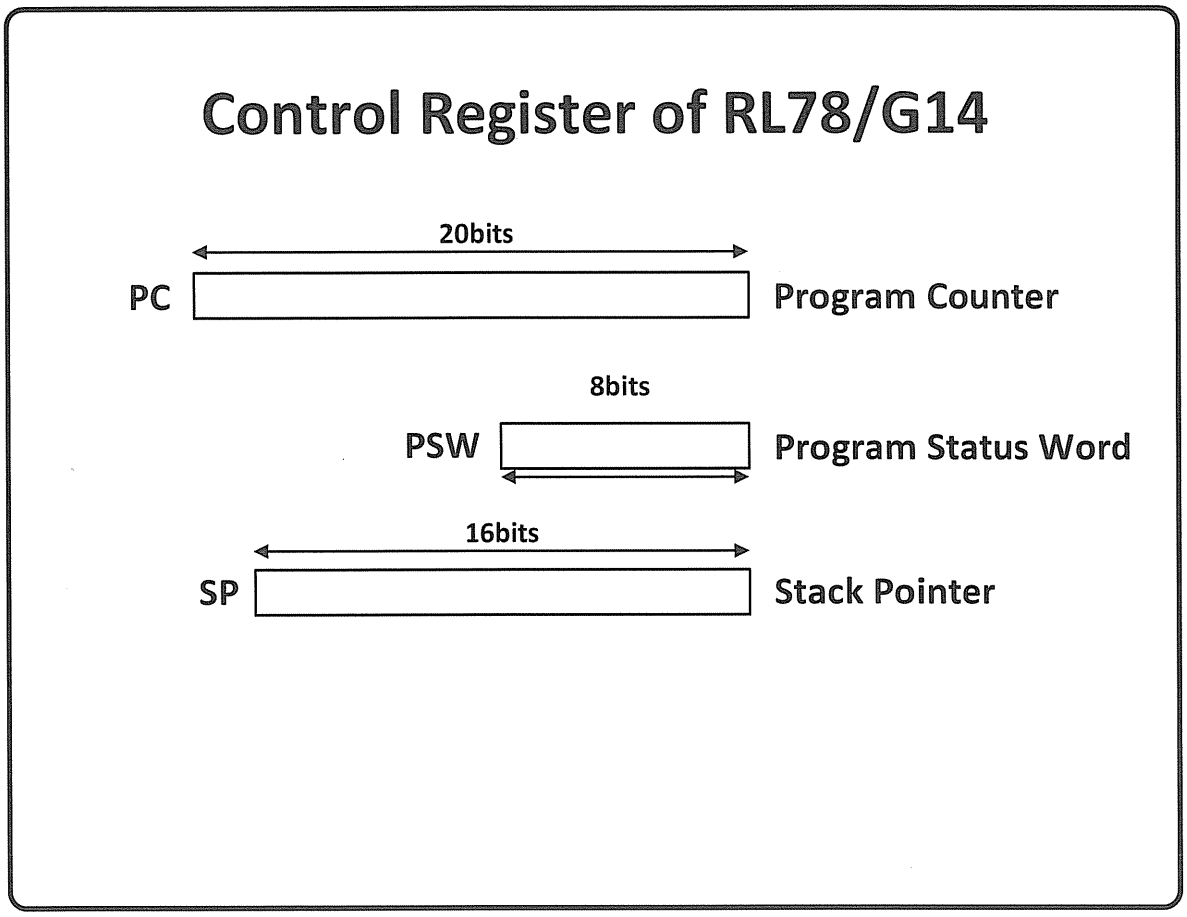
Register Bank

These registers consist of 4 banks, each bank consist of eight 8-bit register (X, A, C, B, E, D, L and H). The bank can be chosen by executed CPU control instruction "SEL RBn".

Length of Register of RL78/G14

The length of each register is 8-bits. And in addition, two 8-bit registers in pairs can be used as a 16-bit register (AX, BC, DE and HL).

Notes: General-purpose registers can be described in terms of functional names (X, A, C, B, E, D, L, H, AX, BC, DE and HL) and absolute names (R0 to R7 and RP0 to RP3).



Control Registers

Control registers control the program sequence, statuses and stack memory. There are three registers: Program counter (PC), Program status work (PSW) and Stack pointer (SP).

Program Counter

A 20-bit register holds the address information of the next program instruction to be executed.

Program Status Word

An 8-bit register which consists of various flags to be set/reset by instruction execution.

Stack Pointer

A 16-bit register holds the start address of the memory stack area. The SP is decremented ahead of writing (saving) to the stack memory and is incremented after reading (loading) from the stack memory.

Be sure to initialize the SP contents after reset signal before using.

Program Status Word

[Bit configuration]

	b7							b0
PSW	IE	Z	RBS1	AC	RBS0	ISP1	ISP0	CY

[Bit state after reset]

PSW	0	0	0	0	0	1	1	0

Flag setting instruction

```
SET1    CY      ; CY flag is set to 1
CLR1    CY      ; CY flag is cleared to 0

SEL     RB2     ; RBS1 is 1 and RBS0 is 0
```

Status Register

A status register is a collection of status flag bits for a microcomputer. It contains information about the state of the microcomputer.

Program Status Word

RL78/G14 call Status register flag as Program Status Word that is configured with 8 bits.

<<SET1 Instruction>>

This is an assembly instruction and used to set the particular bit in the register to 1.
For details, refer to "SET1" of RL78/G14 Software Manual.

<<CLR1 Instruction>>

This is an assembly instruction and used to set the particular bit in the register to 0.
For details, refer to "CLR1" of RL78/G14 Software Manual.

<<SEL RBn Instruction>>

This is an assembly instruction and used to select the nth Register Bank.
For details, refer to "SEL RBn" of RL78/G14 Software Manual.

Assembly Language Introduction

Example) Transfer 0600H to AX register

***in machine language:**

0011 0000 .... 30H opcode

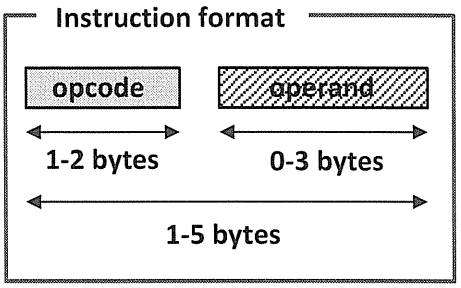
0000 0000 .... 00H Operand (Low data)

0000 0110 .... 06H Operand (High data)

***in assembly language (mnemonic):**

MOVW AX, #0600H

MOVW BC, #0000H



Address	Memory	
00100H	0011 0000 B	MOVW AX, #0600H
00101H	0000 0000 B	
00102H	0000 0110 B	
00103H	0011 0010 B	MOVW BC, #0000H
00104H	0000 0000 B	
	⋮	

Machine Language

Machine language is a set of combination of binary "0" and "1" instructions that executed directly by CPU to control the microcomputer operation. The instruction part that describes the execution function of the instruction is called "Op (=operation) Code". The instruction part that indicates execution object is called "Operand". The machine language instruction is composed of an opcode with/without one or more operands.

Assembly Language

Assembly language is a low-level programming language that uses mnemonic to represent each low-level machine instruction. The corresponding relationship between assembly language and machine language is generally 1-to-1. The program is translated into machine language, and then can be executed. Assembly language can directly manipulate general-purpose register, control register, and special functions of the microcomputer.

Assembly Language and Assembler

The process to translate Assembly language into machine language is called "Assemble". And the tool to translate Assembly language into machine language is called "Assembler".

Assembly Instruction Set

Type	Group	Type	Group
Transfer instructions: 9	Move data Set one instruction Clear instruction Exchange data	Jump instructions: 16	Call return subroutine Conditional branch Unconditional branch
Arithmetic /Logical instructions: 35	Addition & Subtraction Logical operation Comparison Increment/Decrement Shift operation Rotate operation Multiplication Division	Stack instructions: 6	Push & Pop stack instruction Move stack address operation Add & Subtract stack address
Bit operation instructions: 7	Move bit transfer Bit logical operation Set bit instruction Clear bit instruction	Others: 12	Conditional skip instruction Select Register bank Interrupt control instruction Control CPU operation clock

Instruction Set

Instruction set is a part of microcomputer architecture related to programming. The table above shows the instruction set of RL78/G14 that includes 85 instructions.

Instructions can be classified into transfer, arithmetic, program control and others instructions based on their functions.

Reference to Software Manual

This manual provides the detailed description of each instruction, such as function, selectable operands, flag change, operation, description examples, and cautions, etc. RL78/G14 describes instructions in detail in "Software Manual". Please make sure to refer to it when you use instructions.

Move Data Instruction

Mnemonic	Syntax
MOV	MOV dst, src
MOVW	MOVW dst, src

8-bit data src → dst	Description example
Register (8-bit) → Register (8-bit)	MOV A, X
Register (8-bit) → Memory (8-bit)	MOV [HL], A
Immediate value → Register (8-bit)	MOV A, #012H
Immediate value → Memory (8-bit)	MOV ram, #012H

16-bit data src → dst	Description example
Register (16-bit) → Register (16-bit)	MOVW AX, BC
Register (16-bit) → Memory (16-bit)	MOVW [HL], AX
Immediate value → Register (16-bit)	MOVW AX, #1234H
Immediate value → Memory (16-bit)	MOVW ram, #1234H

Move data instruction

In RL78/G14 series, specify operands (include src, dst) and transfer data from src to dst.

src

It indicates the source from which data is transferred. Its value is not changed after the instruction is executed.

For the possible source usage of each instruction please refer to RL78/G14 Software Manual.

dst

It indicates the destination to which data is transferred. Its value would be changed after the instruction is executed.

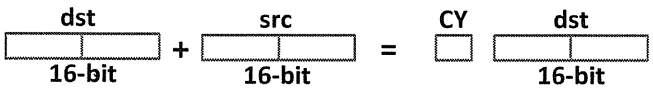
For the possible destination usage of each instruction please refer to RL78/G14 Software Manual

Add/Subtract Instruction

Mnemonic	Syntax
ADD	ADD dst, src
SUB	SUB dst, src
ADDW	ADDW dst, src
SUBW	SUBW dst, src

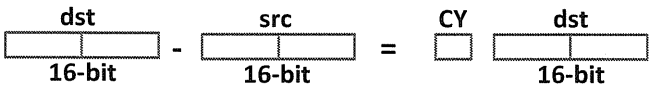
Example: ADD A, X

ADDW AX, BC



SUB A, X

SUBW AX, BC



Add and Subtract instruction

Add and Subtract instruction of RL78/G14 performs the arithmetic operation between "src" and "dst" and the result is stored in the "dst" operand and CY flag.

For the possible operands usage of each instruction please refer to RL78/G14 Software Manual

CY (Carry Flag)

This flag stores an overflow or underflow upon add/subtract instruction execution. It stores the shift-out value upon rotate instruction execution and functions as a bit accumulator during bit manipulation instruction execution.

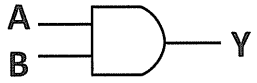
For ADD instruction: this flag is set to 1 when add operation result generates a carry out for bit 15 (word) or 7 (byte). Otherwise this flag is cleared to 0.

For SUB instruction: this flag is set 1 when sub operation result generates a borrow out of bit 15 (word) or 7 (byte). Otherwise this flag is cleared to 0.

Logical Operation Instruction

Logical AND

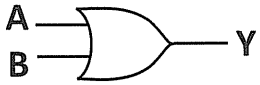
Mnemonic
AND
Syntax
AND dst, src



Input		Output
A	B	Y
1	1	1
0	1	0
1	0	0
0	0	0

Logical OR

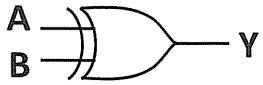
Mnemonic
OR
Syntax
OR dst, src



Input		Output
A	B	Y
1	1	1
0	1	1
1	0	1
0	0	0

Exclusive OR

Mnemonic
XOR
Syntax
XOR dst, src



Input		Output
A	B	Y
1	1	0
0	1	1
1	0	1
0	0	0

Logical Operation Instruction

Logical operation instructions of RL78/G14 perform bitwise logical operation between "src" and "dst" then store the result in "dst".

For details, refer to "AND", "OR" and "XOR" instructions in RL78/G14 Software Manual.

Data Manipulation Using Logical Operation Instructions

(1) Mask Processing: Fixing the other bits to "0" or "1" in order to pick up a specified bit is called "Mask Processing". This technique is only used to grasp the changing of the specified bit.

(2) Bit Inverting: Logically inverting the value "0" or "1" of a bit is called "Bit Inverting".

AND (Masking Processing)	OR (Masking Processing)	XOR (Bit Inverting)
1010 1010 B	1010 1010 B	1010 1010 B
0000 1111 B	0000 1111 B	0000 1111 B
0000 1010 B	1010 1111 B	1010 0101 B

Comparison Instruction

Mnemonic	Syntax
CMP	CMP dst, src
CMPW	CMPW dst, src

Z and CY flag status after executing comparison instruction

	src > dst	src = dst	src < dst
Zero flag (Z flag)	Z = 0	Z = 1	Z = 0
Carry flag (CY flag)	CY = 1	CY = 0	CY = 0

Example

CMP A, #10 When A is 10: Z = 1, CY = 0

CMPW AX, BC When AX is 0000H, BC is 0020H: Z = 0, CY = 1

Comparison instruction

These instructions will perform subtraction (result is not stored) to compare, and change the contents of Program Status Word register. You can do =, > or < judgment according to the status flag value.

For details, refer to these instruction usages in RL78/G14 Software Manual.

Conditional combination

Zero flag (Z) and Carry flag (CY) in Program Status Word register will be modified when executing a comparison instruction. By using of these modified values, combination with conditional branch instructions (described later) or conditional skip instructions can change the flow of a program.

Conditional Branch Instruction

Mnemonic	Syntax
B*Cnd*	B*Cnd* label

	Branch instruction of assembly language	
Condition name	Branch when condition is "0"	Branch when condition is "1"
Carry CY	BNC	BC
Zero Z	BNZ	BZ
Higher Z v CY	BH	BNH

Branch processing

Assembly language provides conditional branch instructions to change the processing flow after comparing data.

Carry (CY flag)

This flag stores an overflow, underflow or shift-out value upon arithmetic instruction execution.

Zero (Z flag)

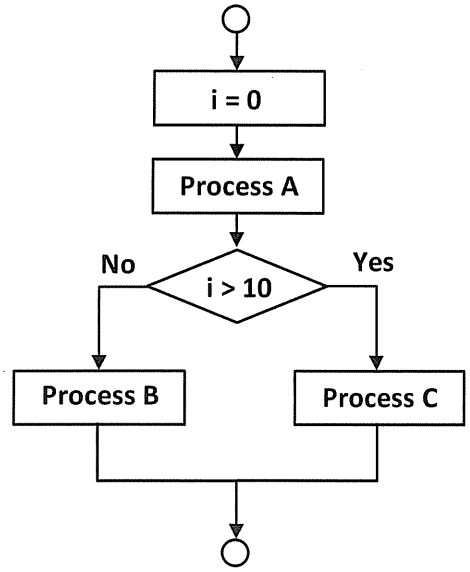
This flag is set to "1"when the operation result is zero. Otherwise, it's cleared to "0".

Higher (Z v CY combination)

When first operand content is larger than second operand the combination of Z v CY is "0". Otherwise, the combination of Z v CY is "1".

Conditional Branch Processing

Example) A conditional branch when comparing a number with 10



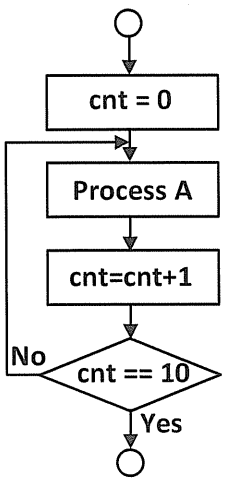

```
Assembly Language
        ⋮
    mov   a, #0H
    Process A
    cmp   a, #10
    bh    L0
    Process B
    br    L1
L0:
    Process C
L1:
        ⋮
```

Conditional Branch processing

We can change the processing flow of the program by combination of compare and conditional branch instruction. In the above example, after executing CMP instruction, the status flag's value is changed. Then based on the changed flag's value, the process jump to the label (address) specified by the conditional branch instructions following CMP instruction. In C language, such branch operation is performed by using if-else or goto statement. In assembly language, to branch to a line we want, we just need to specify the label that represents that line, and then the branch can be completed. This allows writing a very high-free program, but it is difficult to write a structured program. Moreover, the code is low-readable and -maintainable.

Loop Processing

Example) A loop process with 10 times




```
Assembly Language
        ⋮
    mov   a, #0H
L0:
    Process A

    inc   a

    cmp   a, #10
    bnz   L0
        ⋮
```

Loop processing

We can change the processing flow of the program by combination of compare and conditional branch instruction. In the above example, after executing CMP instruction, the status flag's value is changed. Then based on the new flag's value, the process jump to the label (address) specified by the conditional branch instructions following CMP instruction. In C language, such loop operation is performed by using while or for statement. In assembly language, like branch operation described above we use label to specify a line the branch instruction will jump to. This allows writing a very high-free program, but it is difficult to write a structured program. Moreover, the code is low-readable and -maintainable.

Addressing Modes

- **What is addressing mode**
  It indicates a way to access some address.
- **Addressing modes**
  The modes vary with processor's type. Usually, they have three modes.
  (1) Direct addressing - Operand is the object to be operated on.
  (2) Indirect addressing - The value indicated by the content of address register constitutes the effective address to be operated on.
  (3) Based addressing - The value indicated by displacement plus the content of address/special register, added not including the sign bit, constitutes the effective address to be operated on.

Merits of Abundant Addressing

The abundant addressing such as between memory - register, register - register, register - memory, memory - memory, makes it to be possible that data transfer and arithmetic can be accomplished with a fewer instructions and a high-free program can be generated. Consequently, the efficiency of compiling C code is improved.

Addressing of RL78/G14 Series

Addressing depends on the selected "src" and "dst" that are operands of instruction. The selectable addressing modes, as well as "src" and "dst" for each instruction are different.

For details, refer to "Explanation of Instructions" for each instructions format in RL78/G14 Software Manual.

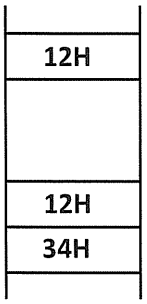
<<Difference of Instruction Size/Number of Cycles>>

Even if the same instruction is executed, the required instruction size and number of cycles will vary depending on addressing you choose.

For details, refer to "Operation List" in RL78/G14 Software Manual.

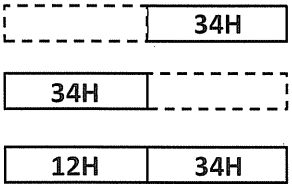
Register/Direct Addressing

Direct Addressing	[Example]
#IMM8, #IMM16, #IMM20	MOV 0FE40H, #12H ---▶ FFE40H MOVW 0FE20H, #1234H --▶ FFE20H



Register Addressing	[Example]

X, A, C, B, E, D, L, H, AX, BC, DE, HL	MOV X, #34H
	MOV A, #12H
	MOV AX, #1234H



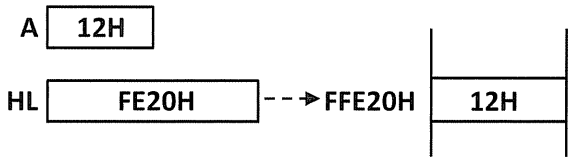
Register Addressing

The specified register is the object to be operated on.

Direct Addressing

The immediate data operand is the object to be operated on.
#IMM8: Specifies an immediate whose size is 8 bits (Ranging from FFE20H to FFF1FH)
#IMM16: Specifies an immediate whose size is 16 bits. (Ranging from F0000H to FFFFFH)
ES: #IMM16: Specifies a combination immediate whose size is 20 bits.

Register Indirect Addressing

Register Indirect Addressing	[Example] MOV [HL], A
[DE], [HL]	

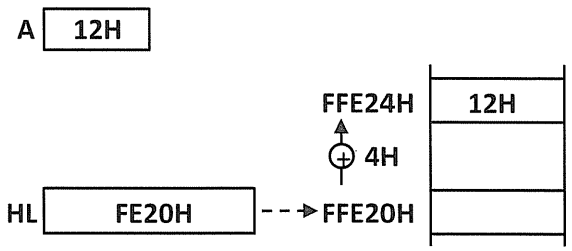
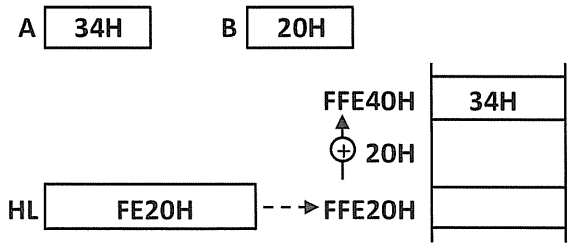
Register Indirect Addressing

The value indicated by the contents of address register pair (DE, HL) specified the address is the object to be operated on.

[DE], [HL]: Range from F0000H to FFFFFH is specifiable

ES:[DE], ES:[HL]: Range from 00000H to FFFFFH is specifiable

Based Addressing

Based Addressing	[Example] MOV [HL+4H], A
[HL+byte], [DE+byte], [SP+byte], word[B], word[C]	
Based Index Addressing	[Example] MOV [HL+B], A
[HL+B], [HL+C]	

Based Addressing

The contents of a specified register pair will be used as a base address, and 8-bit or 16-bit immediate data as offset address. The sum of these values is used to specify the target address.

[HL+byte], [DE+byte], [SP+byte]: Range from F0000H to FFFFFH is specifiable
word[B], word[C]: Range from F0000H to FFFFFH is specifiable
word[BC]: Range from F0000H to FFFFFH is specifiable
ES:[HL+byte], ES:[DE+byte]: Range from 00000H to FFFFFH is specifiable
ES:word[B], ES:word[C]: Range from 00000H to FFFFFH is specifiable
ES:word[BC]: Range from 00000H to FFFFFH is specifiable

Based Index Addressing

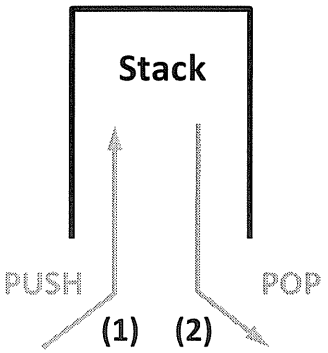
The contents of a specified register pair will be used as base address, and the contents of B or C register will be used as offset address. The sum of these values will be used to specify the target address.

[HL+B], [HL+C]: Range from F0000H to FFFFFH is specifiable
ES:[HL+B], ES:[HL+C]: Range from 00000H to FFFFFH is specifiable

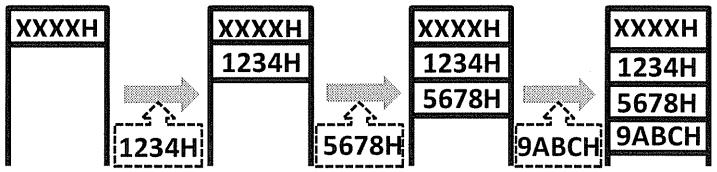
Stack Manipulation Instructions

Mnemonic	Syntax
PUSH	PUSH src
POP	POP dst

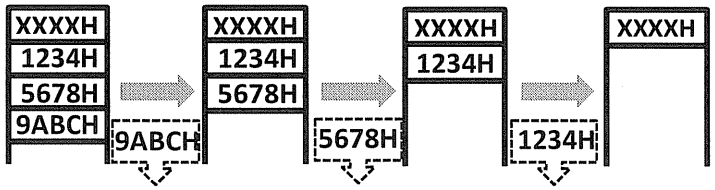
FILO (First In Last Out) method



(1) Save data (PUSH)



(2) Load data (POP)



Stack

A data structure where data stored in-first is output last (FILO: First In Last Out).

Stack Manipulation Instruction

Stack manipulation instructions of RL78/G14 perform operation to manipulate the stack area.

For details, refer to "Stack Manipulation Instructions" in RL78/G14 Software Manual.

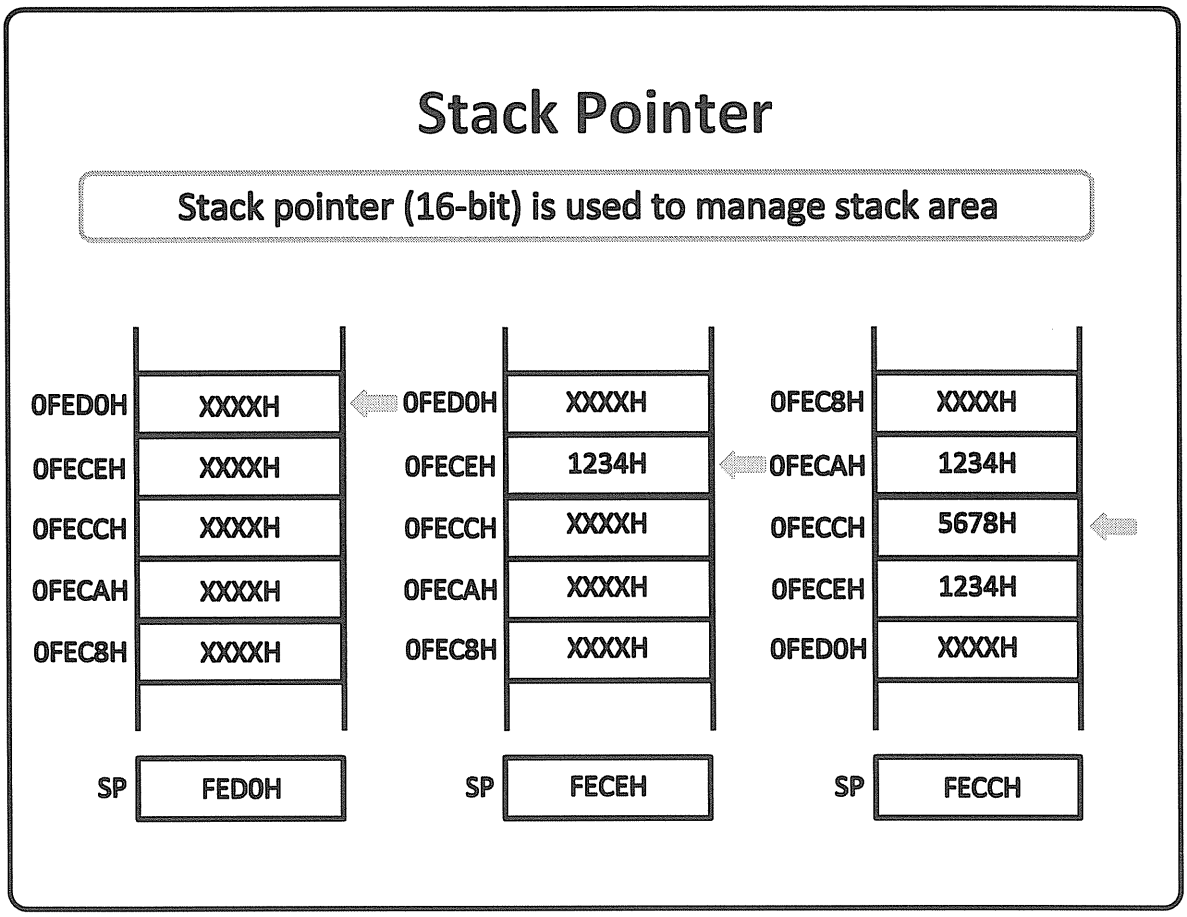
Stack Area

The memory area allocated to stack for temporary storing data is called "stack area".

The stack area is configured in memory as FILO data structure and is managed by Stack Pointer.

Stack Overflow

Stack is used to store temporary data. So in general, you must pop the data when not used. If you go on pushing data into stack, but not popping data out, the stack area would be exceeded and result in accessing to the other memory area. This behavior is called "Stack Overflow", and it will cause bugs to happen.



Stack Pointer (SP)

The 16-bit register that holds the start address of the memory stack area.

Stack pointer is decremented ahead of save (push) to the stack area and is incremented after load (pop) from the stack area.

The value of stack pointer must be set to even number. If the odd number is set, the least significant bit is automatically cleared to 0.

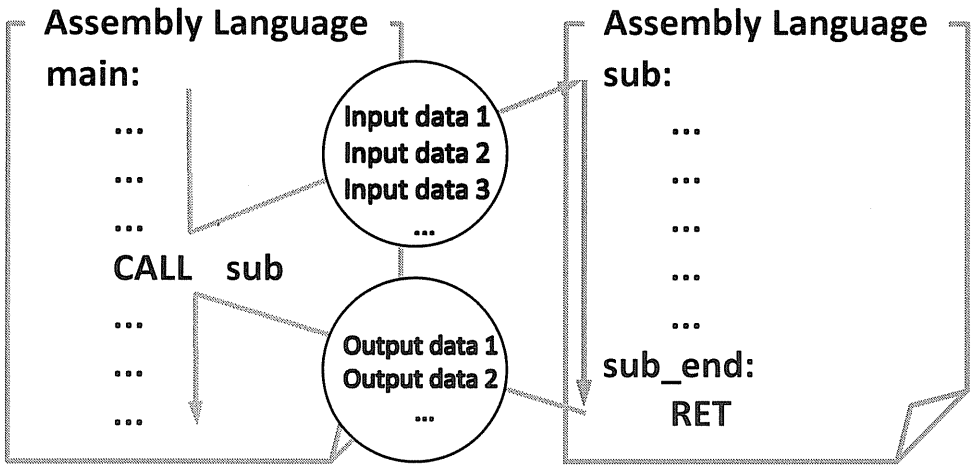
After reset signal generation, stack pointer contents are undefined, so make sure to initialize the stack pointer value before using the stack.

Stack Pointer Area

Stack pointer can be specified only within internal RAM (Target range is from F0000H to FFFFFH).

Call Return Instructions

Mnemonic	Syntax
CALL	CALL target
RET	RET



Call Instructions

This is a subroutine call with 20/16-bit absolute address or register indirect address.

The start address of the next instruction is saved in the stack and PC is branched to the address specified by the target operand.

For details, refer to "CALL" instructions in RL78/G14 Software Manual.

Return Instruction

This is a return instruction from subroutine call made by CALL instructions.

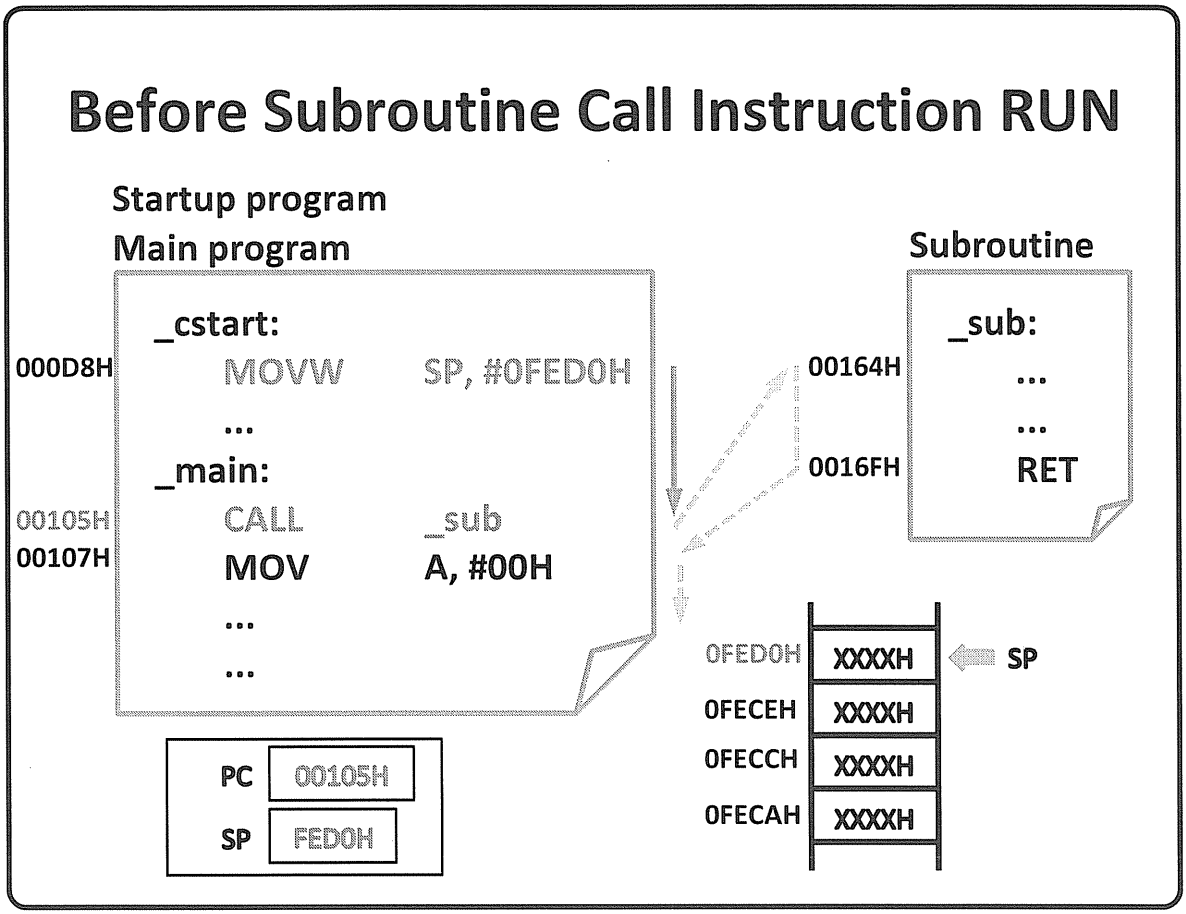
The word data saved to the stack returns to the PC, and the program returns from the subroutine.

For details, refer to "RET" instructions in RL78/G14 Software Manual.

Subroutine

Subroutine is a sequence of program instructions that perform a specific task like C/C++ function.

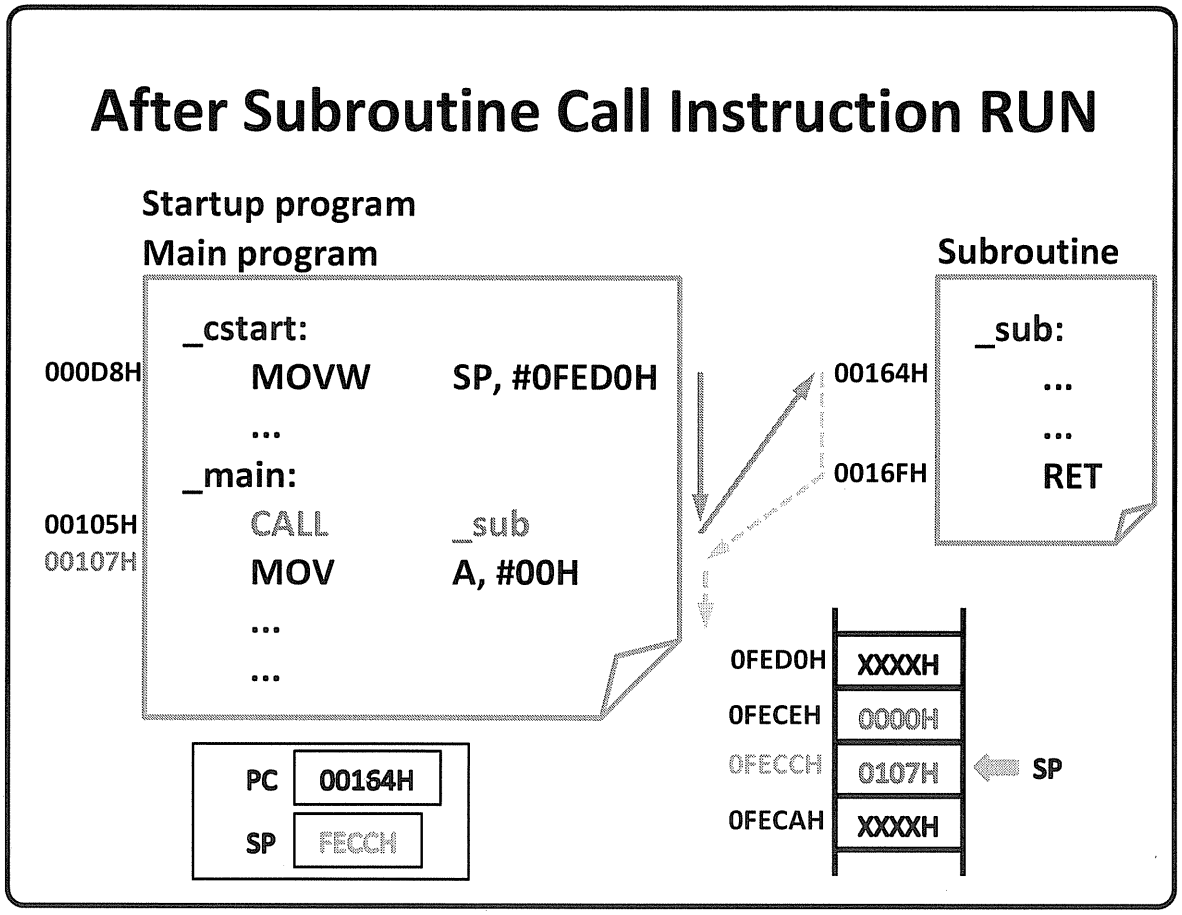
Usually, data is transferred using registers. If there are many arguments needed to pass to subroutine, they can be pushed to stack.



Initialize Stack Pointer

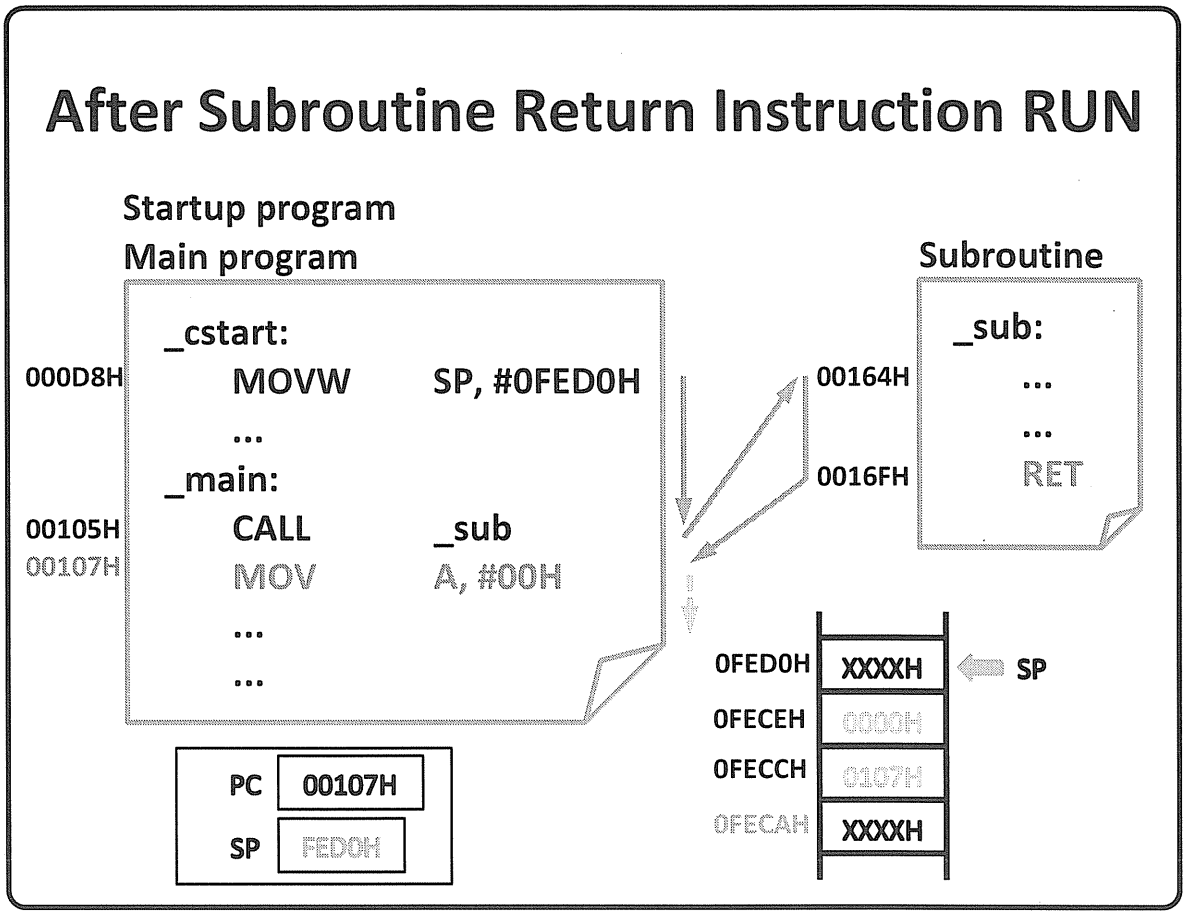
When using subroutine, you must initialize the stack pointer.

When you do not initialize the stack pointer, the program may get hang up when calling "main" subroutine. So, please be careful at this point.



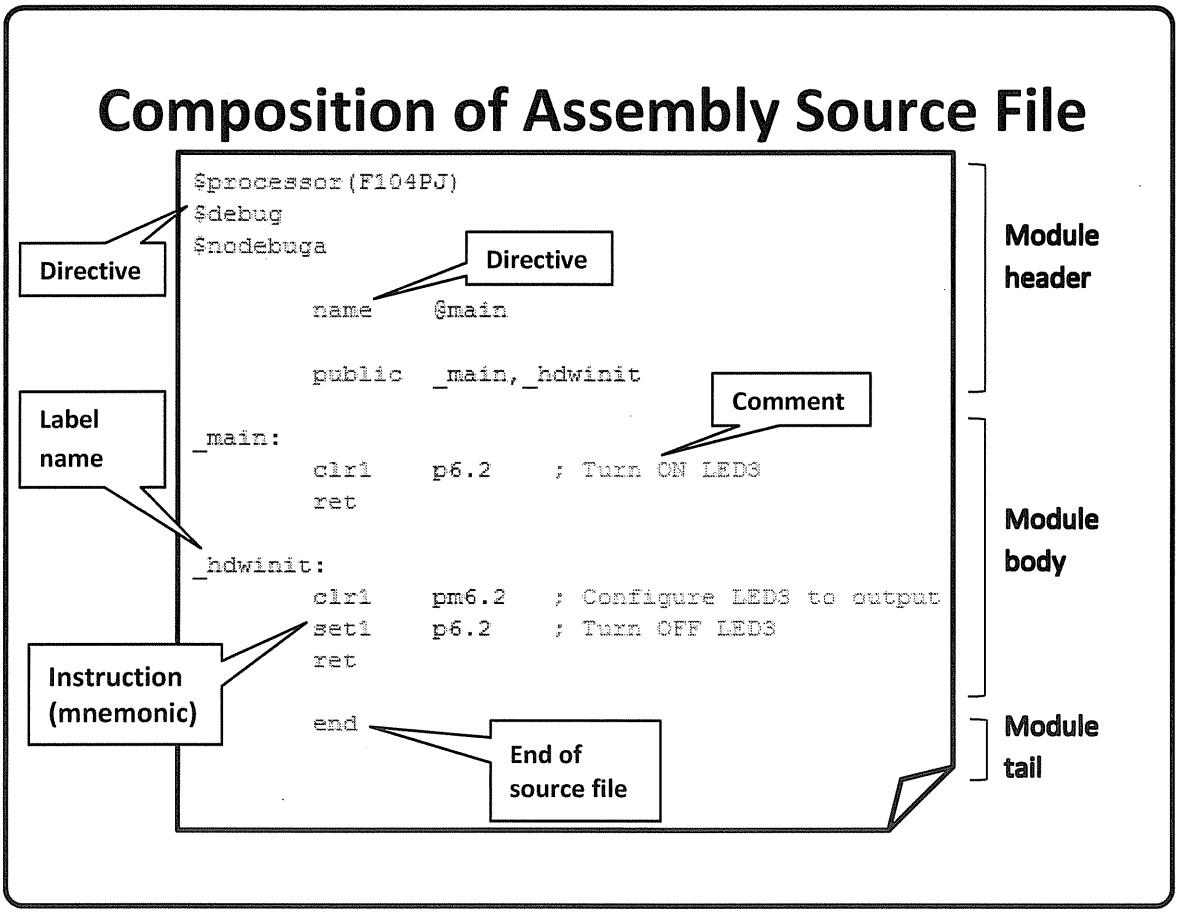
Save Return Address from Subroutine

When subroutine "CALL" instruction is executed, the control flow is changed to the start address of subroutine. After executing the subroutine, the control flow must be returned to the original place, where subroutine is called, to continue the control flow. For this purpose, the subroutine "CALL" instruction will pushes the next instruction address to stack area to save the place to return before branching to subroutine start address.



Return from Subroutine

As soon as the return instruction from subroutine is executed, the return address stored in stack area will be written to PC, and allowing the control flow to return back. The stack pointer also returns to the original value before calling the subroutine. The return address is retained in stack area, but it is rewritten when the next subroutine is called.



Description of Mnemonic and Register / Flag name

The names of register/flag and mnemonics in Assembly code of RL78/G14 series are reserved words. You cannot use the reserved words as the label, section name, etc.

Label

Colon ":" should be written after a label. A label name specifies a given memory address.

Comment

A comment should be preceded by a semicolon ";".

Numeric Expression

Binary, decimal, and hexadecimal can be expressed. But, do not forget to attach the unit.

Binary notation: 1010101B

Decimal notation: 125 (no unit)

Hexadecimal notation: 55H and the value starting with A~F should be preceded by "0"

Directive Commands of Assembler

Type	Directives
Segment definition	CSEG, DSEG, BSEG, ORG
Symbol definition	EQU, SET
Memory initialization, area reservation	DB, DW, DG, DS, DBIT
Linkage	EXTRN, EXTBIT, PUBLIC
Object module name declaration	NAME
Branch instruction automatic selection	BR, CALL
Macro	MACRO, LOCAL, REPT, IRP, EXITM, ENDM
Assemble termination	END

Directives Command to Assembler

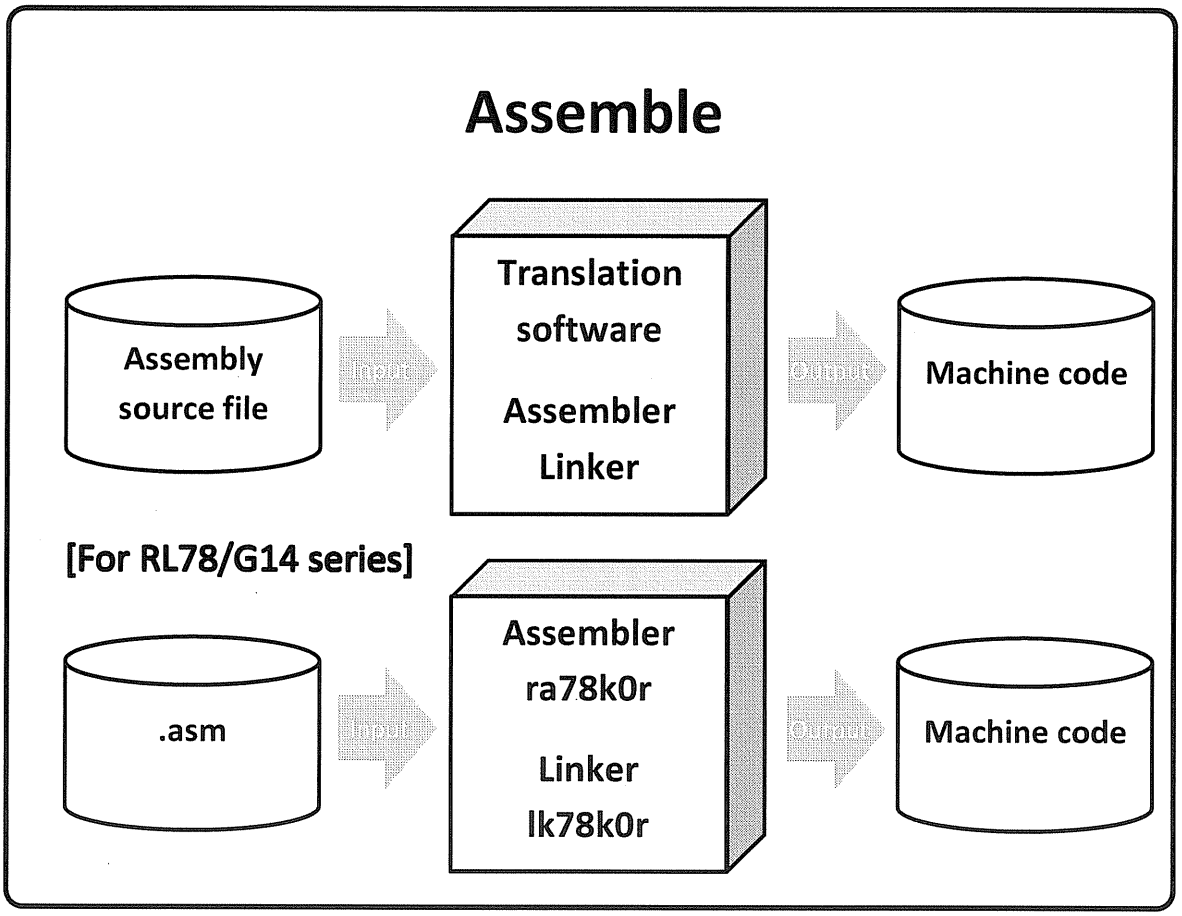
Directives are instructions that direct all types of instructions necessary for RL78/G14 assembler to perform a series of processes. Instructions are transplanted into object codes (machine code) as result of assembling, but directives are not converted into object code in principle.

Directives Functionality

Directives are used to facilitate description of source program.

Specify the memory area, initialize memory and reserve memory areas.

Directives provide required information for assembler and linker to perform their intended processing.



Assemble

Translating assembly language into machine language is called "Assemble". Assemble is done with a professional software (assembler). This assembler has two types: one is absolute assembler to fix the memory configuration of a program; the other is re-locatable assembler that can allocate memory again.

Linkage Editor

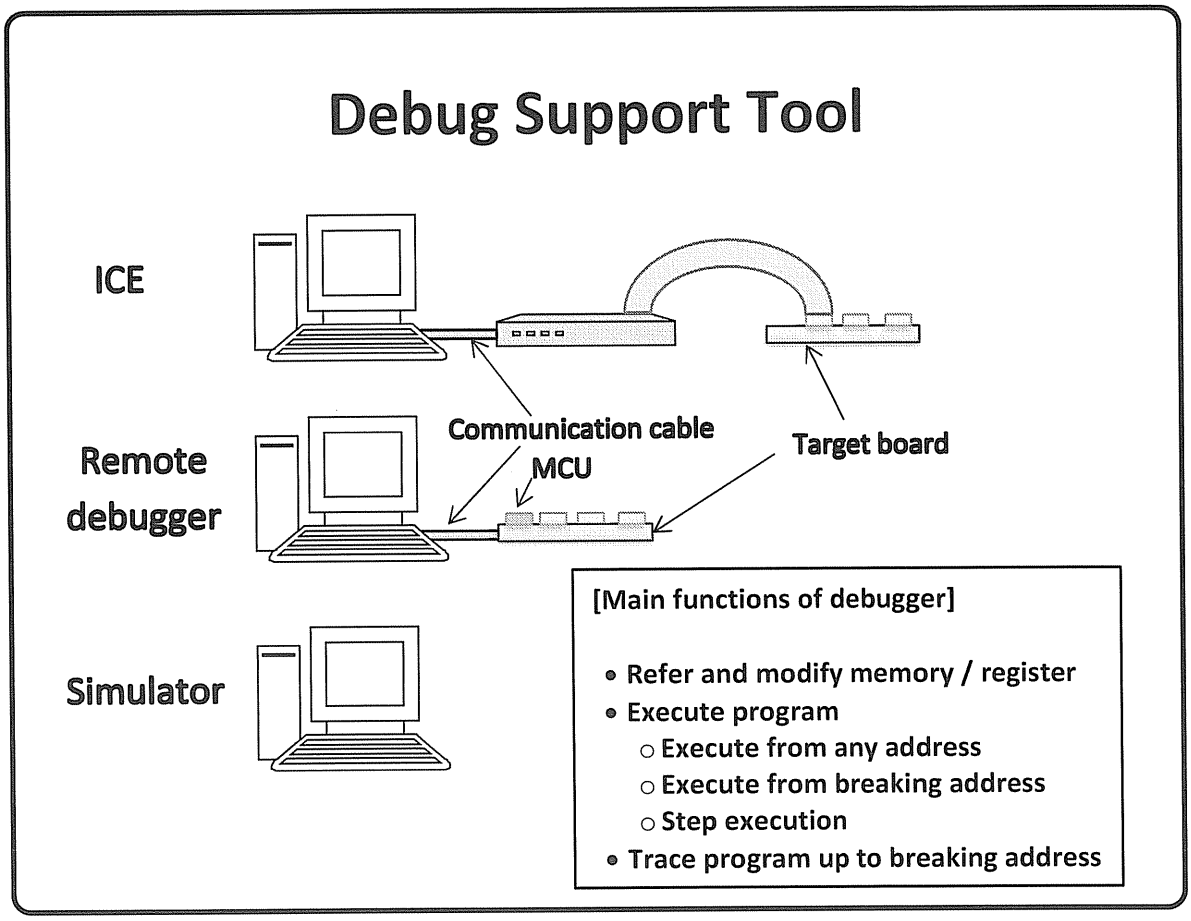
Combining intermediate files that are generated in translating from an assembly language to a machine language, and determine the address by library and etc., and generating machine code is called "linking". The software to perform such task is called "Linkage Editor" or "Linker".

ra78k0r

This is the re-locatable assembler for RL78/G14 Series.

lk78K0r

This is the linkage editor for RL78/G14 Series.



Debugger

The syntax mistake of a program can be specified by compiler error messages. But to check mistakes of logical flow, we need debug support tools to help us find out the logical error in the program which the compiler/assembler cannot detect. The most commonly used debug support tool is ICE, remote debugger, and simulator.

ICE (In-Circuit Emulator)

ICE executes the program instead of target microcomputer. You can control and monitor the result from Personal Computer.

Remote Debugger

Download the debugging program into the memory of target board, and control it from PC. Execution is done on the target microcomputer without using the expensive ICE. However, the available functions are limited because of actual resource are used.

Simulator

Simulate the microcomputer operation only on PC rather than the target board.
It is often used when target hardware or debug function on hardware is not available.

(This page intentionally left blank)

(This page intentionally left blank)

Chapter 4.

Embedded C Language

In this chapter:

- Variable Allocation in ROM and RAM
- Review on C
- Special Function Register
- Volatile Variable
- Startup Process and ROMization

Introduction to Embedded C

- ✓ There are thousands of programming languages: C, C++, Java, Fortran, Pascal...
- ✓ There are many programming languages suitable for writing embedded code: C, C++, Ada, Forth, Assembly...
- ✓ Advantages of C:
  - "Low level" high level language
  - Simple to write code yet still can access hardware devices
  - Easy porting

Embedded C Language

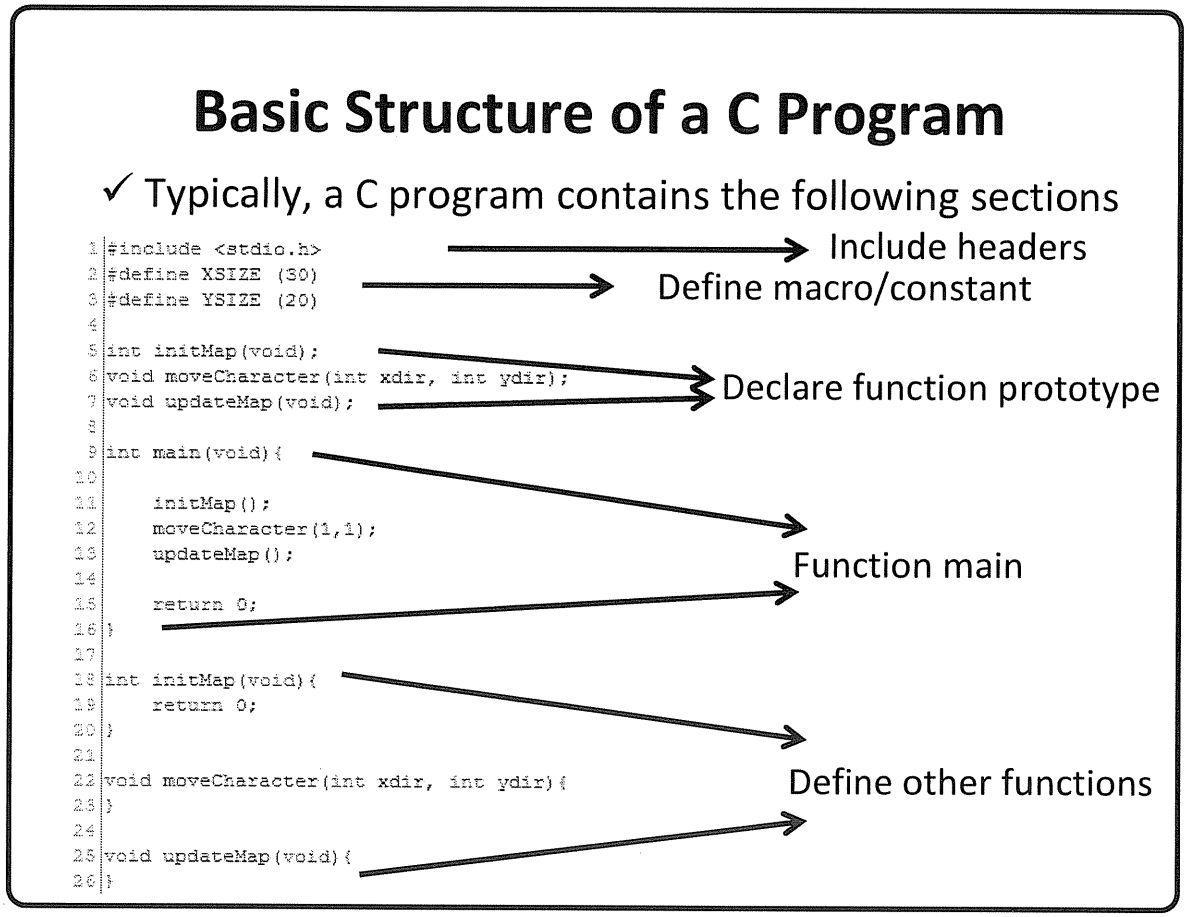
C has become a popular choice among many other languages for programming embedded system, thanks to its simplicity and portability.

C Language in this training

This training will be using RL78/G14 board (Renesas Demonstration Kit - RDK) with CA78K0R compiler. This compiler supports ANSI C syntax and provides some extended features such as: microcontroller dependencies, extended functions...

Hardware accessing

In C, the hardware devices can be accessed through their addresses in the memory. To do this, we will need to use pointer variable in C.



Headers

In case of using library or additional source files, we use the directive #include to include their header files.

The main function

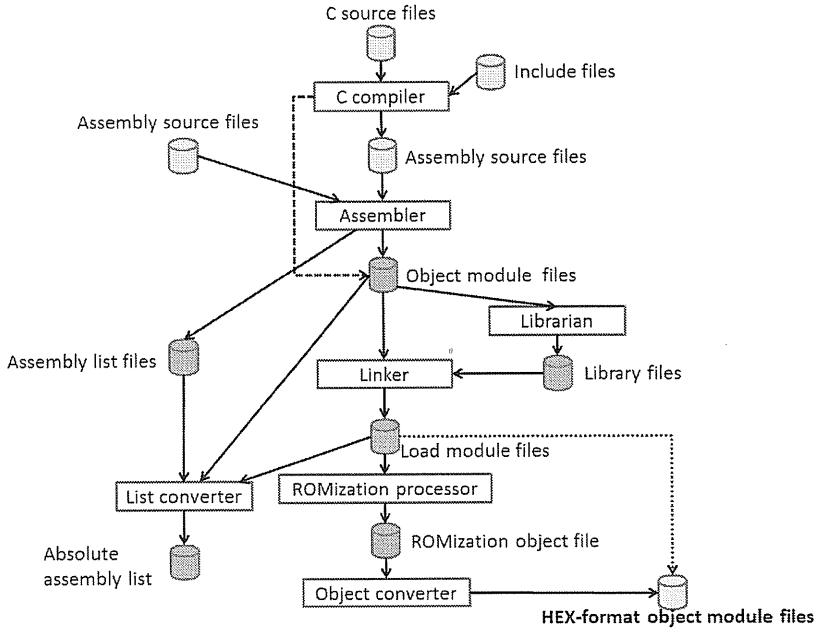
In a C program for computer, main is the entry point. It means, when the operating system executes the program, it first enters the main function. In embedded C program, main is not the entry point. The embedded C program needs to have another function called startup which is responsible for initializing hardware and then for calling the main function.

Other functions

Beside the main function, we often need to write many other functions to realize a big program or software. We must declare their prototype before defining and using them. For a better management, we normally organize a program into smaller modules. This technique is called *modular programming*.

Compilation Process

✓ How C code is built?



Compiler & Assembler

Compiler will compile C source files and headers to assembly files, or directly to object module files. Assembler then will assemble assembly source files to create object files or assemble list files depending on the options passed to it.

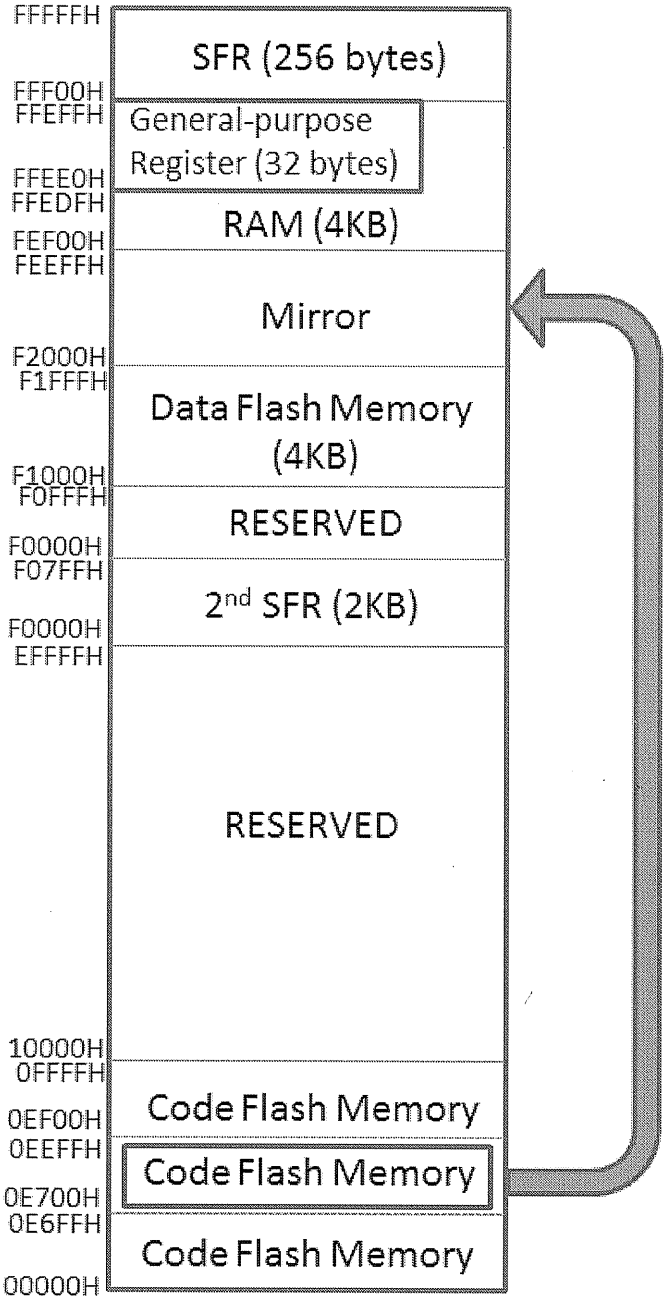
Linker

Object module files can be linked together to create library files. Linker will link all object module files with necessary library files to create Load module files.

ROMization

The final files obtained throughout this process are HEX-format object module files that can be executed by microcontroller. For details, refer to CA78K0R Compiler Manual.

Memory Map (RL78/G14)



- ✓ 4KB of RAM
- ✓ General-purpose Register (32 bytes on RAM)
- ✓ 4KB on-chip flash memory for data
- ✓ SFR: Special Function Register

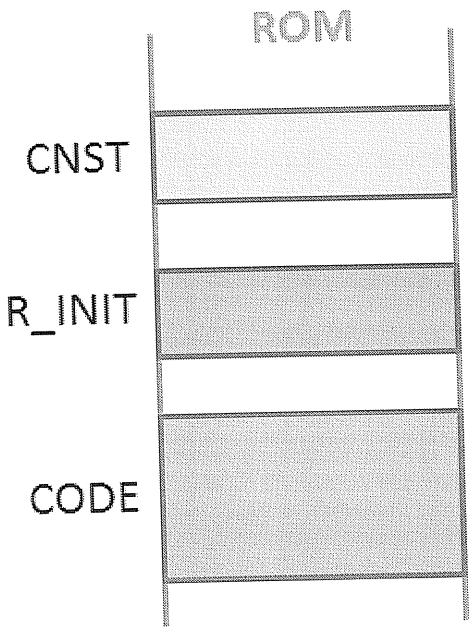
Code Flash Memory Mirror

A segment of code is mirrored from 0E700H to F2000H. In the mirror area, we can use shorter instructions to access the data memory (RAM). This will help to reduce execution time.

Variable Declaration (ROM)

✓ Depending on the types of variables, they will have different storing locations

```
1  int var11 = 110;
2  int var12;
3
4  const int var21 = 210;
5
6  static int var31 = 310;
7  static int var32;
8
9
10 int main(void)
11 {
12     int var41 = 410;
13     int var42 = 420;
14     int var43;
15
16     static int var51;
17
18     var43 = var41 + var42;
19
20     return 0;
21 }
```



Constant variable

Constant value is stored in ROM, in segment @@CNST.

Variable with initial value

The values of variable and static variable with initial value are stored in ROM, in segment @@R_INIT, while the variables themselves are stored in RAM (see the next page). During ROMization, these values are copied from ROM and assigned to variables in RAM.

Code function and its instructions

The code and its instructions are stored in ROM, in segment @@CODE and @@CODEL.

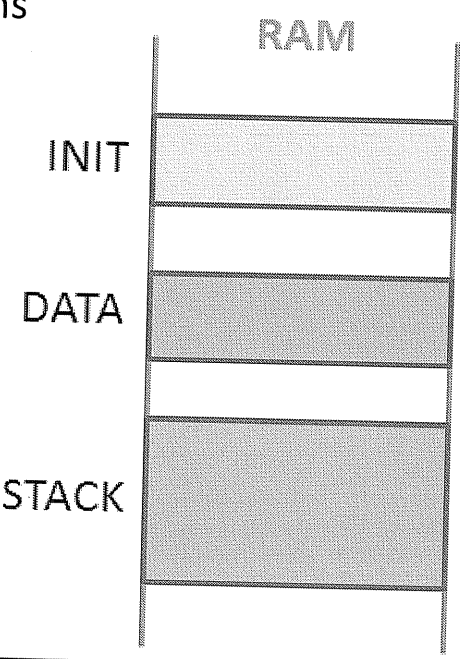
Segment name

Find to understand more about other segments (compiler manual, section 3.5).

Variable Declaration (RAM)

✓ Depending on the types of variables, they will have different storing locations

```
1  int var11 = 110;
2  int var12;
3
4  const int var21 = 210;
5
6  static int var31 = 310;
7  static int var32;
8
9
10 int main(void)
11 {
12     int var41 = 410;
13     int var42 = 420;
14     int var43;
15
16     static int var51;
17
18     var43 = var41 + var42;
19
20     return 0;
21 }
```



Variable

Variable itself is stored in RAM. If the variable has initial value, it is stored in segment @@INIT of RAM; otherwise, it is stored in segment @@DATA.

Auto variable

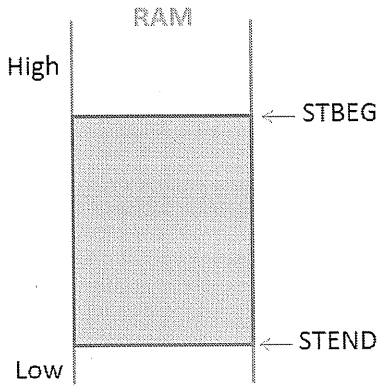
Variable in a function without being declared as static with/or without initial value is called auto variable. Auto variable is stored in the stack in RAM with push instruction and retrieved from the stack by pop instruction.

Question

- How about var32, var51?

Stack Area

✓ Setting the stack: CA78K0R (Build Tool) → Property → Link Options → Stack → Generate stack... Yes(-s)




```
NUM          FFEE0H    _@STBEG
NUM          F9FC2H    _@STEND
```

(Memory map file)

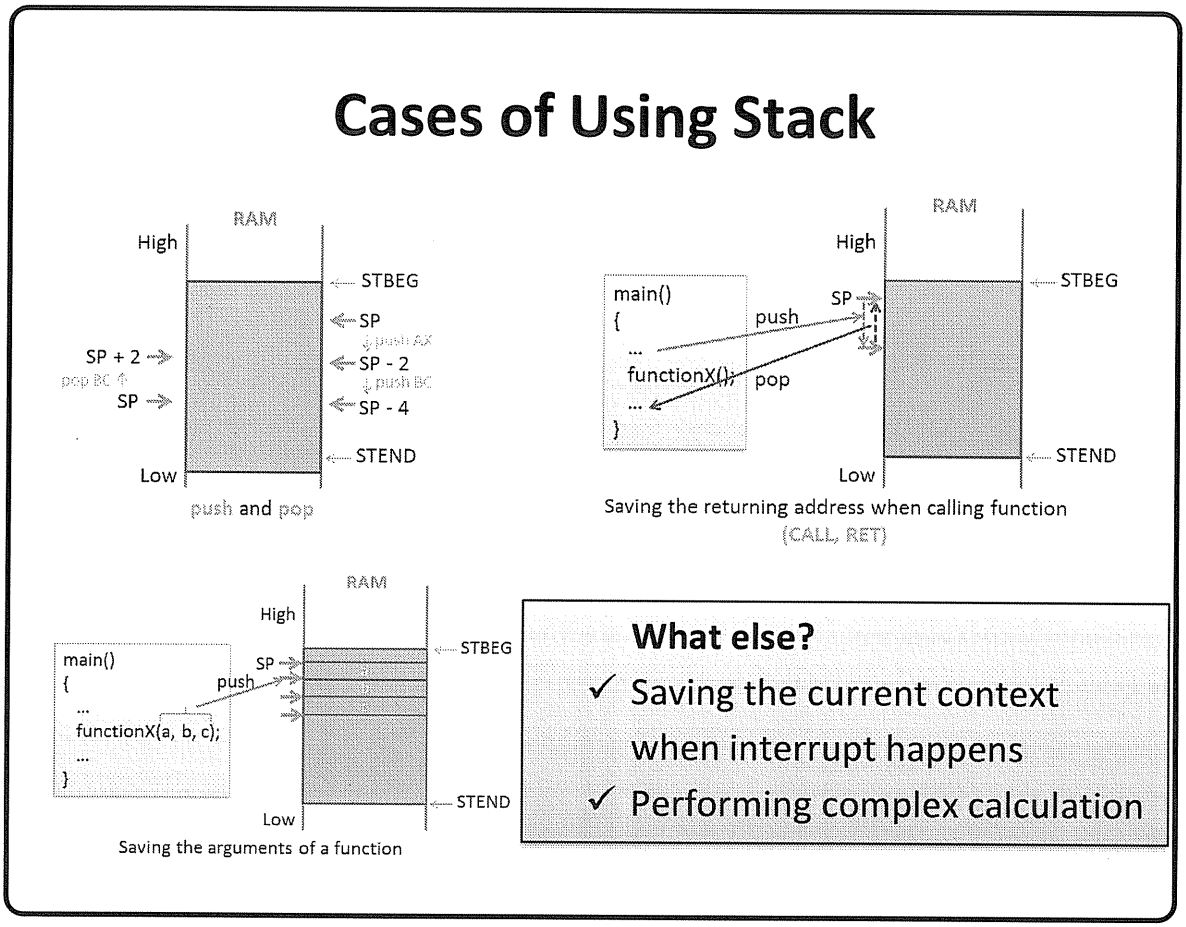
```
1  int main(void)
2  {
3      //ASM code
4      #asm
5          movw ax, #0FH
6          movw bc, #0EH
7          push ax
8          addw ax, bc
9          movw bc, ax
10         pop ax
11     #endasm
12
13     return 0;
14 }
```

Resolution symbol

If we want to specify a resolution symbol name for stack area, in the option for linking in CubeSuite+, we need to select "*Yes(s)*" to "*Generate stack resolution symbol*".

Stack beginning and ending address

After having built the source code, a memory map file is generated (*.map). Inside this file, the ending and beginning addresses of the stack area are shown.



push and pop

When there is auto variable in the function, the assembler will automatically use push to store the variable in the stack. In assembly program, we also often use push and pop to store and restore the value of variable.

Saving the returning address when calling a function

See chapter 3.

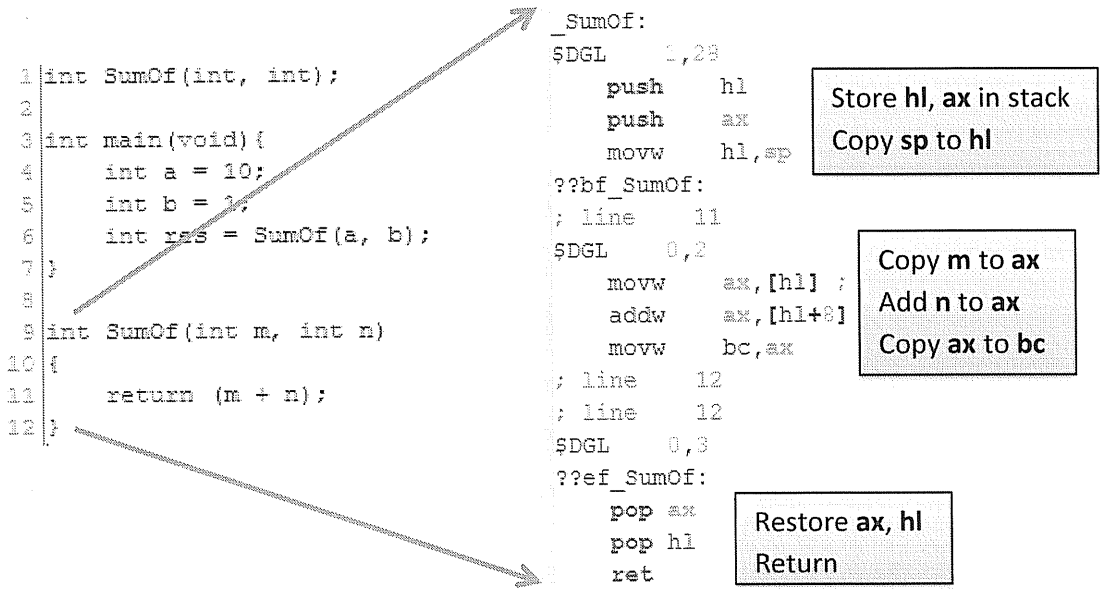
Storing the arguments of a function

The arguments of a called function are stored in the stack. When performing calculation with these arguments, we can access their values via stack pointer.

Other usages of stack

Find out or share your understanding on the other common usages of stack?

Example of Using Stack: Calling Function



Base register HL

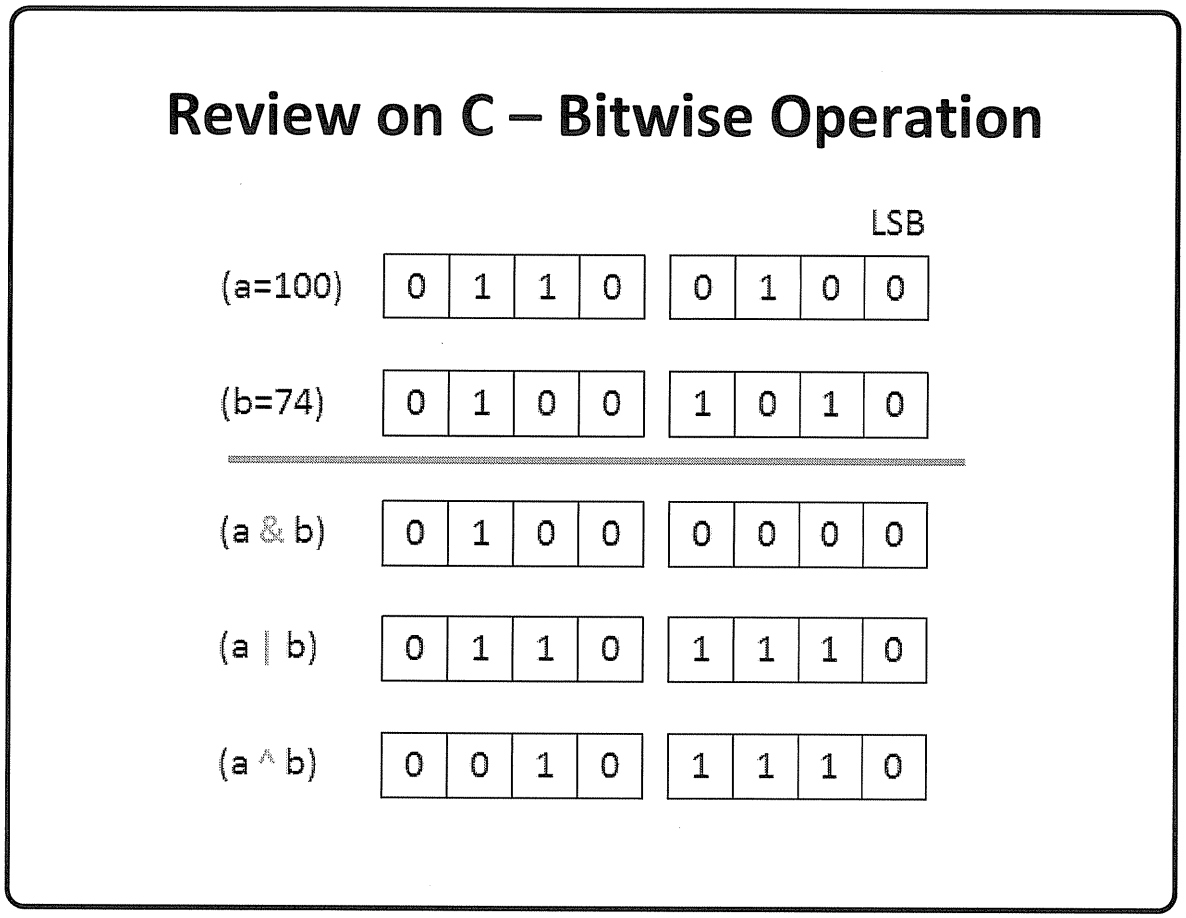
HL is the base register (16-bit) of a function when it is called.

Using HL

We must pay attention on using HL to access the input arguments of size larger than 16-bit of a function.

Question

Try to explain why there is "8" in the instruction "addw ax, [hl+8]"of the generated assembly code?



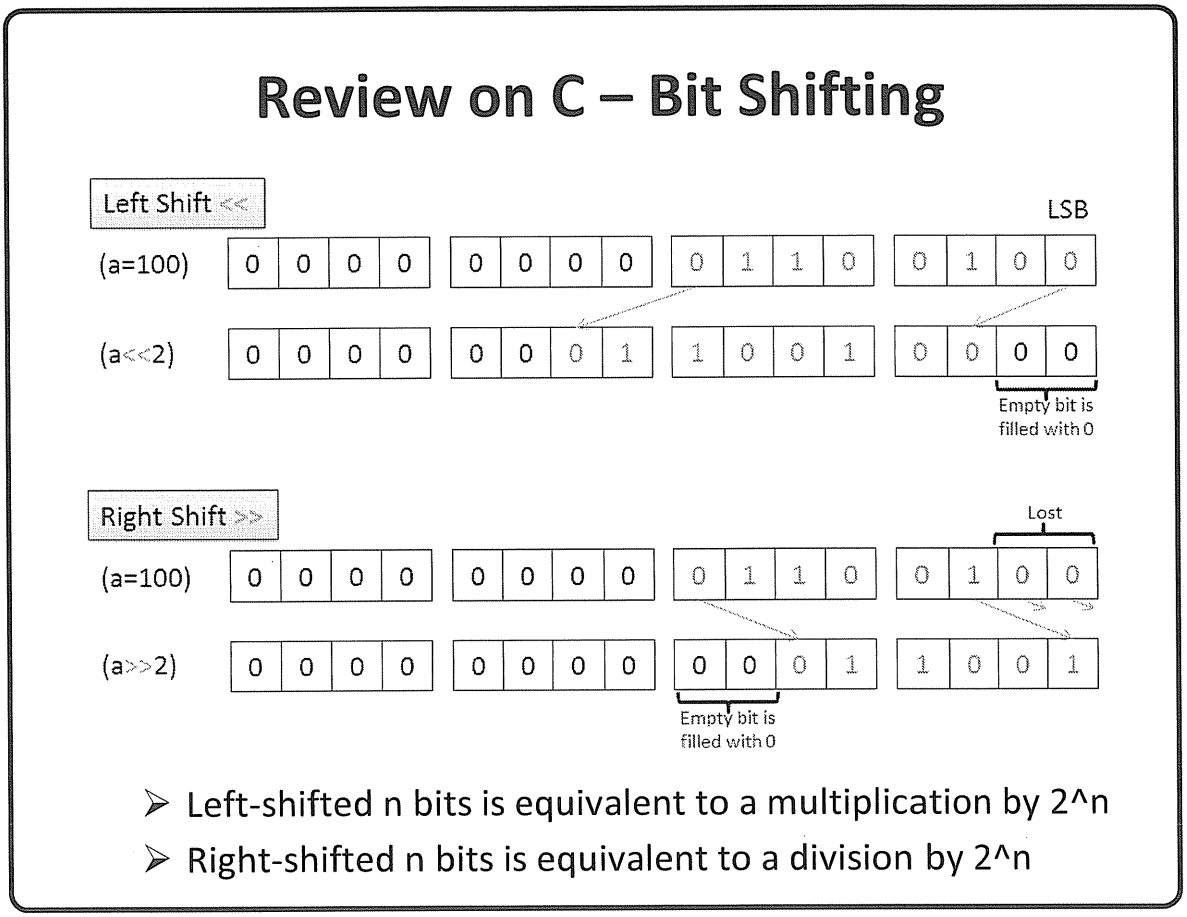
Bitwise operation

There are three operators: AND (&), OR (|), XOR (^). The numbers are represented in binary system and each of their bits will be applied bitwise operation. The usage of bitwise operation is to manipulate certain bits of registers and extract useful bit value for further control of a system. For example, if bit number 5 of the register A is equal to 1 equivalent to shut down the machine. We can set this bit to 1 by using OR operator as follows: A | $(0010\ 0000)_b$.

Logical operator

Do not confuse bitwise operators with logical operators:

Logical AND	&&
Logical OR	||
Logical EQUAL	==
Logical NOT EQUAL	!=



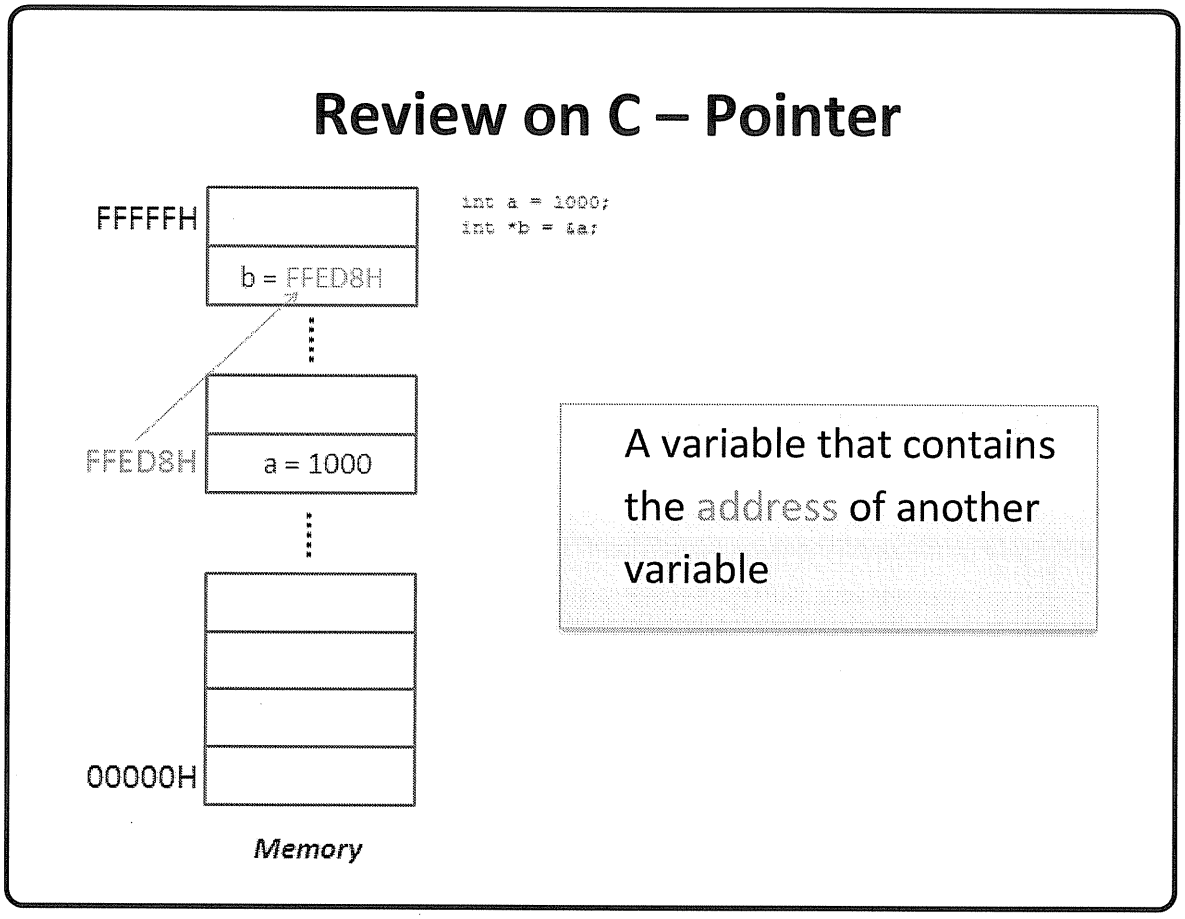
Logical bits shifting

Shifting bits with left and right shift is a simple and efficient way of performing multiplication and division of a number by 2^n (where n is the number of shifted bits). In case of multiplication, make sure that the return value can be contained in its pre-defined data type.

In logical bits shifting, the freed bits will be filled with 0 as shown in the above examples.

Arithmetic bits shifting

Arithmetic right shift has a difference compared to logical right shift. In arithmetic right shift, the vacant bits will be filled with the value of the MSB bit. Arithmetic left shift typically has no difference with logical left shift.



Declaration

To declare a pointer, we use * (asterisk) symbol. The address of a variable in C programming language is retrieved with & (and) symbol. A pointer must points to NULL (e.g. int *p = NULL) or to the address of an existing variable (e.g. int *p = &a).

If b is a pointer, the value of b will be the address of another variable that it points to. To access the value of the variable itself, we will use * symbol in front of b. It means, in the example above, we have: *b is equal to 1000.

Modifying the value of a variable via pointer

To modify the value of a in the above example, we can use *b. For example, if we write *b = 500, the value of a will be changed to 500.

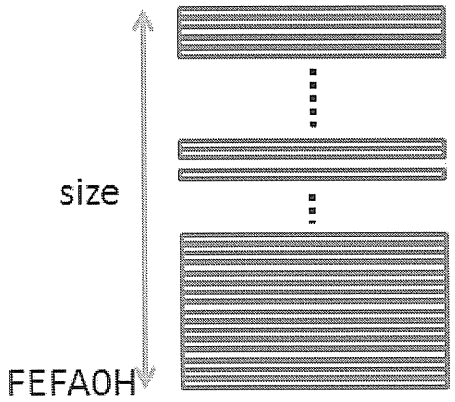
Usage of pointer

Share your understanding on the usefulness of pointer.

Review on C – Pointer Usage (1)

```
void timeConverter(int *hour, int *minute, int *second)
```

```
void passingBigData (int *my_data, size)
```



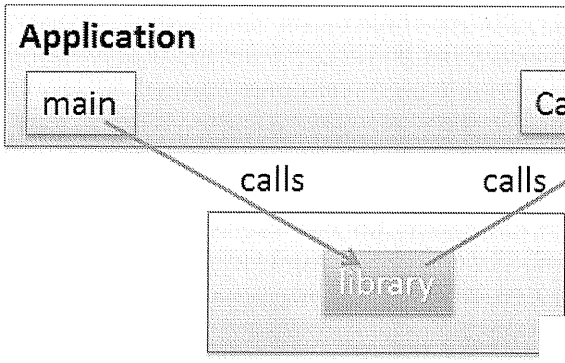
Modifying the values of the input arguments of a function

By using pointer, we can modify the values of the input arguments of a function. It is because the value at the address pointed by the pointer is now modified. This is a one of the common usages of pointer.

Passing big data into a function

In case of array, data structure, buffer stream... the input data may be very huge. Instead of passing all of the elements of the data into a function (which normally costs time and resources), we can just pass the address of the data and its size. This technique will help to improve significantly the performance of the code.

Review on C – Pointer Usage (2)




```
1  int sumOfTwo(int a, int b)
2  {
3      return a+b;
4  }
5  int main(void){
6      int (*fp)(int, int);
7      int a = 100, b = 200, c = 0;
8
9      fp = sumOfTwo;
10     c = (*fp)(a,b);
11     return 0;
12 }
```

Callback function

Another popular usage of pointer is function pointer which is used as callback function. A callback function is a function created in a higher context level but called by another function in a lower context level. Callback function is often used in interrupt. When there is interrupt, the interrupt handler function will be called to handle it.

Example of callback function

In Win32 API programming, Windows provides a library (windows.h) for creating and handling the behavior of objects (Button, Edit Text, Menu...). A callback function is created to set the properties of the windowing application. While the application is running, upon receiving interaction from user (e.g. a click on one of its buttons), the application will call the callback function accordingly to response to the action of user.

Review on C – Array

```
1  int main(void){
2
3
4      int myArayInt[5] = {0, 3, 5, 7, 9};
5      char myArrayChar[4] = {0, 4, 5, 6};
6      char myString[] = "Hello, World";
7
8      int my2DArrayInt[4][5] = {0};
9      int i = 0, j = 0;
10
11     for (i=0; i<4; i++){
12         for (j=0; j<5; j++){
13             my2DArrayInt[i][j] = i + j;
14         }
15     }
16     return 0;
17 }
```

The first index of an array is 0.

2D array is an array whose each element is an 1D array

	(j=0)	(j=1)	(j=2)	(j=3)	(j=4)
(i=0)	0	1	2	3	4
(i=1)	1	2	3	4	5
(i=2)	2	3	4	5	6
(i=3)	3	4	5	6	7

Multidimensional array

It is very often to see multidimensional array in programming. It may be the data of a 2D picture or a RGB data of a video frame. Like 1D array, when using multidimensional array as function argument, we should pass it by pointer.

Dynamic memory allocation

When the exact size of an array is unknown at the time of writing code (variable size), we use dynamic allocation technique to reserve necessary memory for it in runtime. Depending on the actual size of the array when the code is executing, the processor will allocate the necessary space to it. There are two important functions we use in dynamic allocation:

- malloc: Allocate the memory with a given amount of bytes.
- free: Release the allocated space back to the system.

Always remember to free the memory after using it to avoid memory leakage!

Review on C – #define and Macro

```
1  #define PI (3.1415)
2  #define MULTI_16(X) (X<<4)
3
4  #define RESET_ARRAY(X, N) \
5  do{\
6      int n = 0; \
7      for (n=0; n<N; n++){\
8          X[n] = 0;\
9      }\
10 }while(0)
11
12 int main(void){
13     int a = 100;
14     int x[10] = {0,1,2,3,4,5,6,7,8,9};
15
16     int c = MULTI_16(a);
17
18     RESET_ARRAY(x,10);
19
20     return 0;
21 }
```

Macro

Macro is used to automatize tasks.

#define directive

The #define directive can be used to define constants and in programs. Macros can be designed to perform simple calculations or even to execute more sophisticated code. When calling a macro in the program, the calling instruction is actually replaced by the code of the macro. Be careful on using parentheses in macro code to prevent unexpected result.

Review on C – Structure and Union

```c
#include <string.h>

typedef struct STRUCT_FRAME{
    int ID;
    int width;
    int height;
    char stream[10];
} STRUCT_FRAME;

int main(void){

    STRUCT_FRAME frame1;

    frame1.ID = 1;
    frame1.width = 1024;
    frame1.height = 1024;
    strcpy(frame1.stream,"MPEG4");

    return 0;
}
```

Data structure

```c
typedef union UNION_PORT{
    unsigned char all;
    struct STRUCT_BIT{
        unsigned char bit0:1;
        unsigned char bit1:1;
        unsigned char bit2:1;
        unsigned char bit3:1;
        unsigned char bit4:1;
        unsigned char bit5:1;
        unsigned char bit6:1;
        unsigned char bit7:1;
    } BIT;
} PORT;

int main(void){

    PORT portLed01;
    portLed01.all = 0xFF;
    portLed01.all = 0x00;
    portLed01.BIT.bit6 = 1;

    return 0;
}
```

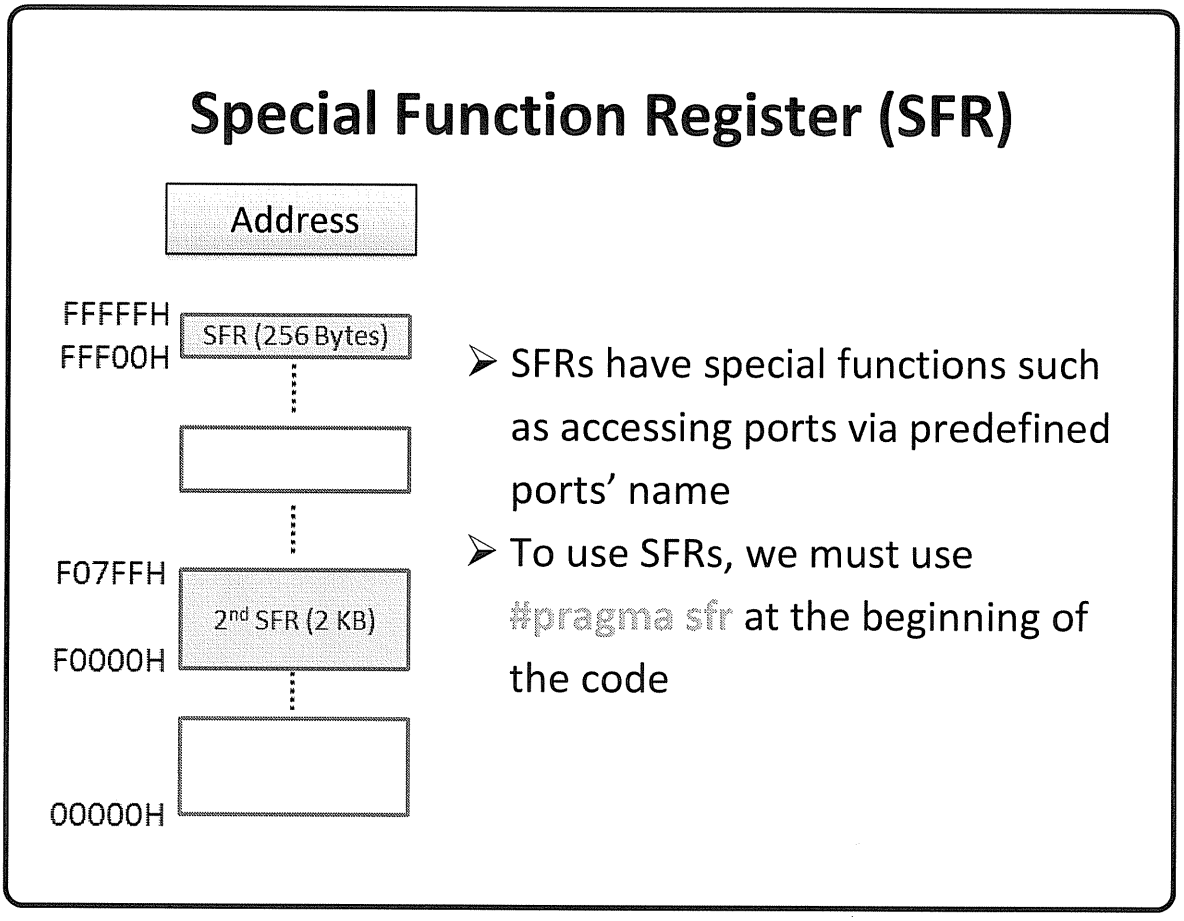
Union

Structure

Unlike array, data structure (struct) can contain different data fields that have different data types. To access each data field, we use symbol . or -> in case the spoken field is a pointer. Data structure is very useful because it allows us to group many data fields to represent certain "object" in our program. In the example above, the structure STRUCT_FRAME contains the parameters of a video frame.

Union

Another kind of data structure is union, which is very useful for manipulating bit. One advantage of using union is resources saving. In the example above, the variable portLed01 just takes one byte (8 bits). The fields all and BIT share the same memory space. As such, to modify all the bits of this portLed01, we can apply on the field all; and to modify each individual bit, we can make use of each bit in the structure BIT.



Special Function Register

In RL78/G14 microcontroller, there are 2 zones of SFR whose the addresses are shown in the above schema. SFRs predefine the ports by short names and bit fields for users to easily access and control them.

#pragma sfr

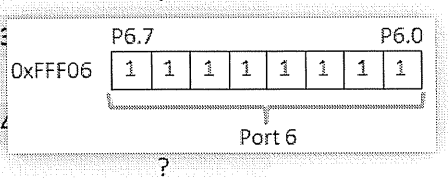
In order to access the ports' name, we must declare #pragma sfr at the beginning of C code. Otherwise, there will be error when compiling the code.

Register name (for RL78/G14)

We can refer to the hardware manual of the board to know the full list of registers predefined for using as SFR. Some of them are, for instance, PMC (Processor Mode Control register), CS register, ES register, PSW (Program Status Word), Port register (P0, P1,... P15), Timer register (TDR00, TDR01L), Port mode register (PM0, PM1,... PM15), Serial data transfer register (TXD0, RXD0...).

Example of SFR for LEDs

LED #	Color	SFR Port	Address of SFR Port	SFR Port Mode	Address of SFR Port Mode
3	Red	P6.2	0xFFF06H	PM6.2	?
5		P6.3		?	?
7		P6.4		PM6.3	
9		P6.5		?	
11		P6.6		PM6.4	
13		P6.7		?	?
4	Green	P4.2	?	?	0xFFF24H
6		P4.3		PM4.3	
8		P4.4			
10		P4.5		?	
12		P15.2	?	?	?
14		P10.1	?	?	0xFFF2A



Port mode register

Port mode register allows us to set a corresponding port as input or output. If the port is connected to LED and used to light on the LED, we have to set it to output mode, for example: PM6.2 = 0. Each port and port mode are associated to fixed addresses in the memory map. For example, it is 0xFFF06H in the case of Port 6.

Port 15

Unlike the other ports in the table above, Port 15 offers more features (e.g. Analog/Digital conversion). Therefore, its usage is a bit more complicated than the others.

Your task

- Refer to the hardware manual of RL78/G14 RDK and the one of the microcontroller to find out the missing information in the above table.
- Investigate how to enable Port 15 as analog output port.

Example of Using SFR

```
1  #pragma sfr
2  extern void WaitMe();
3
4  int main(void){
5
6      //set 6.2 (LED 3) as output
7      PM6.2 = 0;
8
9      while (1U){
10         //off
11         P6.2 = 1;
12         WaitMe();
13
14         //on
15         P6.2 = 0;
16         WaitMe();
17     }
18
19     return 0;
20 }
```

- WaitMe(): a function written in ASM that "generates" a delay
- This sample code "flashes" the LED #3 on the board

Pausing the program

In a C program running on personal computer, we can use sleep() function to make the program wait for certain amount of time. However, such a function does not exist in embedded C. In this case, we normally write by ourselves a function to make the code wait. It can be:

- a C function having a finite loop
- an assembly function (or subroutine)

There are other techniques that we will learn in chapter 6 on Timer. We will see how we can implement software timer and hardware timer to control the time in our program.

Accessing Hardware via Direct Address

```
1  extern void WaitMe();
2  #define PORT6 (unsigned char *)0xFFF06
3  #define PORTM6 [redacted]
4
5  int main(void){
6
7      //set 6.2 (LED 3) as output
8      *PORTM6 &= 0xFFFFB;
9
10     while (1U){
11         //off
12         *PORT6 |= 0x00004;
13         WaitMe();
14
15         //on
16         *PORT6 &= 0xFFFFB;
17         WaitMe();
18     }
19
20     return 0;
21 }
```

- No need #pragma sfr
- Need to know where the register is in the memory map
- Need to use pointer

Direct address access

One of the methods for accessing register in embedded programming is to use its direct address. At first, we need to define a pointer pointing to the address of the register being used. Then, using * operator to modify the value held at this address. In the example above, we manipulate the behavior of the LED #3 via the address of Port 6.

Bitwise operation

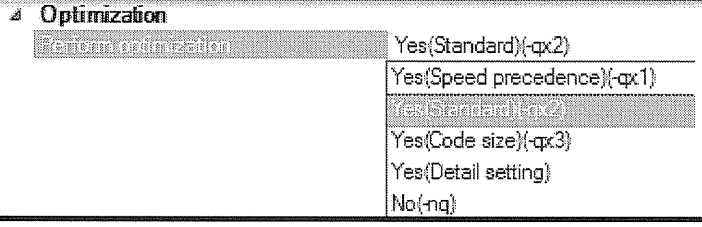
To change the value of a particular bit in the address pointed by the pointer, we can make use of bitwise operators. In the example above, we have used AND and OR operators.

Your task

- To run the above code successfully, complete the address of the port mode register (PM6).
- Verify the operand of the above bitwise calculations. Try to understand why they are like this.

Volatile Variable

- ✓ The generated code (ASM) from C source is affected by the optimization options of the compiler.
- ✓ The generated instructions can be different from what C source intends to do; and the executable program may act unexpectedly.
- ✓ It is important to declare variable as volatile to prevent the complier from doing assumption during optimization.



Optimization options in CA78K0R compiler

To select optimization level, in CubeSuite+ we can go to CA78K0R (Build Tool) from the Project Tree. Then, right click -> Property -> select Compiler Options tab -> Optimization category -> Perform optimization.

We can disable optimization (No) or select one of the optimization levels: standard, speed, code size, or customized (Detail setting). For command line usage and for further details of customized optimization, refer to the Build Help documentation of the compiler.

Volatile variable

volatile is a type qualifier (like const). For a variable which is not declared with volatile (for example: int i = 0), the compiler will make assumption that this variable will not be modified by external program. Based on this assumption, the compiler will perform optimization in order to, for example, increase the speed of the program; reduce the code size... In embedded programming, the values of a register can be often altered while the program is running due to, for example, interrupt; reading signal from external environment; or

data written to a port by some other processes. For safety reason, the variable should be declared as volatile letting the compiler know that it is subject to be change at all time. The compiler will not do unexpected optimization.

Preventing Unexpected Bugs Using volatile and const

- ✓ Declaring I/O port and control register as volatile.
- ✓ Declaring read-only data are as const to prevent unexpected writing to it.
- ✓ Declaring pointer variable as const to fix a reference-indirect area, and to prevent from changing this reference-indirect area.

const variable

It is also important to declare read-only data as const variable to prevent unexpected data to be written to it.

Example of Optimization & Volatile (1)

```
1 int main(void){
2     int i = 0;
3     i = i + 1;
4     i = i + 2;
5 }
```

```
1 int main(void){
2     volatile int i = 0;
3     i = i + 1;
4     i = i + 2;
5 }
```

No optimization

```
    clrw    ax
    movw    [hl],ax ; i
; line      3
$DGL    0,3
    incw    ax
    movw    [hl],ax ; i
; line      4
$DGL    0,4
    addw    ax,#02H ; 2
    movw    [hl],ax ; i
```

```
    clrw    ax
    movw    [hl],ax ; i
; line      3
$DGL    0,3
    movw    ax,[hl] ; i
    incw    ax
    movw    [hl],ax ; i
; line      4
$DGL    0,4
    movw    ax,[hl] ; i
    addw    ax,#02H ; 2
    movw    [hl],ax ; i
```

With volatile (code on the right)

The variable i is stored in register HL. Whenever an instruction performs calculation on this variable, it always refers to the latest value of i (which is the content of the address pointed to by HL).

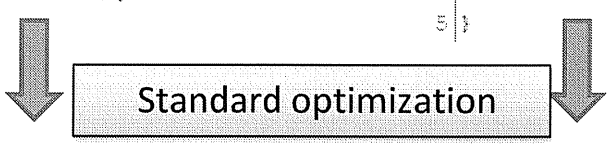
Without volatile (code on the left)

The variable i is not updated before executing an instruction. Here, the compiler is assuming that i is not altered by other program other than the current one. As a result, no updating process is needed.

Example of Optimization & Volatile (2)

```
1 int main(void){
2     int i = 0;
3     i = i + 1;
4     i = i + 2;
5 }
```

```
1 int main(void){
2     volatile int i = 0;
3     i = i + 1;
4     i = i + 2;
5 }
```




```
; line     2
$DGL    0,2
    clrw    ax
; line     3
$DGL    0,3
    incw    ax
; line     4
$DGL    0,4
    incw    ax
    incw    ax
```

```
; line     3
$DGL    0,3
    movw    ax,[hl] ; i
    incw    ax
    movw    [hl],ax ; i
; line     4
$DGL    0,4
    movw    ax,[hl] ; i
    incw    ax
    incw    ax
    movw    [hl],ax ; i
```

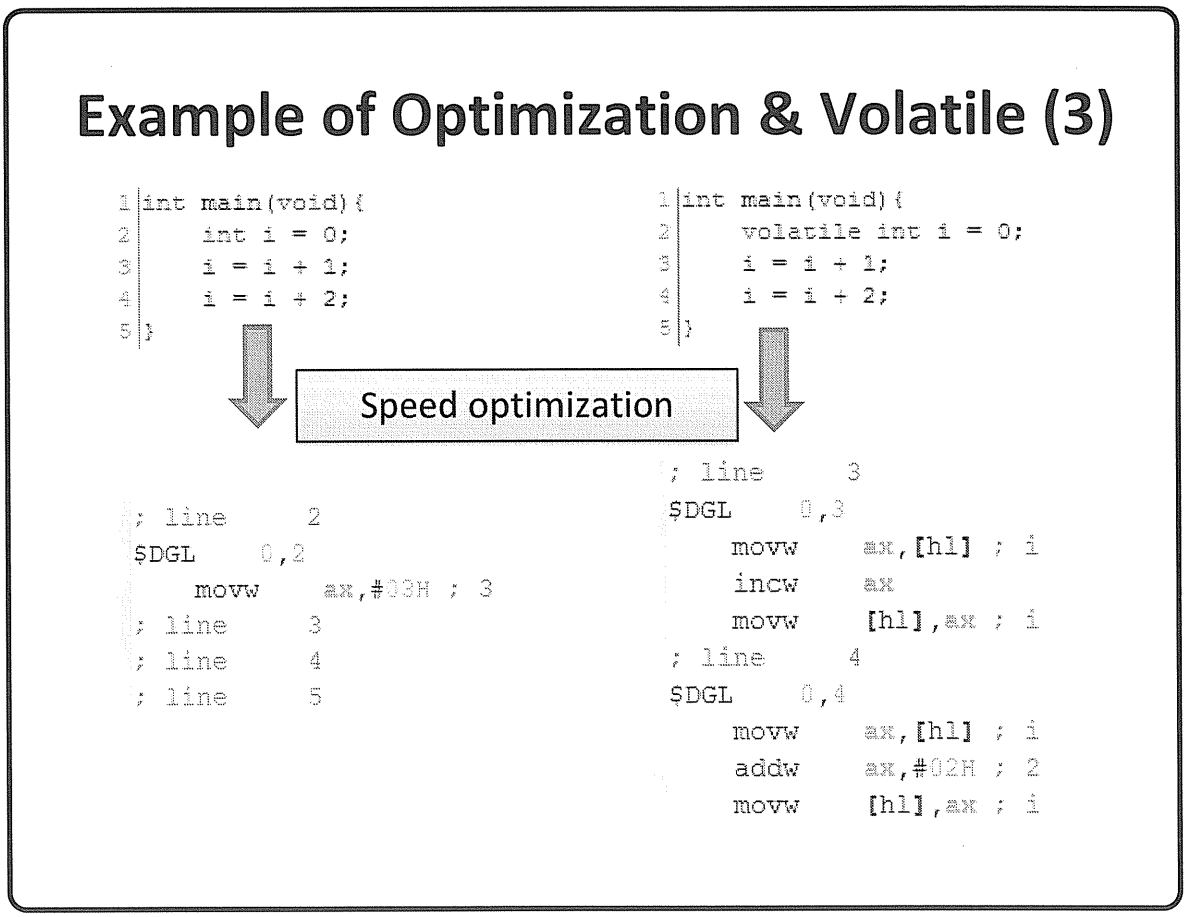
With volatile (code on the right)

There are not many changes compared with the case (1). We notice that only the instruction i = i + 2 is now compiled to "incw ax" twice.

Without volatile (code on the left)

Now, there is even no usage of register HL to store the result of i. Every instruction is now executed through the intermediate of register AX. If the value of i is modified by an external process while this code is running, the final result of AX will be inaccurate.

Example of Optimization & Volatile (3)



With volatile (code on the right)

The code has no difference compared with the case (1). We can say that by using **volatile** the compiler will suppress optimization.

Without volatile (code on the left)

The compiler implicitly replaces two instructions (at line 3 & 4) by only one instruction. It does not care whether or not the value of i will be changed during the time the 2 instructions are executed.

Summary: It will be bug prone if we define variables inappropriately. All assumptions that the compiler will make for optimization may cause the code to run unstable.

ROMization

- ✓ Recall: Variables are stored on different locations on ROM and RAM, depending on their types (const, static, auto variable, with or without initialized value...).
- ✓ ROMization is a process of placing initial values into ROM and then copying them to variables in RAM when the system is executed.

The startup program cstart.asm

The startup program contains a portion of code performing ROMization process. See the next page for some descriptions of this program, and refer to the compiler manual for further details.

Startup Process

- ✓ (1) Preprocessing
- ✓ (2) Initializing memory
- ✓ (3) Initializing hardware
- ✓ (4) Executing ROMization
- ✓ (5) Performing entry process for C code (main)
- ✓ (6) Performing exit process for C code (exit)

Preprocessing

The startup program performs settings for library to use; defines directive labels and segment labels for ROMization process.

Initializing memory

The startup program initializes the memory area for each library to be used, for the stack; and selects register bank. It also initializes external variables without initial values.

Initializing hardware

By default, the function hdwinit is called to initialize hardware such as LCD display, wireless device... If this file is empty, the startup process will replace it by a simple return instruction (RET).

Executing ROMization

The code of ROMization is also included inside the startup program. In this process, initial values will be placed in ROM and copied to the variables declared in RAM.

Segment Labels (in ROM)

✓ To understand well the ASM code generated by the compiler and the startup program, we should understand the definition of different segment labels.

Segment	First Label	End Label	Meaning
@@R_INIT	_@R_INIT	_?R_INIT	External variable with initial value (when allocated to near area)
@@RLINIT	_@RLINIT	_?RLINIT	External variable with initial value (when allocated to far area)
@@R_INIS	_@R_INIS	_?R_INIS	**sreg** variable with initial value (when allocated to near area)
@@CNST	-	-	(Refer to the compiler manual)
@@CNSTL	-	-	(Refer to the compiler manual)
@@CODE	-	-	(Refer to the compiler manual)
@@CODEL	-	-	(Refer to the compiler manual)

ROM

Segment name

The initial values of the external variables and the code instructions are stored in ROM with the segments shown in table above. Depending on the nature of the variable (for example: constant, allocated to near or far area...), it will be allocated to different segment. The full list of all segments can be found in the compiler manual.

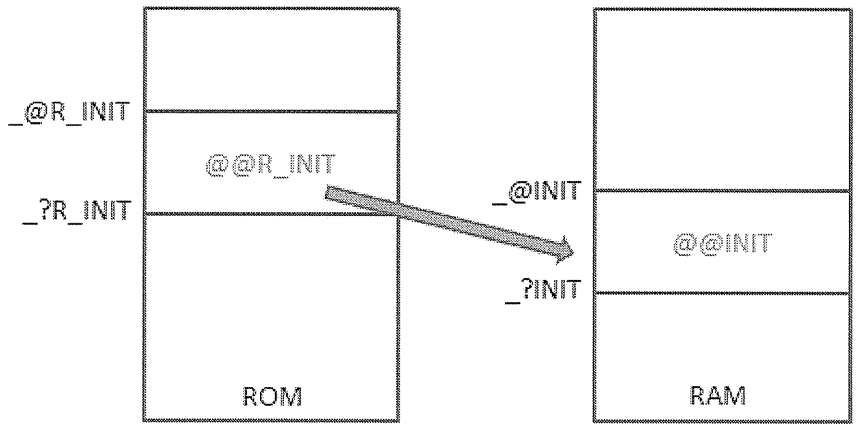
Segment Labels (in RAM)

✓ Similarly, there are labels defined for different segments in RAM.

Segment	First Label	End Label	Meaning
@@INIT	_@INIT	_?INIT	External variable with initial value (when allocated to near area)
@@INITL	_@INITL	_?INITL	External variable with initial value (when allocated to far area)
@@DATA	_@DATA	_?DATA	External variable without initial value (when allocated to near area)
@@DATAL	_@DATAL	-	External variable without initial value (when allocated to far area)
@@INIS	-	-	(Refer to the compiler manual)
@@DATS	-	-	(Refer to the compiler manual)
RAM			

ROMization in Startup Program

✓ The startup program contains assembly code, which is a loop, to copy the values from ROM to the variables in RAM; initialize the variables without initial values...



ROMization performs the following tasks

- Copying initial values in the segment @@R_INIT of ROM to variables in the segment @@INIT of RAM
- Copying initial values in the segment @@RLINIT of ROM to variables in the segment @@INITL of RAM
- Copying initial values in the segment @@R_INIS of ROM to variables in the segment @@INIS of RAM
- Initializing variables in the segment @@DATA of RAM to zero
- Initializing variables in the segment @@DATAL of RAM to zero
- Initializing variables in the segment @@DATS of RAM to zero

Reading Linker Address Map file

After building the code, this file is generated containing the information on the address map of all symbols and segments used. It is also necessary to understand how to read the information of this file.

Linker Address Map File

Sample Code

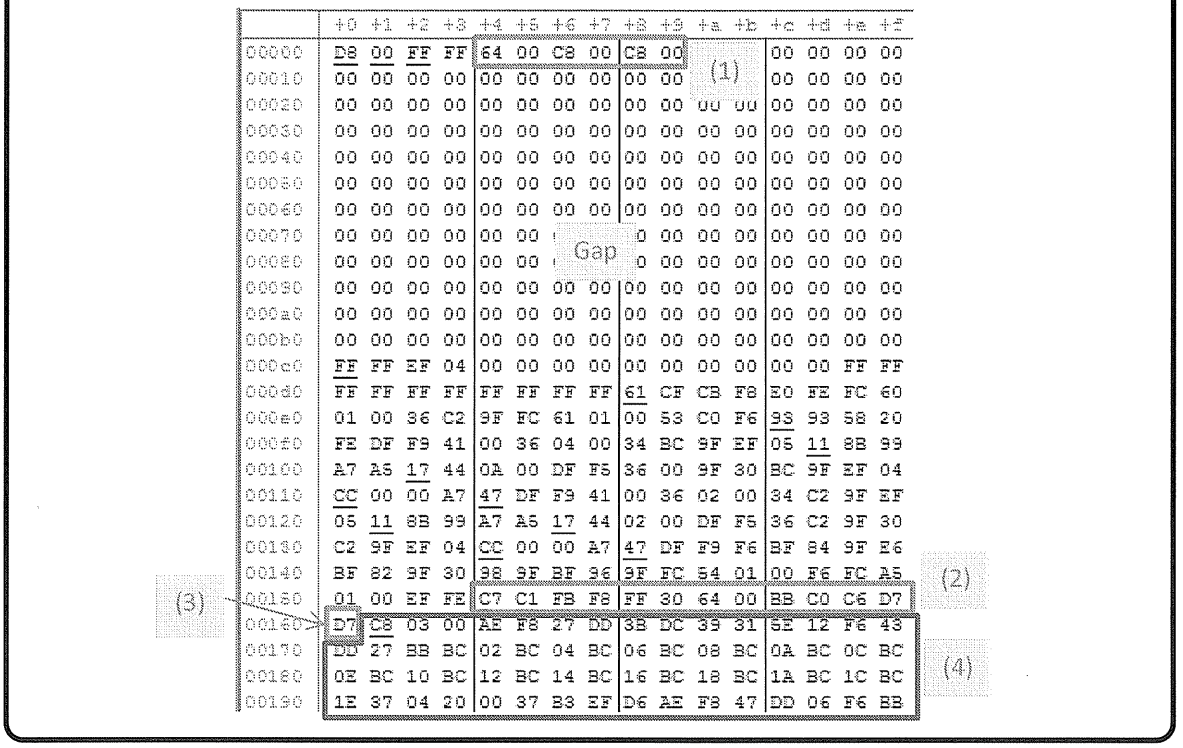
```
1  int i = 100;
2  int j = 200;
3  char k = 200;
4
5  int a;
6  int b;
7
8  const int c = 100;
9
10 int main(void){
11     int n = 100;
12 }
```

```
MEMORY=ROM
BASE ADDRESS=00000H   SIZE=40000H
       OUTPUT   INPUT     INPUT      BASE      SIZE
       SEGMENT  SEGMENT   MODULE     ADDRESS
       @@VECT00                      00000H    00002H
                @@VECT00  @cstart    00000H    00002H
                          ...
   (1) @@R_INIT                      00004H    00006H
                @@R_INIT  @cstart    00004H    00000H
                @@R_INIT  main       00004H    00006H
                @@R_INIT  @rom       0000AH    00000H
   (2)                    ...
       @@CODEL                       00154H    0000CH
   (3) @@CODEL   main     ...        00154H    0000CH
       @@LCODEL                      00160H    00061H
                @@LCODEL  @hdwinit   00160H    00001H
                @@LCODEL  @stkinit   00161H    00044H
                @@LCODEL  exit       001A5H    0001CH
                          ...
MEMORY=RAM
BASE ADDRESS=F9F00H   SIZE=06100H
       OUTPUT   INPUT     INPUT      BASE      SIZE
       SEGMENT  SEGMENT   MODULE     ADDRESS
   (4) @@DATA                        F9F00H    000BCH
                @@DATA    @cstart    F9F00H    000B8H
                @@DATA    main       F9FB8H    00004H
                @@DATA    @rom       F9FBCH    00000H
   (5) @@INIT                        F9FBCH    00006H
                @@INIT    @cstart    F9FBCH    00000H
                @@INIT    main       F9FBCH    00006H
                          ...
```

Explanation

- (1) In ROM: The segment @@R_INIT is found in 3 codes cstart, main and rom. And in the main code, the size of this segment is 6 bytes (which corresponds to the 3 initial values of i and j and k).
- (2) In ROM: The segment for code @@CODEL is found the function main and the size of its instructions is 12 bytes.
- (4) In RAM: The segment @@DATA for the function main has the size of 4 bytes (which corresponds the 2 variables a, b).
- (3) and (5): Try to explain their meaning?
- How about the variable n?

Reading Memory Information (CS+)



Memory view

In debugging mode, we can use the memory view tool to observe the variables and instructions in the memory when the code is running. To open this tool: Menu View → Memory → Memory 1.

Explanation

The memory view reflects exactly the memory map in the previous slide. For example: (1) is the 6 bytes of the segment @@R_INIT in which the values are 100, 200, 200 respectively for the variables i, j and k declared in the program. (2) is the instructions of the main function. The size of all the instructions is 12 bytes from the address 0x00154 to 0x0015F. (3) is the return instruction of the hdwinit. As this function is empty, the compiler just adds a return function which corresponds to 1 byte of code instruction. (4) is the code of the stkinit process (stack initialization).

(This page intentionally left blank)

Chapter 5.

Linking C and Assembly

In this chapter:

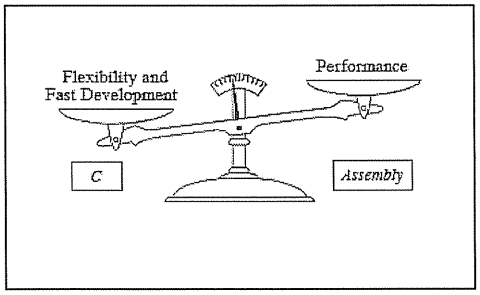
- Data Transfer between C Functions
- Linking C and Assembly
- Rules on Argument Transfer and Return Value
- Rules on Symbol Conversion
- Calling C Function in Assembly Subroutine and vice versa
- Inline Assemble Function

Programming Language

	Advantage	Usage
C	C can describe a program similar to human language, so program readability, productivity and maintainability can be improved. ➢ A variety of operators is usable: +,-,*,/,%,^,!,==,... ➢ Control structure can be used: if statement, if-else statement, for statement, while statement.. ➢ Data structure can be used	➢ Data analysis ➢ Data processing ➢ User interface
Assembly	Assembly language can describe program similar to machine language. ➢ It can control processor, e.g.: ✓ Register operation ✓ Stack operation ✓ Interrupt operation ✓ Control by using CPU own instruction ➢ Access CPU peripheral devices	➢ Processor control ➢ Hardware control ➢ High-speed process

Which language is the best for our application?

It depends on what is more important to us. If we need flexibility and fast development, choose C. On the other hand, use assembly if we need the best possible performance.



Programming Language

Here are some things should be considered:

1. How complicated is the program? If it is large and intricate, we will probably want to use C. If it is small and simple, assembly may be a good choice.
2. Are we pushing the maximum speed of our program? If so, assembly will give us the last drop of performance from the device. For less demanding applications, assembly has little advantage, and we should consider using C.
3. How many programmers will be working together? If the project is large enough for more than one programmer, lean toward C and use in-line assembly only for time critical segments.

C program vs. assembly code

When a C program is compiled, does the executable code resemble *efficient* or *inefficient* assembly program? The answer is that the compiler is able to generate *efficient* code.

Every compiler provides options to optimize code. If our program is simple, the compiler will be able to generate executable which is as optimized as a program written directly in assembly. However, if we are developing something strange or complicated, expect that an assembly program will execute significantly faster than a program written in C, because it is not obvious for the compiler to automatically optimize a complicated program.

Programming Language

Here are some things should be considered (cont.):

4. Which is more important, product cost or development cost? If it is product cost, choose assembly; if it is development cost, choose C.
5. What is our background? If we are experienced in assembly (on other microprocessors), choose assembly for our program. If our previous work was in C, choose C for our program since many things could be re-used.
6. Which programming language does the customer request us to use?

Customer requirement

Suppose that our customer requests using assembly since they prefer performance and they are ready to pay more. How we can reject this requirement!

Assume that our customer lets us choose a programing language. We will ask HW manufacturer which language to use, and they will tell us: *"Either C or assembly can be used, but we recommend C."* We had better take their advice! What they are really saying is: *"Our HW system is so difficult to program in assembly that we will need long time of training to assembly it deeply."* On the other hand, some HW systems are easy to program in assembly.

Product cost

It refers to the costs used to create a product. These costs include direct labor, direct materials, consumable production supplies, and factory overhead. Product cost can also be considered the cost of the labor required to deliver a service to a customer.

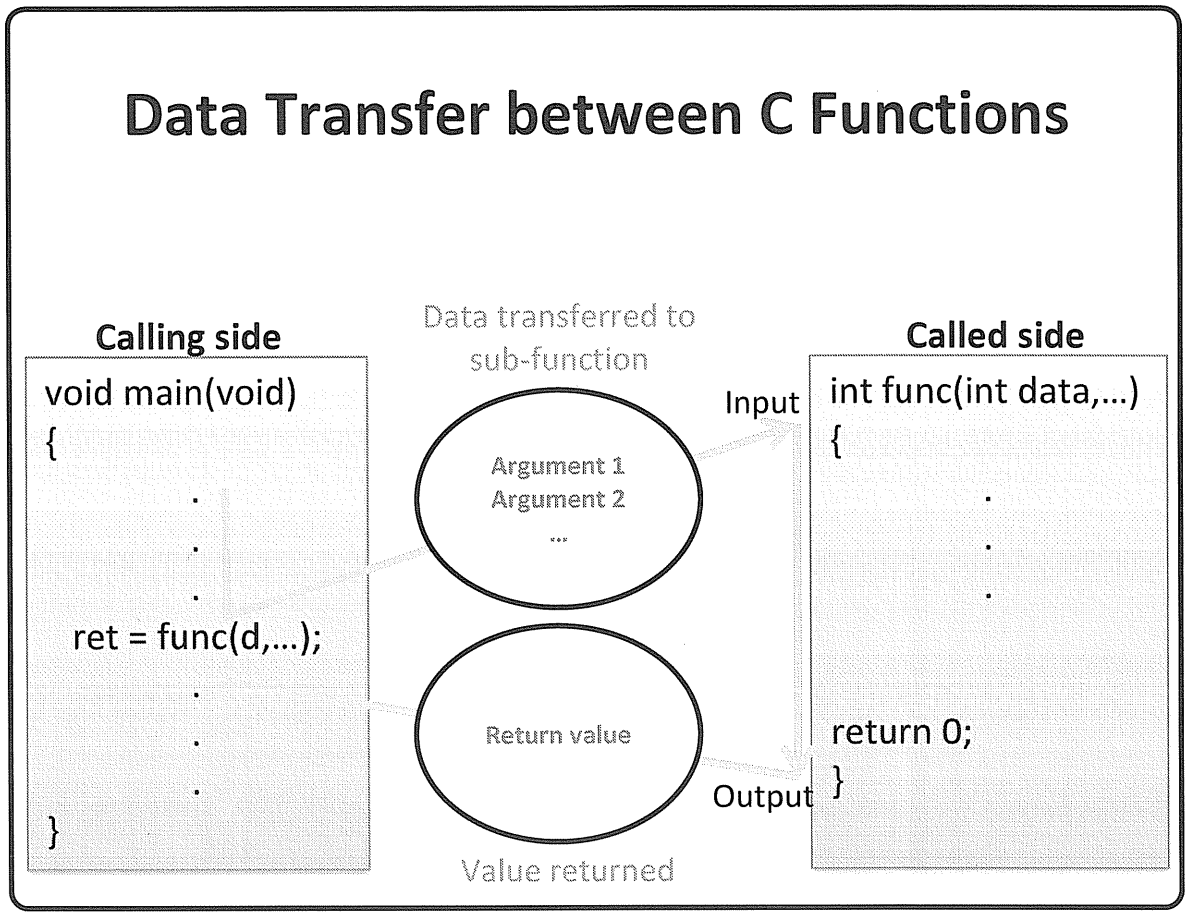
Development cost

It is the total of all costs incurred from initiation to implementation of a project.

Necessity of Linking C and Assembly

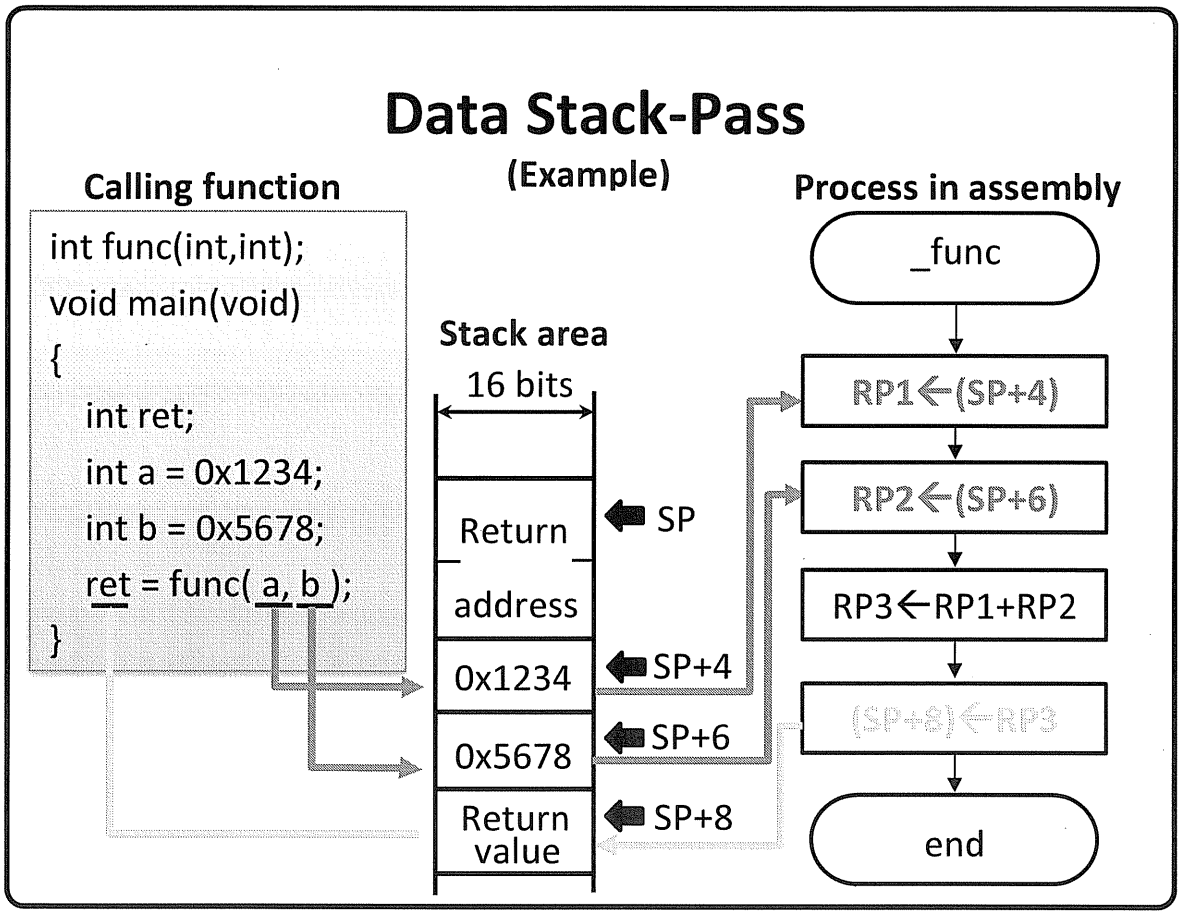
Although software is developed in C, some parts of programs sometimes need to be developed in assembly

(1) Some special instructions cannot be described in C language.
Example: Access control registers, interrupt process, etc.
(2) Many resources were already written in assembly language.
(3) Processing speed and ROM efficiency of program can be improved.



Arguments and return value of C function

C language allows transferring multiple arguments between two C functions via register or stack. But, only one value is allowed to be returned from the function being called.



Stack-pass

The method that uses stack as a temporary storage area to save the arguments and the return value of a function is called "stack-pass".

Calling function by stack-pass

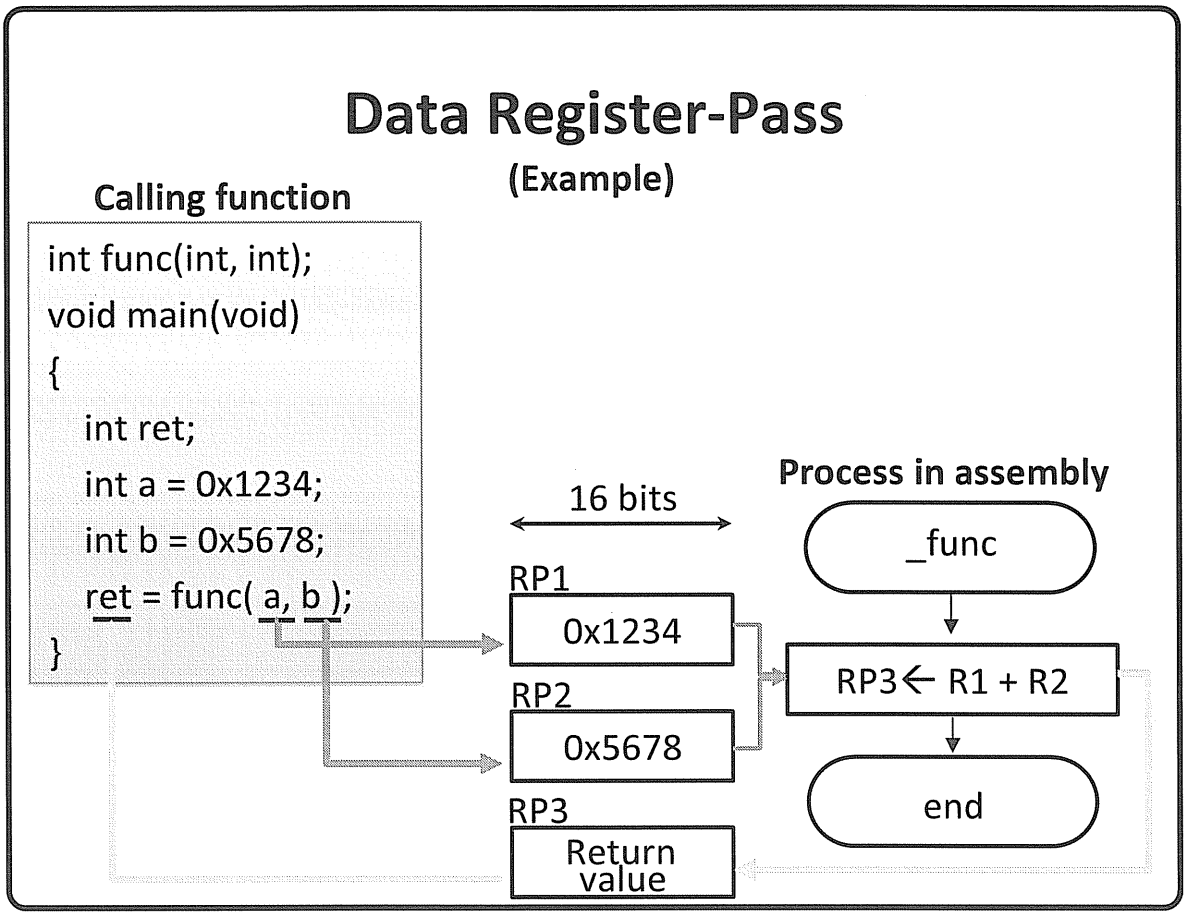
When calling func(), we copy the value of its arguments to the stack area for data transfer. Its return value is also stored on the stack.

Argument transfer and return value of a function via stack

In order to perform arithmetic calculations in func(), arguments are read from the stack area to registers (some compilers, however, allow to directly access memory). The calculation result is copied to the return value area ensured in the stack beforehand.

What is different in reality?

In the above example, the stack pointer SP is decremented. How would it be in reality (CA78K0R compiler & RL78/G14 board)?



Register-pass

The method that uses registers as temporary storage areas to save the arguments and the return value of a function is called "register-pass".

Calling function by register-pass

When calling func(), we copy the value of arguments to registers for data transfer. The return value is also stored on a register.

Argument transfer and return value of a function via register

When func() goes on arithmetic, it is not necessary to read out the arguments from the stack like in stack-pass technique because the value of the arguments has already been copied to registers. Therefore, arithmetic operations can be done immediately, and the arithmetic result is stored in register too.

Linking C and Assembly Language

The following points should be noted when linking C and assembly language.

1. Rules on argument transfer
2. Rules on return value
3. Rules on symbol conversion

Interfacing functions

Linking between assembly and C programs has to be done with respect to some rules about function interfacing.

The rules for function interfacing depend on the specifications of a given compiler. Therefore, developers must read the user's manual of the compiler and carefully confirm the contents of the rules.

Rules on Argument Transfer and Return Value

Rules on Argument Transfer

The rules define that temporary data should be allocated to either stack or register when transferring arguments between functions.

Rules on Return Value

The rules define that temporary data should be allocated to either stack or register when transferring return value between functions.

Stack-Pass

Advantage: The number of arguments and their size depend on the stack size.
Disadvantage: Processing speed is slow and stack is used.

Register-Pass

Advantage: Processing speed is high and stack is not used.
Disadvantage: The number of arguments and their sizes are limited.

Importance of understanding the rules

When passing data from C function to assembly function, the data will not be passed properly unless we comply with the rules on argument and return value transfer that are required by the compiler.

As the compiler will access registers and stack in accordance with these rules, developers have to pass data to the specified registers or stacks along with the rules when programming. In addition, the symbol of function and variable name must also be converted based on the rules; otherwise, linking error will occur.

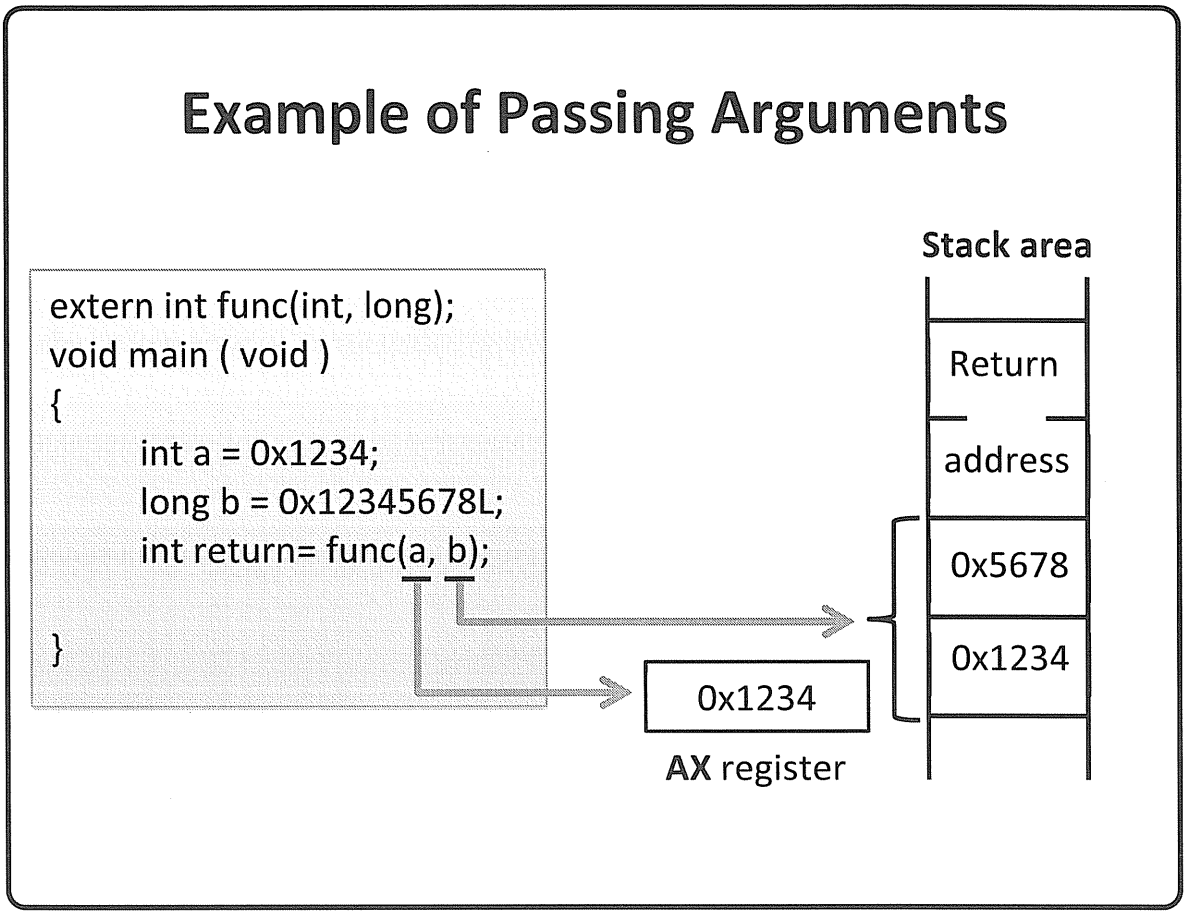
Rules on Argument Transfer
(CA78K0R)

1. The second and following arguments are passed to functions on the stack.
2. The first argument is passed to the function definition side via a register or stack. The location where the first argument is passed is shown in the table below.

Data type	Storage location
1-byte data 2-byte data	AX
Pointer to near data	AX
3-byte data 4-byte data	AX, BC
Pointer to function Pointer to far data	AX, BX
Floating-point number	AX, BC
Other	On the stack

Registers

AX and BC are the 2 general-purpose registers in the processor.



Order of passing arguments to stack

The schematic above just gives a simple explanation on how the 1st and the 2nd argument are passed to register and stack respectively. In reality (CA78K0R compiler and RL78/G14 board), this process is a bit different. Refer to the compiler manual to see how exactly the 2nd argument is transferred to the stack. (Hint: Pay attention on the order of passing the return address, the higher 16 bits and lower 16 bits of the 2nd argument.)

Rules on Return Value

(CA78K0R)

The return value of a function is stored in registers or carry flags. The location where the return value is stored is shown in the table below.

Type	Storage location
1-bit	CY
1-byte integer 2-byte integer	BC
Near pointer	BC
4-byte integer	BC (lower), DE (upper)
Far pointer	BC (lower), DE (upper)
Floating-point number	BC (lower), DE (upper)
Structure	The structure to be returned is copied into private storage for the function, and the address of the copy is stored in BC and DE.

Register

BC is a general-purpose register in the processor.

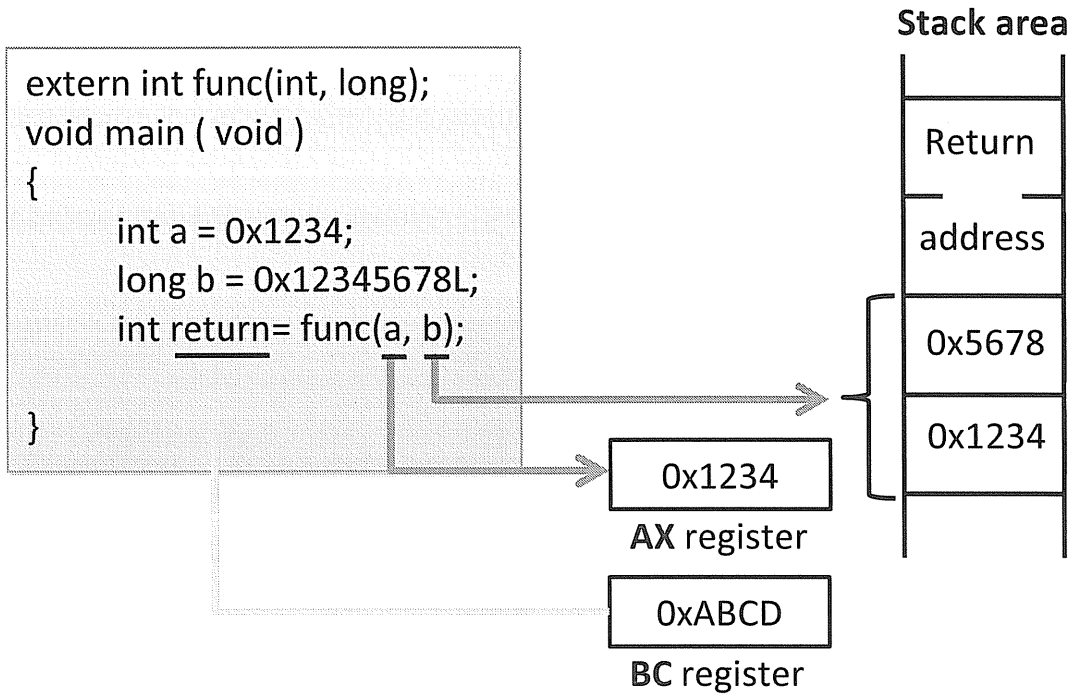
Flag

CY is a carry flag which is the bit 0 of Program Status Word (PSW) register.

PSW

The Program Status Word is an 8-bit register that stores the "status" of the program. For example: Interrupt is enabled or not? The result of an operation is zero or not? Which register bank is used? The result of a subtraction/addition is overflow or not? (Note: Refer to chapter 3 for details.)

Example of Return Value



Rules on Symbol Conversion (1)

(CA78K0R)

The rules define how to express the symbol of function name and variable name when calling between assembly and C functions.

(Example)	Symbol conversion	
	C	Assembly
Function name	main() func()	_main() _func()
External variable name	abc	_abc

Rules on symbol conversion

When interfacing C and assembly functions, we have to take care of the rules on symbol conversion. The function names and variable names defined in C function have to be properly converted to assembly names following the rules predetermined by the compiler.

An example illustrating the rules is given in the next page.

Rules on Symbol Conversion (2)

(CA78K0R)

Referencing variables defined in C from assembly.

```
extern void subf (void) ;
char c = 0 ;
int i = 0 ;

void main (void)
 {
       subf ( ) ;
}
```

C

```
$PROCESSOR (F1166A0)
PUBLIC _subf
EXTRN _c
EXTRN _i
@@CODE CSEG
_subf :
MOV !_c, #04H
MOVW AX, #07H
MOVW !_i, AX
RET
END
```

ASM

Rules on symbol conversion

If external variables defined in a C language program are referenced in an assembly routine, the "extern" declaration must be used.

Underscores "_" are added to the beginning of the variable names defined in the assembly routine.

Rules on Symbol Conversion (3)

(CA78K0R)

Referencing variables defined in assembly from C

C

```
extern char c ;
extern int i ;
void subf (void)
{
        c = ' A ' ;
        i = 4 ;
}
```

ASM

```
NAME ASMSUB
PUBLIC _i
PUBLIC _c
ABC DSEG BASEP
_i : DW 0
_c : DB 0
END
```

Rules of symbol conversion

If variables defined in an assembly routine are referenced from a C function, they have to be declared with the keyword PUBLIC.

Calling ASM Subroutine from C (1)

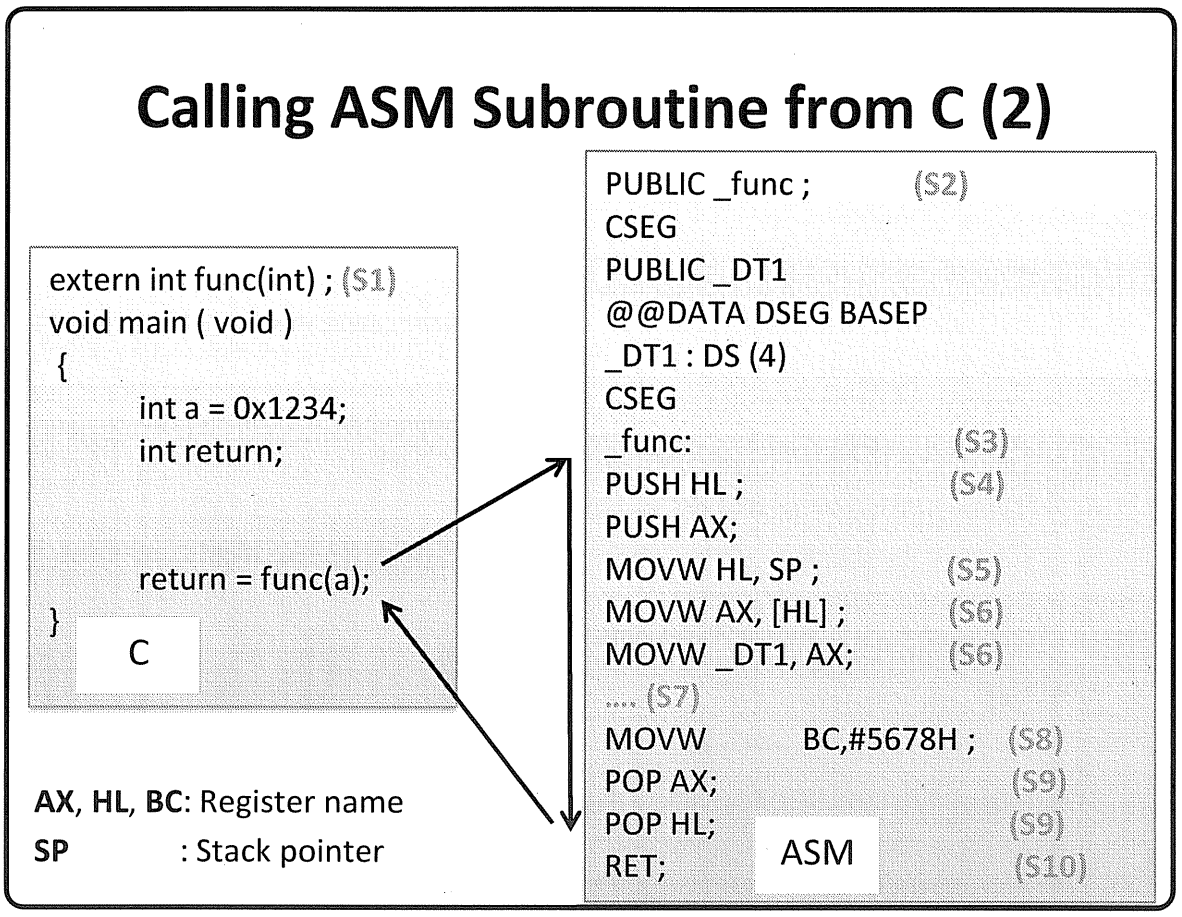
Calling an ASM subroutine (assembly function) from C function should be on the following steps:

1. Declare the prototype of the ASM subroutine to be called
2. Declare the ASM subroutine with global type
3. Define a label of the assembly subroutine
4. Save the base pointer, **saddr** area for register variable
5. Copy the stack pointer (**SP**) to the base pointer (**HL**)
6. Fetch out argument(s)
7. Write code
8. Set the return value
9. Restore the saved register
10. Return from subroutine (assembly function)

Saving base pointer and work register

A label with "_" is prefixed to the function name declared in the C source. Register HL is used as a base pointer for accessing arguments and automatic variables stored on the stack. After the label is specified, base pointer (HL) and work registers (AX, BC and DE) are saved after the label of ASM subroutine in ASM source code.

In the case of programs generated by the C compiler, other functions are called without saving the saddr area for register variables. Therefore, if ASM subroutine changes values of these registers for those other called functions, be sure to save the values beforehand. However, if register variables are not used on the call side, saving the saddr area for register variables is not required.



Assembly function

The function func() being called is described as an assembly subroutine in an assembly source code. The method for calling an assembly subroutine is the same as for a C function.

Prototype declaration

In C function, the assembly subroutine must be declared as external (extern) because its source code is created in another source file which is different from the one of the C function.

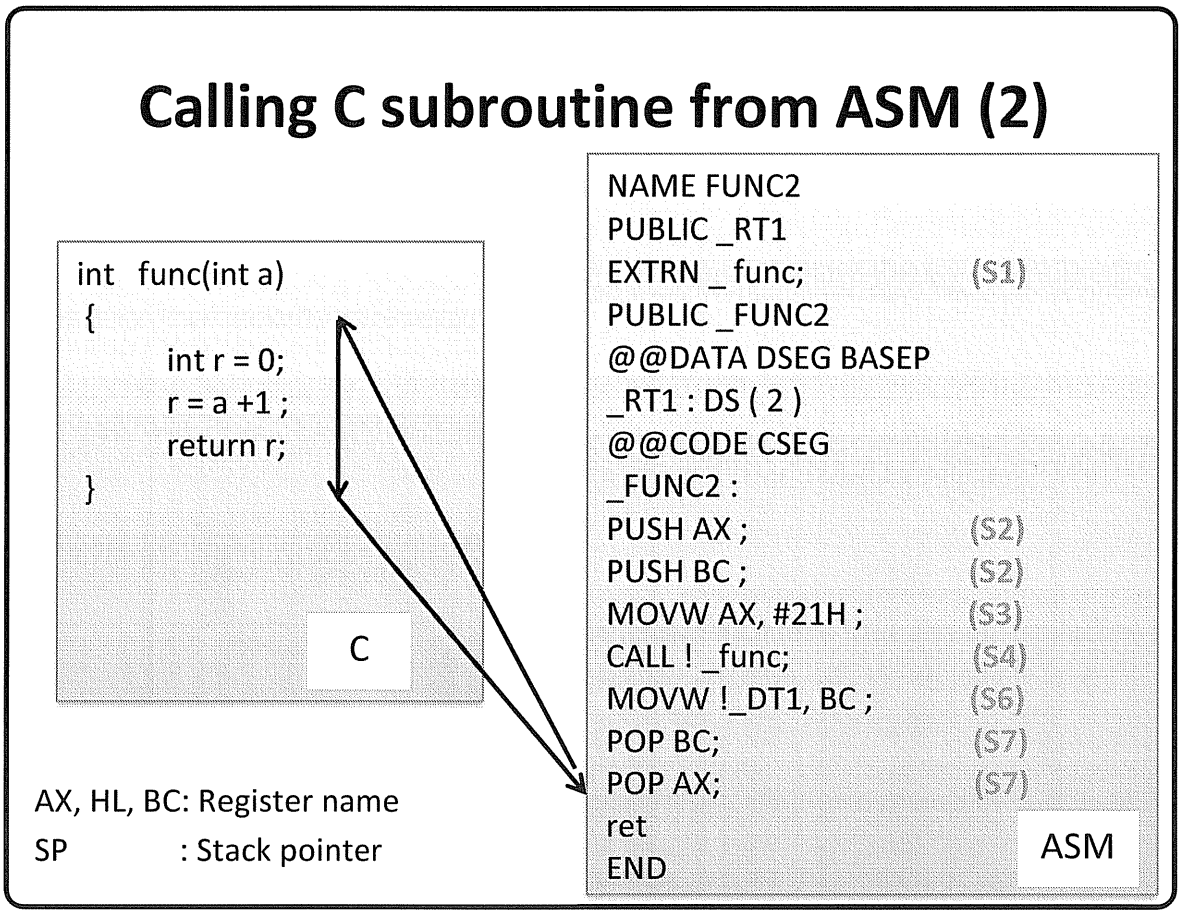
Global declaration of assembly subroutine

The body of the assembly function is created in another file separate from the C source file. Therefore, for the assembly subroutine to be called from within C source file, the symbol of the assembly function needs to be defined as *global* (the keyword PUBLIC in CA78K0R).

Calling C Subroutine from ASM (1)

Calling a C subroutine (C function) from an assembly function should be on the following steps:

1. Declare the C subroutine to be called with the keyword **EXTERN**
2. Save registers
3. Transfer arguments
4. Call C subroutine
5. Restore the stack pointer **SP**
6. Reference the return value
7. Restore the saved registers



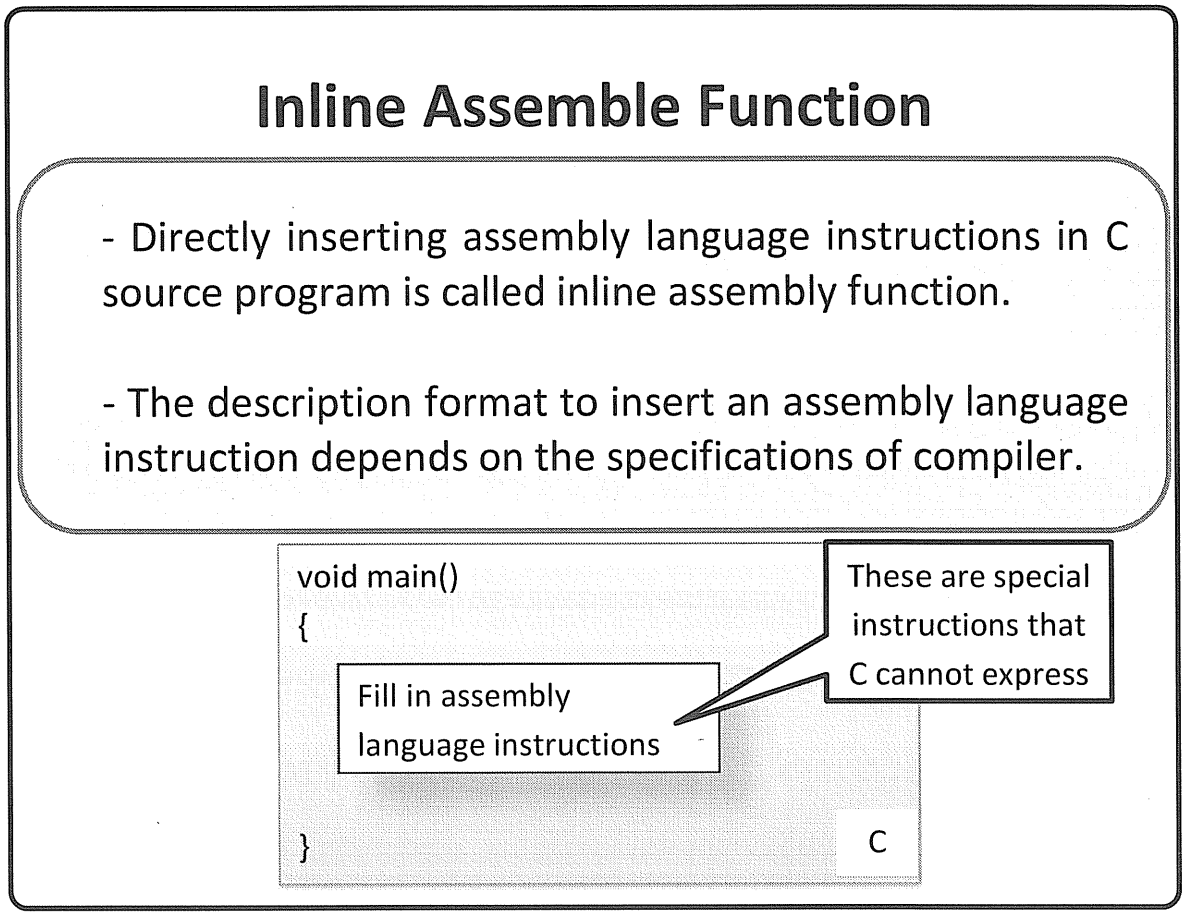
Saving work registers (AX, BC, and DE)

The 3 registers AX, BC, and DE are used in the C language. Their values are not restored after returning from C subroutine. Therefore, if the values in registers are needed, they should be saved on the calling side (i.e. assemble code in this case).

The HL register is always saved on the side of the C language when it is used in the C language.

Restoring the stack pointer (SP)

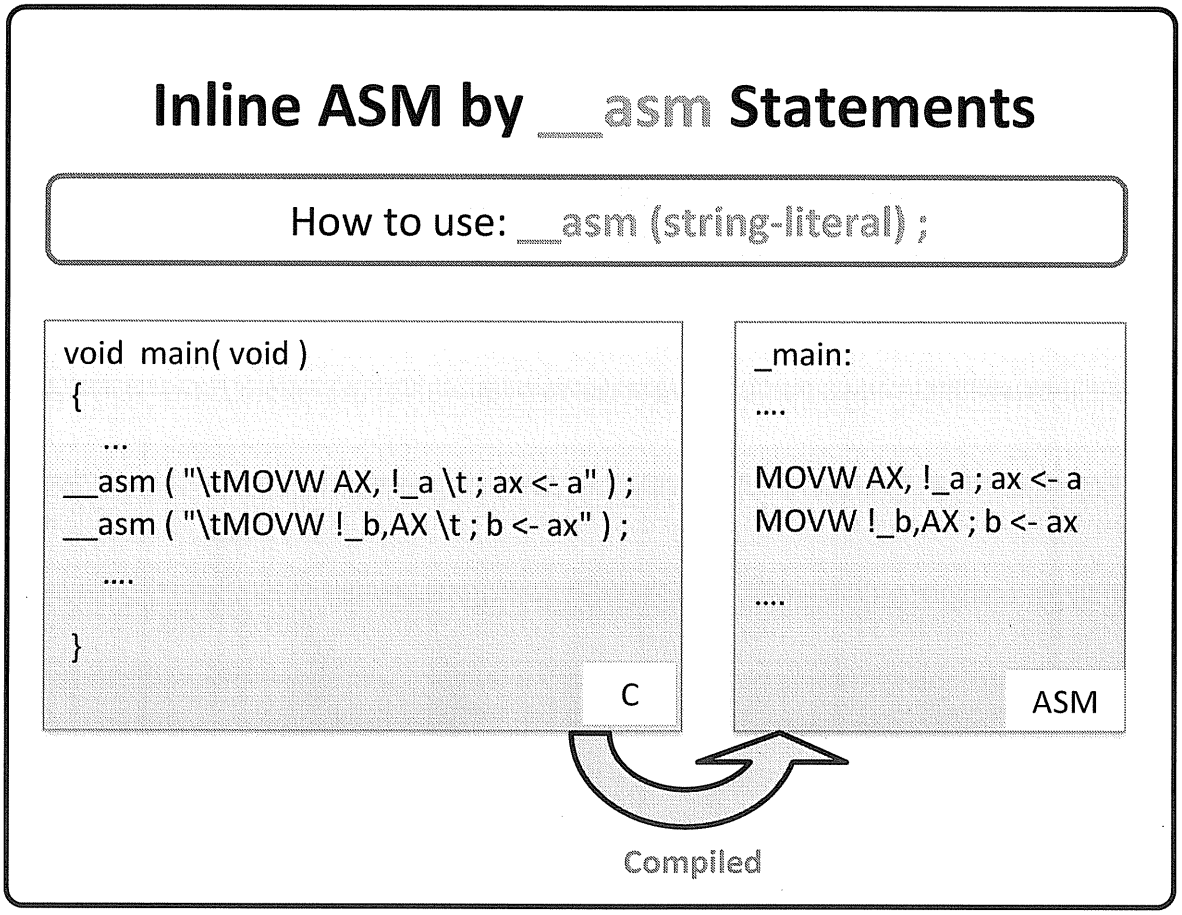
The stack pointer is restored by a number of bytes that were used to hold the arguments.



Inline assemble function

When we need to write a special command (e.g. to access control register, to work on flag register...) that cannot be expressed in C language, it is necessary to directly insert an assembly language instruction in C source via a special function. Such a special function is called "inline assemble" function.

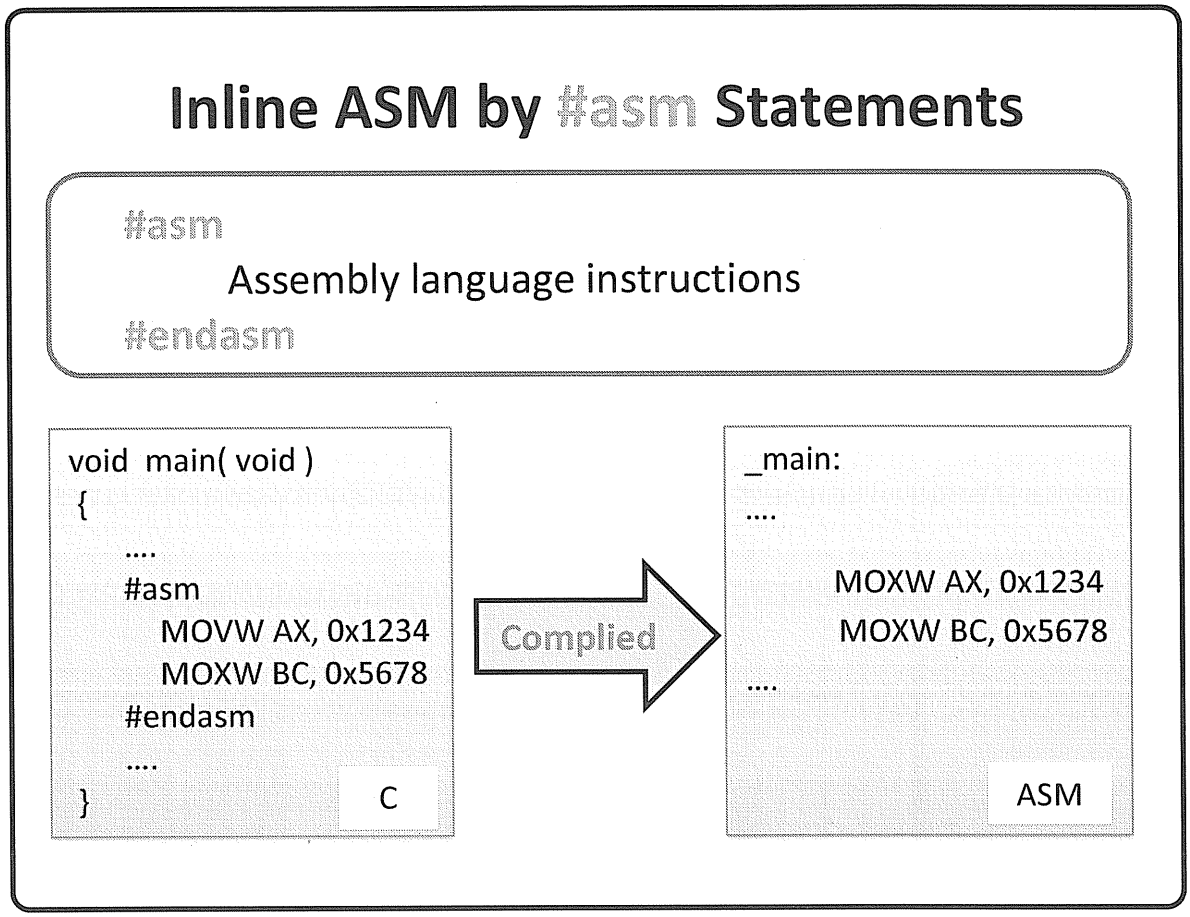
The inline assemble function name and its description format rely on compiler specifications (not regulated in ANSI C). Developers should check and follow the user's manual of each compiler to understand how to use this feature.



__asm

The __asm function can be described inside/outside a function. It is often used to directly control a flag or register, or in the case we need high-speed processing.

The character string (including space and tabs) surrounded by quotation mark can be directly copied to assembly program.



#asm, #endasm

When we want to insert multiple lines of assembly language in C program, using #asm and #endasm of CA78K0R is very convenient.

(This page intentionally left blank)

(This page intentionally left blank)

Chapter 6.

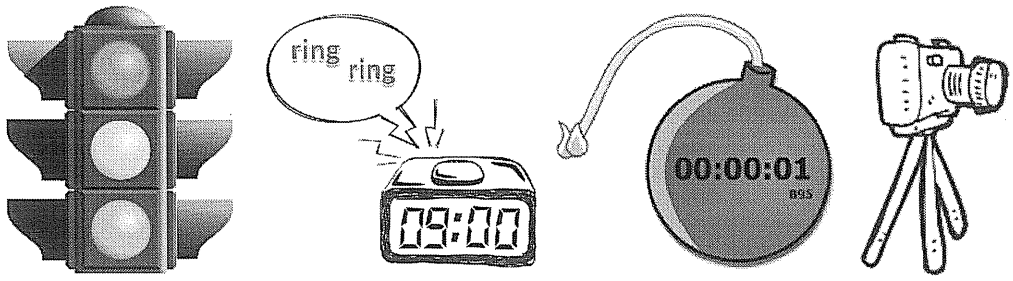
Time Management

In this chapter:

- **Necessity of Time Management**
- **Software Timer**
- **Hardware Timer**
- **Usage of Timer**
- **Time Management in RL78/G14**

Necessity of Time Management (1)

When we need time management?

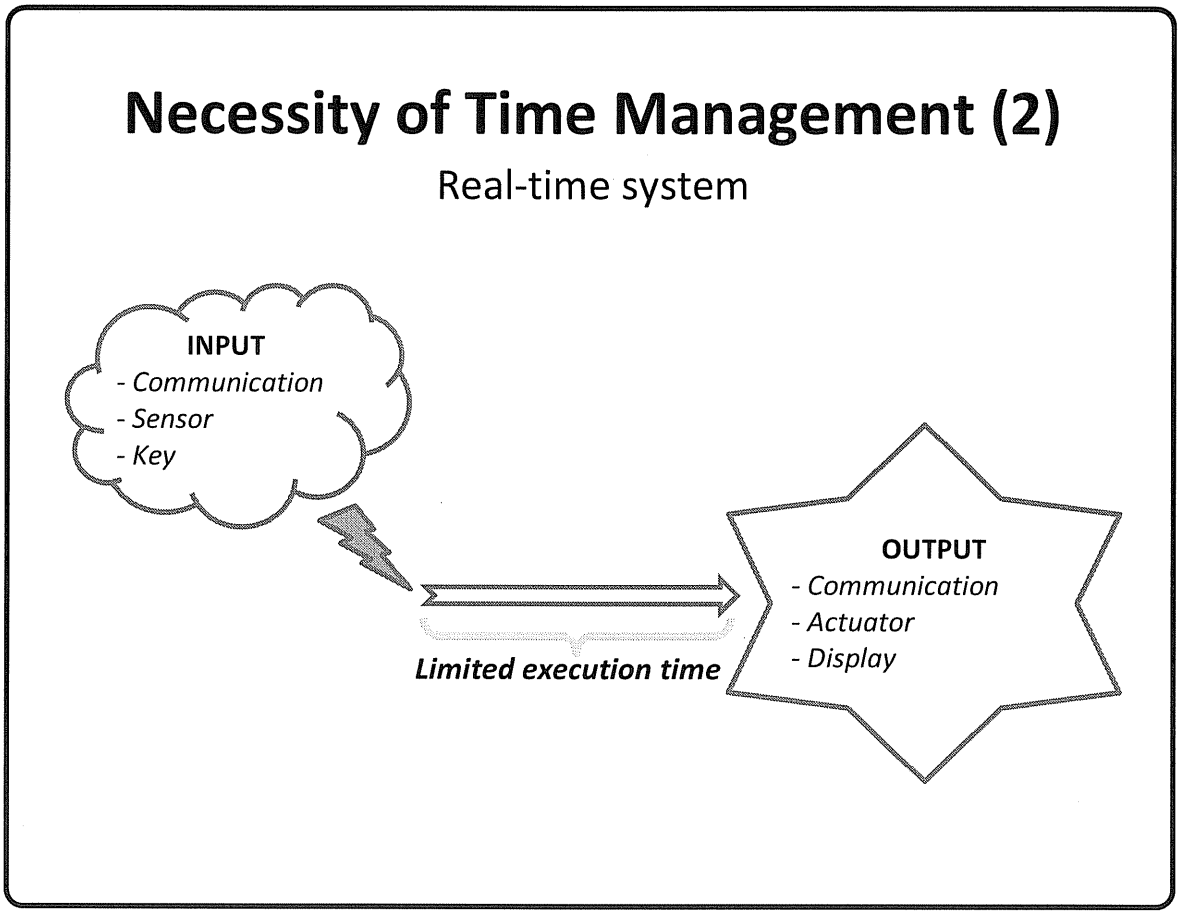


Introduction to timer

A timer is a software program or hardware device that either keeps track of how much time has been spent on doing something or counts down a specified duration of time. For example:

- A person may have a timer to keep track of how much time he has spent on surfing websites.
- A count-down timer is used to let a person know when to stop or start something.

Timer is widely used in electronic devices such as traffic lights, alarm clock, camera, time bomb, etc...



Real-time processing

In embedded system, multiple events often occur periodically or randomly, and it is required to respond to each of the events in a limited time. Such a system is called real-time system, and the response processing is called "real-time processing".

Calculation of program execution time

- How to calculate 1 cycle time?
- How to calculate the execution time of an instruction?

Instruction execution cycle

One period of the internal clock is called "1 cycle". How many cycles are required to execute an assembly language instruction depends on the instruction itself.

Necessity of Time Management (3)

Time requirement in embedded system

1. Time requirement for embedded systems
   - Process within certain time.
   - Process after certain time.
   - Process at certain time (real-time).
   - Process after each every given amount of time (periodic process).

2. Kinds of timer:
   - Software timer
   - Hardware timer

Software timer

This is a technique that generates a waiting time using the execution time of program. It is also called as "software wait".

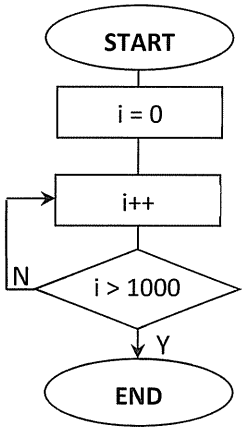
Hardware timer

Timer function is implemented by a built-in timer of microcontroller; or connected to an external circuit.

Software Timer (1)

What is software timer?

Example: A loop with 1000 iterations



Execution time calculation (assuming one iteration takes one execution cycle):

Execution time of one cycle (F = 32.768 MHz):

.....

.....

Number of cycles needed for this program:

.....

.....

.....

Execution time of this program:

.....

Using software timer:

With software wait, the waiting time relies on the clock frequency of the microcontroller and the number of execution cycles. In other words, we should calculate accurately how many execution cycles needed in order to attain a given waiting duration.

In implementation, software wait may be a simple loop in C (like the example above) or assembly code (see the next page). It is important to point out that one iteration in C code is normally not equivalent to one execution cycle of the microcontroller. It depends on the microcontroller and on how compiler compiles C code to assembly instructions.

Hence, when applying software wait technique, it is advisable to write code in assembly language.

Software Timer (2)

How to generate software timer?

Example: Generate a timer of 1 ms:

F = *32 MHz* -> In 1s, 32 * 10^6 cycles are executed.

➔ To wait 1ms (10^{-3} s), we need:
*(10^{-3}) * (32 * 10^6) ≈ 32000 cycles*

Formula:

To generate the timer of N (s), at clock f = F (Hz). The number of cycles C we need is:

$$C = N * F$$

Software wait example

- Waiting for 1ms:

+ Total cycles inside L0: *8 (AX>0) or 6 (AX=0)*

+ Total cycles outside L0: *10*

+ Number of iterations of L0: *3999*

+ Total number of cycles in _Wait1Ms function:

10 + (8 * 3999-2) = 32000 cycles

```
_Wait1Ms:
    push hl
    push ax

    movw AX, #3999
    L0:
        SUBW AX, #1
        NOP
        NOP
        NOP
    BNZ $L0
    NOP
    NOP
    NOP
    NOP
    NOP
    pop ax
    pop hl
RET
```

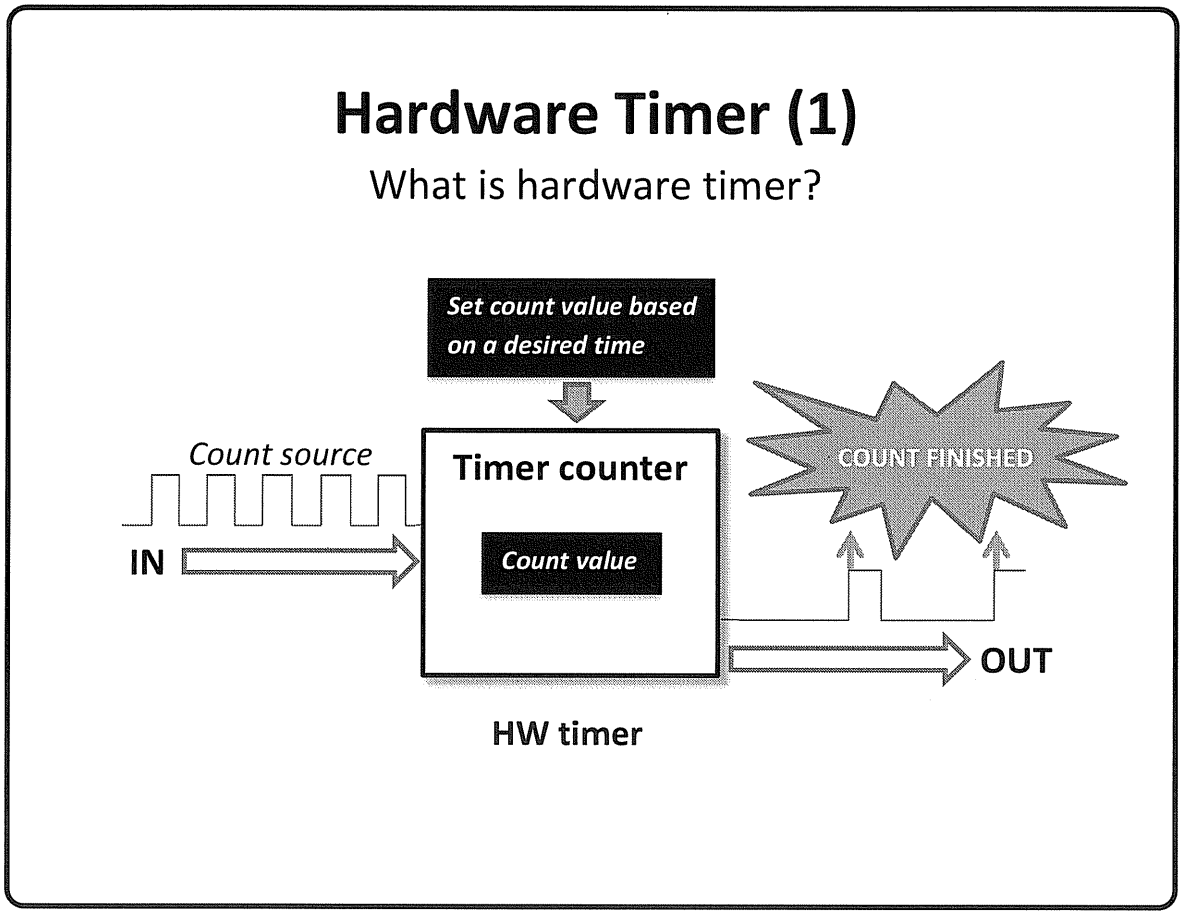
Number of Loops

8 cycles per Loop

4 cycles: Satisfied
2 cycles: Unsatisfied

Exercise

- How to generate a software wait with a waiting time of 1s? Write an example code for it.



Hardware timer

Generally, hardware timer is a circuit that can generate some signal when the desired time we set is reached.

Timer has a counter whose value can be set. As soon as the final condition regarding to count value is satisfied, timer will output a pulse or invert a specified bit (i.e., flag) to inform the desired duration has been achieved. This desired duration is determined by the frequency of count source and the value set in timer counter (count value). The count value will be decremented every time when counter receives 1 pulse.

Hardware Timer (2)

What is hardware timer?

A typical HW timer will consist of four parts:

+ Prescaler.
+ Counter register.
+ Capture registers.
+ Compare registers.

There will also be control and status registers to configure and monitor the timer.

Prescaler

The prescaler takes the basic timer clock frequency and divides it by some value before feeding it to the timer accordingly to how the prescaler register is configured. The purpose of the prescaler is to allow the timer to be clocked at the rate we desire. For shorter (8 and 16-bit) timers, there will often be a tradeoff between resolution and range.

Timer register

The timer register itself is typically an N-bit up-counter, with the ability to read and write the current count value, and to stop or reset the counter.

Capture register

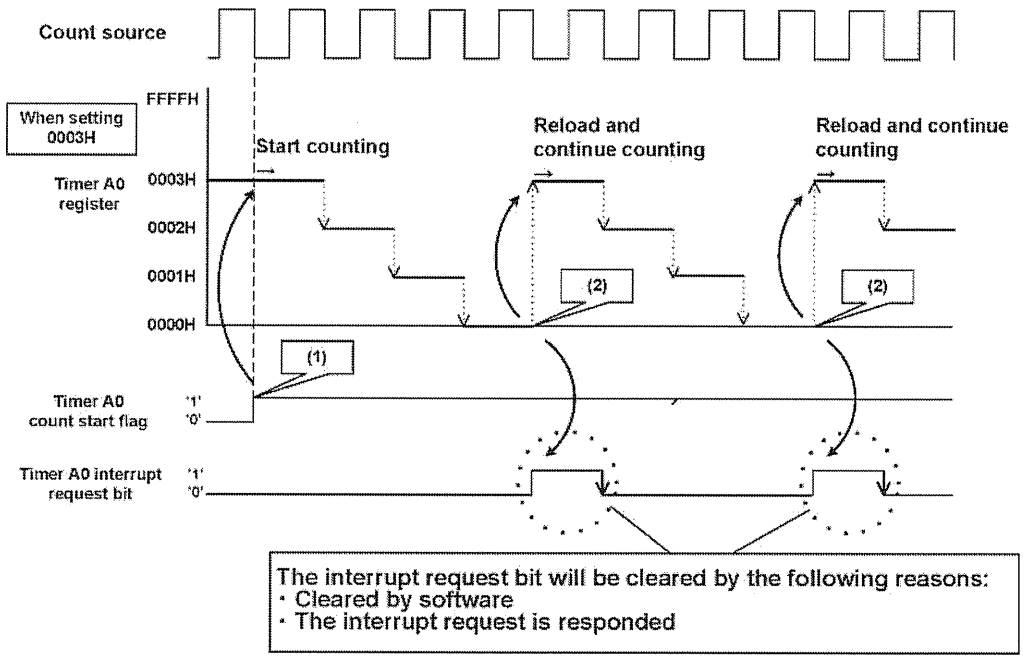
A capture register is a register which can be automatically loaded with the current count value upon the occurrence of some event, typically a transition on an input pin. Capture registers can be used, among other things, to time the intervals between pulses, to determine the high and low times of input signals, and to time the intervals between two different input signals.

Compare/Match register

Compare registers (sometimes also called match registers) hold a value against which the current timer value is automatically and continuously compared. A compare register is used to trigger an event when the value in the timer register matches the value in the compare register.

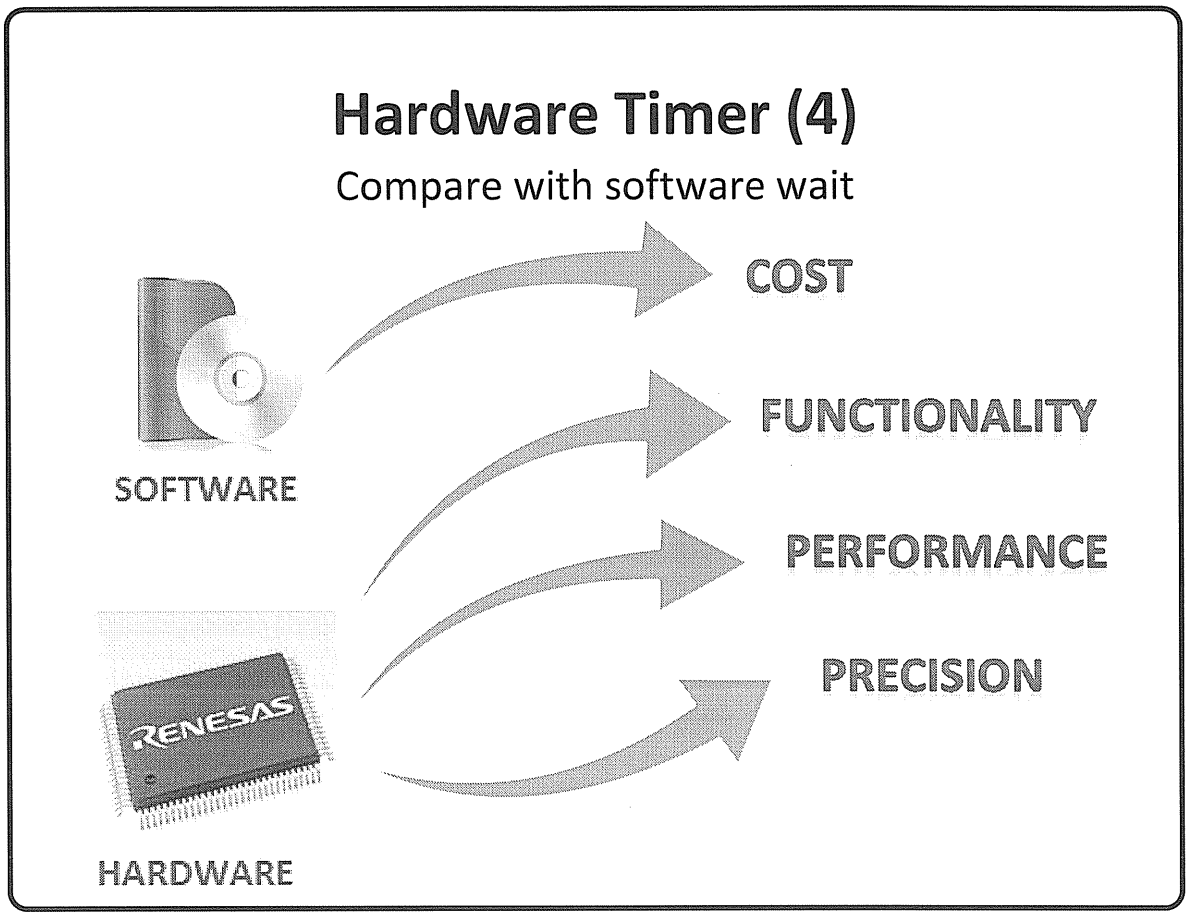
Hardware Timer (3)

Using hardware timer



Two styles of using hardware timer

- Polling for a timer value: User can use loop to read the value of timer counter register. By this way, the hardware timer register can be monitored.
- Using timer interrupt: When the desired time is over, it will trigger an interrupt timer. User can implement interrupt routine for any purpose.



Cost

- With software wait, we do not need any additional circuit to generate timer.
- User need to investigate and understand HW Manual if they want to use HW Timer.

Functionality

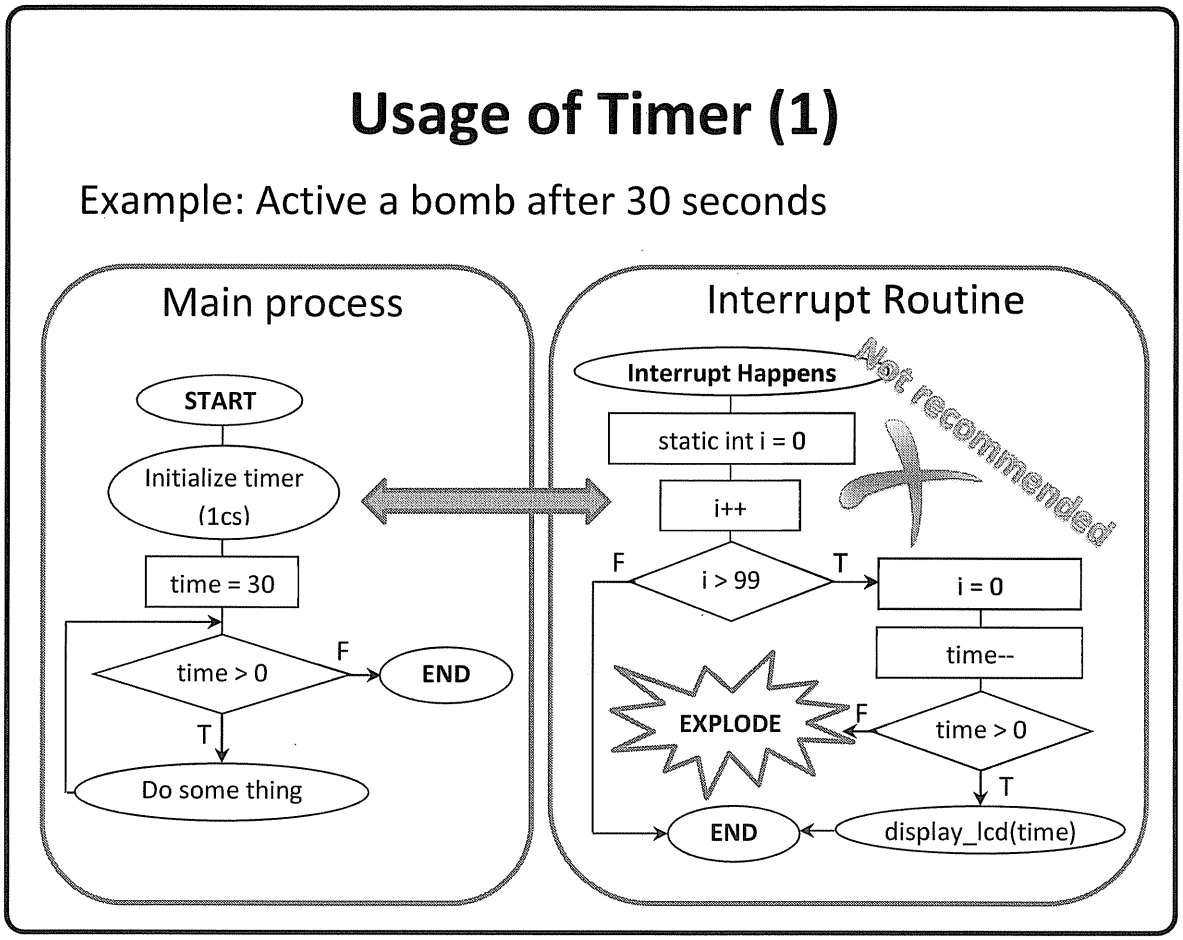
- Software wait and main process are dependent on each other. This imposes some limitation in terms of functionality to software wait.
- Software wait cannot be used in multitask, multithread processing environment.

Performance

- Software wait requires main process to be suspended, and wait for time to be over.
- Hardware timer works in parallel with main process, and does not require main process to be suspended.

Precision

- With software wait, user has to calculate carefully to generate the time (or estimate the number of execution cycles). On the contrary, with hardware timer, user can easily calculate and design the timer without the worry about precision.



Process after certain time

This example illustrates the idea of a program which performs tasks after a desired time:

- A variable "time" is declared holding the spoken desired time.
- After each every second, "time" is decremented and its value is displayed on a LCD device.
- As soon as time is up (30 seconds in this example), a bomb will be activated.

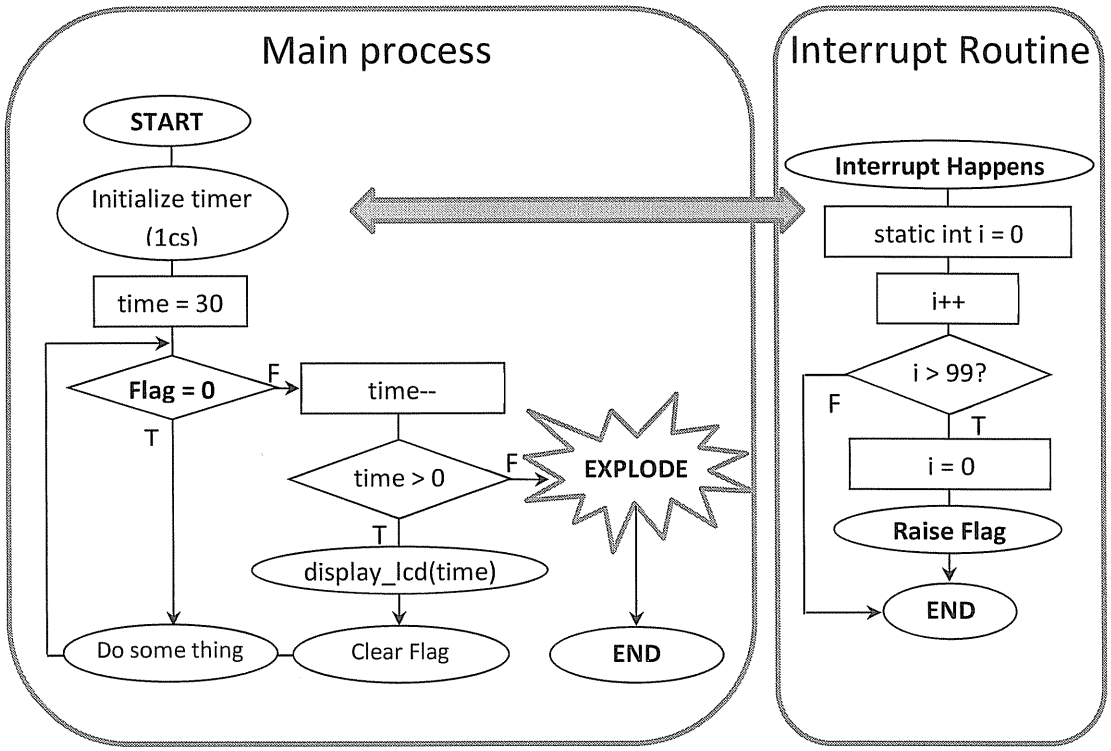
The 1 second elapsed time in the program can be precisely determined by using interrupt timer which raises an interrupt signal after each every 1 centi-second.

Recommendation

In the right diagram, we make use of interrupt routine to display some information (e.g. the current time) on a LCD, and to active a bomb. There is a risk that these actions will not be performed at an expected time if they are implemented inside interrupt routine and their processing times are long. A recommendation for using interrupt routine is described in the next page.

Usage of Timer (2)

Example: Active a bomb after 30 seconds



Explanation

As we initialize the timer with 1 centi-second, interrupt routine will be executed after each every 1 centi-second has elapsed, and it will increase the value of its static local variable i. When 1-second passed, interrupt routine will raise a flag to inform main process. With this mechanism, it only takes a very short time for interrupt routine to be processed, leading to a safer and more precise program.

Main process can check the flag. It will then process appropriate actions accordingly:

- If flag is set (raised), main process will calculate the new time and display it on to an LCD.
- If the time is over (0), main process will active the bomb and terminates the program.

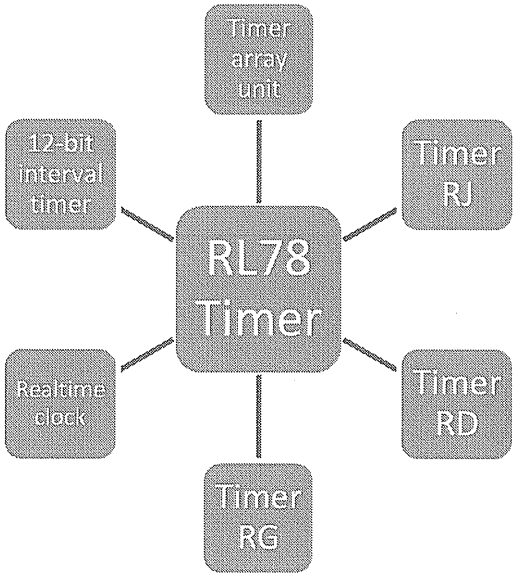
Time Management in RL78/G14

RL78/G14 board has many kinds of timer:

+ ***Real-time clock:*** Used to count the real-time such as year, month, week, day, hour, minute and second.

+ ***12-bit interval timer:*** Used to generate interrupt at any specified time interval.

+ ***Timer Array Unit, Timer RJ, RD, and RG:*** They have many functionalities and configurations. We can use it for different purposes.

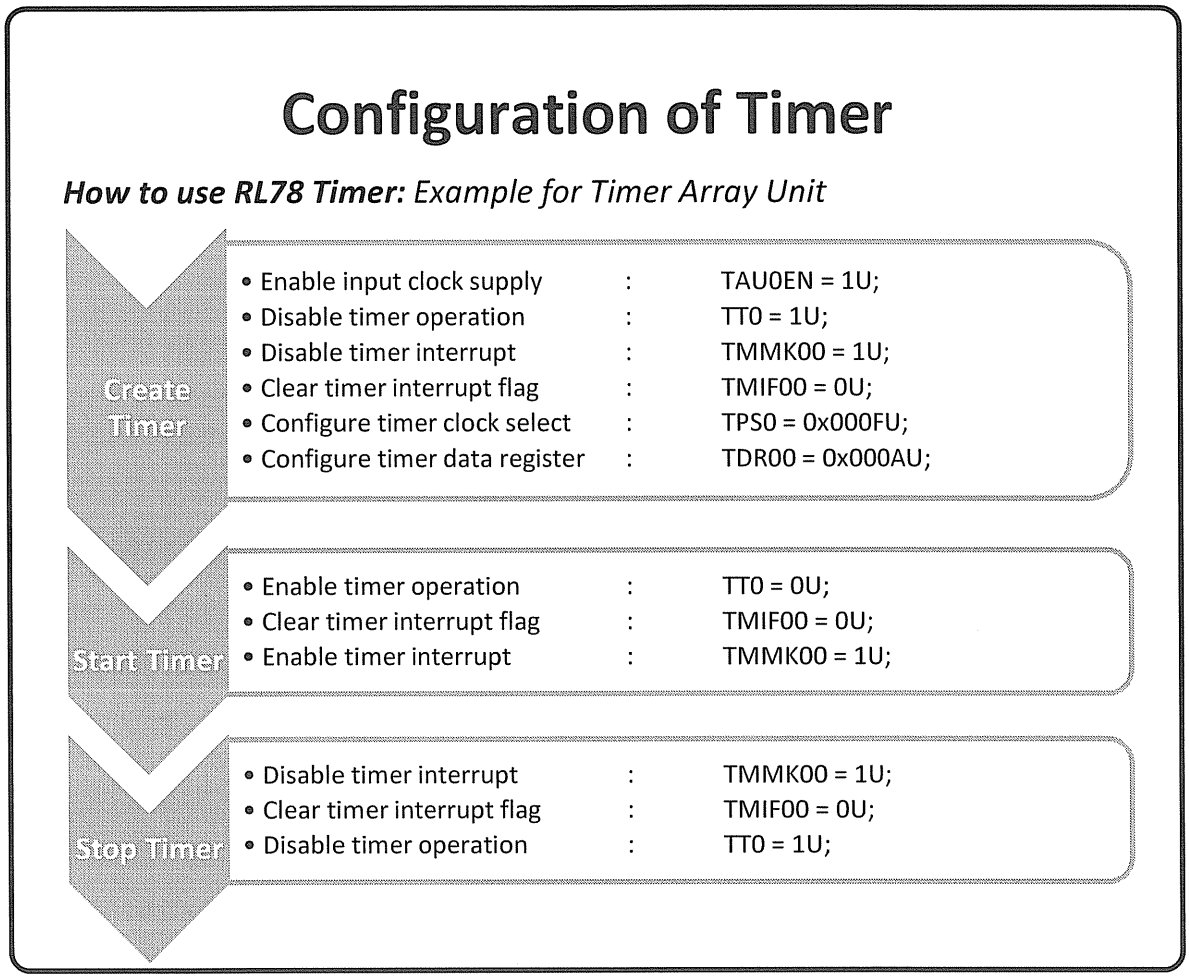


Timer configuration

Each timer has different configuration. For details, refer to RL78/G14 Hardware Manual, chapter 6 to 11.

Item	Configuration
Timer/counter	Timer count register mn (TCRmn)
Register	Timer data register mn (TDRmn)
Timer input	TI00 to TI03, TI10 to TI13 Note 1, RxD0 pin (for LIN-bus)
Timer output	TO00 to TO03, TO10 to TO13 pins Note 1, output controller
Control registers	<Registers of unit setting block> • Peripheral enable register 0 (PER0) • Timer clock select register m (TPSm) • Timer channel enable status register m (TEm) • Timer channel start register m (TSm) • Timer channel stop register m (TTm) • Timer input select register 0 (TIS0) • Timer output enable register m (TOEm) • Timer output register m (TOm) • Timer output level register m (TOLm) • Timer output mode register m (TOMm)
	<Registers of each channel> • Timer mode register mn (TMRmn) • Timer status register mn (TSRmn) • Input switch control register (ISC) • Noise filter enable registers 1, 2 (NFEN1, NFEN2) • Port mode register (PMxx) Note 2 • Port register (Pxx) Note 2

Example: Timer Array Unit Configuration



Explanation

Typically, the use of a timer involves three steps: Create it, start it and stop it. In the first step, creating timer, we have to enable the input clock supply, clear the timer interrupt flag so that when interrupt happens, this flag will be set to "1"...

The configurations above are not exhaustive. We should refer to the use of a specific timer in the hardware manual to get more information on how to configure it and use it.

(This page intentionally left blank)

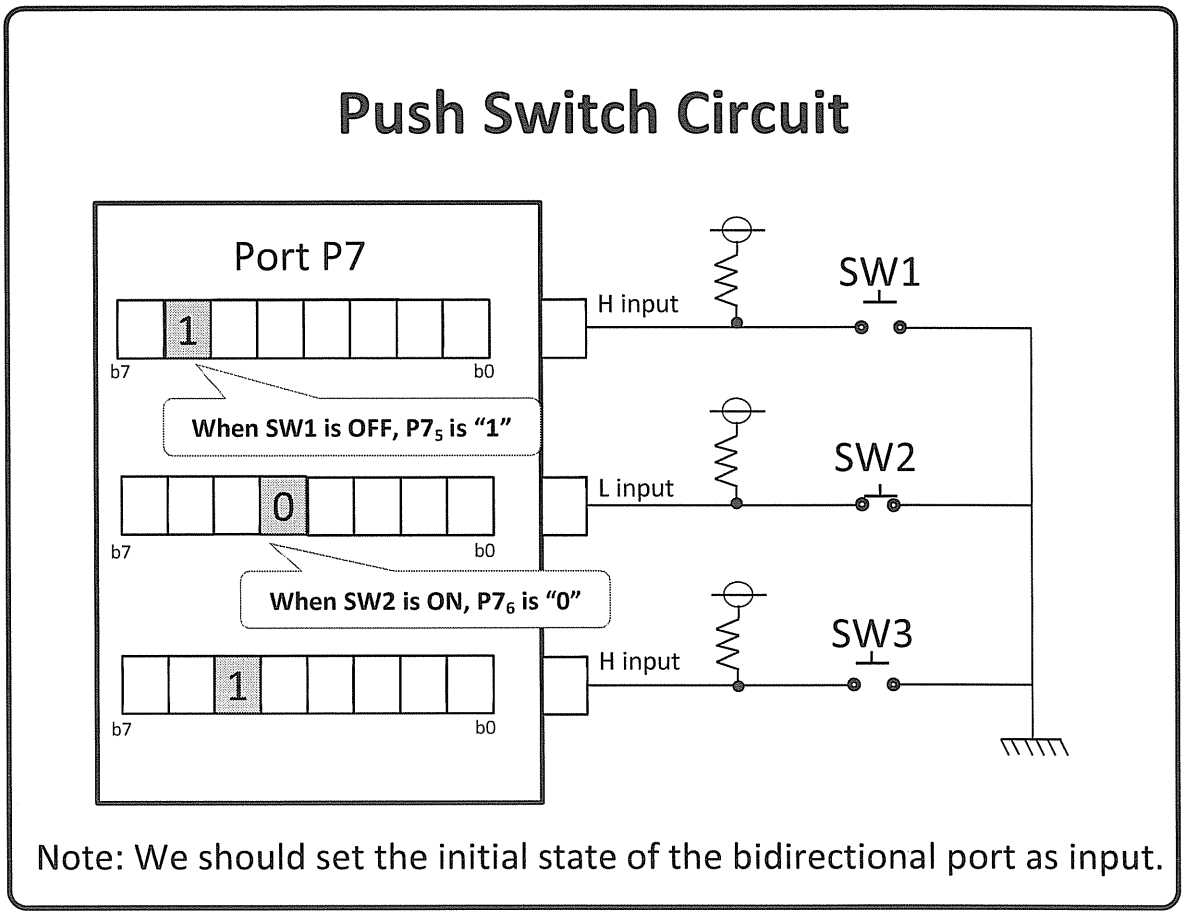
(This page intentionally left blank)

Chapter 7.

I/O Control

In this chapter:

- Key Input Process (Push Switch)
- Chattering Elimination (Push Switch)
- LCD Display
- LCD Command List
- Glyph APIs



Example of push switch circuit

- **Bit6 pin of port P7**

  This pin is an input port for SW1. The level of this port pin is regarded as "L" when SW1 is in ON state and "H" when SW1 is in OFF state.

- **Bit4 pin of port P7**

  This pin is an input port for SW2. Just like SW1, ON is "L" and OFF is "H".

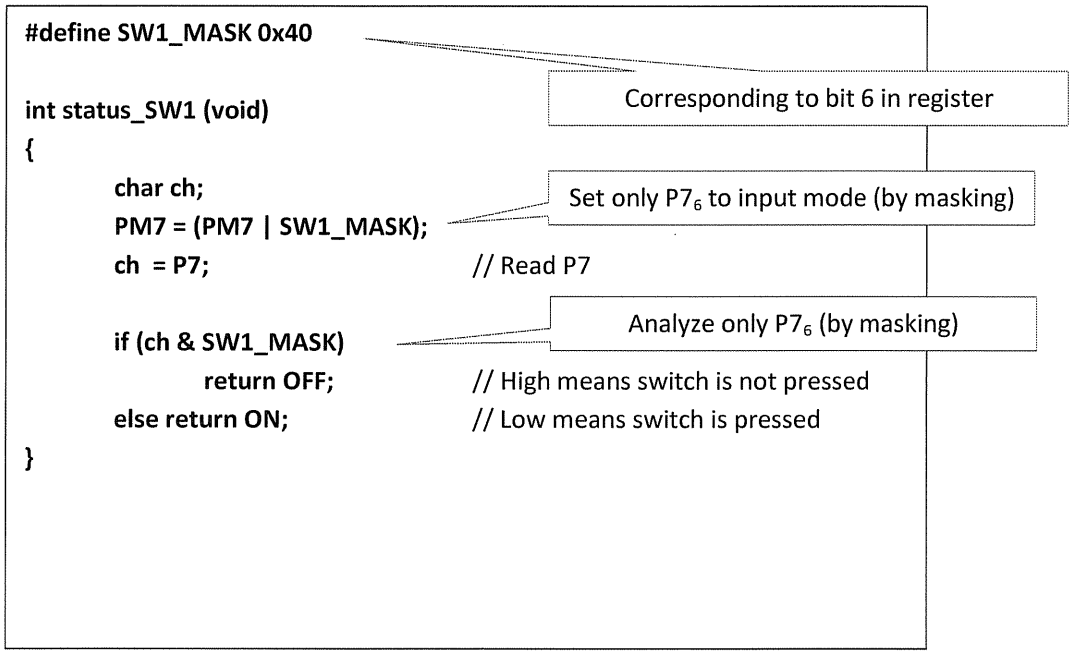
- **Bit5 pin of port P7**

  This pin is an input port for SW3. Just like SW1, ON is "L" and OFF is "H".

Pull-up resistor

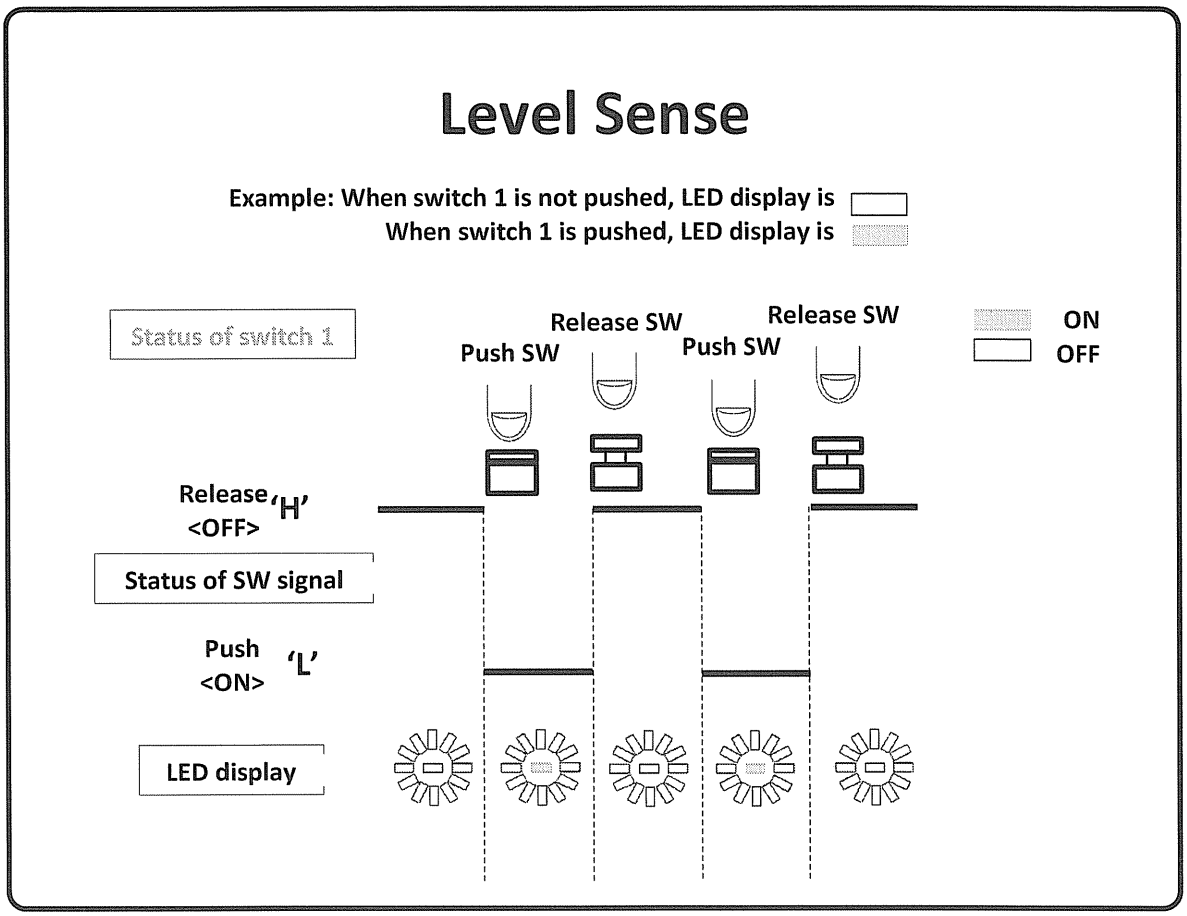
The resistor, that is connected to a power supply and limits the current, is called "pull-up resistor".

Process Key Input Data



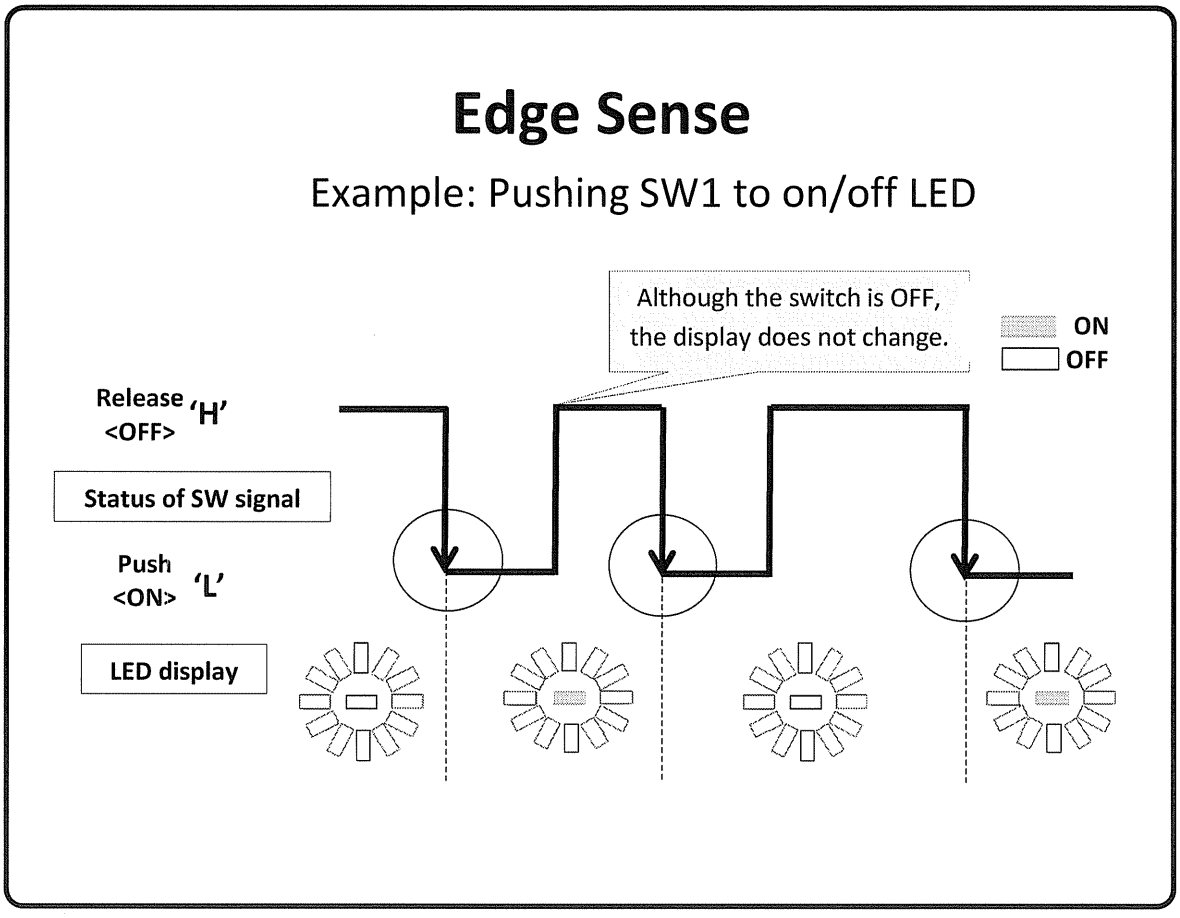
Masking

Masking means using a bit mask in bitwise operation to read or change specific bit(s). This process ensures other unrelated bits are not affected.



Level sense

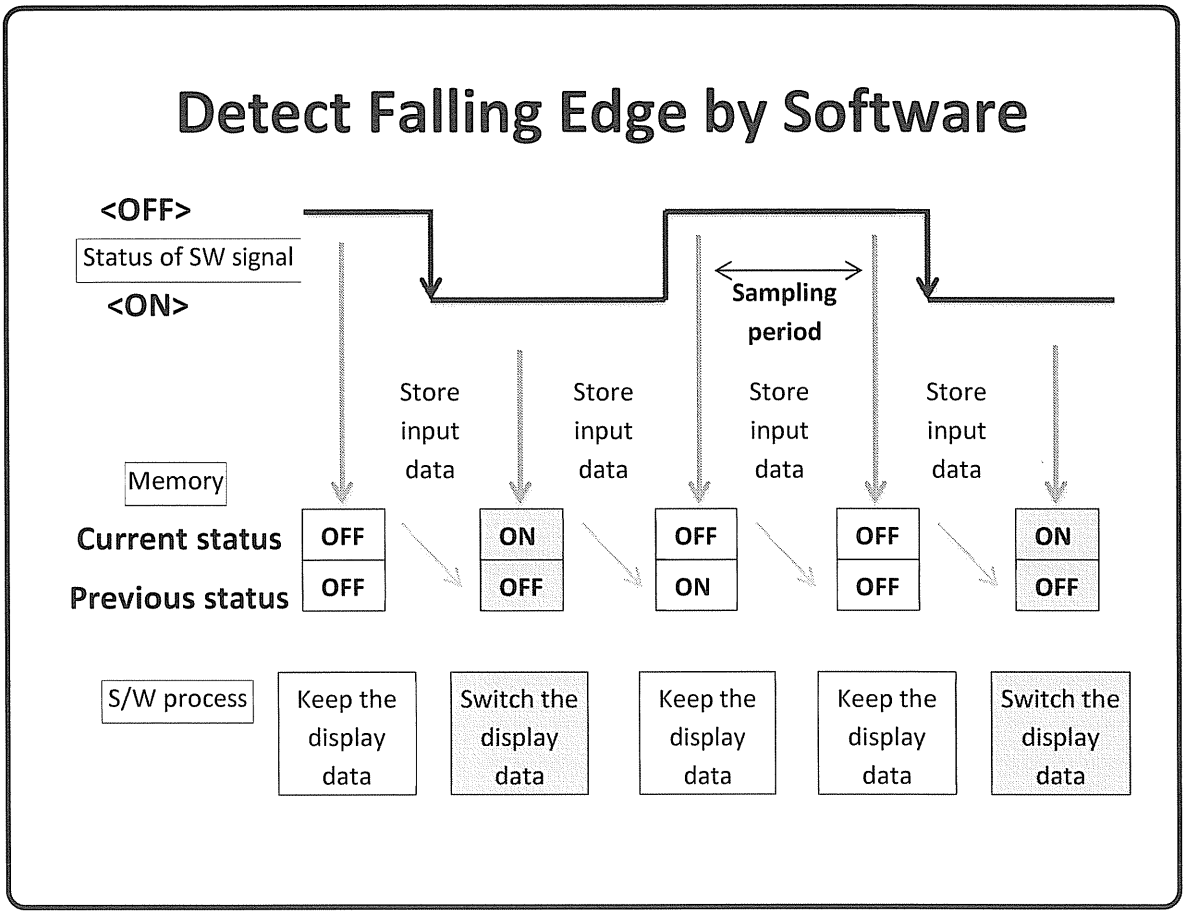
Judging the level of input data to be "1" (H) or "0" (L) is called "level sense".



Edge sense

This is the method to detect the state change of input signal by comparing the previous input data with the current one. Possible state change is from "H" to "L" or from "L" to "H". To perform edge sense, we need to store the previous state and compare it with the current state.

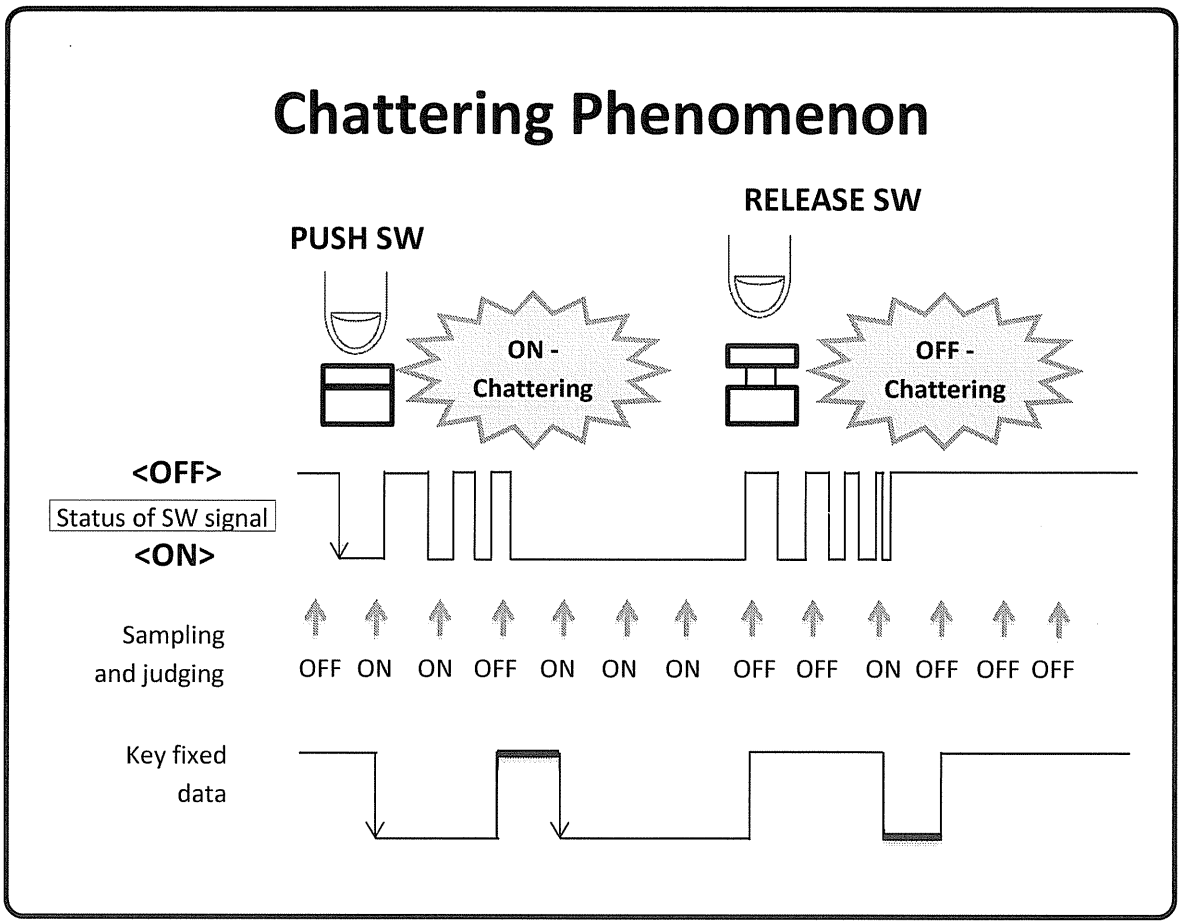
The change from "H" to "L" is called "negative edge (falling edge)" and from "L" to "H" is called "positive edge (rising edge)".

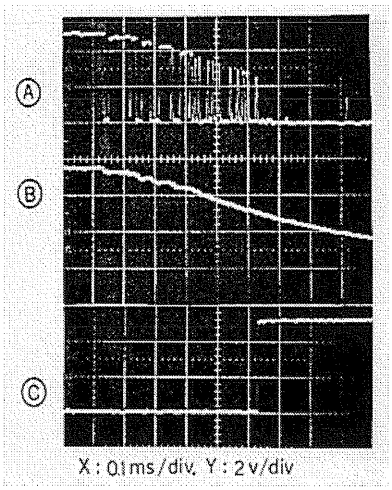


How to detect falling edge by software

The above example shows that in order to detect falling edge, the previous state of the switch is stored first, and the judgment is done by comparing it with current input data. Then, current input data, instead of the previous one, is saved for the next comparison and so on.

Hence, to determine the change in the switch's state, the value of the port needs to be read periodically. This technique is called "polling".





Removing chattering signal with HW

A: Chattering signal

B: Signal filtered by integrated circuit

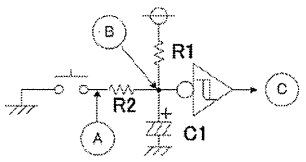
C: Signal modulated by Schmitt trigger IC

Chattering phenomenon

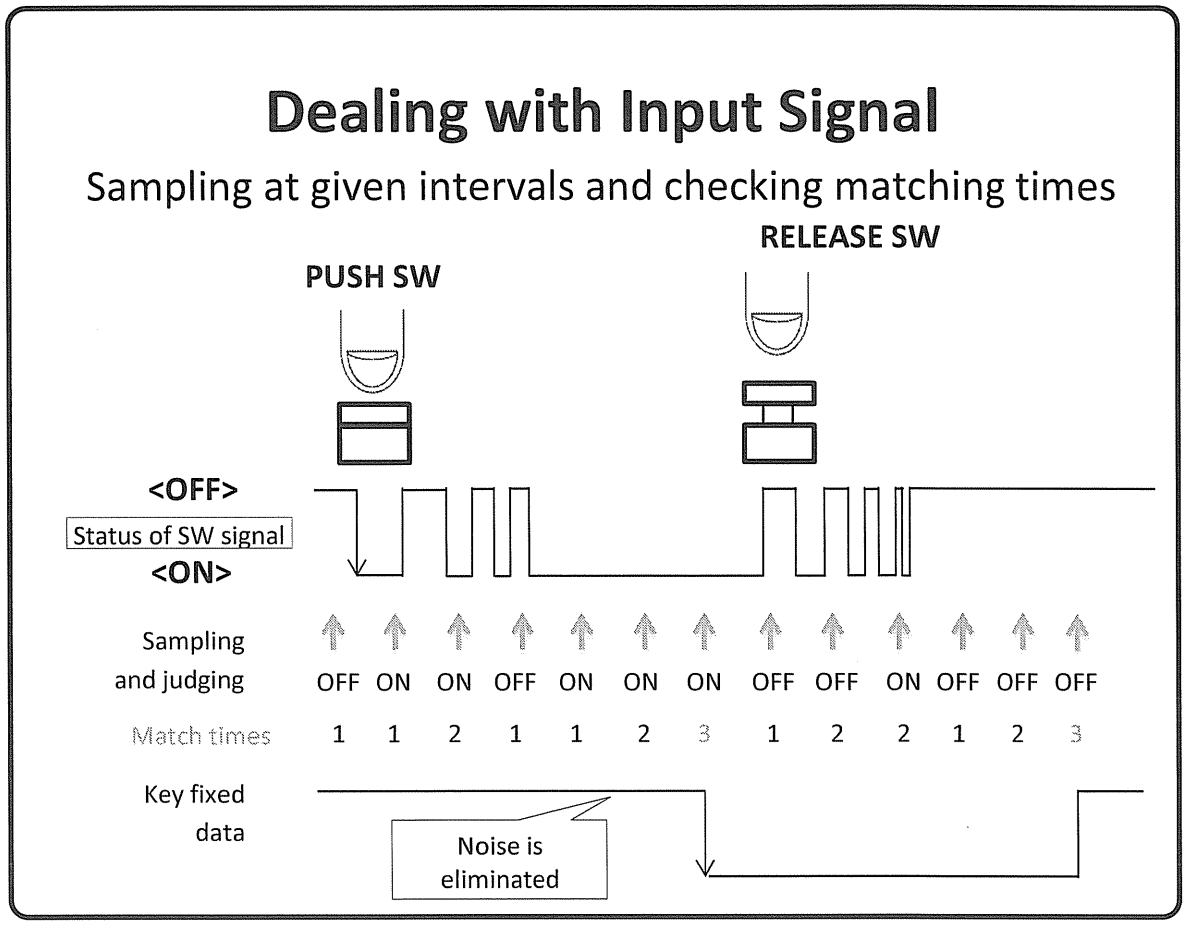
When switching SW from OFF to ON or from ON to OFF, the vibration at SW connecting point occurs. This is called "chattering phenomenon". This vibration would probably last more than 10ms for some types of SW. If we change the switch's state during this time, one switching operation may be treated as many.

Removing chattering

A circuit can be implemented to convert a short signal into gradual change to eliminate chattering. Generally, the combination of Schmitt trigger IC and integrated circuit is used (see figure below).



We also can remove chattering by software way. This technique consists of sampling input data at regular intervals and checking multiple-matching data.



Treating input signal

In order to prevent malfunctioning due to, for instance, noise, we often sample input data at regular intervals, check matching times, and fix the input data if and only if it is matched for a few times.

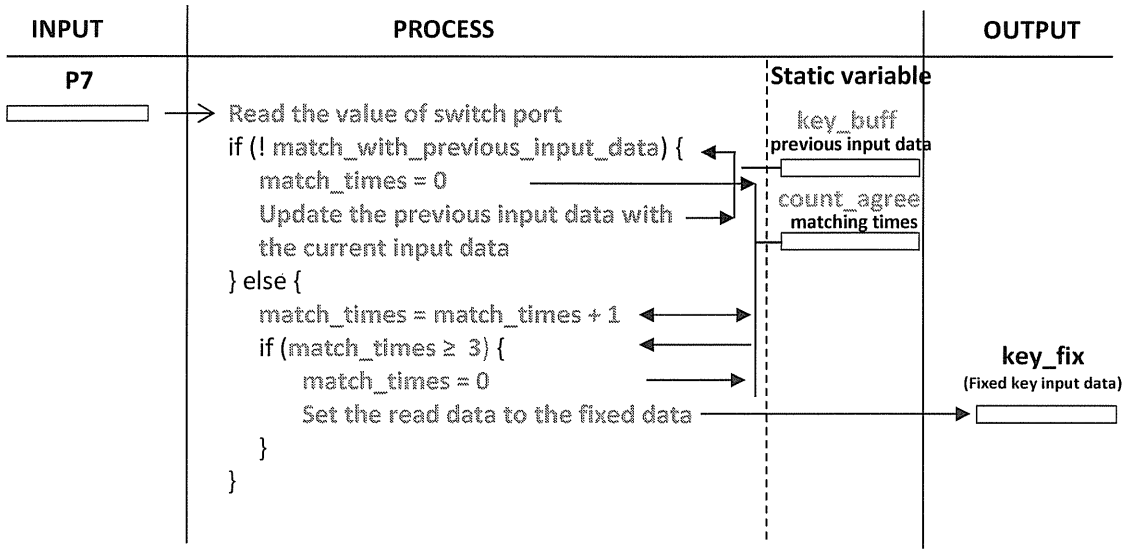
Sampling period and the value of matching times are determined in accordance with device characteristics and response features required by program specifications.

Matching times

It is the number of appearances of the same state (ON or OFF) detected consecutively.

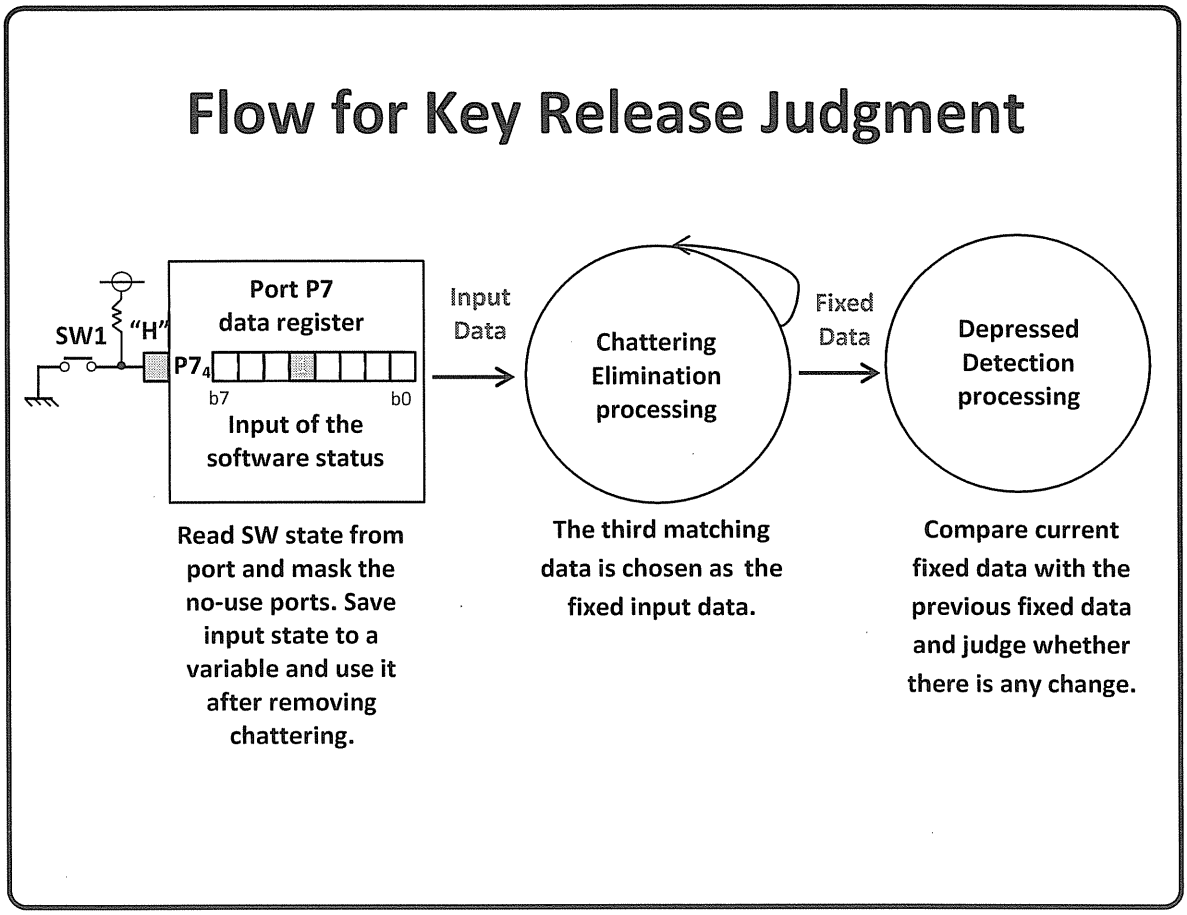
Algorithm to Eliminate Chattering

Multiple-matching method for fixing input data
(three-time agreement)



Three-time agreement technique

If the input data is matched three times, this data is fixed and the previous input data is replaced with this fixed data. If the current data does not agree with the previous one, reset the match-times counter ("0"), and restart reading. This method is often used to read the data from external devices such as sensors. It is one of the effective programming techniques used to reduce the effect of noise, chattering, etc...



Key lockout method

Only accept the input of the first key after all keys are released, and refuse to accept the subsequent keys' inputs. That is to say, only the first key's input is valid. The disadvantage of this this technique is slow because key input can be accepted if and only if all the other keys have already been released.

N-Key rollover method

Memorize the first key's input, and accept new valid key's inputs (if two keys are pushed at the same time, both are valid). As an example, two-key rollover method indicates that up to two key's inputs at the same time are acceptable, but three or more keys' inputs at the same time are not acceptable. The keyboard of a personal computer is an example of using N-key rollover method.

Introduction to LCD Device (RL78/G14)

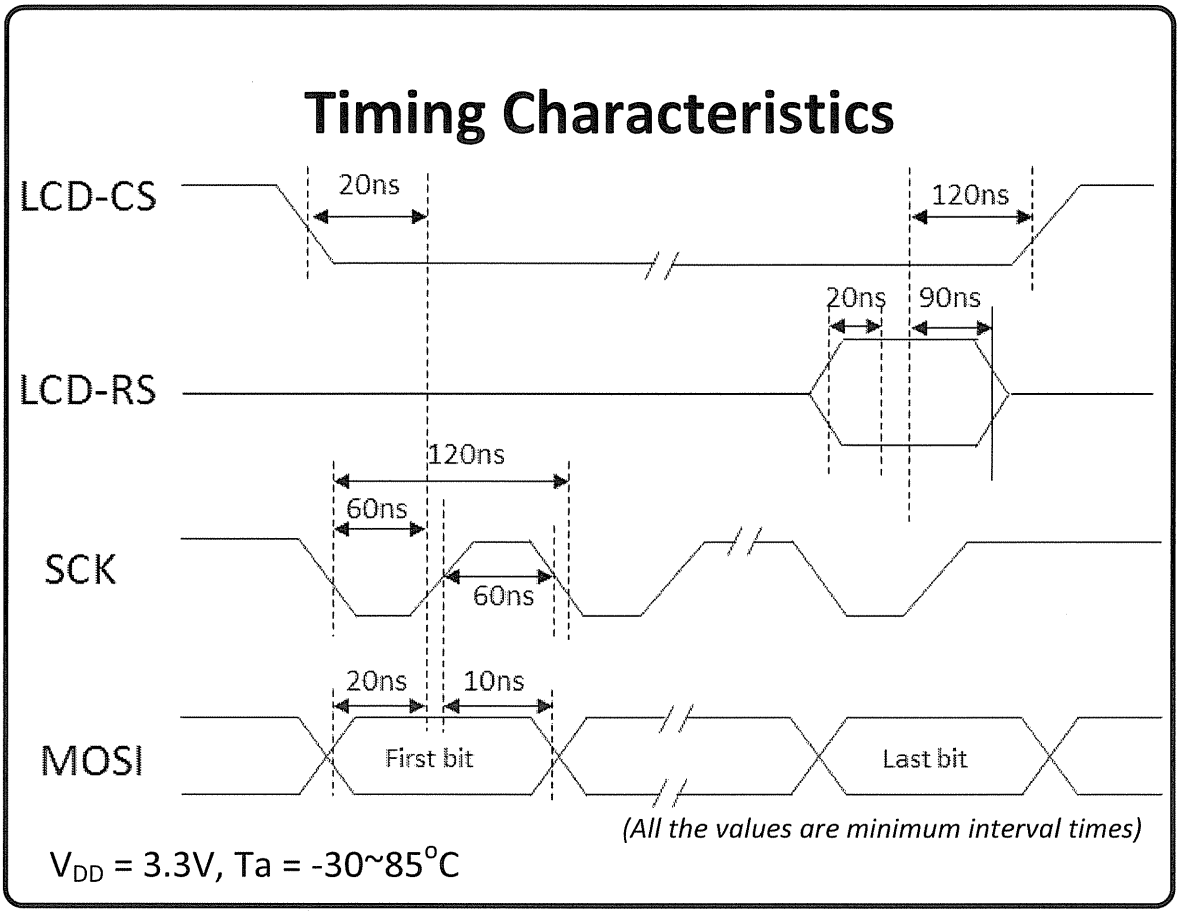
Features	Standard Value
Display type	96 x 64 dots
Logic supply voltage	3.3 V
Viewing direction	6 o'clock
Interface	8-Line interface
Backlight	White LED
Driver condition	LCD Module: 1/68 Duty, 1/9 Bias

Note

- White LED backlight is ON by default.
- It can be toggled OFF by setting P00 (BL-ENA, pin 97) LOW.

Interface Pin Description

PIN	Circuit net name	Function	RL78 Port
1	+5V Backlight Positive Anode	-	-
2	GND	-	-
3	GND	-	-
4	GND	-	-
5	GND	-	-
6	LCD-CS	Chip select signal. Active "L".	**P145 (pin 98)**
7	RSTOUT#	Reset input pin. When RESB is "L", internal initialization is executed.	**P130 (pin91)**
8	LCD-RS	It determines whether the access is related to data or command. RS = "H": Indicates that D [7:0] is displaying data. RS = "L": Indicates that D [7:0] is controlling data. RS is not used in 3-line SPI interface and should fix to "H" by VDD.	**P146 (pin 73)**
9	+3.3V	-	-
10	+3.3V	-	-
11	SCK	When using serial interface: D0=SCLK: Serial clock input. When CSB is non-active (CSB="H"), D [7:0] pins are high impedance.	**P70 (pin 40)**
12	MOSI		**P72 (pin 38)**
13	MOSI		**P72 (pin 38)**
14	MOSI		**P72 (pin 38)**
15	+3.3V		-
16	+3.3V		-
17	+3.3V		-
18	+3.3V		-
19	GND	-	-
20	+3.3V	-	-



Writing timing of LCD

When writing from CPU to LCD module, it is necessary to control the port output program of microcomputer, so that the control of each signal can satisfy the timing condition (described in figure above).

Note: Timing characteristics are provided in the datasheet of the LCD module.

Sample Code (Pseudo)

This sample demonstrates how to keep the timing characteristics of LCD module

```
// Set P70, P72, P145, P146 to OUTPUT (P130 is already fixed as output port)
PM7.bit0 = PM7.bit2 = PM14.bit5 = PM14.bit6 = 0

// Do not output clock SCK21 to P70 (SCK), control the clock manually to assure the timing
 ST0.bit = 1   //Stop auto serial transfer (do not use SOE0.bit1 as this register is READ ONLY)

SCK = 1          // P7.bit0
LCD_CS = 0       // P14.bit5
LCD_RS = x       // P14.bit6

SCK = 0          MOSI = x        Wait till 60ns    SCK = 1          Wait till 60ns
...
SCK = 0          MOSI = x        Wait till 60ns    SCK = 1          Wait till 60ns

LCD  CS = 1      // P14.bit5
```

LCD Command List (Not Exhaustive)

Instruction	LCD-RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0	Description
H[1:0] Independent instructions (Common functions)											
NOP	0	0	0	0	0	0	0	0	0	0	No operation
Function set	0	0	0	0	1	MX	MY	PD	H1	H0	Set PD (Power down) and H[1:0] (select instruction table)
Read status	0	1	PD	0	0	D	E	MX	MY	DO	Read status bits
Read data	1	1	D7	D6	D5	D4	D3	D2	D1	D0	Read display data
Write data	1	0	D7	D6	D5	D4	D3	D2	D1	D0	Write display data
H[1:0] = (0,0) (LCD_FUNCTION_ZERO)											
Set Y address of RAM (page)	0	0	0	1	0	0	Y3	Y2	Y1	Y0	Set Y address $0 \le Y \le 9$
Set X address of RAM	0	0	1	X6	X5	X4	X3	X2	X1	X0	Set X address $0 \le X \le 101$
H[1:0] = (1,1) (LCD_FUNCTION_THREE)											
RESET	0	0	0	0	0	0	0	0	1	1	Software reset
Frame control	0	0	0	0	0	0	1	FR2	FR1	FR0	Frame-rate control

Note

- Refer to the appendix at the end of this chapter for a full list of commands.
- R/W bit is not supported in current mode, so Read status and Read data commands are unusable.
- The command list is divided into different function sets which can be selected by two bits H [1:0].
- To use a function in another function sets, set H[1:0] to a corresponding set first (by Function set command).

LCD APIs (1)

* **Function Name:**	GlyphGetXY	
* **Description:**	Get the value of X and Y positions in a given Glyph Handle. The filled values are the current top left position of the next character to be placed onto the screen.	
* **Argument:**	aHandle	- Glyph handle setup by the LCD and communications.
	aX	- X position to obtain.
	aY	- Y position to obtain.
* **Return Value:**	success = 0, error != 0	
* **Function Name:**	GlyphSetXY	
* **Description:**	Set the value of X and Y positions in a given Glyph Handle. This position is used as the top left position of the next Glyph on LCD screen.	
* **Argument:**	aHandle	- Glyph handle setup by the LCD and communications.
	aX	- X position.
	aY	- Y position.
* **Return Value:**	success = 0, error != 0	

Using LCD APIs

LCD command is complex, and directly using it to display something on LCD is not recommended. To reduce the complexity and to save time, LCD vendor normally provides a set of APIs for developers to use to control LCD module. In our case (RL78/G14), it is Glyph APIs.

LCD APIs (2)

* **Function Name:**	GlyphRead
* **Description:**	Read from Glyph data storage using a Glyph Register value.
* **Argument:**	aHandle - handle created and returned. aAddress - may be used for communications.
* **Return Value:**	success = 0, error != 0
* **Function Name:**	GlyphWrite
* **Description:**	Write to the LCD using a Glyph Register value, to Glyph data storage or run an LCD Glyph Command.
* **Argument:**	aHandle - handle created and returned. aAddress - may be used for communications.
* **Return Value:**	success = 0, error != 0
* **Function Name:**	GlyphGetVersionInfo
* **Description:**	Fills the given T_glyphVersionInfo with version data.
* **Argument:**	aHandle - Glyph handle setup by the LCD and communications. aInfo - version info structure to fill.
* **Return Value:**	success =0 , error != 0

LCD APIs (3)

* **Function Name:**	GlyphSetFont
* **Description:**	Set a selected font value from the font enumeration to be used as the next font for the next character placed onto the screen. This font will stay selected for every character until we decide to change it. The font must have been previously turned on before compiling time. Fonts can be turned on and off and then recompiled using the following file: GlyphUseFont.h, which is to be stored inside our project root directory.
* **Argument:**	aHandle - Glyph handle setup by the LCD and communications. aFont - A value from the Font Enumeration.
* **Return Value:**	success = 0, error != 0
* **Function Name:**	GlyphGetFont
* **Description:**	Retrieve the enumeration value for the currently selected font.
* **Argument:**	aHandle - Glyph handle setup by the LCD and communications. aFont - A value retrieved from the font enumeration.
* **Return Value:**	success = 0, error != 0

LCD APIs (4)

*** Function Name:** GlyphSetDrawMode

*** Description :** The draw mode function. Cause a draw procedure to happen on LCD display. The procedure is chosen by using the T_glyphDrawMode enumeration in the Glyph.h file.

Operations are described as:

GLYPH_CMD_NOP	-- Do Nothing.
GLYPH_CMD_SCREEN_CLEAR	-- Clear the screen to blank.
GLYPH_CMD_SCREEN_INVERT	-- Invert the entire screen. Black becomes white.
GLYPH_CMD_SCREEN_REGULAR	-- Used on startup to create a normal screen.
GLYPH_CMD_SCREEN_SLEEP	-- Put the LCD in a low power blank screen.
GLYPH_CMD_SCREEN_WAKE	-- Restore the LCD from low power sleep.
GLYPH_CMD_TEST_PATTERN	-- Display a test pattern.
GLYPH_CMD_DRAW_BLOCK	-- Fill X to X2 and Y to Y2.
GLYPH_CMD_ERASE_BLOCK	-- Erase X to X2 and Y to Y2.

*** Argument:** aHandle - Glyph handle setup by the LCD and communications.

aMode - A value from the draw mode enumeration.

*** Return Value:** success = 0, error != 0

LCD APIs (5)

* **Function Name:**	GlyphChar
* **Description:**	Draw a character in the currently chosen font at the current position (X,Y).
* **Argument:**	aHandle - Glyph handle setup by the LCD and communications. aChar - A given character.
* **Return Value:**	success = 0, error != 0
* **Function Name:**	GlyphString
* **Description:**	Draw a string in the currently chosen font at the current position X and Y. Each character of the string is set from the start of the last character by the number of given aSpacing. So, if the font is 8 pixels wide, a good number to use here might be 10 pixels.
* **Argument:**	aHandle - Glyph handle setup by the LCD and communications. aString - A given character string. aLength - The number of characters in the string to be displayed on LCD.
* **Return Value:**	success = 0, error != 0

LCD APIs (6)

*** Function Name:**	GlyphDrawBlock
*** Description:**	Draw a set of dark pixels from (X,Y) position to (X2,Y2) position
*** Argument:**	aHandle - Glyph handle setup by the LCD and communications. aX1 - X position. aY1 - Y position. aX2 - X2 position. aY2 - Y2 position.
*** Return Value:**	success = 0, error != 0

*** Function Name:**	GlyphEraseBlock
*** Description:**	Clear to a set of white pixels from (X,Y) position to (X2,Y2) position
*** Argument:**	aHandle - Glyph handle setup by the LCD and communications. aX1 - X position. aY1 - Y position. aX2 - X2 position. aY2 - Y2 position.
*** Return Value:**	success = 0, error != 0

LCD APIs (7)

* **Function Name:**	GlyphSetContrast
* **Description:**	Set the contrast of the LCD display. Any positive number is acceptable. If the number is too large, the contrast will not change. Refer to the documentation of the LCD module for details.
* **Argument:**	aHandle - Glyph handle setup by the LCD and communications. nContrast - A contrast setting value.
* **Return Value:**	success = 0, error != 0
* **Calling Functions:**	main

* **Function Name:**	GlyphSetContrastBoost
* **Description:**	Set the boost for the contrast of the LCD Display. Any positive number is acceptable. If the number is too large, the contrast boost will not change. Refer to the documentation of the LCD module for details.
* **Argument:**	aHandle - Glyph handle setup by the LCD and communications. cContrastBoost - A contrast booster setting value.
* **Return Value:**	success = 0, error != 0

LCD APIs (8)

* **Function Name:**	GlyphClearScreen
* **Description:**	Clear the LCD screen to white.
* **Function Name:**	GlyphInvertScreen
* **Description:**	Invert every pixel on the screen from white to dark and dark to white.
* **Function Name:**	GlyphNormalScreen
* **Description:**	Set up the normal LCD screen right after initialization startup.
* **Function Name:**	GlyphSleep
* **Description:**	Put the LCD in a low power blank screen.
* **Function Name:**	GlyphWake
* **Description:**	Take the LCD out of low power sleep mode and redraws the current message.
* **Argument:**	aHandle - Glyph handle setup by the LCD and communications.
* **Return Value:**	success = 0 , error != 0

Appendix: Full LCD command list

Instruction	LCD-RS	R/W	D7	D6	D5	D4	D3	D2	D1	D0	Description
H[1:0] Independent instructions (Common functions)											
NOP	0	0	0	0	0	0	0	0	0	0	No operation
Reserved	0	0	0	0	0	0	0	0	0	1	Do not use
Function set	0	0	0	0	1	MX	MY	PD	H1	H0	Set PD (Power down) and H[1:0] (select instruction table)
Read status	0	1	PD	0	0	D	E	MX	MY	DO	Read status bits
Read data	1	1	D7	D6	D5	D4	D3	D2	D1	D0	Read display data
Write data	1	0	D7	D6	D5	D4	D3	D2	D1	D0	Write display data
H[1:0] = (0,0) (LCD_FUNCTION_ZERO)											
Reserved	0	0	0	0	0	0	0	0	1	x	Do not use
Set VO Range	0	0	0	0	0	0	0	1	0	PRS	VO range L/H select
END	0	0	0	0	0	0	0	1	1	0	Release read/modify/write
Read-modify-Write	0	0	0	0	0	0	0	1	1	1	RAM address at R:+0, W:+1
Display Control	0	0	0	0	0	0	1	D	0	E	Set display configuration
Reserved	0	0	0	0	0	1	0	0	x	x	Do not use
Set Y address of RAM (page)	0	0	0	1	0	0	Y3	Y2	Y1	Y0	Set Y address $0 \le Y \le 9$
Set X address of RAM	0	0	1	X6	X5	X4	X3	X2	X1	X0	Set X address $0 \le X \le 101$
H[1:0] = (0,1) (LCD_FUNCTION_ONE)											
Reserved	0	0	0	0	0	0	0	0	1	x	Do not use
Display Configuration	0	0	0	0	0	0	1	DO	x	x	Top/bottom row mode set data order
Bias system	0	0	0	0	0	1	0	BS2	BS1	BS0	Set bias system (BSx)
Set Start Line (high)	0	0	0	0	0	0	0	1	0	S6	Specify the initial display line S6
Set Start Line (low)	0	0	0	1	S5	S4	S3	S2	S1	S0	Specify the initial display line to realize vertical scrolling
Set VO	0	0	1	V_{OP6}	V_{OP5}	V_{OP4}	V_{OP3}	V_{OP2}	V_{OP1}	V_{OP0}	Set V_{OP} parameter to register

(This page intentionally left blank)

(This page intentionally left blank)

Chapter 8.

Interrupt

In this chapter:

- **Polling Process**
- **Interrupt**
- **Interrupt Enabled Condition and Mechanism**
- **Notes on Shared Memory and Reentrant**
- **Multiple Interrupts**

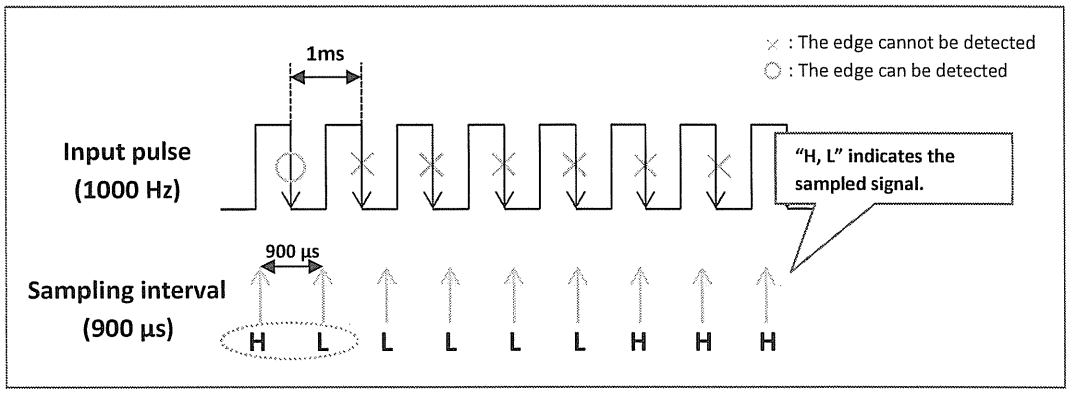
Timing of Polling Process

(Detect edge by software)

Example: Detect the falling edge of 1000 Hz input pulse by polling.

```
while (1)
{
    900 µs Wait
    Edge detection
}
```

If sampling the 1000 Hz input pulse at 900 µs intervals, detection error would occur.



Polling method

Polling method simply uses a code section which checks a particular flag or flags, to know the status of operations. Polling is like picking up our phone every few seconds to see if we have a call.

Period of polling

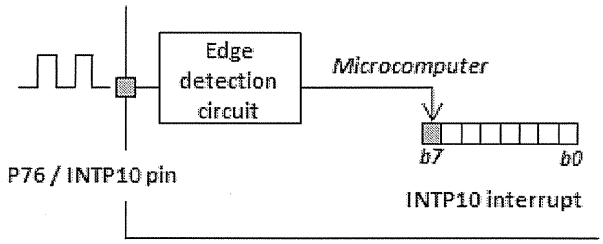
The above example shows that when we use periodic polling process to capture signal state change, the polling period must be shorter than the sum of "H" level time and "L" level time. Otherwise, we cannot capture all signal state changes. If polling interval is shorter than 500µs, we can detect all falling edges. But the CPU time to be allocated to the other process will become shorter.

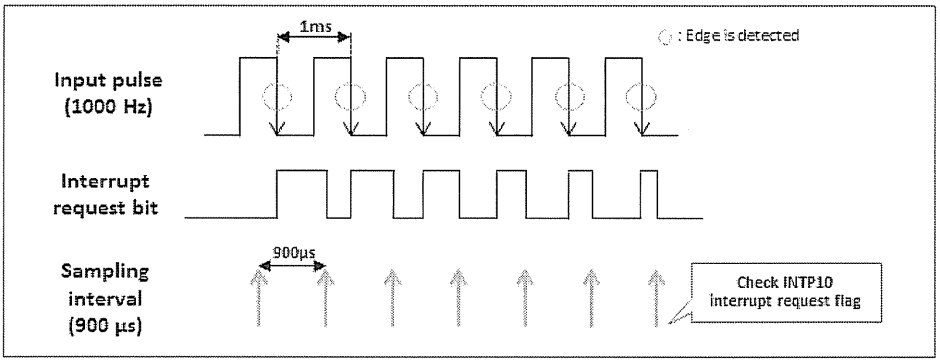
Polling by Interrupt Request Bit

(Detect edge by hardware)

Example: Detect by polling the interrupt request bit.

```
while (1)
{
    900 µs Wait
    if (interrupt request bit is set) {
        Process
    }
}
```





Polling process using interrupt request bit

As shown in the example in the previous page, we have to carefully select the sampling interval because this time follows the external status change. If this status change can be detected by hardware and saved in memory, it is fine to do the process before the next change comes. In the above example, microcomputer function is used to detect edge with hardware. Request bit is set when there is a status change. This example shows that polling process should be finished within 1 cycle of input pulse.

If INTP10 input is assigned to P76 pin, falling edge (negative edge) or rising edge (positive edge) can be chosen by setting edge selector.

If falling edge is selected, INTP10 interrupt request bit will be set to "1" when the change of P76 is from "1" (H) to "0" (L) (falling edge occurrence). This bit will be remained unless we use an instruction to clear it. Therefore, do not forget to clear it within polling process.

Advantages

- **Polling is a simple method to execute. It does not involve any priority and is easy to debug.**
- **The code segment will always execute within a fixed time and in a fixed sequence, has no effect on the execution of other sections of code.**
- **There is no issue with stack and big memory management.**

Good places for polling method

Polling can be used either in the early stages of development where the working of a peripheral is to be verified or in cases where the peripheral is armed or activated just before polling code executes, and is deactivated immediately after that. So the system expects events during very specific times.

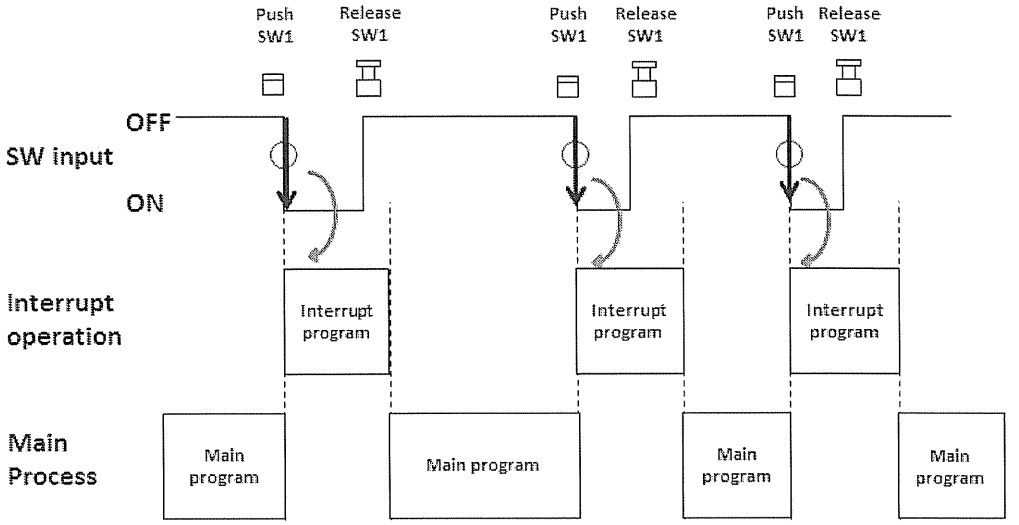
Disadvantages and Limitations

- There is a great chance of missing events. This is because polling code executes only in a particular order and does not execute in response to an event. This may cause two events to occur on succession before the polling code executes. In this case, the program would register only a single event.

- Polling uses the execution time of the microcontroller significantly. The same code segment has to be executed over and over again. This is irrespective of whether the event in question has occurred or not.

What is Interrupt?

An interrupt is a mechanism which causes the processor to execute a specific function (an *interrupt service routine*, or ISR) in response to a specified event occurring, regardless of what part of the program is currently being executed (i.e., *asynchronously*).



Operation of interrupt and main program

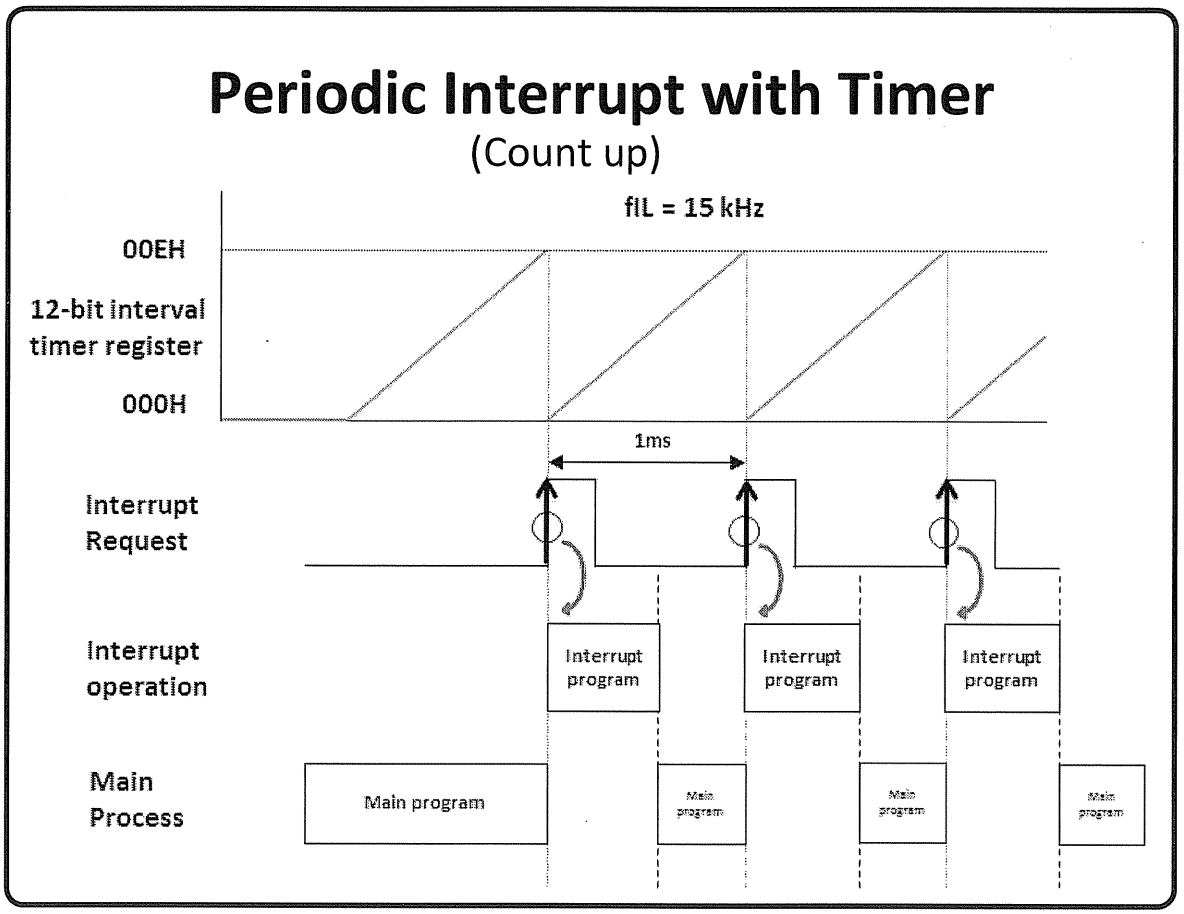
Interrupt processing is similar to telephonic correspondence. When the telephone bell rings, we stop our current work to answer the phone, and resume working after finished talking over the phone. In microcomputer, switching between two programs is done by a modification of the program counter (PC). Once the switched program is finished, microcontroller will restore the PC with its value before switching. As a result, the original program is resumed. Stack is used to save the contextual information (e.g. variables, values...) of the first program.

INTP10 interrupt

SW1 on RL78/G14-RDK board is connected to bit6 pin of port P7. If this pin is used as INTP10 input pin, INTP10 interrupt occurs by falling (negative edge) or rising edge (positive edge) of input signal.

Interrupt of RL78/G14

There are two interrupt types in RL78/G14 series, which are hardware interrupt and software interrupt generated by instruction.



Periodic interrupt using timer

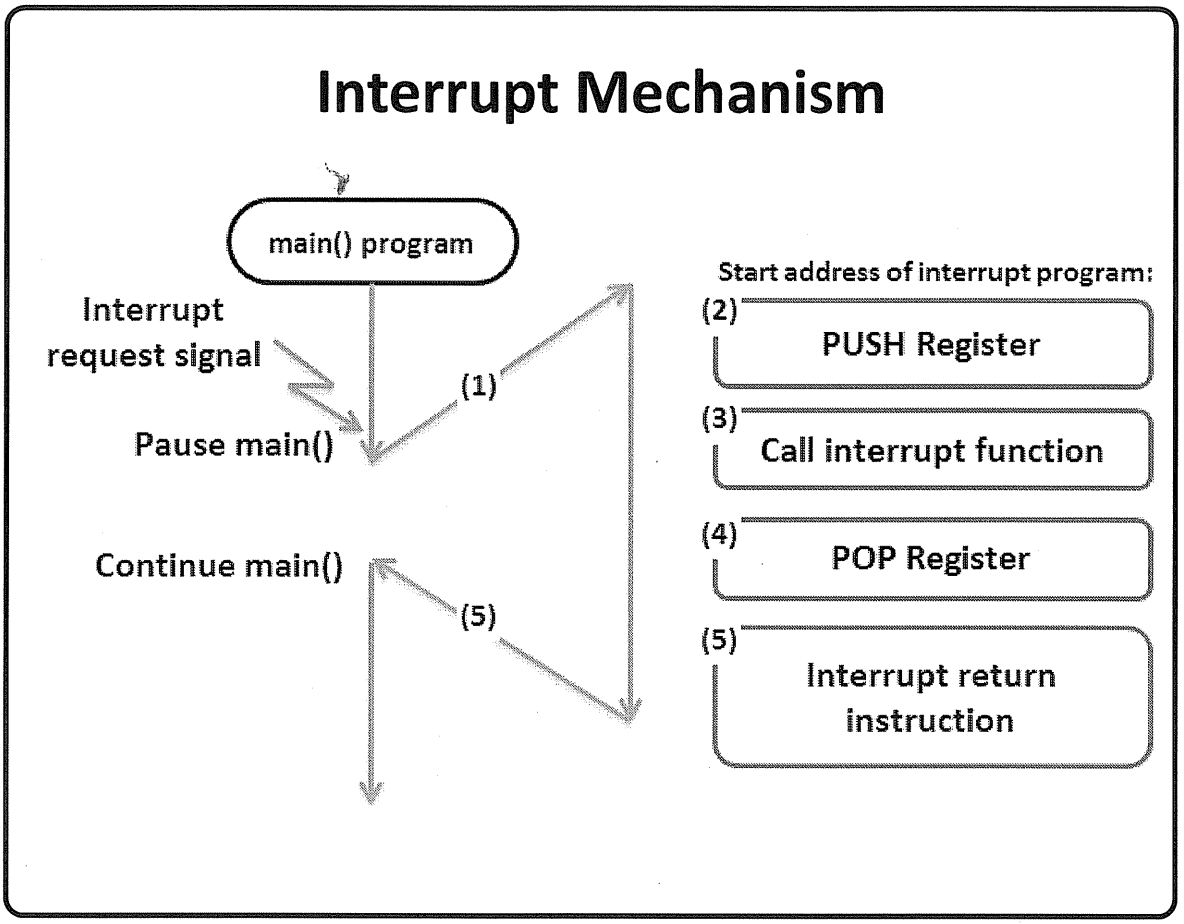
This technique consists of using timer (discussed in chapter 6) to set the period of interrupt generation so that interrupt request can be generated at a constant interval.

Interrupt sources and vector addresses

INTERRUPT TYPE	DEFAULT PRIORITY	INTERRUPT SOURCE NAME	INTERRUPT SOURCE TRIGGER	INTERNAL/ EXTERNAL	VECTOR TABLE ADDRESS	BASIC CONFIGURATION TYPE	128-PIN	100-PIN	80-PIN	64-PIN	52-PIN	48-PIN
Maskable	0	INTWDTI	Watchdog timer interval (75% of overflow time)	Internal	0004H	(A)	√	√	√	√	√	√
	1	INTLVI	Voltage detection		0006H		√	√	√	√	√	√
	2	INTP0	Pin input edge detection	External	0008H	(B)	√	√	√	√	√	√
	3	INTP1			000AH		√	√	√	√	√	√
	4	INTP2			000CH		√	√	√	√	√	√
	5	INTP3			000EH		√	√	√	√	√	√
	6	INTP4			0010H		√	√	√	√	√	√
	7	INTP5			0012H		√	√	√	√	√	√
	8	INTST2/ INTCSI20/ INTIIC20	UART2 transmission transfer end or buffer empty interrupt/CSI20 transfer end or buffer empty interrupt/ IIC20 transfer end	Internal	0014H	(A)	√	√	√	√	√	√

Interrupt vector table

The interrupt vector table holds the 16-bit address (vector) of each ISR (interrupt service routine). The CPU uses this table to load the PC with the correct ISR address. Figure above shows an example part of the interrupt vector information for the RL78/G14 MCU. In some cases, a vector is used by multiple interrupt sources. For example, the reset vector (located at 00000H) is used when the RESET pin is asserted, for power-on-reset, when a low operating voltage is detected, when the watchdog timer overflows, or when an illegal instruction is executed.



Interrupt program

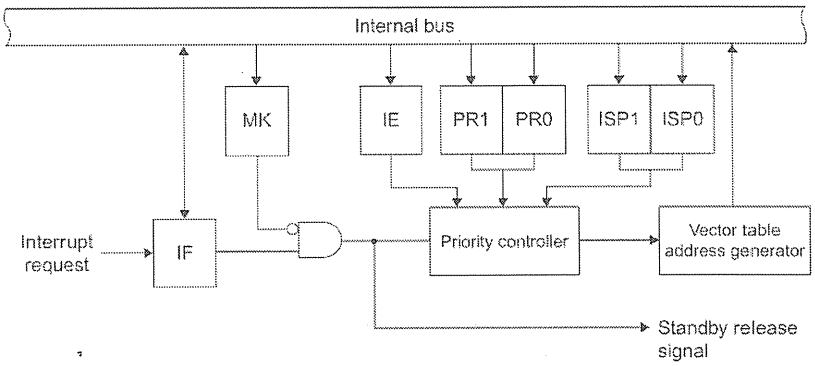
This program is started when interrupt request signal is generated and the interrupt is enabled.

Interrupt mechanism

- (1) Pause the main program and switch the control flow to interrupt program: Switch to the interrupt program with the start address already set in the interrupt vector table.
- (2) Save registers' values into the stack area: In order to prevent the registers' values used in main program from being altered, save them in the stack area.
- (3) Interrupt process: The body of interrupt process is an interrupt function which can be written in C. Interrupt function can be called from interrupt program.
- (4) Restore registers' values: Restore the saved values in the stack area to registers.
- (5) Return to main program from interrupt program: Execute an instruction to return from interrupt to main program (RETI with hardware interrupt and RETB with software interrupt). The control flow is now switched to main program.

Interrupt Enabled Condition

(For RL78/G14)



Interrupt will be enabled when the following conditions are satisfied:

1) IE = 1.
2) MK = 0.
3) PR1 PR0 ≤ ISP1 ISP0
4) IF = 1.

IF: Interrupt request flag

An interrupt request flag (IF) indicates whether an interrupt source has requested an interrupt. The flag is set when an interrupt source requests an interrupt, and is cleared when the processor has acknowledged the interrupt.

MK: Interrupt mask flag

Maskable interrupts are masked (disabled) by setting the source's MK flag to "1". They are unmasked (enabled) by clearing the source's MK flag.

PR1 and PR0: Interrupt priority specification flags

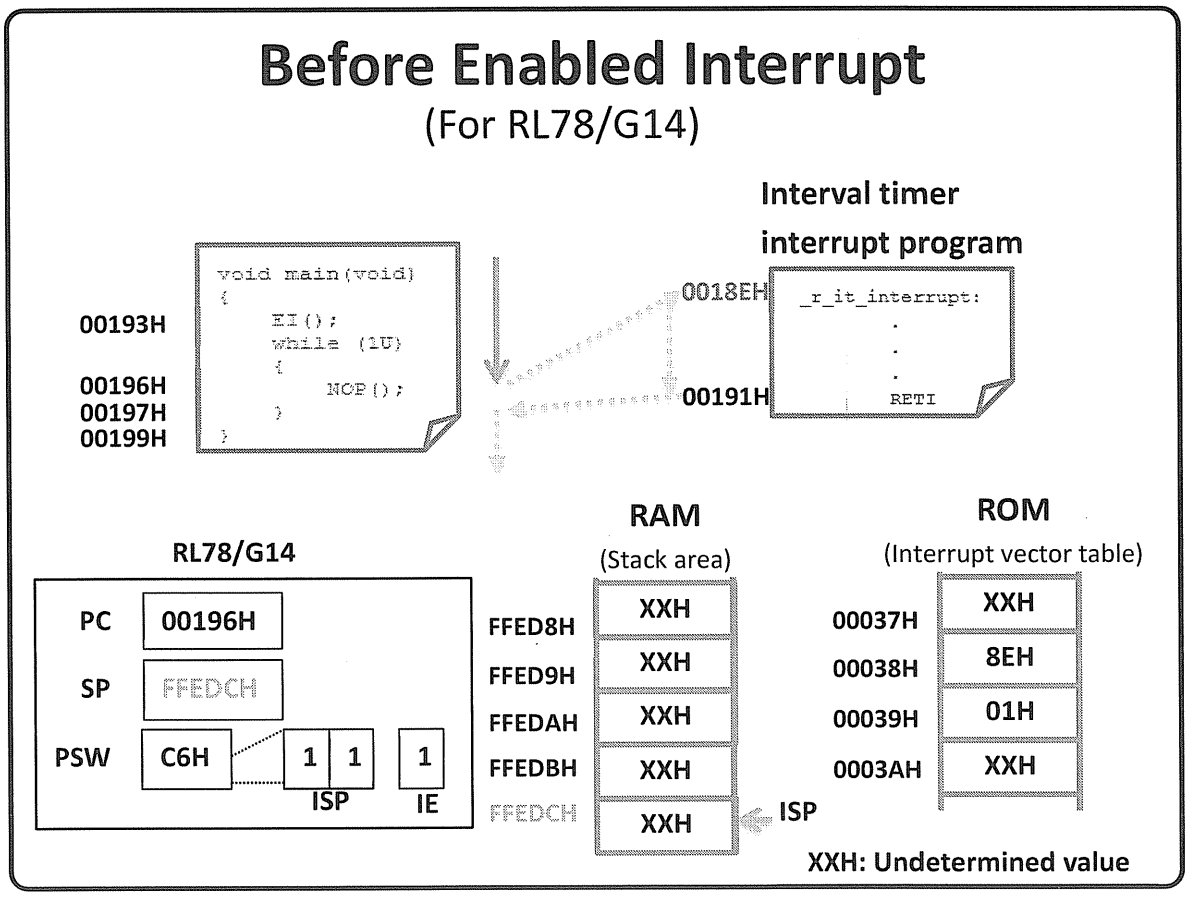
The priority of each interrupt source can be set to one of four levels using the PR1 and PR0 flags. "00" specifies the highest priority.

IE: Interrupt request acknowledgement enabled or disabled

The IE bit in the PSW register controls the masking (disabling) of maskable interrupts. When set to "1", maskable interrupts can be serviced. When cleared to "0", maskable interrupts are masked and are not serviced.

ISP1 and ISP0: Interrupt service priority

The ISP field in the PSW specifies the current interrupt priority level of CPU. "00" represents the highest priority. The CPU will service the interrupts with a priority level which is equal or higher than the one registered in the ISP.



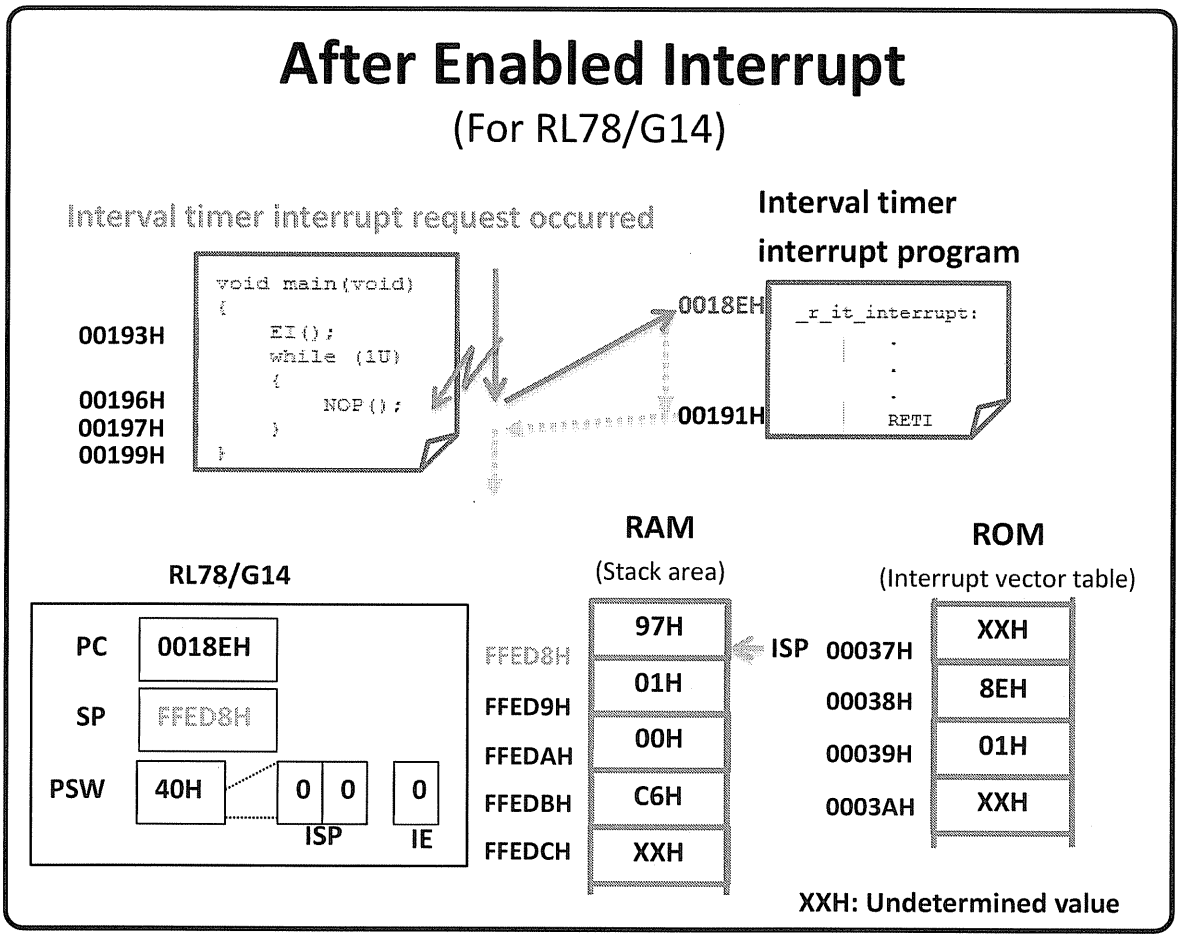
Interrupt vector table

The interrupt vector table holds the 16-bit address (vector) of each ISR. The CPU uses this table to load the PC with the correct ISR addresses. The vector codes that store the program start address are two bytes each, so interrupt jumps to a 64K address area from 00000H to 0FFFFH.

The role of interrupt vector table

User stores the start address of interrupt program in interrupt vector table in advance. When interrupt is enabled, CPU will search for the table corresponding to the interrupt factor and switches the control flow to the address in the table.

Before running main program, it is necessary to set the start address of interrupt program in the interrupt vector table to be used.

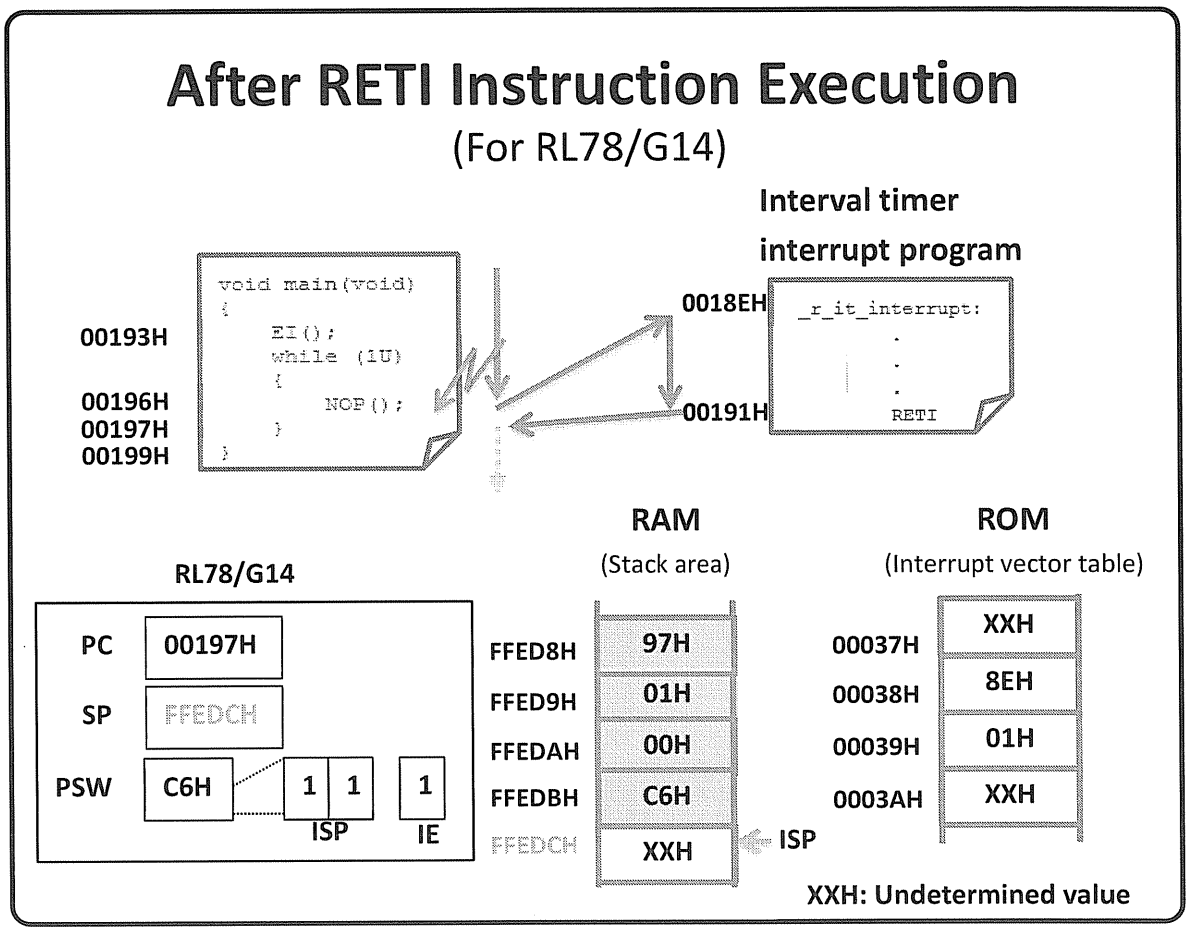


Interrupt sequence

It indicates the processing procedures inside the CPU from interrupt request enable to interrupt program execution.

Interrupt sequence of RL78/G14

- (1) When an interrupt is requested, the CPU will finish executing the current instruction (and possibly the next instruction) before starting to service the interrupt.
- (2) The CPU pushes the current value of the PSW and then the PC onto the stack.
- (3) The CPU next clears the IE bit. This makes the ISR non-interruptible.
- (4) If the interrupt source is a maskable interrupt, the CPU next loads the PSW PR field with the priority of the ISR being serviced.
- (5) The CPU loads the PC with the interrupt vector for the interrupt source.
- (6) The ISR begins executing.



RETI instruction

This is a return instruction from interrupt program. This instruction restores the PC and PSW register that were saved in stack when the interrupt request was accepted, and returns from interrupt to main program. The return of PSW register makes the flags (ISP, IE) be reset to the previous PSW state before interrupt request was accepted.

Difference with RET instruction

RET instruction is used to return from subroutine or C language function call and cannot make PSW register be restored automatically.

How to Use Interrupt

1) **Take priority of microcomputer hardware, select interrupt factor. If selecting external signal as interrupt signal, design it at circuit design state.**
2) **Write initial setting of interrupt in startup program.**
   - **Set interrupt control register.**
   - **Set interrupt vector table.**
3) **Consider the interrupt enable timing; determine the interrupt enable instruction position in program.**
4) **Separate the process from main function, consider the exchange data, and then write the interrupt function.**

Priority level of interrupt factor

The priority level of interrupt factor is determined by hardware. It indicates which one is accepted first from the view of hardware when multiple interrupts occur at the same time. Normally: Reset > NMI > Peripheral IO (priority level is determined by each peripheral function).

Note: NMI = Non Maskable Interrupt (external interrupt that cannot be prohibited).

Initial setting of interrupt

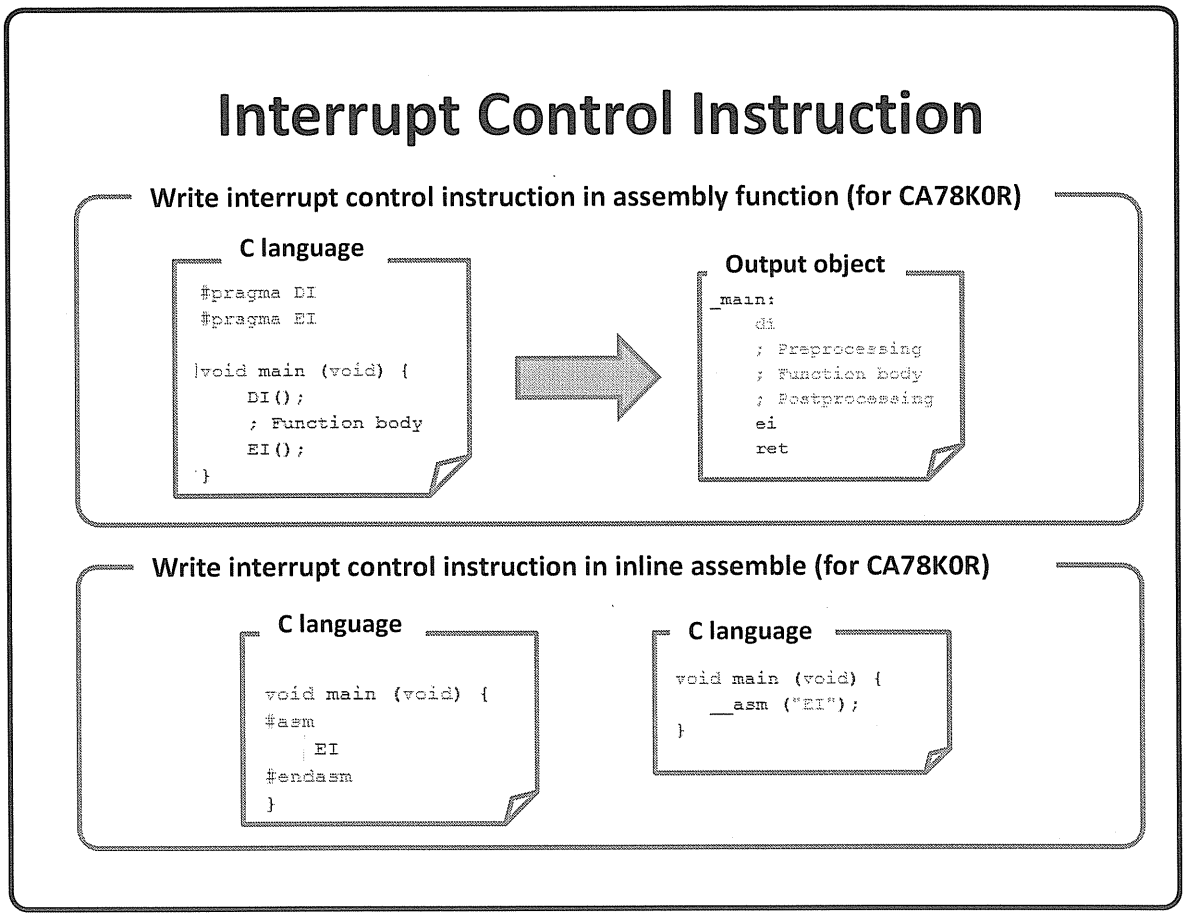
The initial setting of interrupt (interrupt priority level, address at which it will jump to, etc.) is done in startup program. Their descriptions vary with each CPU. The start address of interrupt program must be set in interrupt vector table beforehand.

Interrupt enable

After the initial setting of main function is completed and the conditions of accepted interrupt are satisfied, set interrupt enable flag to "1".

Interrupt function

Interrupt function is as a subroutine and called from the interrupt program written in assembly language.



Write interrupt control instruction

Interrupt control instruction is written in assembly language instruction. There are two methods to write it in C language: one is to write the interrupt control instruction into an assembly language function and call this assembly language function from C language program, the other is to use inline assemble and embed interrupt control instruction (assembly language instruction) in C language program.

ASM statements

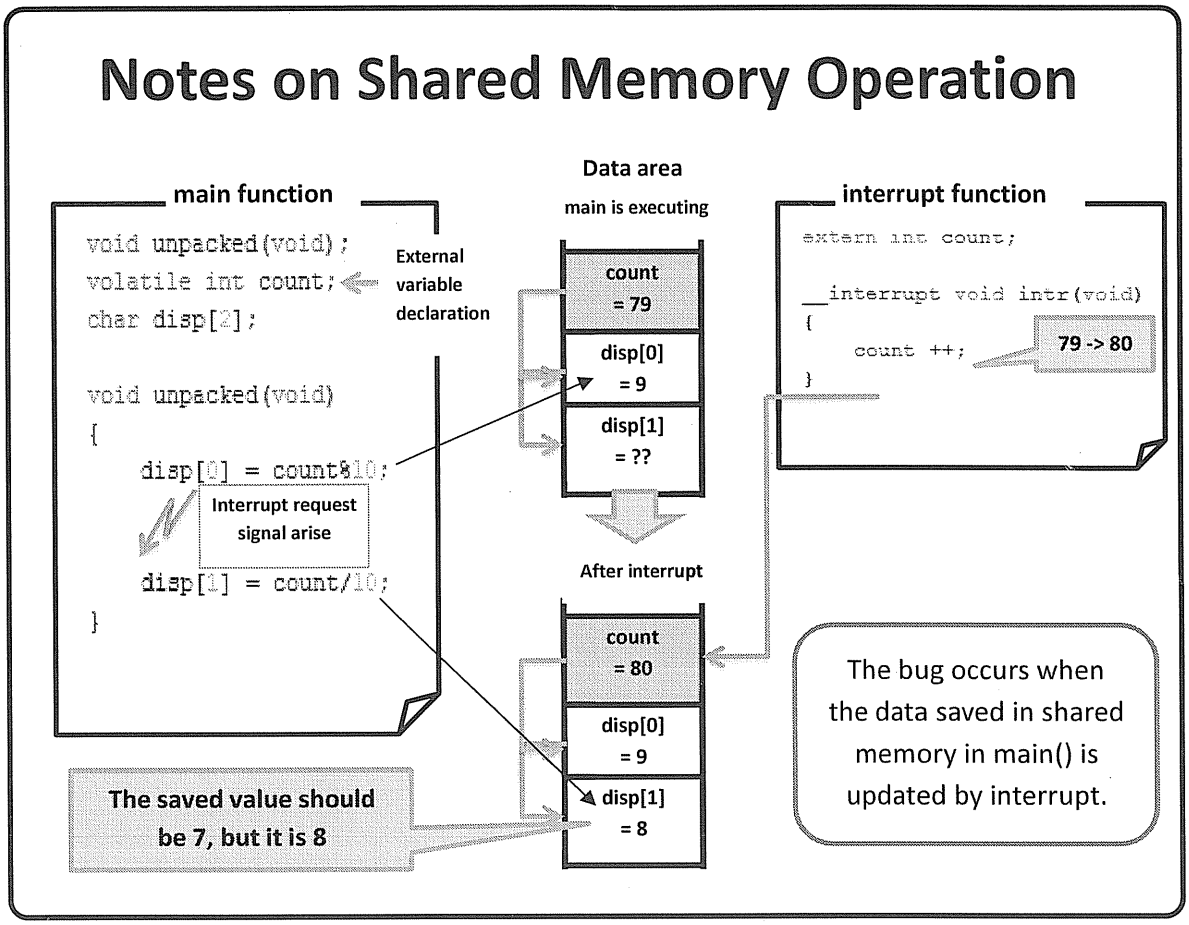
The #asm and __asm directives allow using assembly language statements in C source code (discussed in chapter 5). The statements are embedded in the assembly source code generated by the C compiler.

Notes on Interrupt Programming

1) Set the function shared by both main() and interrupt program to reentrant.

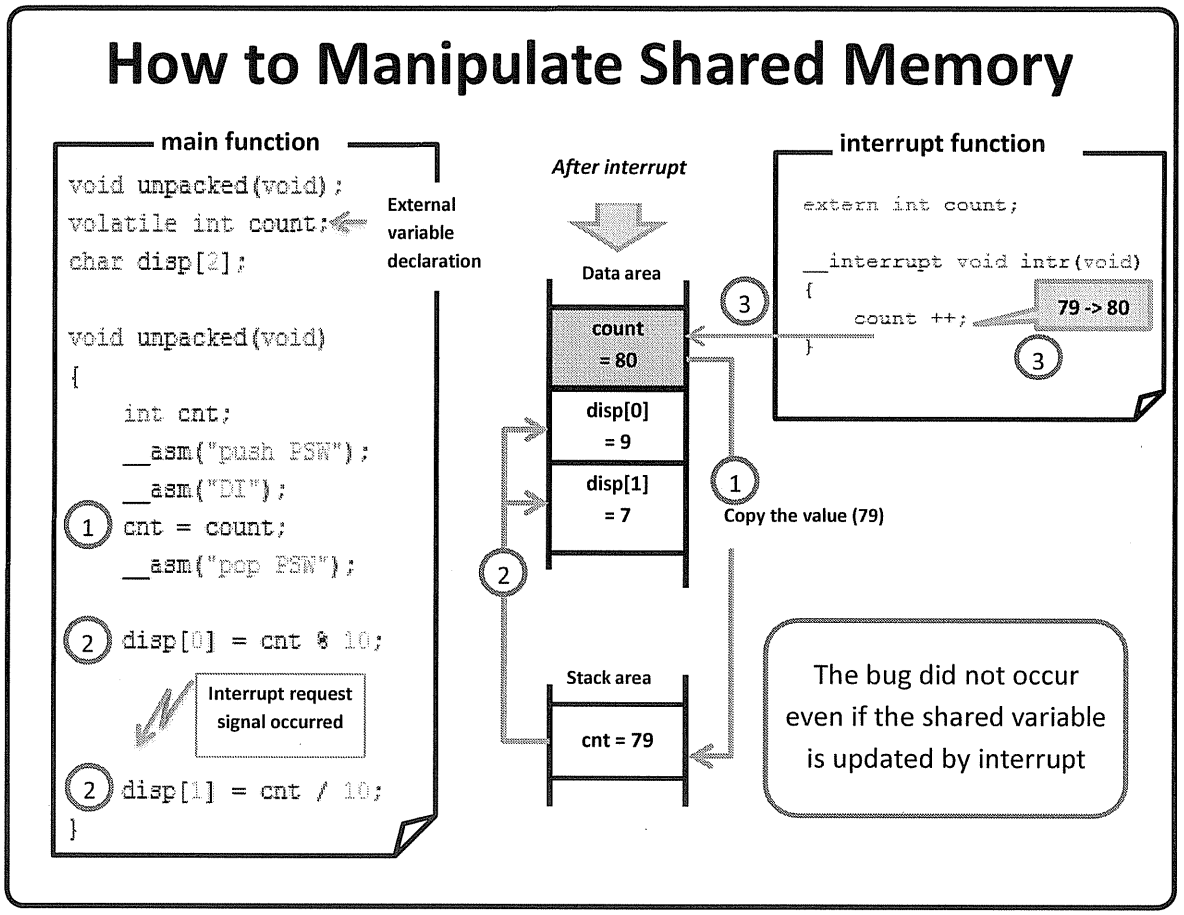
2) Use external variable (global) to exchange the data between main() and interrupt program. Use volatile modifier to prevent compiler from doing unexpected optimization.

3) During manipulation of the shared memory between main() and interrupt program, we should ensure the memory's content is not destroyed even if interrupt occurs.



Bug when operating shared memory

Take for example a variable (shared source) accessed (written) by both main and interrupt program. unpacked() function called by main program divides the value of variable count into upper and lower digit and stored lower digit in disp[0] and upper digit in disp[1]. However, since variable count does not use exclusive control, immediately after storing value in disp[0], interrupt request is generated. Once interrupt function intr() is executed, the value of variable count is modified during dividing. For this reason, after returning to main program, a bug occurs as the value to be divided into upper and lower digital was destroyed, and the saved value is wrong.



How to operate shared memory

Temporarily copy the value of shared memory to the other memory before operating it so that conflicts will not occur even if the shared memory is updated by interrupt. The bug does not occur although in the shared memory operating, interrupt was generated and the value of shared memory was modified.

Remember to prohibit interrupt during the memory-value-copy process in order to prevent this value from being changed.

Exclusive control of shared resource

Allowing accepting only one request for one shared resource (such as in the above example) is called exclusive control. There is no problem when a series of resource operation are performed in order, but it is necessary for us to pay attention on the case that the same resource is probably updated by interrupt processing that is generated at any place of a program. Usually, the operation of exclusive control method is to first set a flag to inform "resource is in use", and control the process. However, if there is not such a wait mechanism

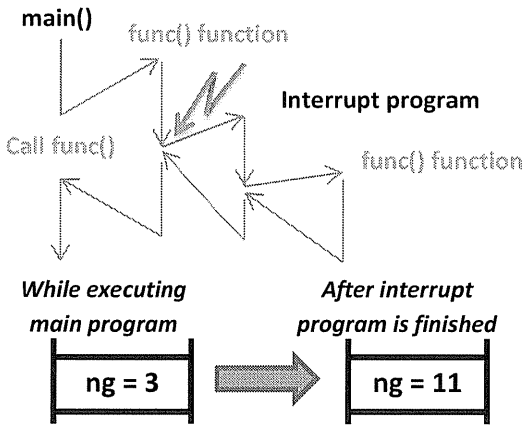
that waits for the flag and executes the next process, the processing like the above example is necessary.

Since embedded OS has the wait mechanism, user can easily realize the exclusive control program.

Reentrant Feature of Function

Function without reentrant

```
void func (void);
void main (void)
{
    func();
}
void func (void)
{
    static int ng;
    for (ng = 0; ng < 11; ng++)
    {
        .
        .
        .
    }
}
```



Data is destroyed when return to main() from interrupt program

Function without reentrant

When external variable and static variable are used within function, the new area to save variables at the entry of function is not ensured. Therefore, it is possible for the value of the variable to be destroyed when returning to the calling function after the called function is executed. We have to be careful on this mechanism.

Create a function with reentrant

In order to create a function with reentrant, do not use external variable or static variable within function but try to use auto variable if possible. This is because the same area would not be destroyed when auto variable is used.

Usage of volatile Qualifier

main function

```
volatile char event_flag;

void main (void)
{
    event_flag = 0;
    while (1)
    {
        if (event_flag == 0){...}
        if (event_flag == 1){...}
    }
}
```

interrupt function

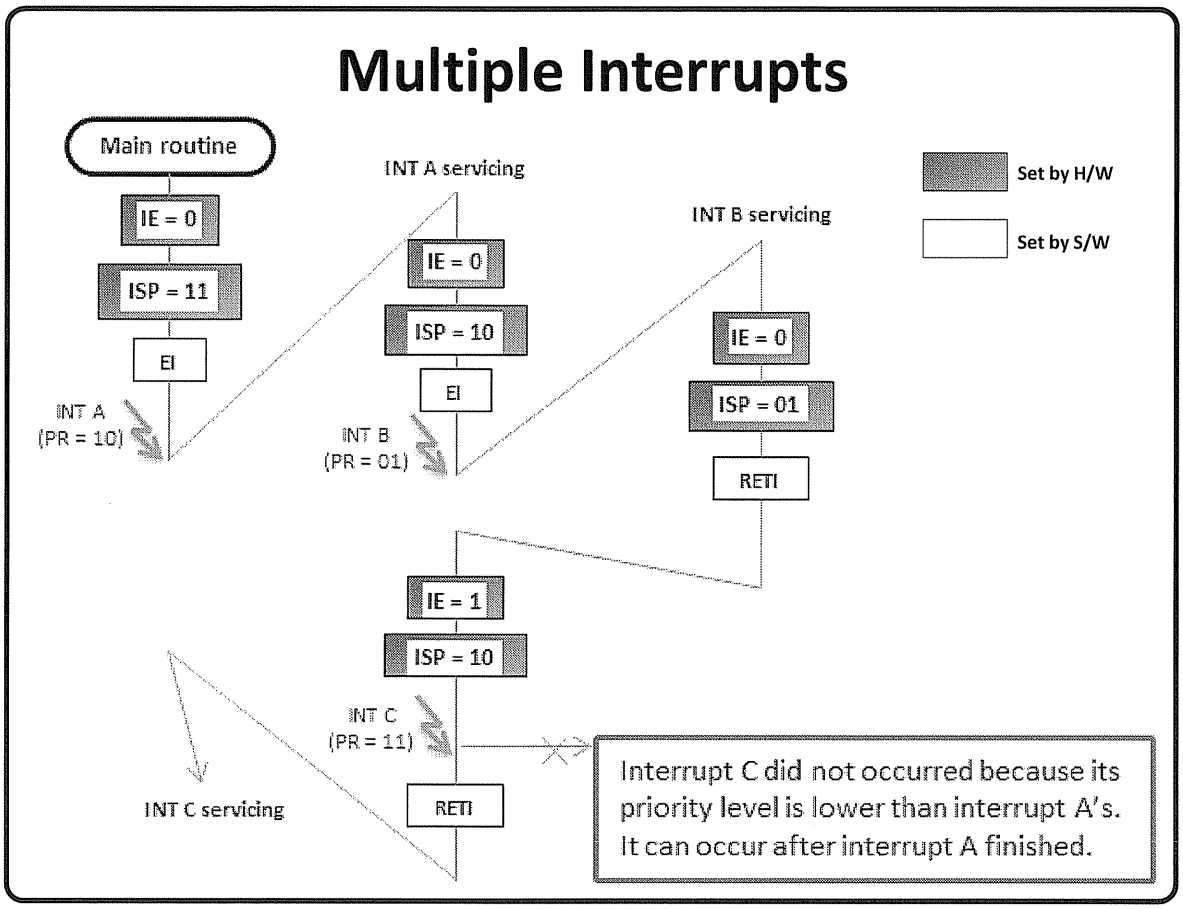
```
extern char event_flag;

__interrupt void intr(void)
{
    event_flag = 1;
}
```

When compiler does optimization, the instruction code of the second if-statement is not generated. If volatile is specified, compiler optimization is suppressed; the second instruction code will be generated.

volatile qualifier

Declaration with volatile qualifier is necessary for all I/O port and control register so that the compiler does not carry out the optimization unintended by the programmer (discussed in chapter 4).



Multiple Interrupt

For the ordinary interrupt processing, as soon as a single interrupt request is accepted, interrupt enable flag (IE flag) is set to "0", and make interrupt is disable state. When you wish to accept the other interrupt while an interrupt is executing, set the interrupt enable flag (EI flag) to "1" at the entrance of interrupt program and make interrupt be in enable state. Such nesting and controlling interrupt is called "Multiple Interrupt".

Control multiple interrupts

- (1) Enable interrupt: Set interrupt enable flag (EI flag) to "1" in the interrupt program and enable interrupt.
- (2) ISP < PR: Interrupt is permitted if and only if interrupt priority level is higher than ISP.
- (3) ISP > PR: If interrupt with the priority level lower than ISP occurs, this interrupt request is kept, but not accepted. When the interrupt priority level becomes higher than ISP, the interrupt will be accepted.

Compiler Directive Dependency

RL78, 78K0R C compiler supports the following #pragma directives, which allow extended function.

#pragma vect **#pragma interrupt**	Generate the interrupt vector table, and output object code required by interrupt in C.
#pragma di **#pragma ei**	Disable and enable interrupts in C.
#pragma brk	Write CPU control BRK instructions in C.
__interrupt / **__interrupt_brk**	The qualifier of an interrupt function in C.

#pragma directives

#pragma directives are one of the types of preprocessing directives supported by the ANSI C standard. A #pragma directive instructs the compiler to translate in a specific way, depending on the string that follows the #pragma.

The keyword after #pragma may be specified in either uppercase or lowercase (case insensitive).

Interrupt is often used in embedded systems in order to implement real-time processing. Most of C compilers for embedded microcomputer have the unique description format so that the interrupt processing can be easily written in C. RL78, 78K0R C compiler provides #pragma to support this function. This can reduce the codes written in assembly language.

(This page intentionally left blank)

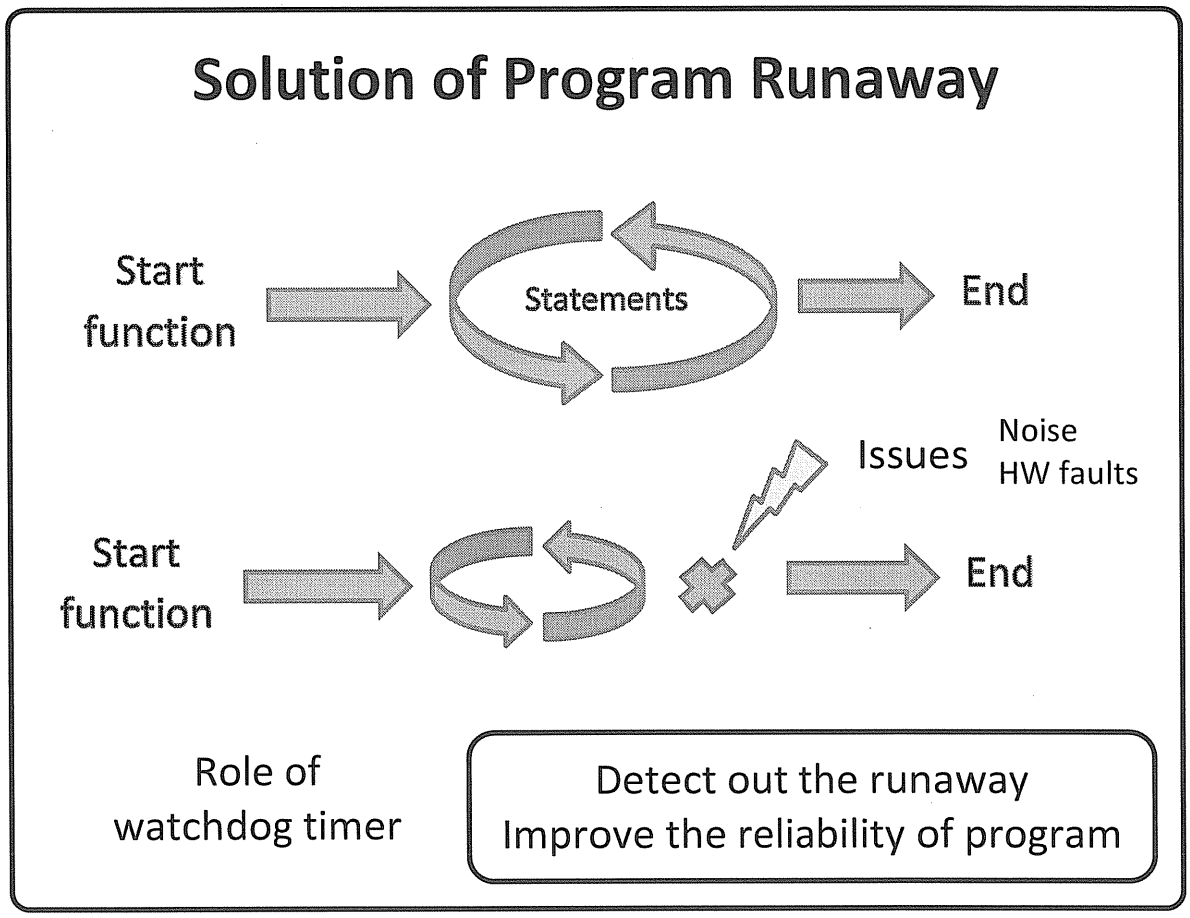
(This page intentionally left blank)

Chapter 9.

MCU Peripherals

In this chapter:

- Watchdog Timer
- ADC
- Serial Communications
- UART
- SPI
- I2C



Watchdog timer

A watchdog is a hardware timer used to detect and recover microcomputer malfunctions (due to noise, bugs, etc...).

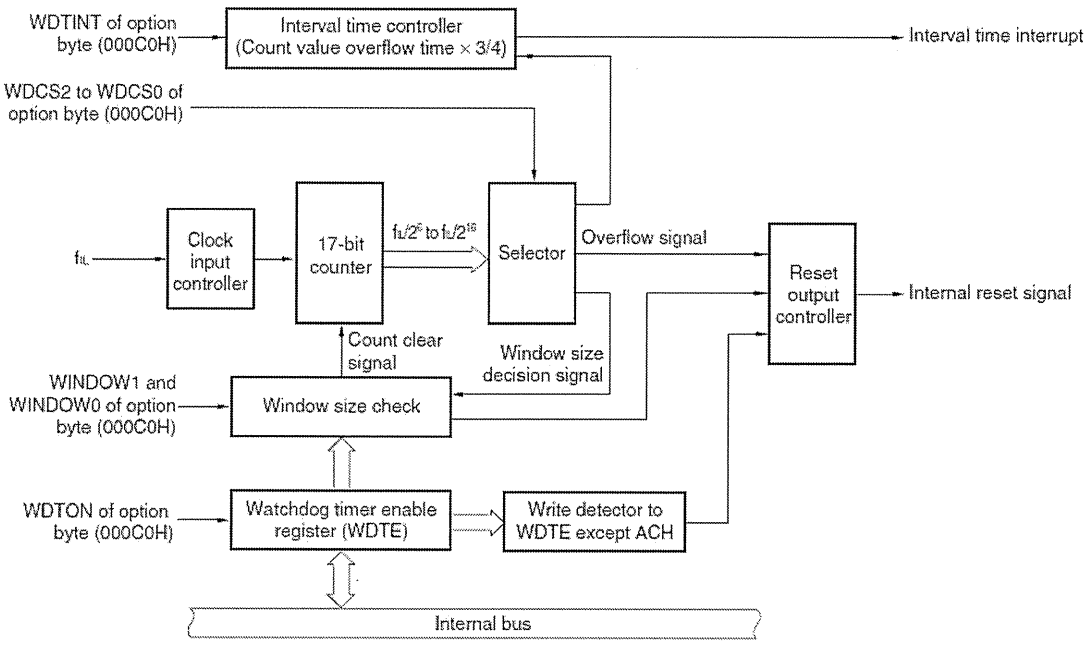
During normal operation, the microcomputer regularly restarts the watchdog timer to prevent it from elapsing, or "timing out". If there is hardware fault or program error, the microcomputer cannot restart the watchdog timer. This timer will elapse and generate an interrupt signal.

There are two kinds of watchdog timer in microcontroller, one is built inside microcontroller and the other is made out of microcontroller.

Watchdog timer of RL78 series

RL78 series has a built-in 17-bit watchdog timer using low-speed on chip oscillator clock (17.25 kHz MAX).

Watchdog Timer Block Diagram



RL78 watchdog timer diagram

Watchdog configuration

Watchdog timer operation is controlled by option byte and watchdog timer enable register.

- Option byte is a flash memory section, is used to configure operation of watchdog timer: interval interrupt, watchdog timer window period, overflow time, operation in HALT/STOP mode.
- Watchdog timer enable register (WDTE): By writing an activation value, watchdog timer will be cleared and restart counting.

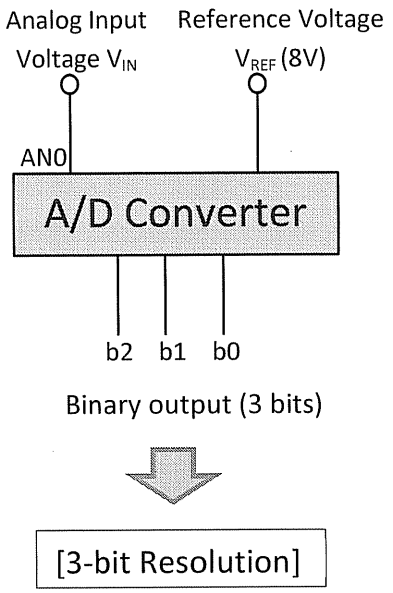
For detail information, please refer RL78 HW manual, chapter 13 WATCHDOG TIMER, page 600.

Watchdog reset signal

An internal reset signal will be generated in case:

- Watchdog timer counter overflows.
- A 1-bit manipulation instruction is executed on WDTE.
- Data other than activation value is written to the WDTE register.
- Data is written to the WDTE register during a window close period.

Analog-to-Digital Signal Conversion



Analog input voltage and digital value relationship

V_{IN} Range	b2	b1	b0
$0.0V < V_{IN} \leq 1.0V$	0	0	0
$1.0V < V_{IN} \leq 2.0V$	0	0	1
$2.0V < V_{IN} \leq 3.0V$	0	1	0
$3.0V < V_{IN} \leq 4.0V$	0	1	1
$4.0V < V_{IN} \leq 5.0V$	1	0	0
$5.0V < V_{IN} \leq 6.0V$	1	0	1
$6.0V < V_{IN} \leq 7.0V$	1	1	0
$7.0V < V_{IN} \leq 8.0V$	1	1	1

(Note: Reference voltage (V_{REF}) is 8V)

Function of A-D converter

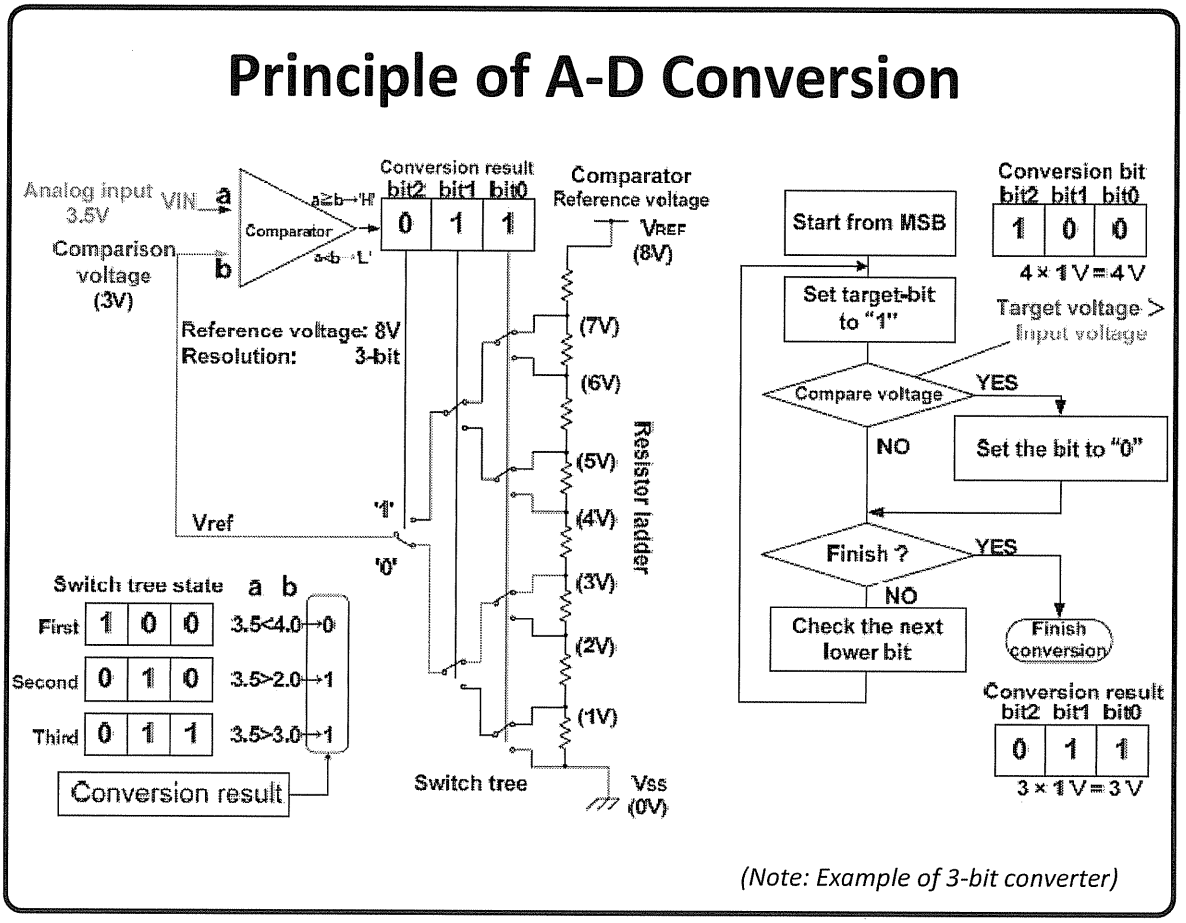
Converting the analog input voltage (V_{IN}) applied to analog input pin into digital value is called "A-D converter".

This function is used to deal with the analog signals such as "room temperature, humidity" of air conditioner, "pressure sensor" of electric scale, "gas sensor" of air purifier, airbag of automobiles and "vibration, acceleration sensor" etc...

A-D Converter in microcomputer

Because A-D conversion takes a little long time, it is necessary to confirm the conversion start of analog input voltage and the conversion end to digital value. Conversion starts with writing to A-D conversion start bit (set 1 to ADCS bit in ADM0 register for RL78/G14) and the conversion end is confirmed by generating interrupt request upon conversion completion.

The conversion time can be reduced using sample and hold function (which allows getting stable conversion results by sampling and holding a high-speed and successive analog input signal).



(Note: Example of 3-bit converter)

Successive approximation method

The "successive approximation method" shown in the above schema is often used in A/D converter. The data is converted from the highest bit to the lowest one in a sequential order.

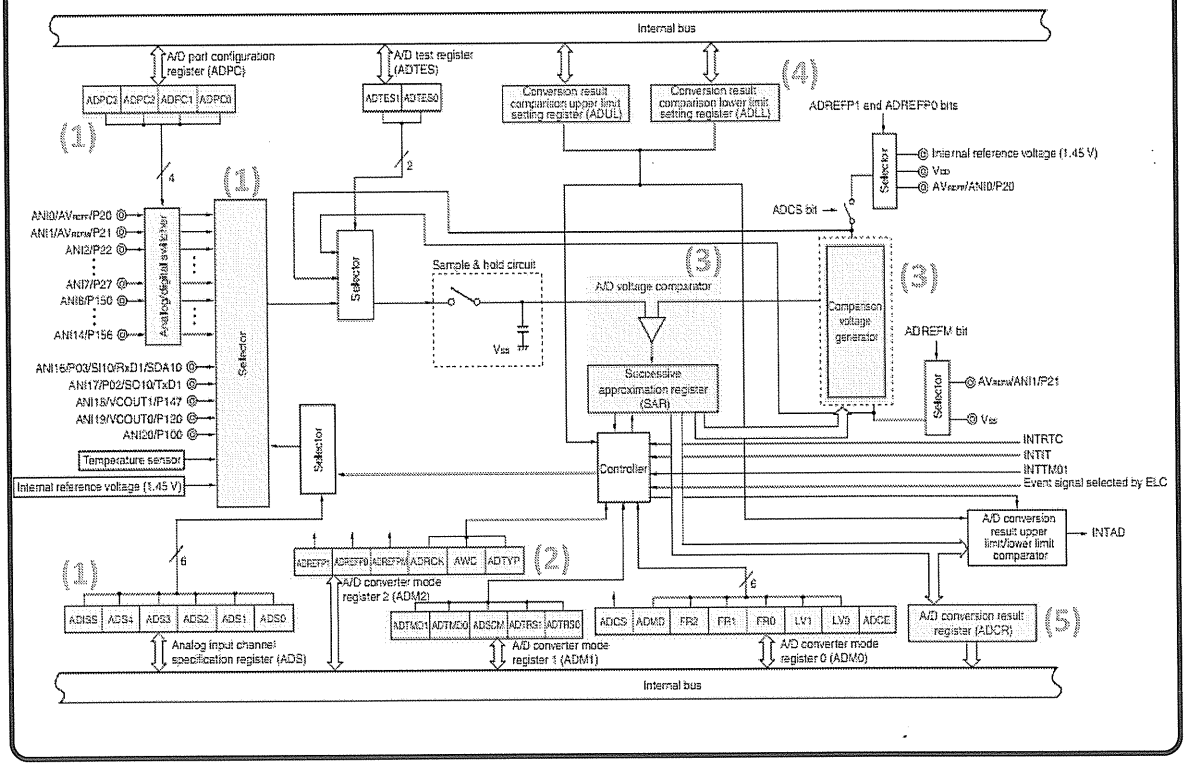
Conversion procedure

In the example of 3-bit convert in the above schema, the conversion will be performed in three steps:

- (1): Bit 2 (MSB) is "1" and the switch tree state is in "100". The comparison voltage is 4.0V at that state, then 3.5V<4.0V (input voltage < comparison voltage), so the conversion result for this bit is "0".
- (2): Bit 2 and bit 1 are now "01" and the switch tree state is in "010". The comparison voltage is 2.0V at that state, the conversion result for bit 1 is "1" because 3.5V>2.0V (input voltage > comparison voltage).
- (3): Bit 2, 1, 0 are now "011" and the switch tree state is in "011". The comparison voltage is 3.0V at that state, the conversion result for bit 0 is "1" because 3.5V>3.0V (input voltage > comparison voltage).

Finally, the conversion result is "011".

A-D Converter Block Diagram of RL78/G14



Selector blocks (1)

A combined use of these blocks is to select which A-D channel will be used to convert the analog input signal to digital value:

- A-D port configuration registers (ADPC) are to select the functionalities of input pins. ADPC will select whether a pin is used as an analog input for the A-D converter, or as a digital input.
- Analog input channel specification registers (ADS) are used together with ADPC to select the A-D converter channel. It not only selects the external signals, but also can select the internal ones such as temperature sensor and internal reference voltage.

A-D converter mode registers (2)

These registers switch A-D converters ON or OFF, select the conversion time, which trigger should be used, or if a wake up from Snooze Mode should be done etc...

Comparison voltage generator and A-D voltage comparator blocks (3)

Comparison voltage generator generates a voltage used to compare with the analog input voltage. The reference voltage supplied to comparison voltage generator can be selected from one of these voltage sources below:

- Internal reference voltage (1.45V)
- V_{DD}
- AV_{REFP} or AV_{REFM}

A-D voltage comparator compares the voltage generated from comparison voltage generator with the analog input voltage. After completing A-D conversion, the contents of the SAR register are held in the A-D conversion result register (ADCR).

Conversion results comparison block (4)

Upper and lower limit compare registers "ADUL" and "ADLL", with which we can set the A-D converter to only generate an interrupt if the A-D conversion result is within or outside a valid range. With this additional functionality, the A-D converter can behave like a comparator.

A-D conversion result block (5)

A-D conversion result register (ADCR) will hold the result of A-D conversion.

Relationship between the input voltage and conversion results

In RL78/G14, the below equation is used to express the relationship between the analog input voltage sent to the analog input pins (ANI0 to ANI14, ANI16 to ANI20) and the theoretical A-D conversion results.

$$SAR = INT\left(\frac{V_{AIN}}{AV_{REF}} \times 1024 + 0.5\right)$$

$$ADCR = SAR \times 64$$

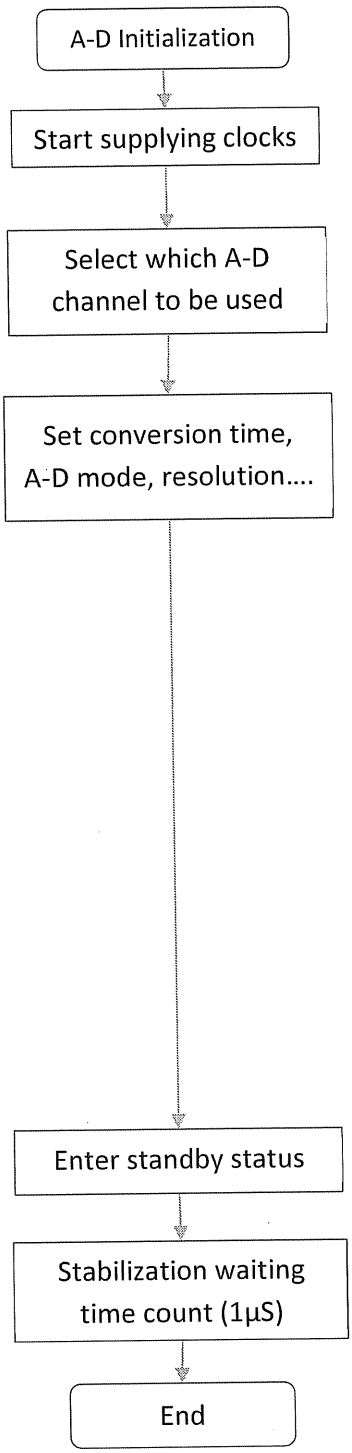
or

$$\left(\frac{ADCR}{64} - 0.5\right) \times \frac{AV_{REF}}{1024} \leq VAIN < \left(\frac{ADCR}{64} + 0.5\right) \times \frac{AV_{REF}}{1024}$$

Where are:

- INT( ): Function which returns integer part of a floating number
- V_{AIN}: Analog input voltage
- AV_{REF}: AV_{REF} pin voltage
- ADCR: A-D conversion result register (ADCR) value
- SAR: Successive approximation register

Configure A-D Converter (1)



- Set ADCEN bit in PER0 register to 1

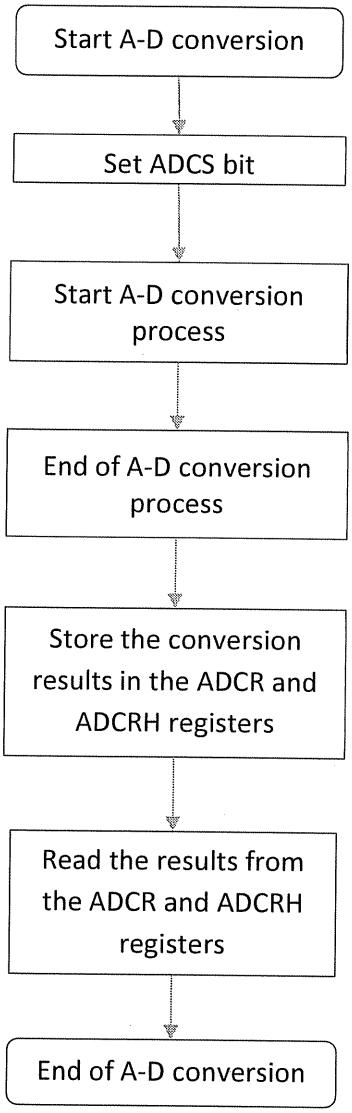
- Select A-D channel:
  ANI0 to ANI14: Setting for ADPC register
  ANI16 to ANI20: Setting for PMC register

- ADM0 register:
  FR2-FR0, LV1-LV0 bits: Select A-D conversion time
  ADMD bit: Select mode/scan mode
- ADM1 register:
  ADTMD1-ADTMD0 bits: Specify software trigger mode
  ADSCM bit: Sequential conversion/one-shot conversion
- ADM2 register:
  ADREFP1, ADREFP0, ADREFM bits: Select the reference voltage source
  ADRCK bit: Select the range for the A-D conversion result
  ADTYP bit: 8-bit/10-bit resolution
- ADUL/ADLL registers:
  Specify the upper/lower limit A-D conversion result comparison values.
- ADS register:
  ADS4-ADS0 bits: select the analog input channels

- Set ADCE bit in ADM0 register to 1

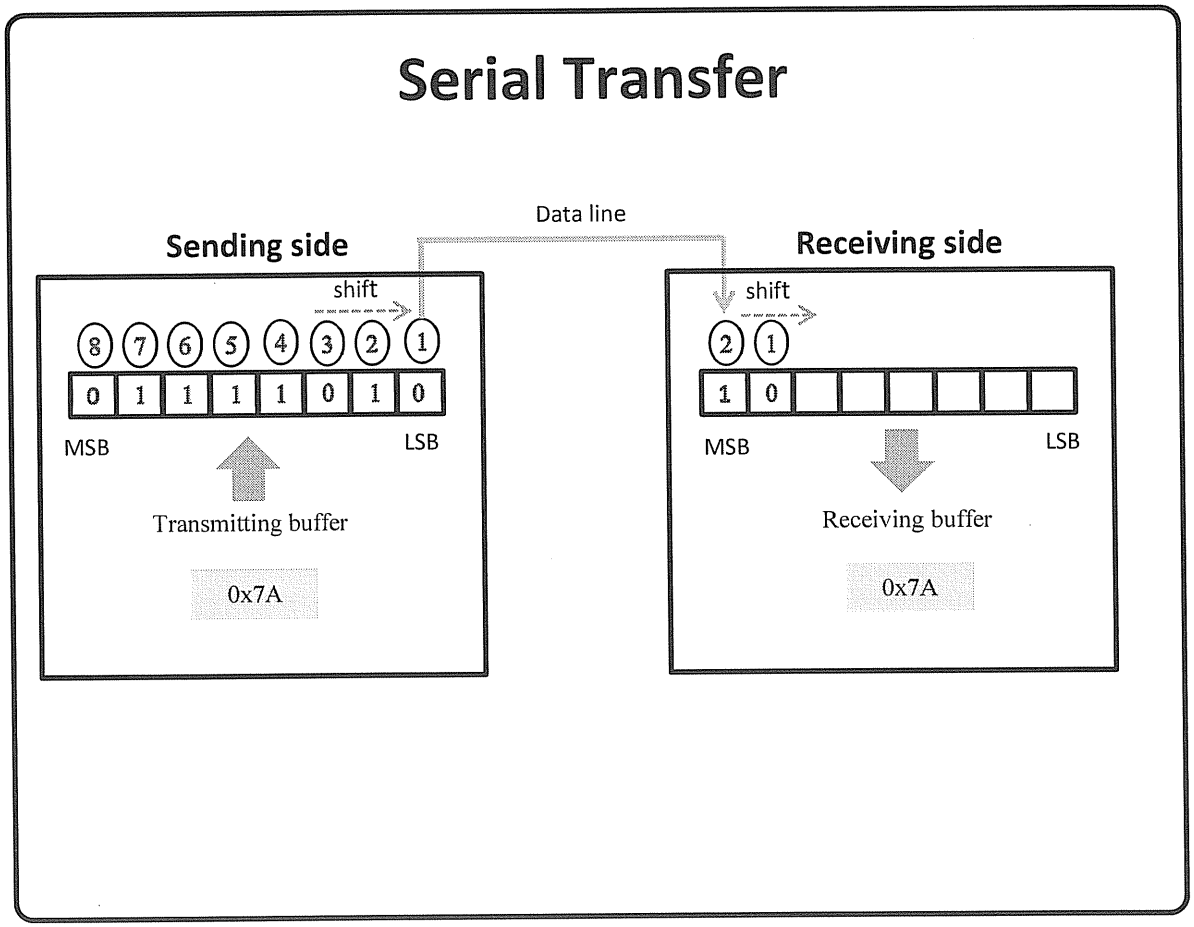
(Note: For details, refer to the hardware manual of the microcontroller)

Configure A-D Converter (2)



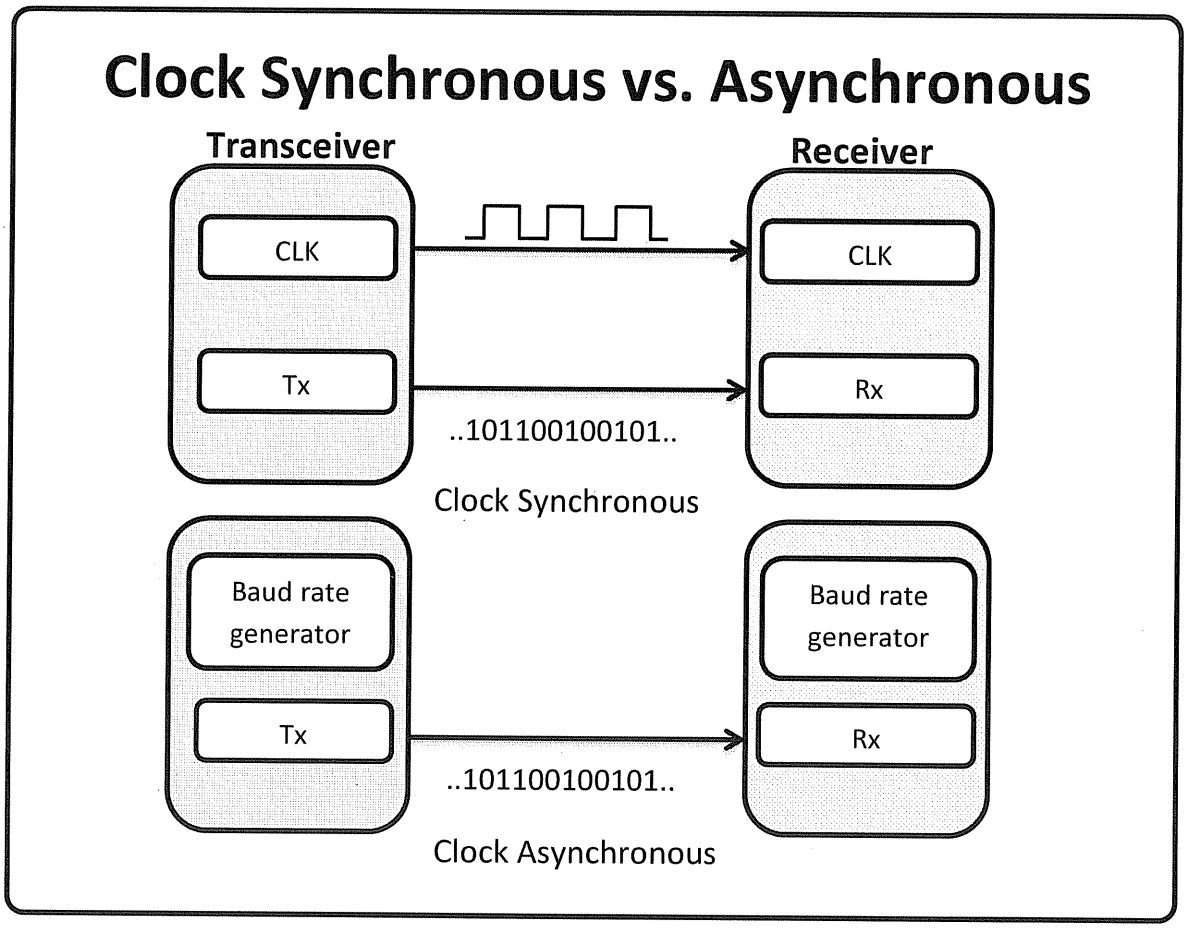
- Set ADCS bit in ADM0 register to 1 to start the A-D conversion

- The A-D conversion end interrupt (INTAD) is generated



Serial transfer

It is a technique of sending data one-by-one bit through a data line at each predetermined time. Comparing with parallel transfer, its benefit is low cost because the number of lines needed for data transfer is lesser.



Clock synchronous method

In serial communication, a transfer method is called clock-synchronous when there is a control line used to inform the transfer timing for each bit, and the data line used to send data. Data is transmitted in synchronization with the control signal (clock signal).

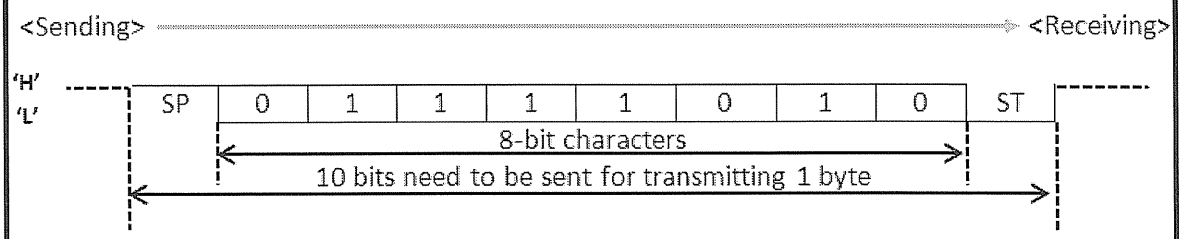
Clock asynchronous method

In serial communication, if the transfer timing of each bit is not informed, the method is called clock asynchronous. This method does not need a signal line for synchronization, so sending and receiving sides should be matched in data format and transfer rate. When there is no data sent out, the data line is applied to "H", the receiving side can detect the start bit "L".

Universal Asynchronous Receiver Transmitter (UART)

Start bit (ST)	1-bit 'L' signal
Transfer data length	Selectable 7, 8 or 9 bits
Parity bit (P)	Odd or Even
Stop bit (SP)	1-bit 'H' signal or 2-bit 'H' signal

Assuming transmission/reception format:
[Transfer data length 8 bits], [1 stop bit] and
[No parity], the data to be sent is [0x7A].



Transfer data format

This format consists of start bit, data character, parity bit and stop bit.

Error detection

Each UART channel has a serial status register (SSRmn), indicates the communication status and error occurrence status of channel. By checking the value of these registers we can know exactly which kinds of error being occurred while transferring data.

- Overrun error: This error is generated when the next data lines up before the content of receiving buffer register is read.
- Framing error: This error is generated when stop bit falls short of the set number of stop bits.
- Parity error: A parity bit is a bit added to the end of a binary code. In case of even parity, if the total number of "1" of the orginal code is odd, the parity bit value is set to "1" so that the total number of "1" of the binary code (including the parity bit) becomes even. A parity error occurs when the parity of the total number of "1" is different from that specified by the parity bit.

Calculating Transfer clock frequency

The baud rate for UART (UART0 to UART3) communication can be calculated by the following expressions:

$$Transfer\ clock\ frequency = \frac{fMCK}{2 * (\mathrm{SDR}[15:9] + 1)}$$

f_{MCK} is operation clock frequency of target channel
SDRmn[15:9] is the value of bits 15 to 9 of serial data register

Data transfer rate

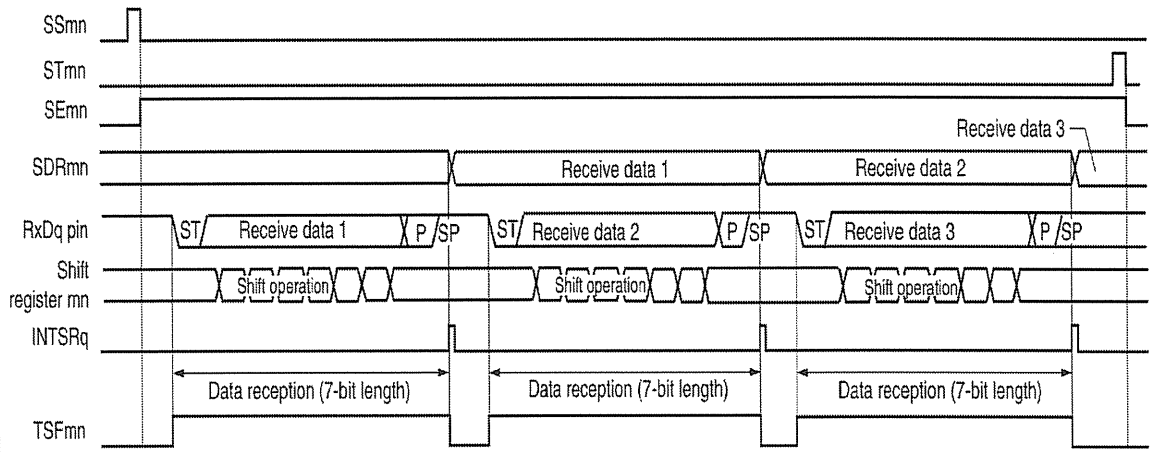
Bps (bits per second) is the unit for data transfer. It indicates the number of bits sent per second. Transmission rate is often called "baud rate". As a transfer speed, there are 9600bps, 4600bps, 2400bps and so on.

Configure baud rate for UART communication

In RL78/G14 MCU, there is no register used to directly set the actually calculated baud rate. In order to select an approriate baud rate for UART communication, we follow these below steps:

- Specify the transfer clock (f_{MCK}) that UART will be used. The clock frequency (f_{MCK}) is calculated based on the setting of SPSm register and bit 15 (CKSmn) of SMRmn register. In RL78/G14 hardware manual, Table 17-4 Selection of Operation Clock for UART, a list of f_{MCK} is available. User can select more easily an appropriate transfer clock for UART based on user own application.
- Set SDRmn[15:9] bits. We can select division ratio for transfer clock of UART by setting these bits. Refer to RL78/G14 hardware manual, Figure 17-9. Format of Serial Data Register mn (SDRmn), to get more information.
- After the transfer clock (f_{MCK}) and divison ratio are selected. The baud rate will be specified by using the above expression.

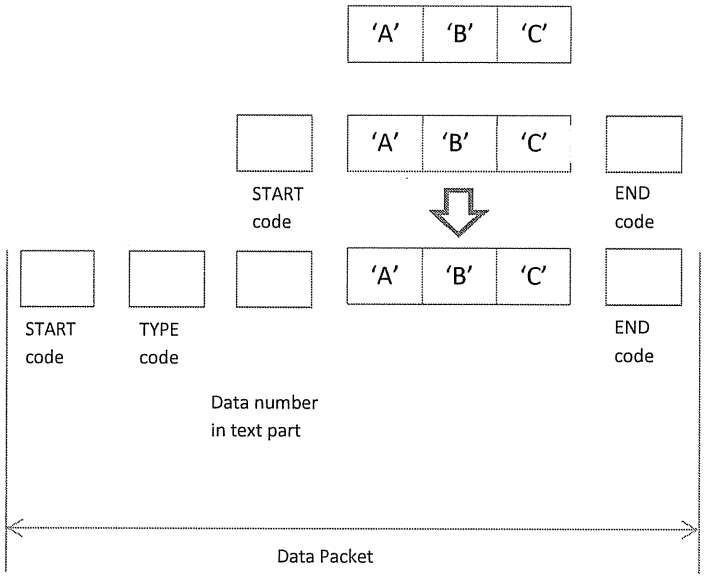
Timing Chart of UART Reception



Timing of reception sequence

- Read the start bit: Received data from RxDq pin is written to lower 8-bit of SDRmn register (hereafter referred to as RxDq register) from the start bit, then bit-by-bit.
- Detect falling edge of start bit: Once the start bit is detected by a falling edge of RxDq pin, the reception data started.
- Check the level of start bit: Checks the center of the start bit level at the rising edge of the next transfer clock, it is as a signal if the level is "L". It is regarded as noise, at this time, stop the reception and return to wait state of start bit.
- Fetch receive data: Following the start bit, the defined data is received, bit-by-bit in sequence, from LSB to MSB or from MSB to LSB based on user own setting. Reception data is read at the rising edge of transfer clock.
- Reception complete: When stop bit(s) is(are) received following the defined data. The content of received data is stored in RxDq register. To know whether the reception is completed, we can check the flag or interrupt request bit.

Packaging the Transferred Data



A data packet consists of:

- Data to be transmitted or received
- Control code (START, END, TYPE, ERROR detection)

Data packet

When any error occurs during transmitting/receiving a large-volume data, we must restart to transmit/receive this data from the beginning. This would take too much time and leads to inefficient communication. In order to solve such a problem, it is recommended to use the method that transmits/receives by dividing the data into some smaller blocks called data packet. This method allows us to only send the corrupted packet again if error occurs. This makes data-resend-processing become lesser and improves the communication efficiency. This also makes it possible to transmit/receive another packet in the midst of sending some packets. Therefore, multiple transmission/receptions can be simultaneously performed using the same data line.

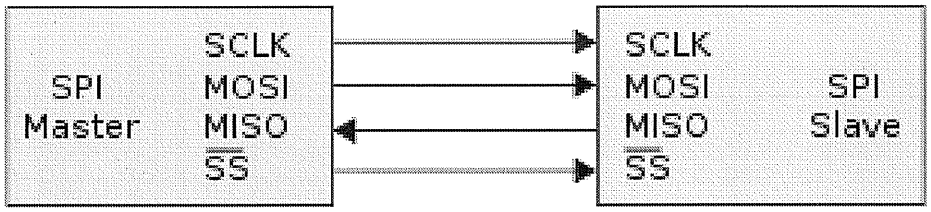
Control code (also called as header) is added into the packet to identify the sender and receiver, as well as to detect errors. Packer format should be determined in advance between sending and receiving sides. Sending side must create the packet according to this format, and

receiving side should analyze the packet and check errors accordingly. The processing when error occurs varies on a particular hardware system.

ECC

ECC stands for Error Correcting Code. It is to correct the data error. Various methods such as check sum and parity bit exist. It is mainly used for communication processing.

Serial Peripheral Interface (SPI)



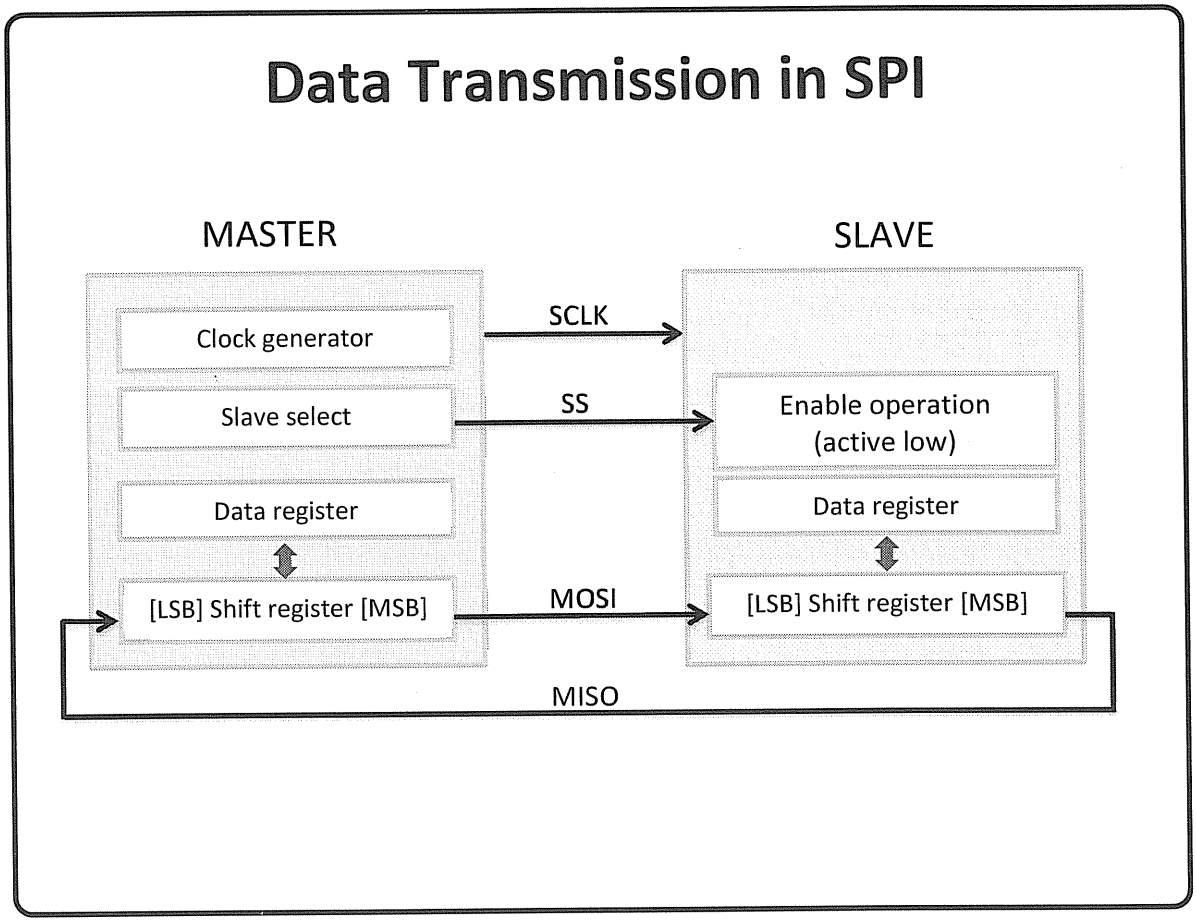
SPI BUS with single master and single slave

Serial Peripheral Interface

Serial Peripheral Interface (SPI) is a synchronous serial communication interface specification used for short distance communication, primarily in embedded systems. The SPI bus has four wires for communication (sometimes, it is called as 4-wire serial):

- SCLK: Serial Clock – represents master clock. This clock determines the speed of data transfer and all transmitting-receiving process is done synchronously to this clock.
- MOSI: Master Output - Slave input.
- MISO: Master Input - Slave output.
- SS: Slave select (active low).

SPI is a full-duplex communication standard (it means allowing communication in both directions) and present in typical applications including sensors, memory card, display control.

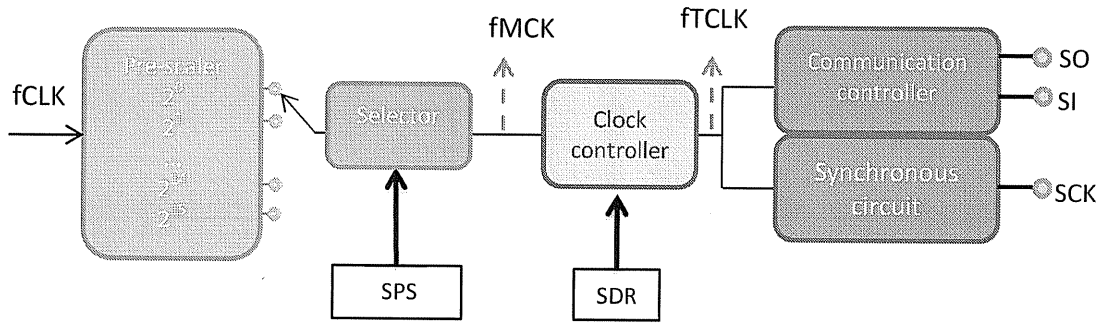


Data transmission

To begin communication, the master configures clock based on a frequencies which are supported by slave device (typically a few MHz). Target slave will be selected by using slave select pin. As long as the SS is at high level, the slave will not accept the SCLK clock and data input-output pins is in high impedance. When SS is at low level, data can be transferred from master to slave and vice versa.

Transmission normally involves two shift registers of some given word size (8 or 16 bits), one in master and one in slave. During each clock cycle, data in each shift register is shifted out with MSB bit first, while shifting a new LSB bit into the same register. After register has been shifted out (some clock pules), the master and slave have exchanged the value in their registers.

Transfer Rate (CSI in RL78)



Clock generation block (master transmission)

$$Transfer\ rate\ [Hz] = \frac{\text{fMCK}}{2 * (\text{SDRx}[15:9] + 1)}$$

fCLK: CPU/peripheral clock
fMCK: operation clock
fTCLK: transfer clock

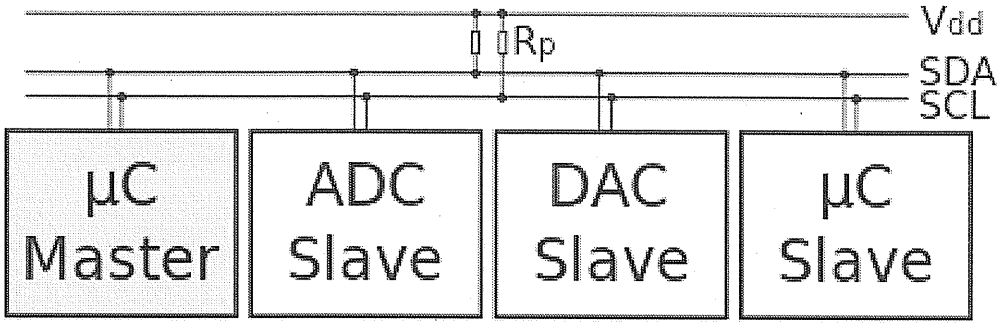
Serial clock select register (SPSx – RL78)

Serial clock select register SPSx is a 16-bit register used to select the operation clock fMCK which is supplied to target channel.

Serial data register (SDRx – RL78)

Serial data register SDRx is 16-bit register used for transmitting/receiving process. Bits [9:15] are used as a register to set the division ratio of the operation clock fMCK. The transfer clock of the channel fTCLK is set by dividing the operation clock fMCK by the value of the higher 7 bits of the SDRx register.

Inter-Integrated Circuit (IIC – I2C)



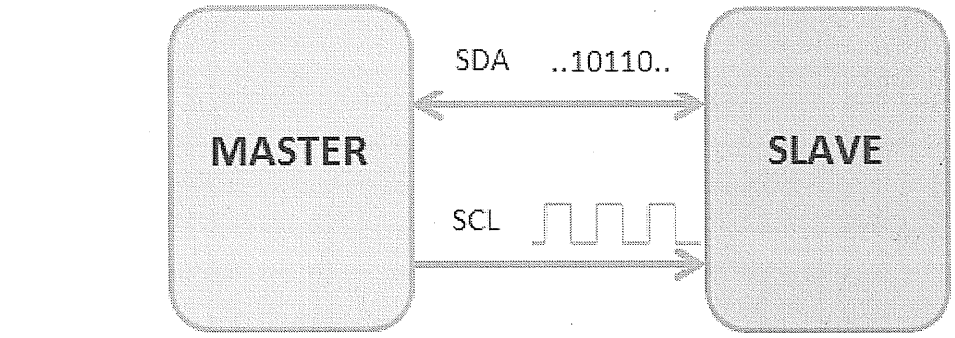
An I2C bus with one master and three slaves

I2C protocol

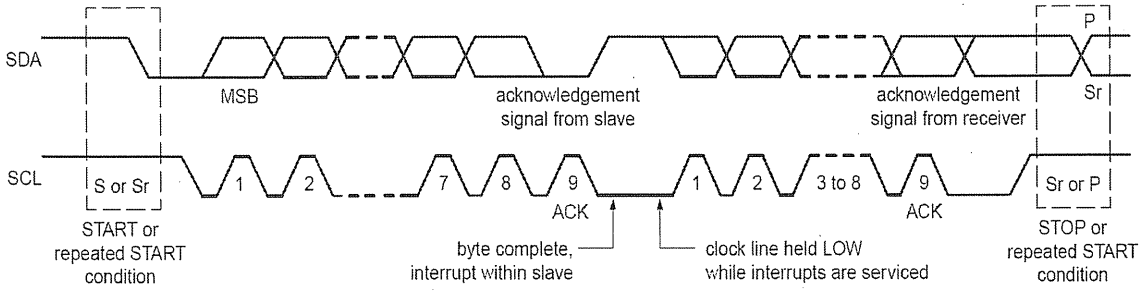
The I2C protocol is a synchronous serial interface which uses only two lines for transmitting and receiving data. There are two kinds of device on an I2C BUS: master (clock supplier) and slave. The BUS is called multi-master which means any number of master devices can be present. Each slave device connected to the BUS has a unique address; the master device will use this address to identify the target slave in communication.

The I2C BUS has five modes: Low speed (10 Kbit/s), standard (100 Kbit/s), fast speed (400 Kbit/s), fast speed plus (1 Mbit/s) and high speed (3.4 Mbit/s).

Data Transmission (1)



I2C BUS (one master – one slave)



Data transfer on I2C BUS

I2C BUS

Only two BUS lines are required: A serial data line (SDA) and a serial clock line (SCL).

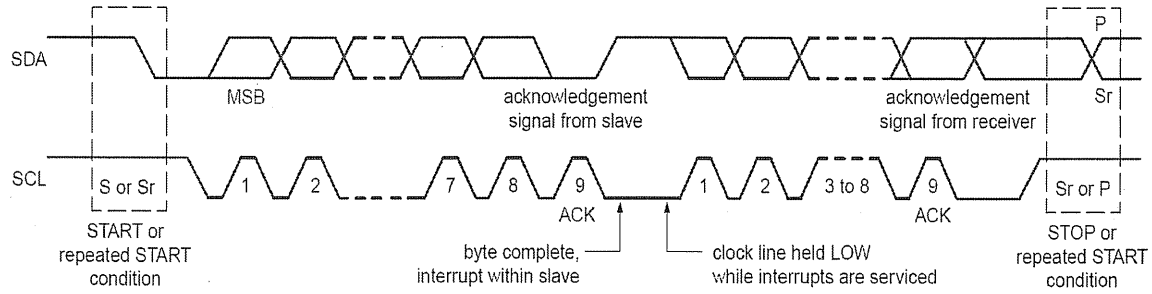
Start and stop conditions

- Start condition: A "H" to "L" transition on the SDA line while SCL line is "H".
- Stop condition: A "L" to "H" transition on the SDA line while SCL line is "H".

Communication condition

All transactions begin with a START (S) and are terminated by a STOP (P). START and STOP conditions are always generated by the master. The BUS is considered busy after the START condition.

Data Transmission (2)



Data transfer on I2C BUS

Acknowledge (ACK) and Not Acknowledge (NACK)

Each data byte on SDA line must be eight bits long and the number of bytes that can be transmitted per transfer section is unlimited.

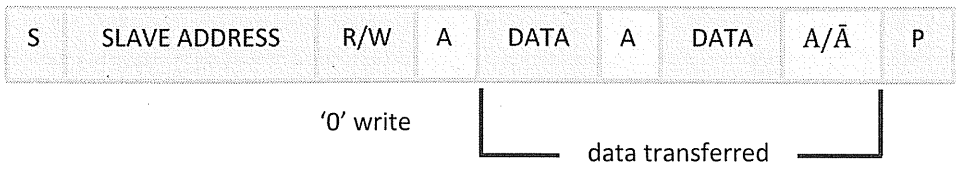
After transmitting-receiving each byte, the master generates a clock pulse (the 9th clock) to take acknowledge bit. The acknowledge bit (ACK) is defined as follows: During the 9th clock pulse, the SDA level is at "L" level. The ACK bit detected after the transmission of a byte means that the slave has received the byte successfully and a next byte can now be transmitted by the master.

The Not Acknowledge bit (NACK) is opposite to ACK. The SDA level will be in "H" level during the 9th clock pulse.

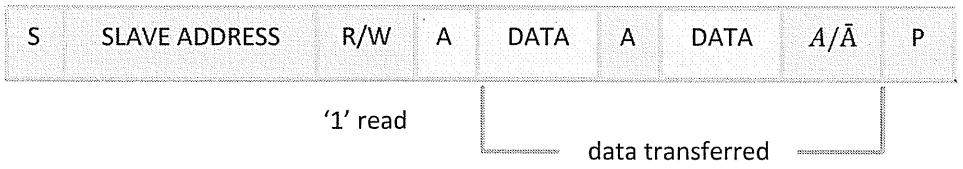
There are 5 conditions that lead to generating an NACK:

- No slave device responds (ACK bit) after slave address has been transmitted.
- Receiving device is unable to transmit or receive.
- During the transfer, receiver gets data or commands that it does not understand.
- During the transfer, the receiver cannot receive any more data bytes.
- The master generates Stop signal to end of the transmission.

Data Transmission (3)



Master transmits to salve (slave address is 7 bits)



Master receives from salve (slave address is 7 bits)

From master to salve

From slave to master

S = Start condition

P = Stop condition

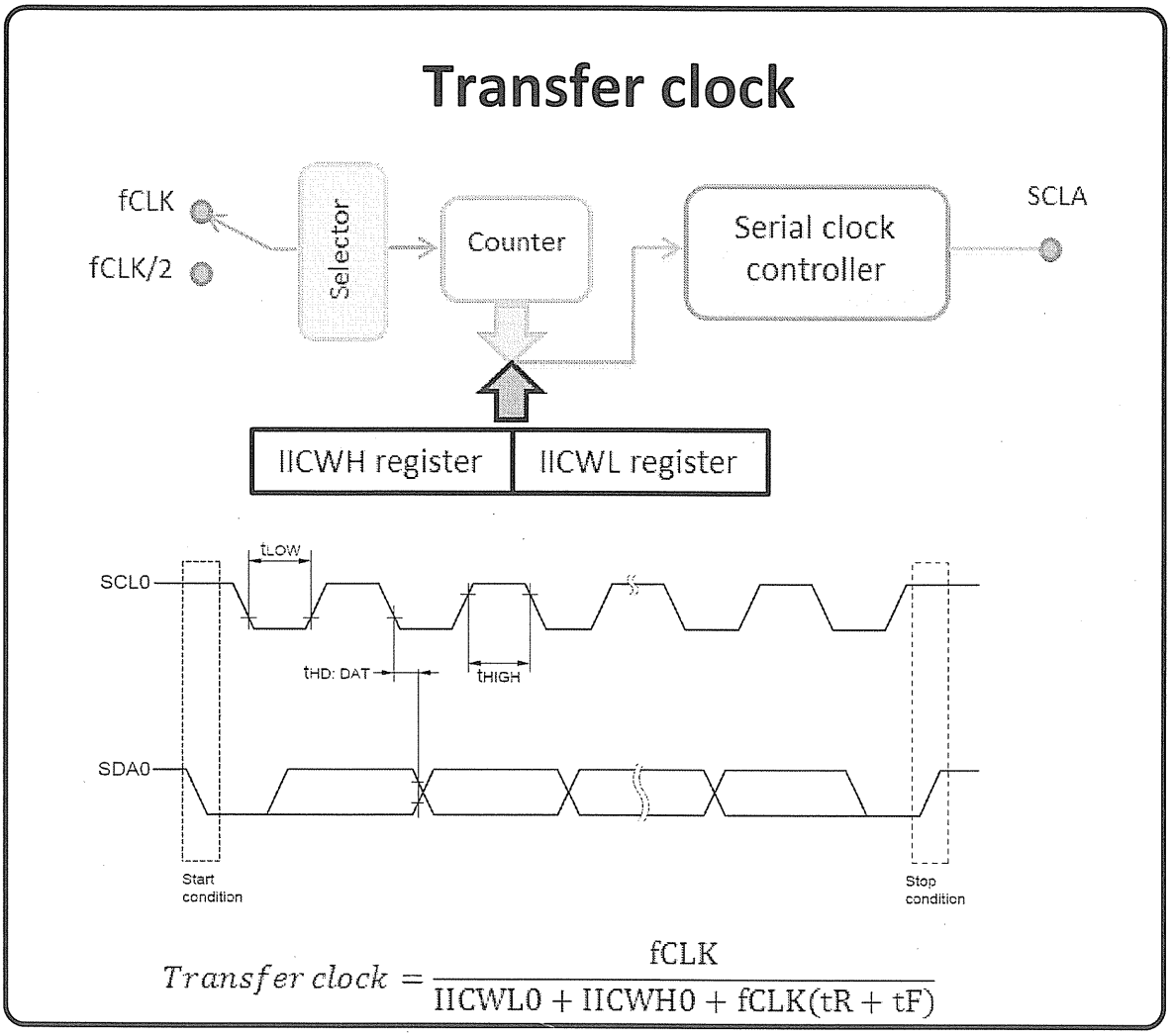
A = Acknowledge

Ā= Not acknowledge

7-bit addressing mode

After the START condition, a slave address is sent out. This address is 7 bits long followed by an 8th bit which is a data direction bit (R/W): 'Zero' indicates a transmission (WRITE); 'one' indicates a request for data (READ).

Data is transferred until a STOP condition is generated by the master. The target slave address can be changed when the master generates Restart signal.



IICA high-level width setting register - IICWH

This register is used to set the high-level width (tHIGH) of the SCLA pin signal that is output by serial interface IICA.

IICA low-level width setting register - IICWL

This register is used to set the low-level width (tLOW) and data hold time (tHD:DAT) of the SCLA pin signal that is output by serial interface IICA.

fCLK

This is the clock frequency of the CPU or peripheral hardware

tR and TF

They are SDA and SCLA signal rising and falling times.

(This page intentionally left blank)

(This page intentionally left blank)

References

1. Micom Embedded Software Course (Renesas)
2. RL78/G14 User Hardware Manual
3. RL78 Family User Software Manual
4. CubeSuite+ V2.01.00 User Manual
5. Renesas Demonstration Kit (RDK) for RL78G14 User Hardware Manual
6. LCD Datasheet (unofficial document from OKAYA – the LCD manufacturer)
7. LCD Driver (Glyph driver, provided in RDK's factory demo)
8. Other resources: Renesas e-learning website (https://elearning.renesas.com)

Revision History

Revision	Date	Notes
Newly created	August 2015	Newly created
1.0	February 2016	Correct typo and mistakes in all chapters, re-structure chapter 9...

Contributors

For any feedback, please contact directly the chapter's creator(s)

Chapter	Creator	Email (@rvc.renesas.com)
1	Khoa Tran	khoa.tran.kc
2	Khoa Tran	
3	Nguyen Nguyen	nguyen.nguyen.ym
4	Quynh Tran	quynh.tran.jy
5	Nguyen Bao Nguyen	nguyen.nguyen.yj
6	Khanh Doan	khanh.doan.eb
7	Hung Tran Thao Nguyen	hung.tran.jy thao.nguyen.yb
8	Khanh Do Phong Huynh	khanh.do.te phong.huynh.eb
9	Khanh Nguyen Tien Le	khanh.nguyen.yj tien.le.px