

MCU Car Kit, Ver. 5.1

Program Explanation Manual

– kit12_rx62t Version

(Version for RX62T)

Version 1.00 [ANDTR101]

March 2014

Renesas MCU Car Rally Secretariat

Important Notice (Revision 1.2)

Copyright

- Copyright of this manual and its contents belongs to the Renesas MCU Car Rally Secretariat.
- This manual is protected under copyright law and international copyright conventions.

Prohibited Use

The user is prohibited from doing any of the following:

- Sale of the manual to a third party, or advertisement, use, marketing, or reproduction of the manual for purpose of sale
- Transfer or reauthorization to a third party of usage rights to the manual
- Modification or deletion of the contents of the manual, in whole or in part
- Translation into another language of the contents of the manual
- Use of the contents of the manual for a purpose that may pose a danger of death or injury to persons

Reprinting and Reproduction

Prior written permission from the Renesas MCU Car Rally Secretariat is required in order to reprint or reproduce this manual.

Limitation of Liability

Every effort has been made to ensure the accuracy of the information contained in this manual. However, the Renesas MCU Car Rally Secretariat assumes no responsibility for any loss or damage that may arise due to errors this manual may contain.

Other

The information contained in this manual is current as of the date of publication. The Renesas MCU Car Rally Secretariat reserves the right to make changes to the information or specifications contained in this manual without prior notice. Make sure to check the latest version of this manual before starting fabrication.

Contact Information

MCU Car Rally Secretariat, Renesas Solutions Corp.
MN Building, 2-1 Karuko-saka, Ageba-cho, Shinjuku-ku, Tokyo, 162-0824, Japan
Tel. (03) 3266-8510
E-mail: official@mcr.gr.jp

Contents

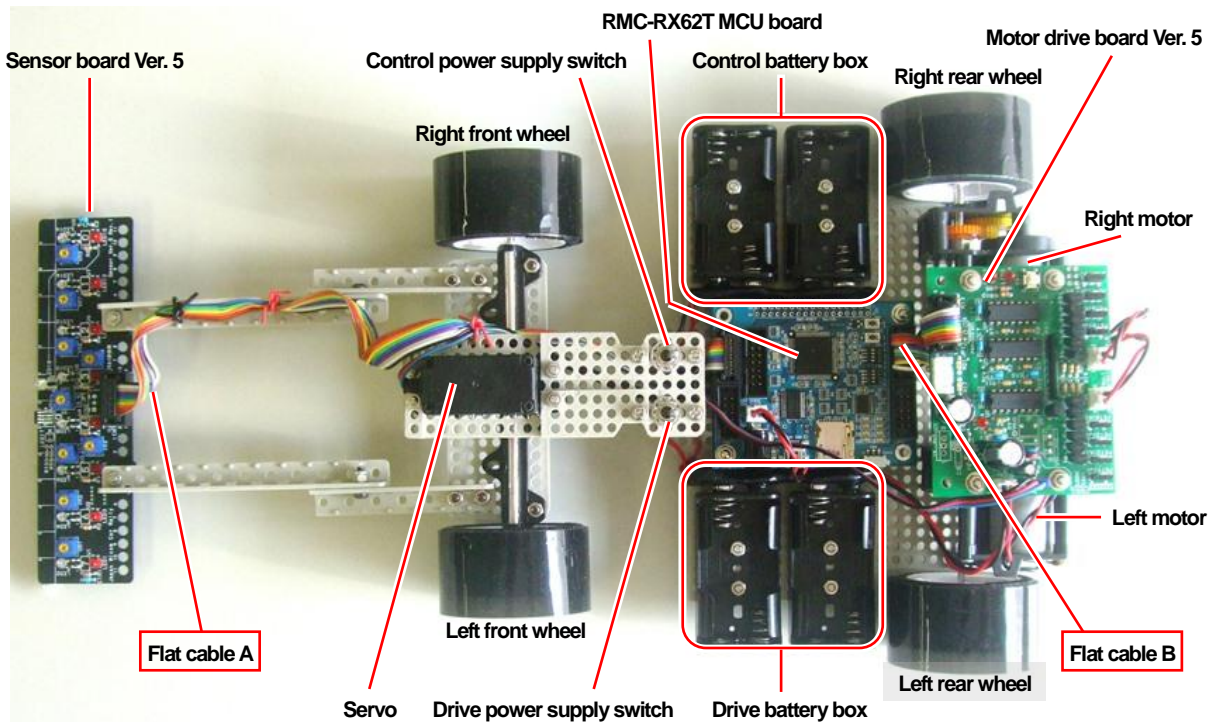
1. Overview of MCU Car Rally Kit Ver. 5.1	1
1.1. Exterior View of MCU Car Rally Kit Ver. 5.1	1
1.2. Power Supply Configuration of Standard Kit	2
1.3. Power Supply Configuration with Boosted Drive Voltage.....	3
2. Sensor Board Ver. 5	5
2.1. Specifications.....	5
2.2. Circuit Diagram	6
2.3. Dimensions	7
2.4. Sensor Mounting Positions	7
2.5. Exterior View.....	8
2.6. Relationship between the Sensor Board Ver.5 CN1 and the RMC-RX62T Board.....	9
2.7. Method of Distinguishing Between White and Black Portions of the Course	10
2.8. Method of Determining Whether Start Bar Is Open or Closed	10
2.9. Output signals of U8 and U9	11
2.10. Operating Principle of Circuit.....	12
2.11. Sensor Adjustment Procedure	13
3. Motor Drive Board Ver. 5	16
3.1. Specifications.....	16
3.2. Circuit Diagram	17
3.3. Dimensions	18
3.4. External Appearance	19
3.5. Relationship between the Motor Drive Board Ver. 5 CN2 and the RMC-RX62T Board.....	21
3.6. Motor Control	22
3.6.1. Role of the Motor Drive Board	22
3.6.2. Operating Principle of Speed Control	22
3.6.3. Operating Principle of Forward and Reverse	23
3.6.4. Brake and free	24
3.6.5. H-bridge circuit	25
3.6.6. Using FETs as the Switches in an H-Bridge Circuit.....	25
3.6.7. P-Channel and N-Channel Short FETs Prevention Circuit.....	29
3.6.8. Free Circuit.....	32
3.6.9. Actual Circuit.....	33
3.6.10. Operation of Left Motor	35
3.6.11. Operation of Right Motor.....	35
3.7. Servo Control.....	36
3.7.1. Operating Principle	36
3.7.2. Circuit.....	37
3.8. LED Control	37
3.9. Pushbutton Control	38
4. Sample Programs	39
4.1. Program Development Environment	39
4.2. Installing the Sample Programs	39
4.3. Opening the kit12_rx62t Workspace.....	41
4.4. Project.....	42

5.	Program Explanation – kit12_rx62t.c.....	43
5.1.	Program Code Listing.....	43
5.2.	Differences between programs for kit07_rx62t.c and kit12_rx62t.c.....	53
5.3.	On-Chip Peripheral Functions of RX62T MCU Used by the Program.....	53
5.4.	Program Explanation	54
5.4.1.	Start	54
5.4.2.	Including External Files	54
5.4.3.	Symbol Definitions	54
5.4.4.	Prototype Declarations	56
5.4.5.	Global Variable Declarations.....	57
5.4.6.	init Function(Clock Choice)	58
5.4.7.	init Function (Port I/O Settings)	58
5.4.8.	init Function (Compare Match Timer Settings)	61
5.4.9.	init Function (Multi-Function Timer Pulse Unit 3 Settings)	62
5.4.10.	Excep_CMT0_CMI0 Function (Interrupt Every 1 ms)	63
5.4.11.	timer Function (Pause)	64
5.4.12.	sensor_inp Function (Read State of Sensors).....	65
5.4.13.	check_crossline Function (Crossline Detection).....	72
5.4.14.	check_rightline function (Right Half Line Detection).....	74
5.4.15.	check_leftline function (Left Half Line Detection).....	75
5.4.16.	dipsw_get Function (Reading DIP Switches).....	76
5.4.17.	buttonsw_get Function (Reading the Pushbutton State in MCU board).....	77
5.4.18.	pushsw_get Function (Reading the Pushbutton State)	78
5.4.19.	startbar_get Function (Reading the Start Bar Detection Sensor)	79
5.4.20.	led_out_m Function (LED Control in MCU board).....	80
5.4.21.	led_out Function (LED Control)	81
5.4.22.	motor Function (Motor Speed Control).....	82
5.4.23.	handle Function (Servo Steering Operation)	88
5.4.24.	Start	90
5.4.25.	Patterns.....	91
5.4.26.	Writing a Program	91
5.4.27.	Pattern Descriptions	93
5.4.28.	Initial while and switch when Using Patterns	94
5.4.29.	Pattern 0: Wait For Button Input	95
5.4.30.	Pattern 1: Check if Start Bar Is Open	97
5.4.31.	Pattern 11: Normal Trace.....	99
5.4.32.	Pattern 12: Check End of Large Turn to Right.....	110
5.4.33.	Pattern 13: Check End of Large Turn to Left	114
5.4.34.	Crank Overview	119
5.4.35.	Pattern 21: Processing at 1st Crossline Detection	120
5.4.36.	Pattern 23: Trace, Crank Detection After Crossline	122
5.4.37.	Patterns 31 and 32: Clearing from Left Crank	125
5.4.38.	Patterns 41 and 42: Right Crank Clearing Processing.....	129
5.4.39.	Right Lane Change Outline.....	133
5.4.40.	Pattern 51: Processing at 1st Right Half Line Detection	134
5.4.41.	Pattern 53: Trace after Right Half Line	137
5.4.42.	Pattern 54: Right Lane Change End Check.....	139
5.4.43.	Left Lane Change Outline	141
5.4.44.	Processing at 1st Left Half Line Detection.....	142
5.4.45.	Pattern 63: Trace after Left Half Line	145

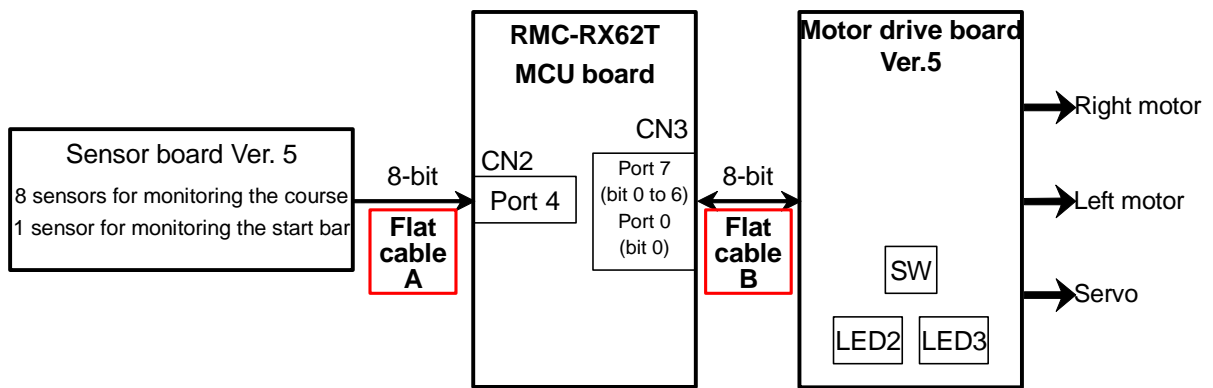
5.4.46. Pattern 64: Left Lane Change End Check	147
6. Adjusting the Servo Center and Maximum Turn Angle	149
6.1. Outline	149
6.2. Install the communication program Tera Term	150
6.3. Adjusting the Servo Center	154
6.4. Determining the Maximum Turning Angle of the Servo	162
6.5. Overwriting the kit12_62t.c Program Code	168
7. Hints on Modifying the Program	170
7.1. Outline	170
7.2. Examples of the MCU Car Going Off the Track	171
7.2.1. Crossline Not Detected Correctly	171
7.2.2. Crank Not Detected Correctly	172
7.2.3. Half Line Not Detected Correctly	174
7.2.4. After Clearing from Crank, MCU Car Mistakes Outer White Line for Center Line and Goes off Track..	175
7.2.5. End of Lane Change Not Detected Correctly	178
7.3. Conclusion	179
8. Calculating the Left-Right Motor Speed Differential.....	180
8.1. Calculation Method.....	180

1. Overview of MCU Car Rally Kit Ver. 5.1

1.1. Exterior View of MCU Car Rally Kit Ver. 5.1



The MCU Car Rally Kit Ver. 5.1, comprises a control system consisting of the RMC-RX62T board (a MCU board with an RX62T MCU mounted on it), the sensor board Ver. 5, and the motor drive board Ver. 5, and a drive system consisting of the right motor, the left motor, and the servo.

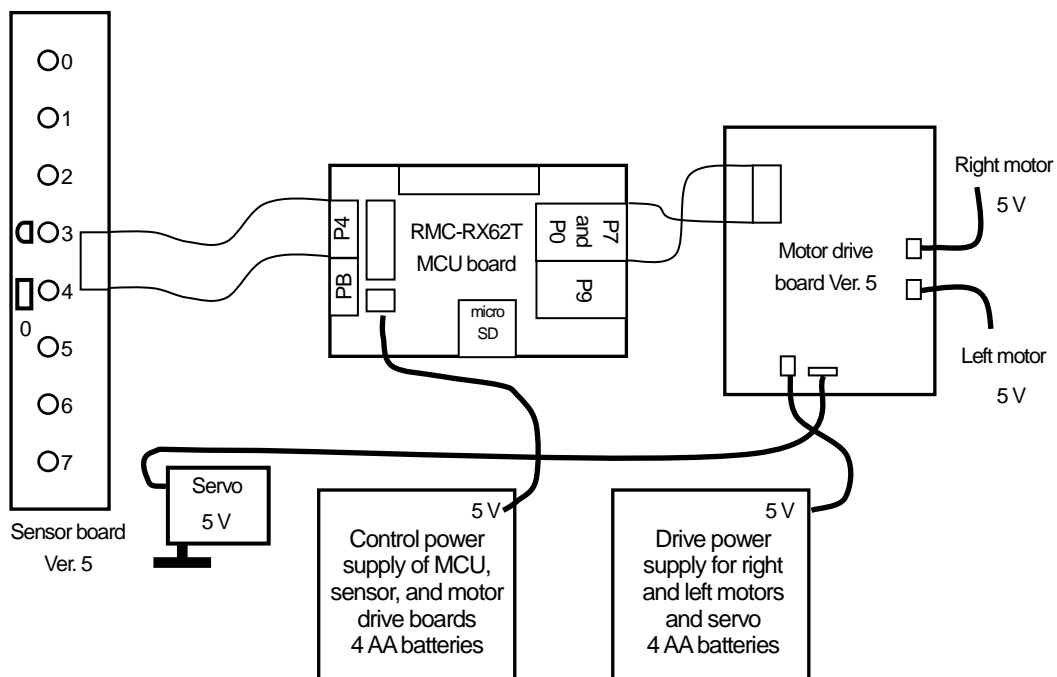


RMC-RX62T MCU board	The MCU board reads the state of the sensors via port 4, calculates the PWM output values for the right and left motors and the turn angle for the servo, and outputs this data to the motor drive board connected to port 7. The manner in which output values for the motors and servo are decided, based on the state of the sensors, is determined by the software program.
Sensor board Ver. 5	8 sensors which detect state of the course are here. They output “0” if bottom of the sensor is white, and output “1” if it is black. ※The program inverts the signal, and it judges white is “1” and black is “0”. There is one sensor which detects if there is the start bar or not. It outputs “0” if the start bar is present, and output “1” if it is absent. ※The rightmost course state detection sensor and start bar detection sensor have an OR relationship connected to bit 0. The rightmost sensor is initially not responding because only the middle of the board should be able to detect the start line, leaving the board able to judge the state of the start bar. After the start, the board does not to look for the start bar and therefore can instead detect the track using the rightmost sensor on the board’s underside.
Motor drive board Ver. 5	The motor drive board converts low-power signals from the MCU board into high-power signals that operate the motors. The motor power supply is also used to drive the servo. A pushbutton is connected to the motor drive board, and the software program is written such that pressing this button starts the MCU car. There are also two LEDs mounted on the motor drive board for debugging.
Batteries	<ul style="list-style-type: none"> Control (MCU) power supply: Four size AA rechargeable batteries (1.2 V × 4 = 4.8 V) are used. Drive (motor and servo) power supply: Four size AA rechargeable batteries or four size AA alkaline batteries (1.5 V × 4 = 6.0 V) are used. Note: Ensure that the voltage of the control system is 4.0 V to 5.5 V.

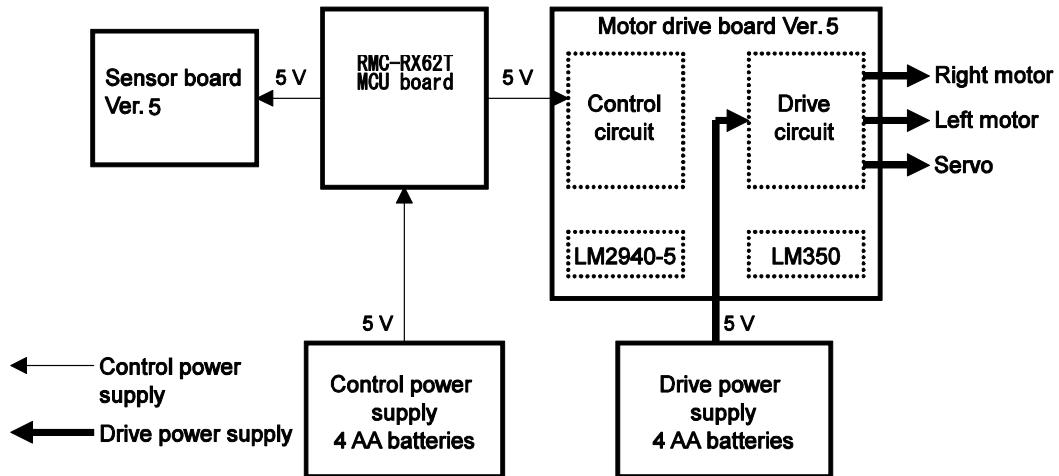
1.2. Power Supply Configuration of Standard Kit

The standard kit uses separate power supplies for the control and drive blocks. This ensures that the MCU will not be reset due to low power no matter how much current the motors and servo consume.

The power supply configuration of the standard kit is shown below.



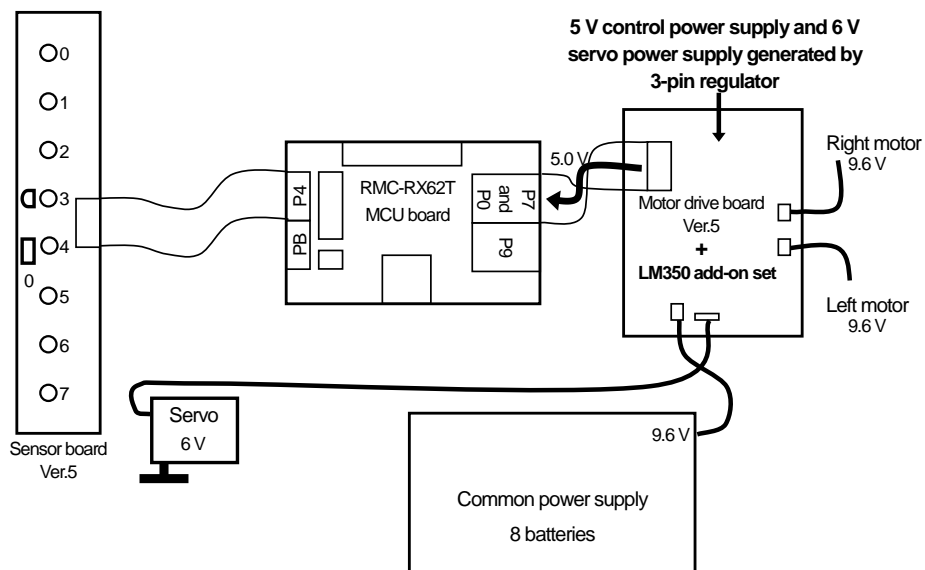
The flow of current from the power supplies is shown below.



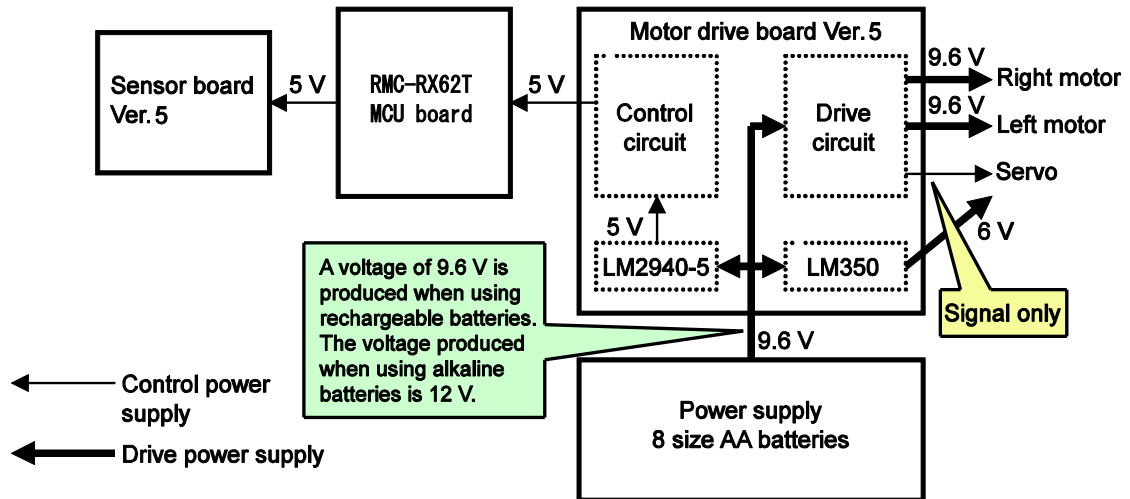
1.3. Power Supply Configuration with Boosted Drive Voltage

It is possible to increase the speed of the motors by boosting the drive voltage (increasing the number of batteries). Using six batteries as the motor power supply increases the voltage to 7.2 V, and using eight batteries increases it to 9.6 V. Note, however, that the maximum number of batteries allowed is eight. This means it is necessary for the control and drive blocks to use a common power supply. Applying a voltage of 9.6 V to the motors will not wreck them (though it is not really desirable, since their rated voltage is 6 V), **but the MCU has a guaranteed operating voltage range of 4.0 V to 5.5 V and applying a voltage exceeding 5.5 V will cause it to stop operating. (The absolute maximum voltage rating is 6.5 V. Applying a voltage in excess of 6.5 V will destroy the MCU.) In like manner, the voltage applied to the servo must not exceed 6 V.** It is therefore necessary to install a three-pin regulator to bring the MCU and servo voltage down to the rated level. Note that when a common power supply is used, the MCU will be reset if the voltage drops below 4.0 V due to large current consumption by the motors, etc. It is necessary to be careful regarding MCU resets when using a common power supply.

When the LM350 add-on set is installed and a power supply voltage of 6 V or greater is used, the LM2940-5 generates a 5 V voltage for the control block, including the MCU, and the LM350 generates a 6 V voltage for the servo.



The flow of current from the power supplies is shown below.



Note: The springs in the battery box of the kit are weak, and this can cause the battery terminals to become disconnected from the battery box contacts when the MCU car accelerates or decelerates. This can reset the MCU (due to a disconnection of several tens of milliseconds). This problem can be prevented by using a battery box that holds the batteries firmly in place.

2. Sensor Board Ver. 5

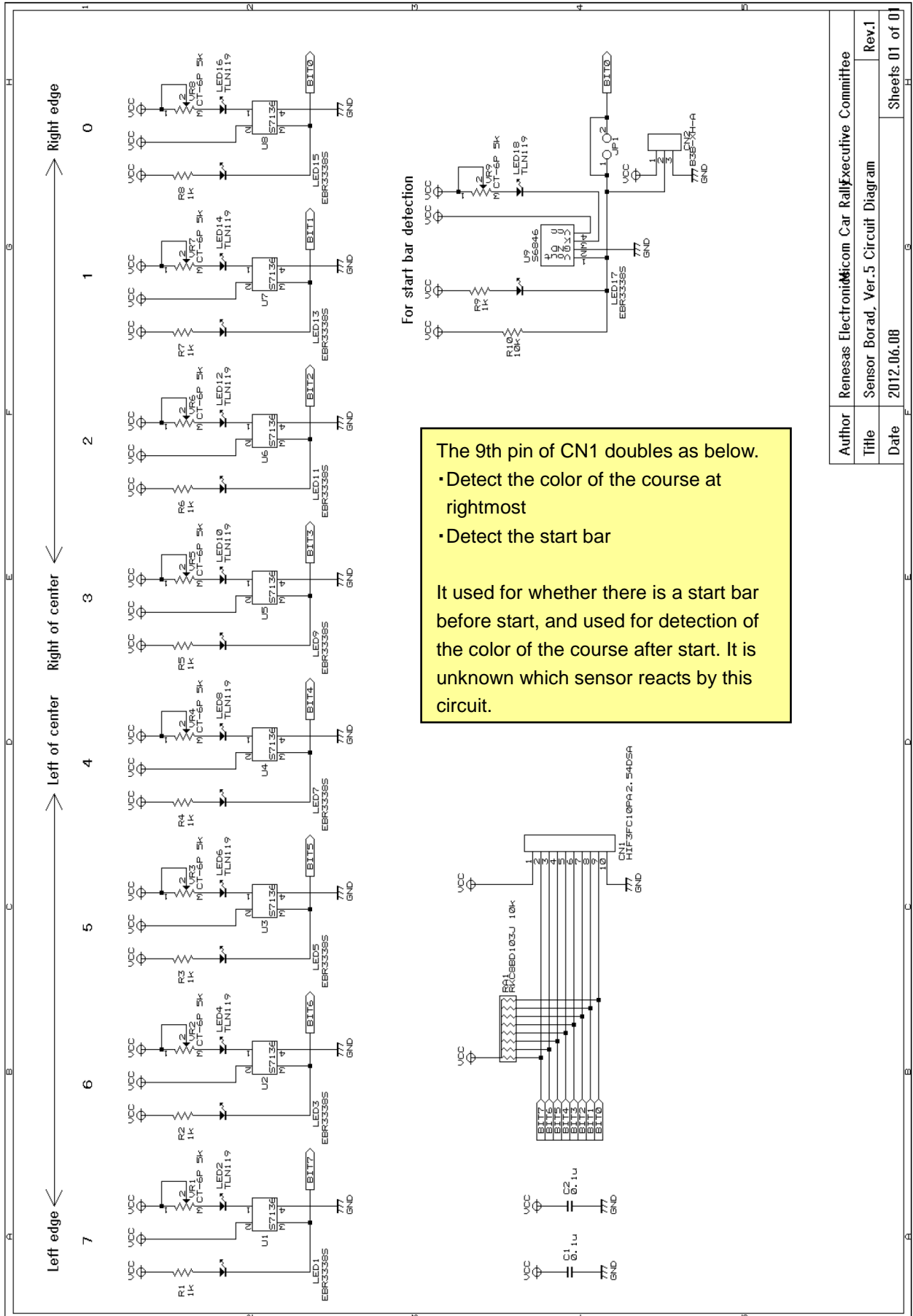
2.1. Specifications

The specifications of the sensor board Ver. 5 are shown below.

Name	Sensor Board Ver. 5
Abbreviation	Sensor Board 5
Contained in kit	MCU car kit Ver. 5.1
Date released for sale	Jun 2013 (Still available as of September 2013.)
Number of boards	1
Number of sensors for monitoring course	8
Number of sensors for monitoring start bar	1
Signal inverter circuit	None (inversion performed by software program)
Connections to MCU board	RX62T: Port4
Voltage	DC5.0 V \pm 10%
Weight (actual measured weight of completed board)	Approx. 18 g
Resist (board colour)	Black
Board dimensions	W 140 \times D 38 \times T 1.2 mm
Dimensions (actual measured dimensions)	Max. W 140 \times D 38 \times H 14 mm

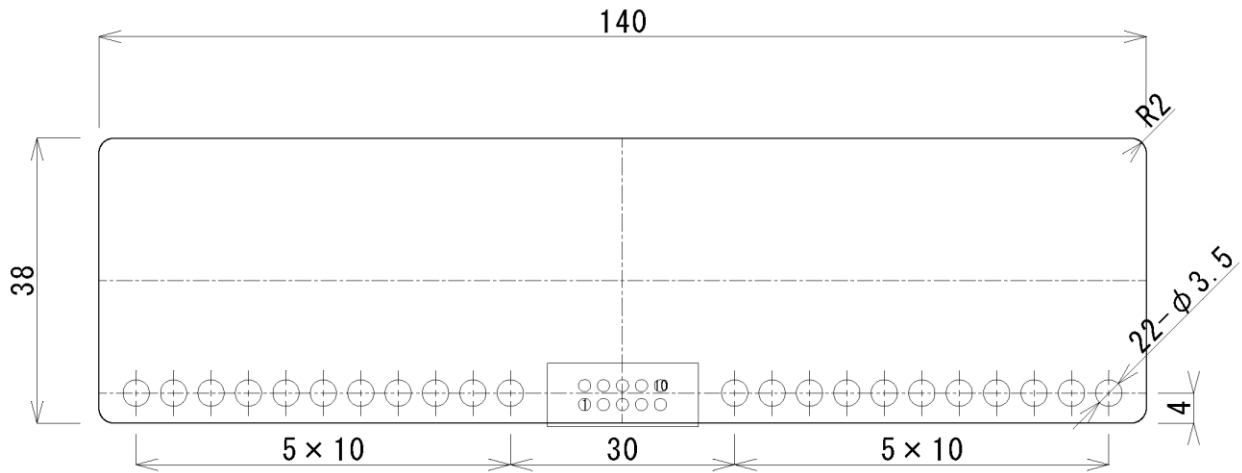
Note: The weight will vary depending on factors such as the length of the lead wires and the amount of solder used.

2.2. Circuit Diagram



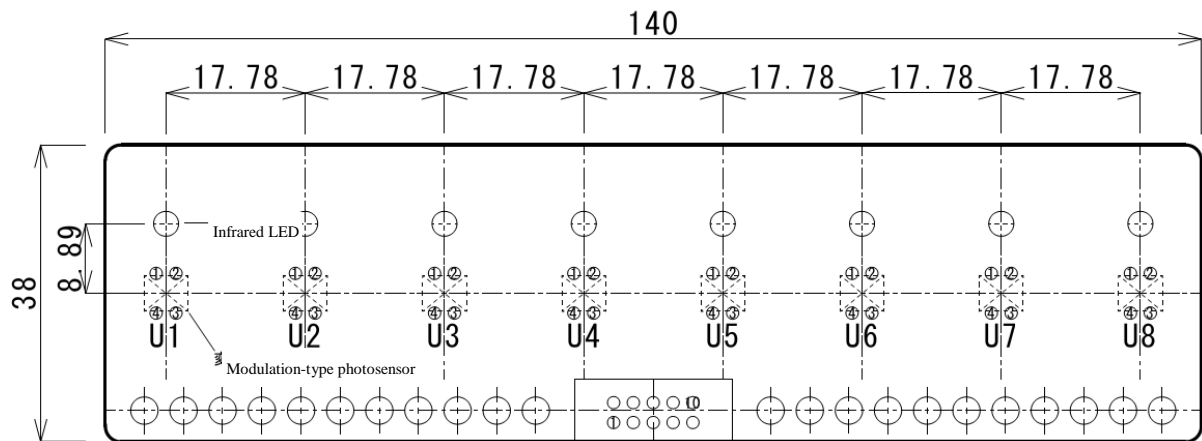
2.3. Dimensions

The sensor board has a total of 22 mounting holes, 11 on the right and 11 on the left. These holes are used to secure the sensor board in place.

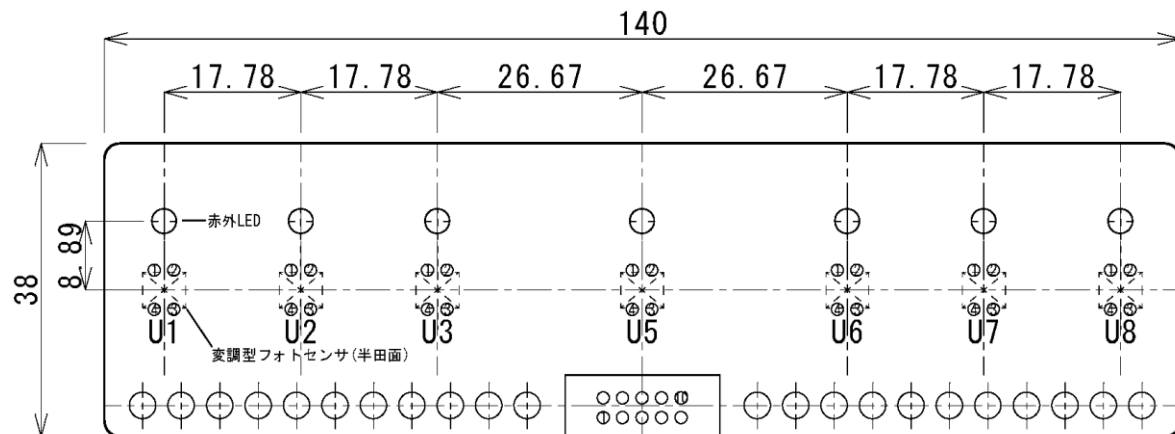


2.4. Sensor Mounting Positions

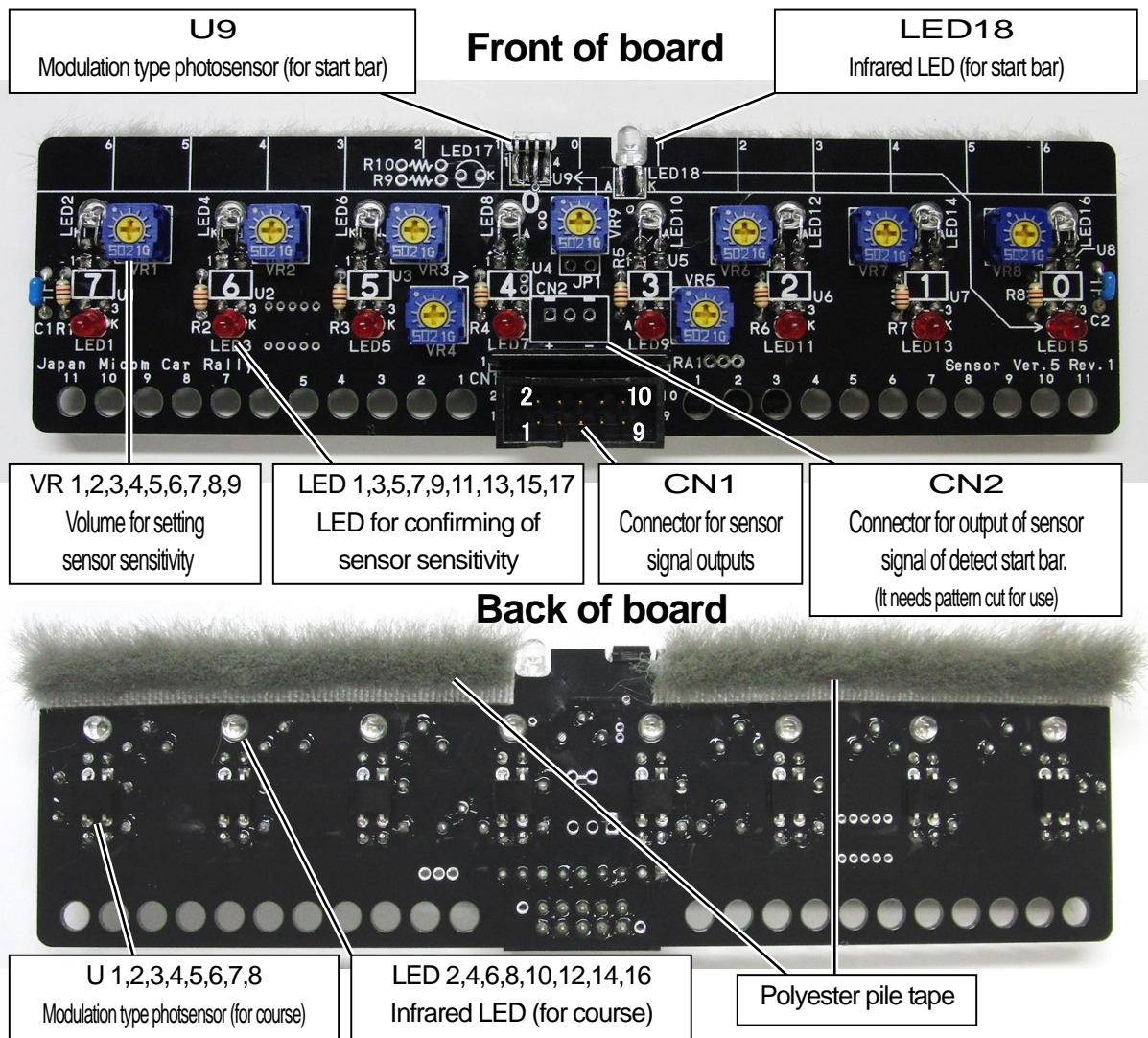
Eight sensors are used to detect the black and white portions of the course. These are mounted on the board in the positions indicated below.



Note: Sensor mounting positions on sensor board Ver.5 for reference



2.5. Exterior View



T

The following shows the connection of connectors and the content of signals:

Parts number	Item	Description
CN1	Connector (connect to MCU board)	Item reference
CN2	Connector (option)	The signal of rightmost of the course state detection sensor doubles as the signal of the start bar detection sensor. It can let CN2 become independent and output the signal of the start bar detection sensor. In addition, it needs to implement the parts (R9, R10, LED17) to let the start bar detection sensor become independent. For more details, refer to Sensor Board Ver.5 Assembly Manual.
LED2,4,6,8,10,12,14,16	Infrared LED	The TLN119 element is used. It emits infrared light. Since the light emitted is in the infrared range, it is not visible to humans. There are eight infrared LEDs for course detection.
LED18	Infrared LED	The TLN119 element is used. There is an infrared LED for start bar detection..
U1,2,3,4,5,6,7,8	Modulation type photosensor	The S7136 element (for course) from Hamamatsu Photonics K.K. is used. Light emitted by the infrared LED is picked up by this element. When infrared light is detected, the current portion of the course is determined to be white. When no infrared light is detected, the current portion of the course is determined to be black. There are eight modulation type photosensors.

U9	Modulation type photosensor	The S6846 element (for start bar) from Hamamatsu Photonics K.K. is used. Light emitted by the infrared LED is picked up by this element. When infrared light is detected, it is determined that there is a start bar present. When no infrared light is detected, it determined that there is no start bar present.
VR1,2,3,4,5,6,7,8	Volume for adjusting sensor sensitivity (for course)	The amounts of light output from infrared LEDs are adjusted in these volumes. Some portions of the MCU car course are grey. By adjusting the sensitivity with the volume, it is possible to make the grey areas be detected as white or as black. The standard software program assumes that grey areas will be detected as white.
VR9	Volume for adjusting sensor sensitivity (for start bar)	The amount of light output from LED18 is adjusted using these volumes. If there is a start bar, it becomes white. If there is not a start bar, there will be no reflection. Adjust this volume to react (for turn lights LED15) when there is a start bar.
LED1,3,5,7,9,11,13,15	LED for confirming sensor sensitivity	The LED lights when white is detected and is dark when black is detected. The LED is used for confirmation when adjusting the sensitivity with the variable resistor.
—	Polyester pile tape	Polyester pile tape is mounted on the solder side of the sensor board and is made a constant height so as to not rub the course and the sensor directly and also to allow the sensor to react appropriately.

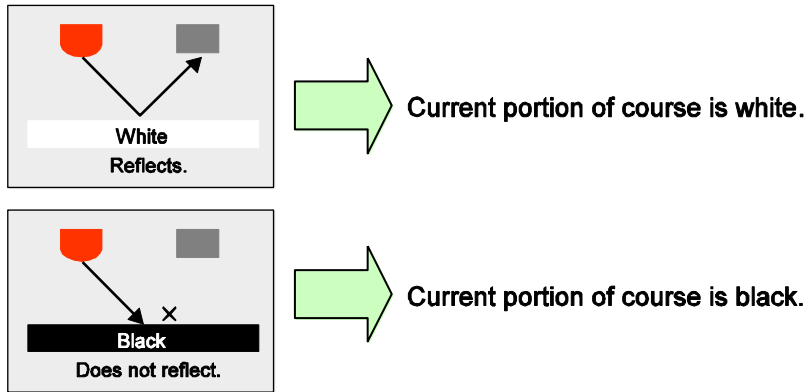
2.6. Relationship between the Sensor Board Ver.5 CN1 and the RMC-RX62T Board

The Sensor Board Ver. 5 connector CN1 and the RMC-RX62T board connector CN2 (port 4) must be connected with a 10-pin flat cable. The following table lists the signals carried by this cable.

RMC-RX62T Board CN2	Signal Direction	Motor Driver Board Ver.5 CN1	Description
Pin 1 (+5V)	—	Pin 1	It provides +5V \pm 10% (4.5~5.5V) to the circuits of the Sensor Board Ver.5.
Pin 2 (P47)	←	Pin 2	Inputs a signal from U1 (The first sensor from left) "0": White (LED1 on) "1": Black (LED1 off)
Pin 3(P46)	←	Pin 3	Inputs a signal from U2 (The second sensor from left) "0": White (LED3 on) "1": Black (LED3 off)
Pin 4(P45)	←	Pin 4	Inputs a signal from U3 (The third sensor from left) "0": White (LED5 on) "1": Black (LED5 off)
Pin 5(P44)	←	Pin 5	Inputs a signal from U4 (The fourth sensor from left) "0": White (LED7 on) "1": Black (LED7 off)
Pin 6(P43)	←	Pin 6	Inputs a signal from U5 (The fourth sensor from right) "0": White (LED9 on) "1": Black (LED9 off)
Pin 7(P42)	←	Pin 7	Inputs a signal from U6 (The third sensor from right) "0": White (LED11 on) "1": Black (LED11 off)
Pin 8(P41)	←	Pin 8	Inputs a signal from U7(The second sensor from right) "0": White (LED13 on) "1": Black (LED13 off)
Pin 9 (P40)	←	Pin 9	Inputs a signal from U8 (The first sensor from right) and a signal from U9 (Sensor which detect start bar). "0": White (LED15 on) "1": Black (LED15 off) or "0": There is start bar (LED15 on) "1": none (LED15 off). Because the car initially sits on the middle of the course, U8 (The first sensor from right) detects black at the start. The data from U9 (start bar) is then checked. After the start the start bar will not be present and the activity of U8(course) is used.
Pin10 (GND)	—	Pin 10	GND

2.7. Method of Distinguishing Between White and Black Portions of the Course

The sensor board is equipped with eight pairs of elements, each pair comprising one element that shines infrared light onto the course and one element that detects reflected infrared light. The system makes use of the fact that white areas reflect light and black areas absorb it. The emitter element is used to shine infrared light onto the course. When this infrared light is reflected back and detected by the receiver element, the current portion of the course is determined to be white. When no infrared light is detected, the current portion of the course is determined to be black.

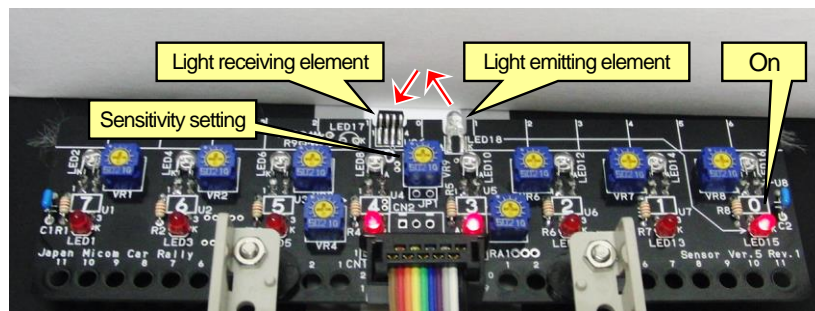


The amount of infrared light emitted can be adjusted by using a variable resistor. Some portions of the MCU car course are grey. By adjusting the sensitivity with the variable resistor, it is possible to make the grey areas be detected as white or as black. **The standard software program assumes that grey areas will be detected as white.**

2.8. Method of Determining Whether Start Bar Is Open or Closed

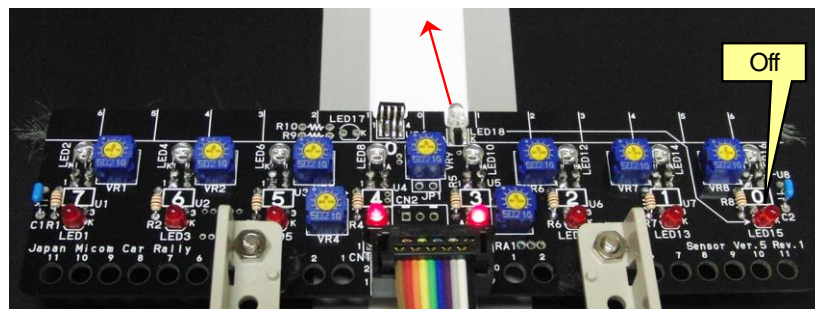
Initially, the white start bar is closed. An infrared LED and S6846 (modulation type photo sensor) are mounted on the board facing forward. The following is determined based on the sensor state.

● Start bar closed



Light is reflected
↓
Start bar is there

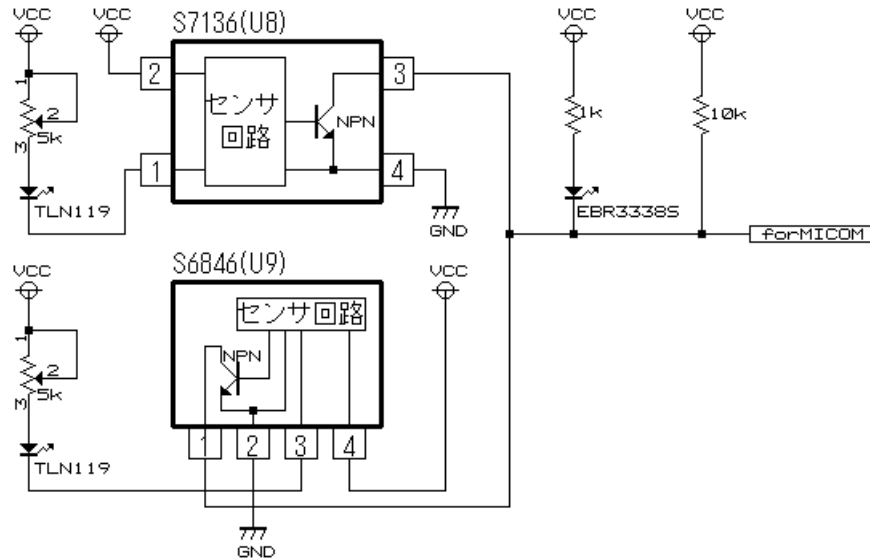
● Start bar open



No light is reflected
↓
No start bar there

2.9. Output signals of U8 and U9

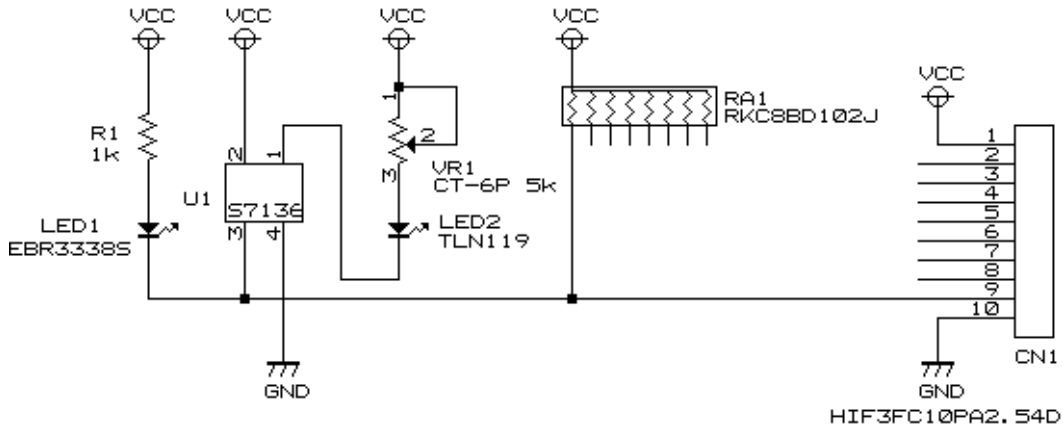
Output of sensor is open collector output, and connects NPN type transistor (type 2SC). Pin 9 of CN1 doubles as output of the sensor which detects the rightmost course sensor (U8) and the sensor which detects the start bar, as in the circuit below.



The behaviour of 2 sensors and output signals are as shown below.

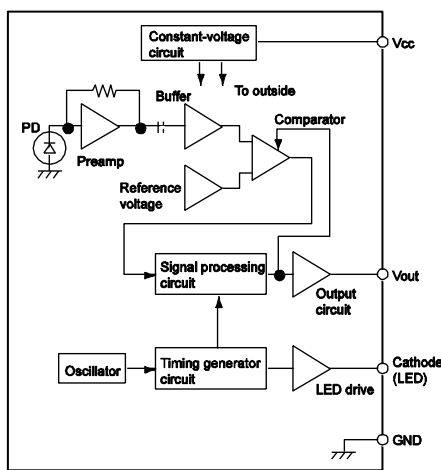
Sensor which detect the course	Sensor which detect the start bar	Circuit	Output
			Mind
Black	absent		open (High impedance) In fact, it outputs 5V after being pulled up.
Black	present		0V The start bar detection sensor is judged to be present if it is 0V before start.
White	absent		0V The course state detection judges the surface to be white if it is 0V after start.
White	present		0V Never both states – OR system.

2.10. Operating Principle of Circuit



1. U1 is a photo sensor. It combines a light receiver and an infrared LED oscillator circuit.
2. Pin 1 of U1 is connected to an infrared LED (LED2). The infrared light emitted by LED2 is received by U1. VR1 is used to adjust the brightness of the infrared LED.
3. The signal indicating whether or not infrared light is being received is output on pin 3 of U1. This pin is connected to an LED (LED1), providing a visual confirmation of whether the signal value is 0 or 1.
4. When light from the infrared LED reaches U1 (course white), 0 is output. The anode of the LED is positive and the cathode is negative, causing the LED to light.
5. When no light from the infrared LED reaches U1 (course black), 1 is output. (See below for details.) The anode of the LED is positive and the cathode is also positive, so the LED is dark.
6. It is stated above that 1 is output when no light reaches U1, but in fact pin 3 of U1 is an open collector output. “Open collector output” means a value of 0 = 0 V and any other value is open, a state in which the pin is not connected to anything. In the digital world, there are no values other than 0 and 1. Therefore, a resistor (RA1) is used to pull up the signal, resulting in a value of 1 when the photosensor is open.

Note: Operating Principle of Modulation — Type Photosensor (S7136) for Reference (from the Product Data Sheet)

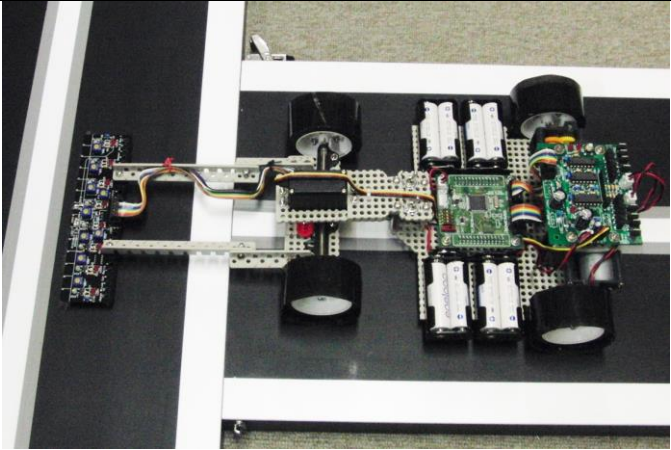
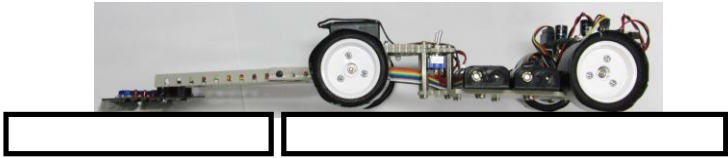


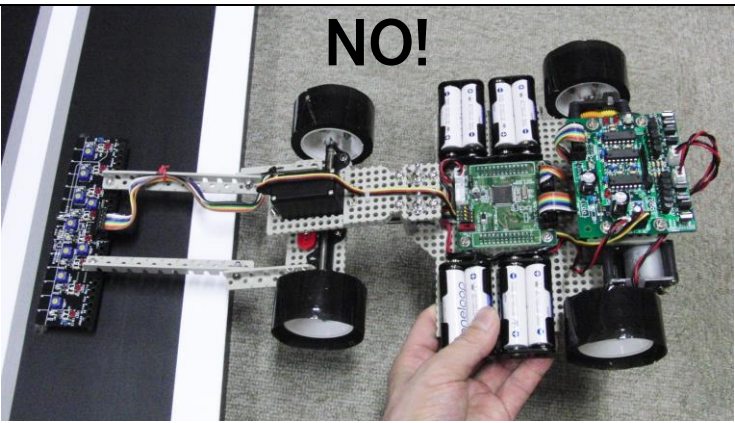
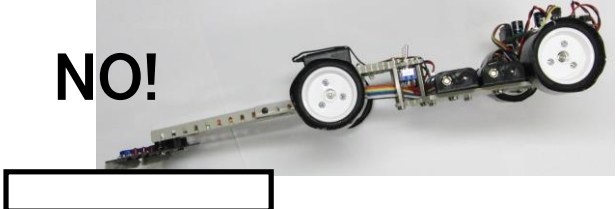
Truth Value Table	
Input	Output Level
LED on	LOW
LED off	OFF

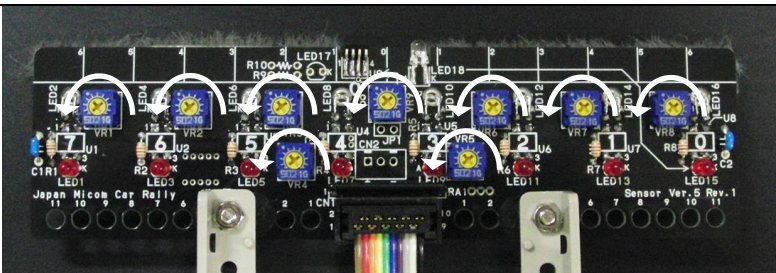
KPIC0002JA

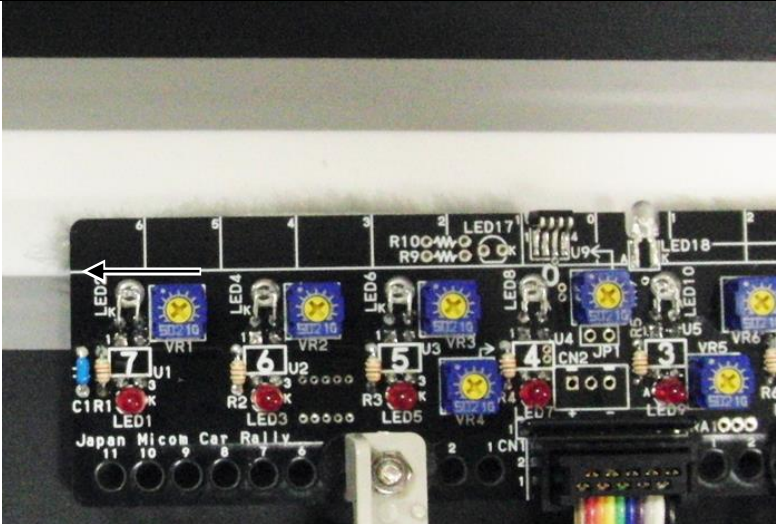
- (a) Oscillator and Timing Signal Generator Circuit
The reference oscillator output is obtained by charging and discharging the built-in capacitor with a constant current. The oscillator output is input to the timing signal generator circuit, which produces the LED drive pulses and the timing pulses used for digital signal processing.
- (b) LED Drive Circuit
This circuit uses the LED drive pulses produced by the timing signal generator circuit to drive a light emitting diode. The drive duty ratio is 1/16.
- (c) Photodiode and Preamp Circuit
The photodiode is of the on-chip type. The photoelectric current from the photodiode is converted into a voltage by the preamp circuit. An independent AC amplifier circuit is used as the preamp circuit. In addition to expanding the dynamic range through increased tolerance for DC and low-frequency ambient light, it boosts the signal detection sensitivity.
- (d) C-Coupling, Buffer Amplifier, and Reference Voltage Circuit
A C-coupling is used to further remove the effects of low-frequency ambient light and to eliminate the DC offset from the preamp. The signal is boosted to the comparator level by the buffer amplifier, and the comparator-level signal is generated by the reference voltage circuit.
- (e) Comparator Circuit
The comparator circuit has an added hysteresis function to prevent chattering caused by tiny fluctuations in the input light.
- (f) Signal Processing Circuit
The signal processing circuit comprises a gate circuit and a digital integrating circuit. The gate circuit prevents malfunctions due to non-synchronous ambient light by distinguishing the input signal during synchronous detection. Since the gate circuit cannot eliminate synchronous ambient light, the digital integrating circuit does so at a later stage.
- (g) Output Circuit
This circuit buffers the output from the signal processing circuit and outputs it externally.

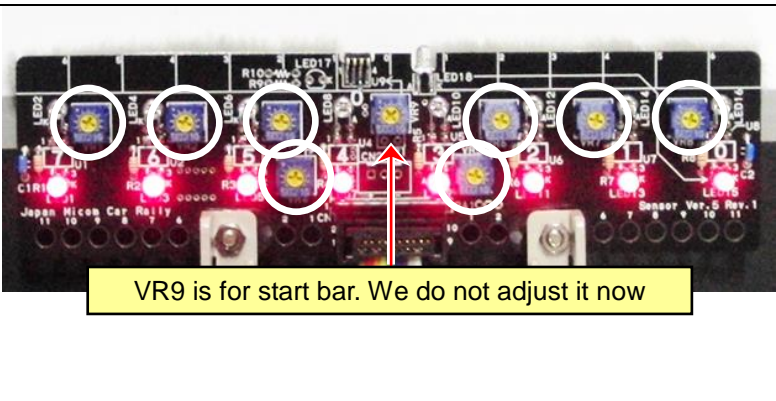
2.11. Sensor Adjustment Procedure

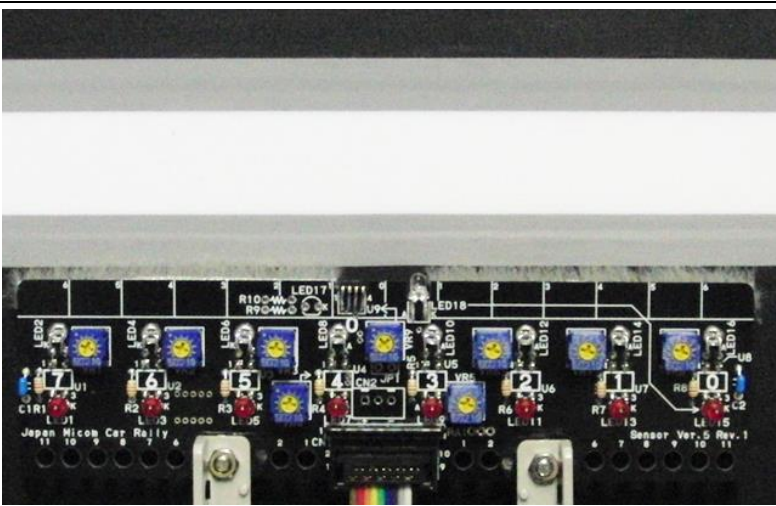
1	 <p>※View from side</p> 	<p>As shown in the photo, place the MCU car such that the edge of the sensor board is parallel with the grey line at the centre of the track. Place the MCU car on a surface that is the same level as the track, just as if it was running on the course.</p>
---	---	---

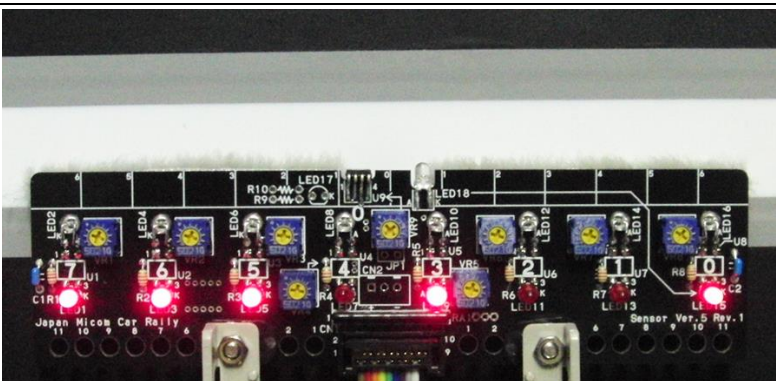
2	 <p>※View from side</p> 	<p>If you try to adjust the sensors by holding the MCU car in your hand as shown here, the results will not be satisfactory because of the unevenness of the gap between the sensors and the track surface. Make sure to place the MCU car on a surface that is the same level as the track</p>
---	--	--

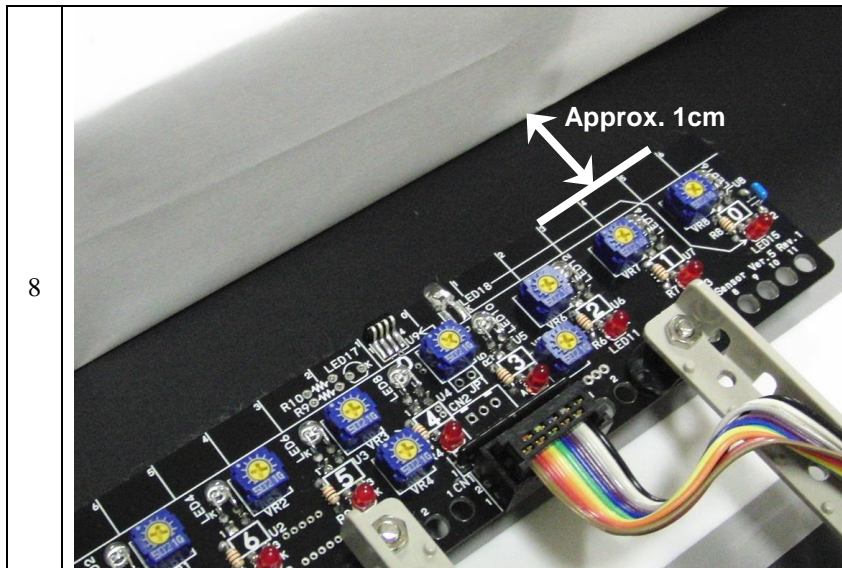
3		<p>Turn all nine of the variable resistors all the way counter clockwise.</p>
---	--	---

4		<p>Align the horizontal line on the board with the line where the white and grey stripes on the track meet. Look straight down from above when doing the alignment.</p>
---	--	--

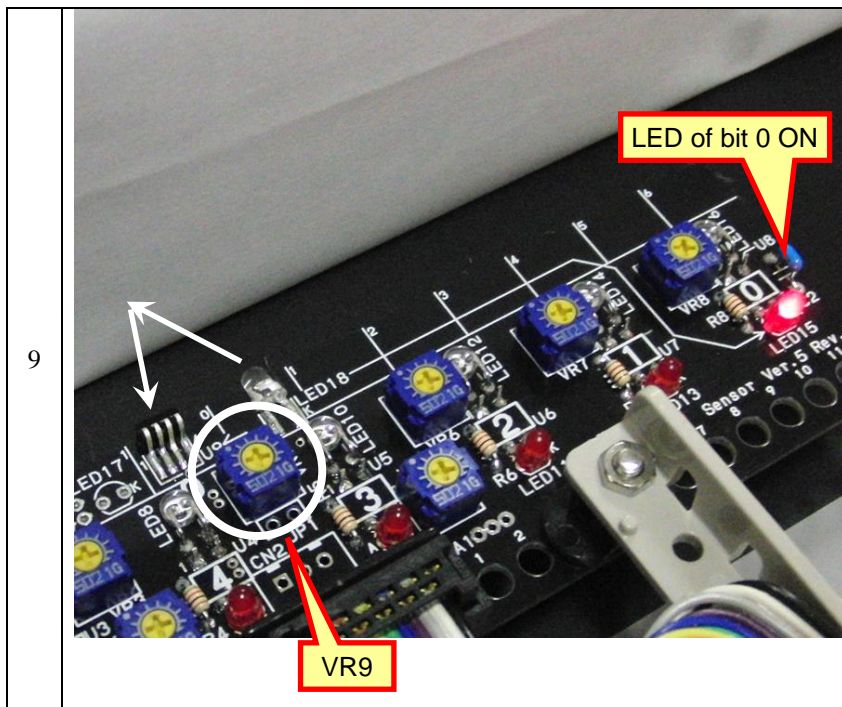
5		<p>Turn each of the eight variable resistors clockwise until the corresponding LED lights. Turn each variable resistor slowly and stop turning the moment the LED lights. Next, adjust the variable resistors so that the sensors also react to the grey stripe. The MCU car kit should be adjusted so the sensors react to both the white and grey stripes.</p>
---	---	--

6		<p>Move the sensors back slightly. The LEDs should all go dark.</p>
---	--	---

7		<p>Once again, slowly move the sensors in parallel toward the grey stripe. If one of the LEDs lights before the others, lower its sensitivity (turn the variable resistor counter clockwise). If an LED does not light, increase its sensitivity (turn the variable resistor clockwise). Repeat the adjustments several times until all the LEDs light at about the same time.</p>
---	--	--



Next, we will adjust the sensor that detects the start bar.
Stand a vertical white panel or sheet of paper several centimetres away from the front of the sensor board. This white panel or sheet of paper will be a substitute for the start bar.
Confirm that under the rightmost sensor which detects the course is black and LED15 is off at that time.



Slowly turn VR9, indicated by the circle ○, clockwise until LED15 lights and then stop turning it.
 Adjust that under LED 15 is black because it doubles as the rightmost of the course state sensor .
 If the LED goes dark when the white panel or sheet of paper is removed, the adjustment is complete.

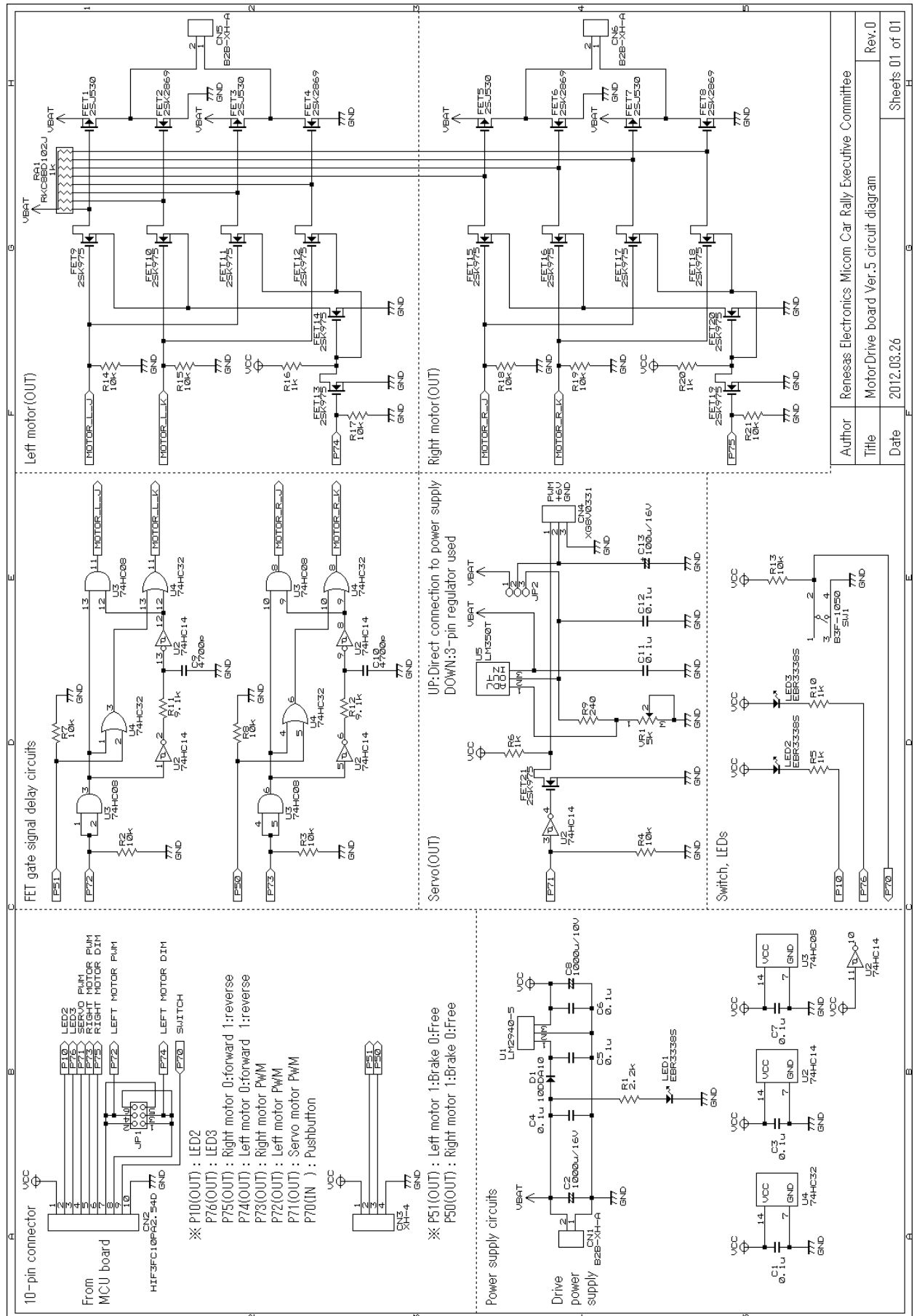
3. Motor Drive Board Ver. 5

3.1. Specifications

The table below lists the specifications of the motor driver board Ver. 5.

	Motor Drive Board Ver. 5
Abbreviation	Drive board 5
Number of components	Components with leads: 66 The component lead pitches are 2.54 mm or greater
Connection to the RMC-RX62T board	Connection using port 7 and bit 0 of port 1
Motors controlled	Two motors (the left and right motors)
Servos controlled	One motor
LEDs turned on/off under program control	Two motors
Pushbutton switches	One switch
Control system voltage (voltage that can be applied to CN2)	DC5.0 V \pm 10 %
Drive system voltage (voltage that can be applied to CN1)	4.5 to 5.5 V or 7 to 15 V Note, however, that if 7 V or higher is used the voltages applied to the microcontroller board and servo board must be limited to 5 V and 6 V respectively with the LM350 Add-On Set.
Servo and motor control period	Motor: 16 ms, Servo: 16 ms Individual setting of these values is not possible.
Motor free-running control	Supported by the addition of the Free-Running Add-On Set. Note: There are two motor stop modes: Brake and Free. See the sections on the Free-Running Add-On Set for details.
Board dimensions	80 × 65 × 1.6 mm (W × D × T)
Dimensions when completed (actual dimensions)	80 × 65 × 20 mm (W × D × H)
Weight	About 33 g Note: The weight varies with the length of the lead wires and the amount of solder used.
Standard software	RX62T microcontroller: kit12_rx62t.c

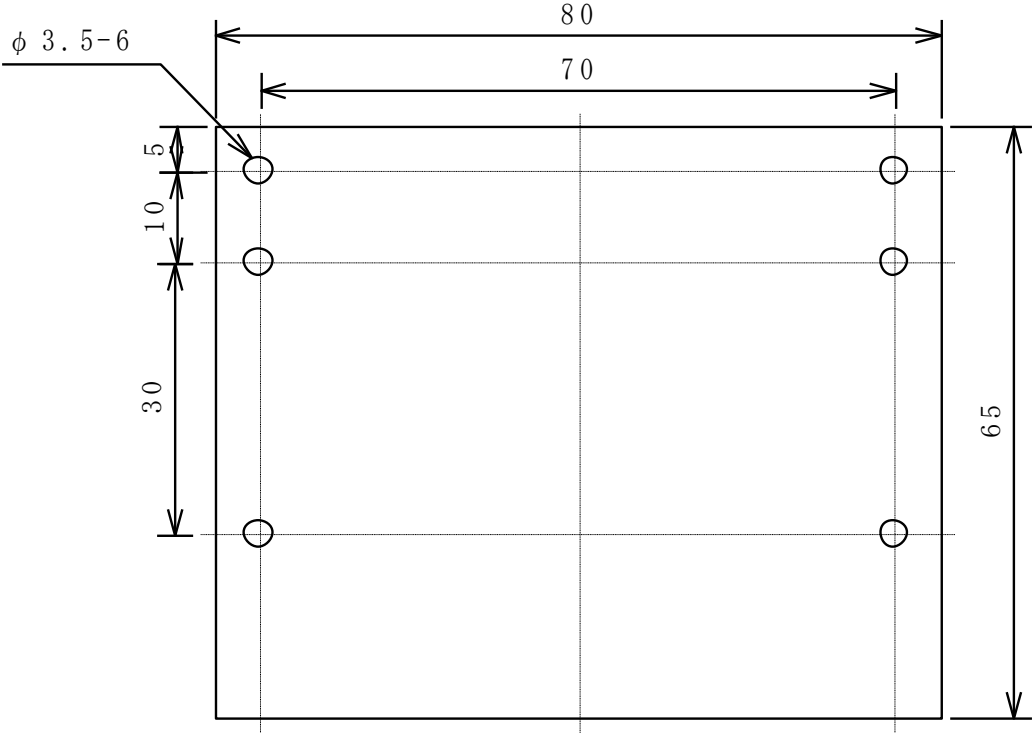
3.2. Circuit Diagram



Author	Renesas Electronics Micom Car Rally Executive Committee
Title	Motor Drive board Ver.5 circuit diagram
Date	2012.03.26
Rev.	Rev.0

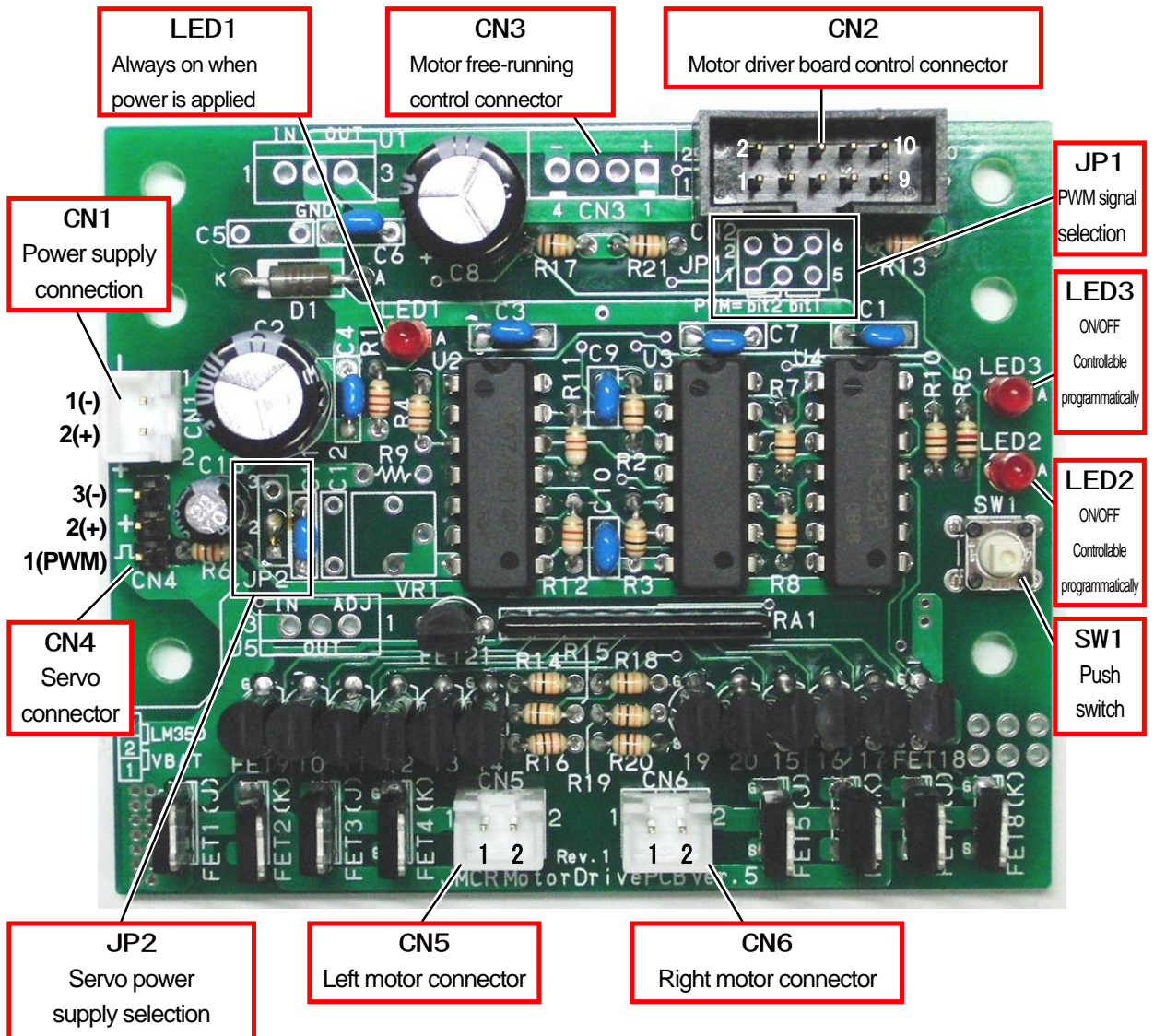
3.3. Dimensions

The motor drive board has six mounting holes. These holes are used to secure the motor drive board to the rest of the MCU car rally kit.



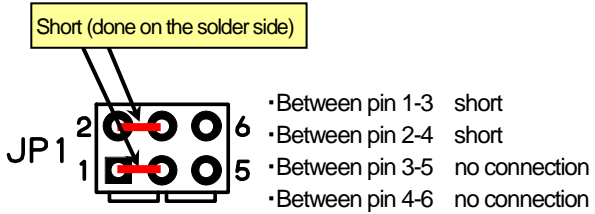
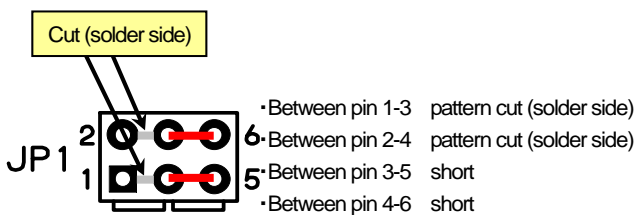
3.4. External Appearance

The photo below shows the external appearance of the motor drive board Ver. 5.



Note: Two-pin connectors CN1, CN5, and CN6 have been changed from IL connectors manufactured by Japan Aviation Electronics Industry, Ltd., to XH connectors manufactured by J.S.T. Mfg. Co., Ltd. This means that the female connectors must be changed as well.

The following table lists the connector connections and the contents carried by these signals.

Part No.	Connects To	Pins	Description
CN1	Power supply input	1	GND
		2	The + power supply connection (4.5 to 5.5 V or 7 to 15 V) Note: However, that if 7 V or higher is used, the LM350 Add-On Set must be installed.
CN2	Connects to MCU board	1 to 10	See next page.
CN3	Connected to the microcontroller board	1	+5 V
		2	Left motor stop state selection. 1: Free, 0: Brake
		3	Right motor stop state selection. 1: Free, 0: Brake
		4	GND
CN4	Servo	1	Servo PWM signal output
		2	Servo power supply (6 V output)
		3	GND
CN5	Left motor	1, 2	Left motor output
CN6	Right motor	1, 2	Right motor output
JP1	PWM signal selection of left motor	1~6	<p>This jumper switches PWM output terminal and direction selection terminal.</p> <p>●RMC-RX62T board and RY_R8C38 Board</p>  <ul style="list-style-type: none"> •Between pin 1-3 short •Between pin 2-4 short •Between pin 3-5 no connection •Between pin 4-6 no connection <p>※It has been short-circuited on the solder side. No need to do in anything in particular.</p> <p>●RY3048FoneBoard</p>  <ul style="list-style-type: none"> •Between pin 1-3 pattern cut (solder side) •Between pin 2-4 pattern cut (solder side) •Between pin 3-5 short •Between pin 4-6 short
JP2	Servo power supply switching	1 to 3	<p>This jumper switches the source for power supply to the servo power supply pin (pin 2 on CN2).</p> <ul style="list-style-type: none"> • If the supply voltage provided to CN1 is under 6 V Short pins 1 and 2 together. Connect the CN1 power supply directly to pin 2 on CN2. • If the supply voltage provided to CN1 is over 6 V Since this would exceed the voltage that can be applied to the servo, the components from the LM350 Add-On Set must be installed and pins 2 and 3 shorted together. A 6 V level will be supplied to pin 2 on CN2 through the LM350 3-terminal regulator.

3.5. Relationship between the Motor Drive Board Ver. 5 CN2 and the RMC-RX62T Board

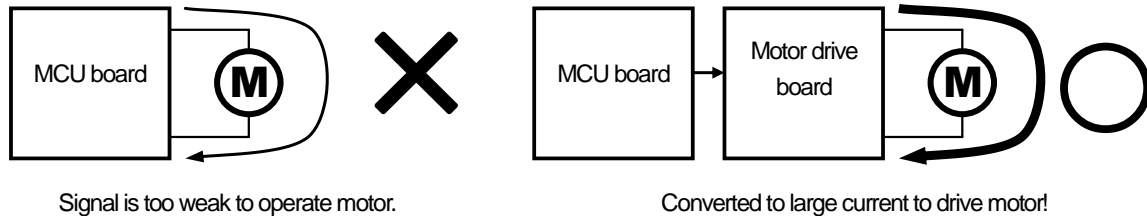
The motor drive board Ver. 5 connector CN2 and the RMC-RX62T board connector CN3 (port 7 and bit 0 of port 1) must be connected with a 10-pin flat cable. The following table lists the signals carried by this cable.

RMC-RX62T Board CN3	Signal Direction	Motor Drive Board Ver. 5 CN2	Description
Pin 1 (+5 V)	—	Pin 1	<p>This is the +5 V level provided to the control system circuits, including the Motor Drive Board Ver. 5 and logic ICs. Regardless of whether or not the LM350 Add-On Set is used, this is always a 5 V source.</p> <ul style="list-style-type: none"> If the control system and drive system power supplies are separate (the LM350 Add-On Set is not used) Here, a 5 V level is supplied to the Motor Drive Board Ver. 5 from the RMC-RX62T board. If the control system and drive system share a power supply (the LM350 Add-On Set is used) The Motor Drive Board Ver. 5 control system circuits and the RMC-RX62T board are supplied from the Motor Driver Board Ver. 5 LM2940-5 (a 5 V output 3-terminal regulator).
Pin 2 (P1_0)	→	Pin 2	<p>Connected to LED 2. 0: LED on, 1: LED off</p>
Pin 3 (P7_6)	→	Pin 3	<p>Connected to LED 3. 0: LED on, 1: LED off</p>
Pin 4 (P7_5)	→	Pin 6	<p>Controls the right motor direction of rotation. 0: Forward, 1: Reverse</p>
Pin 5 (P7_4)	→	Pin 8	<p>Controls the left motor direction of rotation. 0: Forward, 1: Reverse</p>
Pin 6 (P7_3)	→	Pin 5	<p>Outputs a PWM signal to the right motor.</p>
Pin 7 (P7_2)	→	Pin 7	<p>Outputs a PWM signal to the left motor.</p>
Pin 8 (P7_1)	→	Pin 4	<p>Outputs a PWM signal to the servo.</p>
Pin 9 (P7_0)	←	Pin 9	<p>Detects the state of the pushbutton switch. 0: Pressed, 1: Released</p>
Pin 10 (GND)	—	Pin 10	GND

3.6. Motor Control

3.6.1. Role of the Motor Drive Board

The motor drive board operates the motors according to instructions received from the MCU. The signals from the MCU meaning “run motor” or “stop motor” are very weak, so the motors will not operate if they are connected directly to the signal lines. The motor drive board converts the weak signals into signals with a large current level of several hundred to several thousand milliamperes (mA) in order to operate the motors.

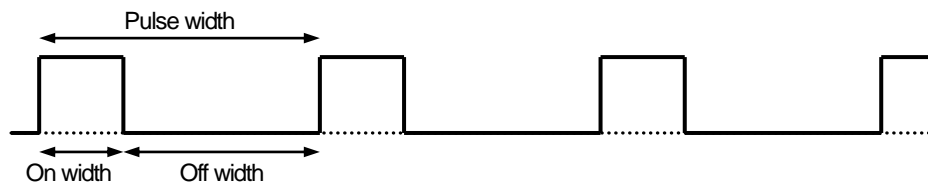


3.6.2. Operating Principle of Speed Control

To make a motor run, it is enough to apply a current. To make it stop, cease supplying the current. But how do you regulate the speed to, say, 10% or 20% of the maximum? How do you make fine adjustments to the speed at which the motor operates?

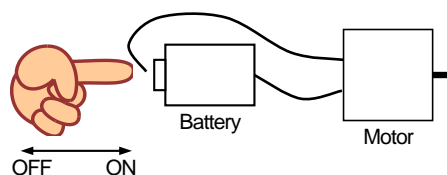
A variable resistor can be used to reduce the voltage. But since a large current flows to the motor, a very large resistance would be required. In addition, the voltage not applied to the motor would be converted to heat by the resistor.

A better way to control the speed of the motor is to switch the power on and off repeatedly at high speed, producing an effect that is equivalent to applying an intermediate voltage. The signal on and off states are controlled by using a fixed cycle and altering the ratio of on and off. This control method is called “pulse width modulation,” abbreviated as **PWM control**. The proportion of the pulse width for which the signal is on is called **the duty cycle**. If the on-width is 50% of the cycle, the duty cycle is 50%. This can also be simplified to “PWM 50%” or just “motor 50%.”



Duty cycle = on-width / pulse width (on-width + off-width). For example, if the pulse duration is 100 ms and the on-width is 60 ms, duty cycle = 60 ms / 100 ms = 0.6 = 60%. If the signal is on for the entire pulse duration, the duty cycle is 100%. If it is off for the entire pulse duration, the duty cycle is 0%.

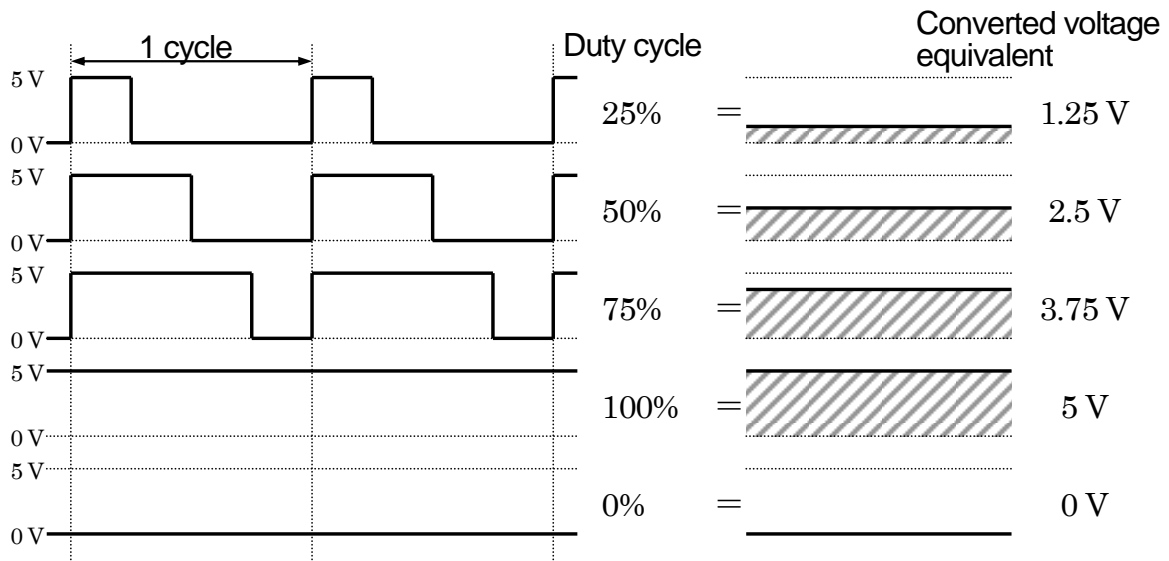
“PWM” sounds complicated, but a simple arrangement like that shown below, in which you control the speed by repeatedly connecting and disconnecting a motor and a battery with a wire, can also be considered an example of PWM. The longer you keep the wire connected, the faster the motor runs. The longer the periods it is disconnected, the slower it gets. A person can repeat this connect, disconnect operation at intervals of a couple of seconds, but the MCU can accomplish it at intervals of a few milliseconds.



Let's assume a waveform consisting of output at 0 V and 5 V. The longer the on-duration is during each cycle, the higher the average voltage value, as shown in the figure below. If output is at 5 V for the entire cycle, the average voltage value is 5 V, as you would expect. This is the maximum voltage. What if the signal is on 50% of the time? This works out to an average of $5\text{ V} \times 0.5 = 2.5\text{ V}$, so the result is the same as changing the voltage.

In this way, if we reduce the on-duration of each cycle to 90%, 80%, and so on down to 0%, the result is equivalent to gradually lowering the voltage until we finally reach 0 V.

By connecting this signal output to a motor, we can change the motor's speed a little bit at a time, making precise speed control possible. If we connect the signal output to an LED, we can change the brightness of the LED. A MCU is capable of performing this operation in microsecond or millisecond increments. Control on this order enables extremely smooth motor control.



Why would we want to use pulse width control rather than voltage control to regulate the speed of a motor? MCUs are very good at handling digital values expressed as zeroes and ones. They are less good at dealing with analogue values such as voltages. This is why we use a system of changing the width of the zeroes and ones to **simulate controlling the voltage. The system is called PWM control.**

3.6.3. Operating Principle of Forward and Reverse

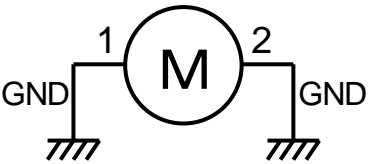
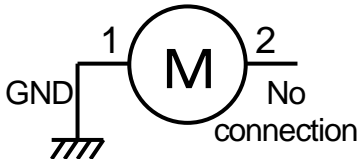
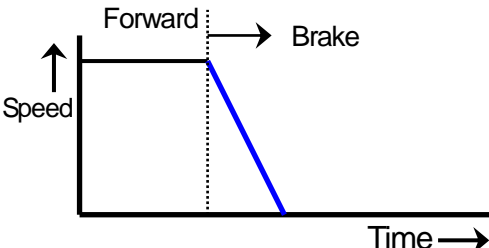
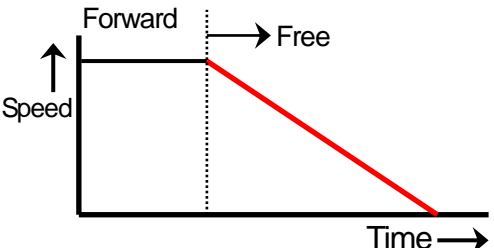
The motor drive board, Ver. 5, can control the forward, reverse, and brake operation of the motors. The voltages applied to the motor terminals for forward and reverse control are shown in the table below.

	Forward	Reverse
Voltages applied to motor terminals	<p>Pin 1 is connected to GND (0 V) and pin 2 to a positive voltage.</p>	<p>Pin 1 is connected to GND (0 V) and pin 2 to a positive voltage.</p>

3.6.4. Brake and free

Stopping and slowing the car with the normal circuit of the Motor drive board Ver.5 is done by breaking. With the free addition set added, there are two methods of halting the car, brake and free.

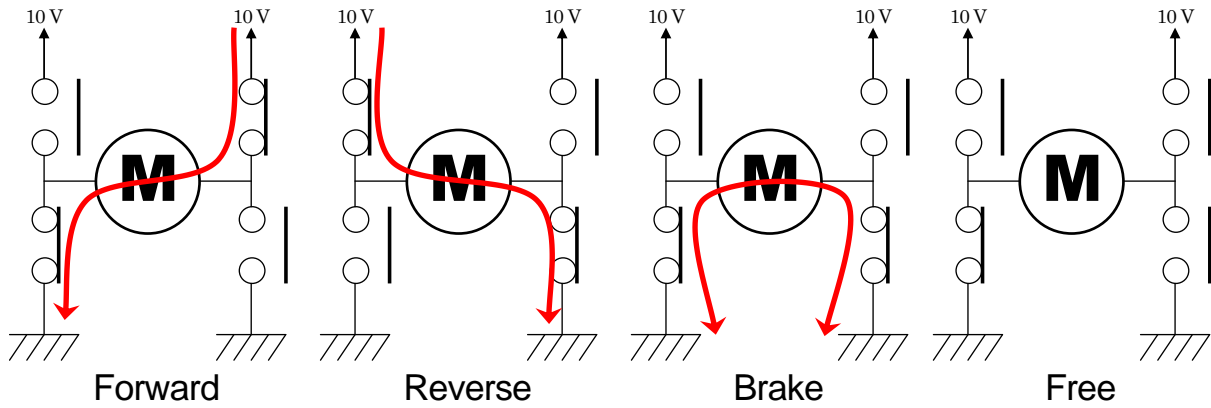
The difference of brake and free is as shown below.

	Brake	Free
Voltages applied to motor terminals	 <p style="text-align: center;">Brake</p> <p>Connect both terminals to GND. As a result, both terminals are short-circuited.</p>	 <p style="text-align: center;">Free</p> <p>Make one side connectionless.</p>
Falling of speed (image)		

As for the free, the slowdown of the stop is slower than brakes. Use the free for the cases that want to cut down speed slowly.

3.6.5. H-bridge circuit

How does this actually work? Four switches are arranged with the motor in the centre, forming an H-pattern, as shown in the figure below. Forward, reverse, brake, and free control is accomplished by turning these four switches on and off in specific combinations. The name “H-bridge circuit” refers to the circuit’s H-pattern.

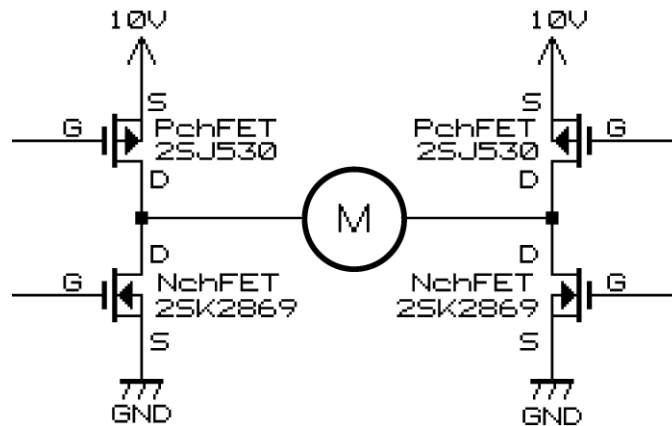


3.6.6. Using FETs as the Switches in an H-Bridge Circuit

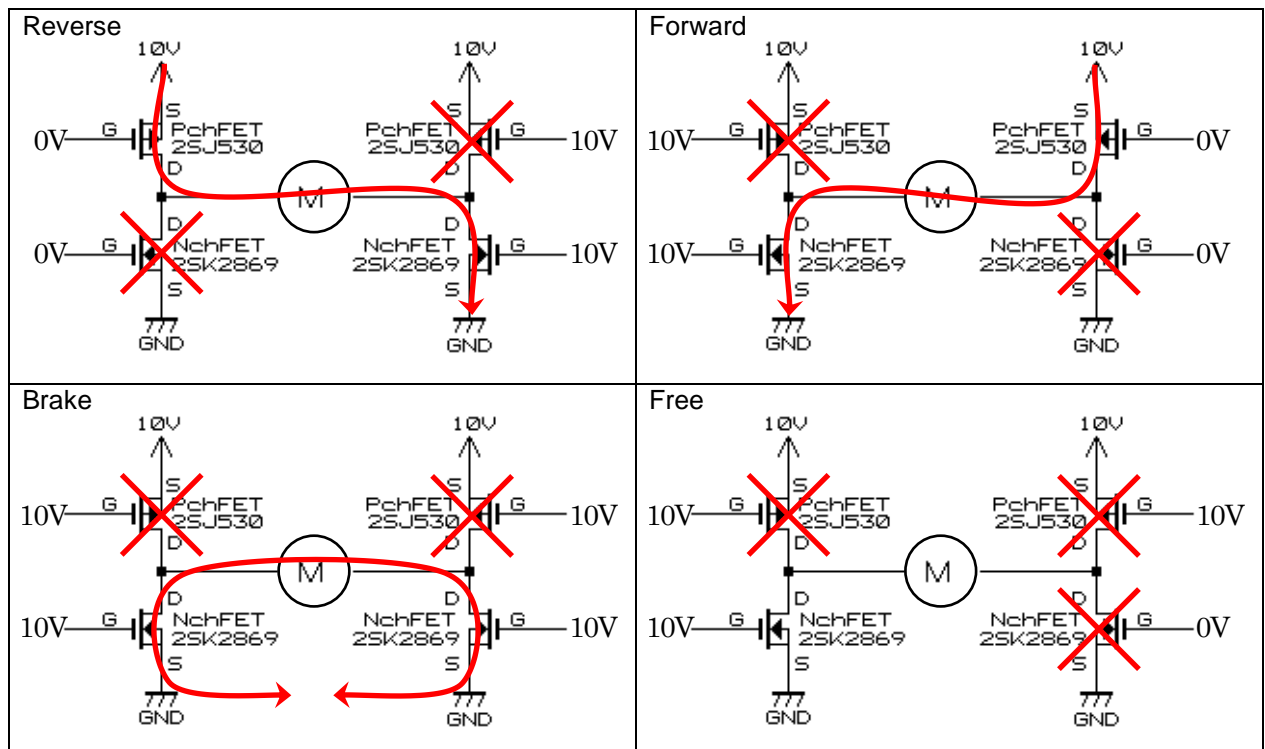
Field-effect transistors (FETs) are used as the switches. A P-channel FET (2SJ type) is used on the positive side of the power supply and an N-channel FET (2SK type) on the negative side.

A P-channel FET allows current to flow between drain and source (D-S) when the gate voltage (V_G) is less than the source voltage (V_S).

An N-channel FET allows current to flow between drain and source (D-S) when the gate voltage (V_G) is greater than the source voltage (V_S).

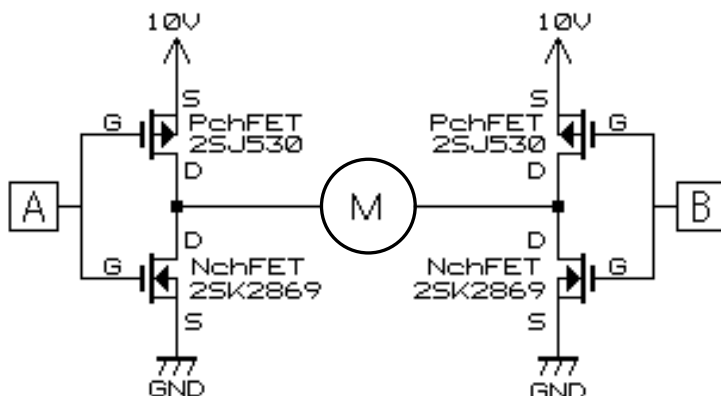


Forward, reverse, and brake operations are performed by changing the voltages applied to the gates of the four FETs.



One point to keep in mind is that the two FETs on the right side or the two FETs on the left side must never both be on at the same time. Having both on at the same time connects the 10 V and GND pins with no load at all, which is the same as shorting them. This could cause the FETs or the trace patterns to burn out, which would be dangerous.

A look at the four gate voltages reveals that the same voltage is applied to the P-channel and N-channel FETs on the right side and to the P-channel and N-channel FETs on the left side. Therefore, we tried using the circuit shown below.



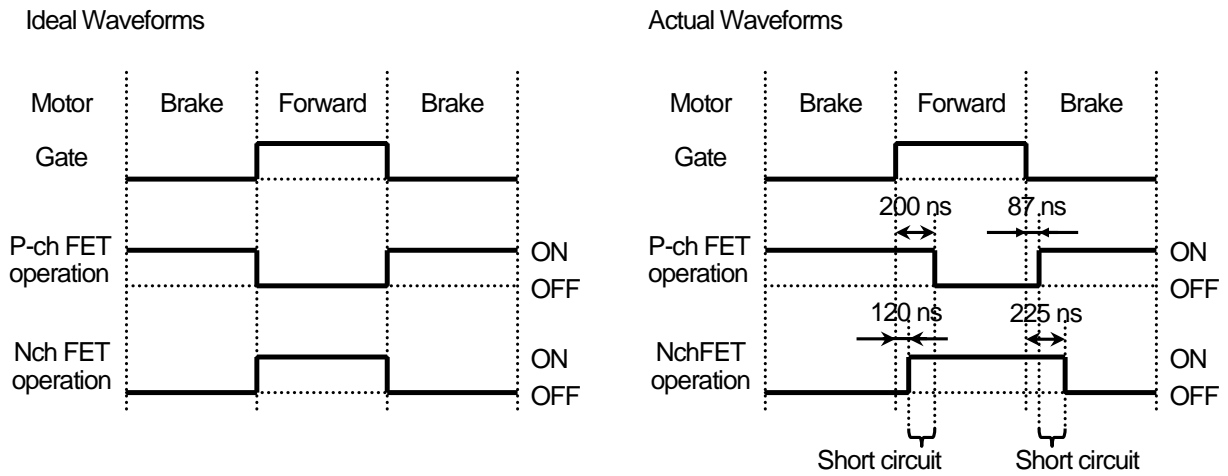
A	B	Operation
0 V	0 V	Brake
0 V	10 V	Reverse
10 V	0 V	Forward
10 V	10 V	Brake

Note: When a power supply voltage of 10 V is input to the G (gate) pin, that voltage is either applied unchanged to the motor or a voltage of 0 V is applied. Note that the voltage applied to the motor differs depending on whether the value of the control signal is 0 or 1.

When we actually input a PWM waveform to the circuit, the FETs became very hot. Why might this be?

We might assume that in on-off switching of the channel between drain and source when signals are input to the gates of the FETs, the P-channel and N-channel FETs would respond instantly as in the Ideal Waveforms figure at

left, resulting in smooth switching between brake and forward operation. In fact, however, the FETs do not operate instantly and there is a time delay. This delay is greater when the FET switches from on to off than when it switches from off to on. Therefore, there is a short duration during which both FETs are in the on state, as shown in the Actual Waveforms figure at right. This state is equivalent to a short circuit.



The delay between the on signal and the start of the response is called the “turn-on delay,” the duration from the start of the on response to the actual on state is the “rise time,” the delay between the off signal and the start of the response is the “turn-off delay,” and the duration from the start of the off response to the actual off state is the “fall time.”

Thus, the actual duration from off to on is the turn-on delay plus the rise time, and the actual duration from on to off is the turn-off delay plus the fall time. These are the delays shown in the figure at right above.

The electrical characteristics of the FETs used on the motor drive board, 2SJ530 and 2SK2869 from Renesas Electronics, are shown below.

2SJ530 (P-channel)

Electrical Characteristics						
(Ta=25°C)						
Item		Min	Typ	Max		
Drain-source destruction voltage	$V_{(BR)DSS}$	-60	—	—	V	$I_D = -10mA, V_{GS} = 0$
Gate-source destruction voltage	$V_{(BR)GSS}$	±20	—	—	V	$I_D = ±100μA, V_{DS} = 0$
Drain cutoff current	I_{DSS}	—	—	-10	μA	$V_{DS} = -60V, V_{GS} = 0$
Gate cutoff current	I_{GSS}	—	—	±10	μA	$V_{DS} = ±16V, V_{DS} = 0$
Gate-source cutoff current	$V_{GS(off)}$	-1.0	—	-2.0	V	$V_{DS} = -10V, I_D = -1mA$
Forward transfer admittance	$ y_{fs} $	6.5	11	—	S	$I_D = -8A, V_{GS} = 10V, V_{DS} = 0$
Drain-source on-resistance	$R_{DS(on)}$	—	0.08	0.10	Ω	$I_D = -8A, V_{GS} = 10V, V_{DS} = 0$
Drain-source on-resistance	$R_{DS(on)}$	—	0.11	0.16	Ω	$I_D = -8A, V_{GS} = 4V, V_{DS} = 0$
Input capacitance	C_{iss}	—	850	—	pF	$V_{DS} = -10V, V_{GS} = 0$
Output capacitance	C_{oss}	—	420	—	pF	$f = 1MHz$
Feedback capacitance	C_{rss}	—	110	—	pF	
Turn-on delay	$t_d(on)$	—	12	—	ns	$V_{GS} = -10V, I_D = -8A$
Rise time	t_r	—	75	—	ns	$R_L = 3.75Ω$
Turn-off delay	$t_d(off)$	—	125	—	ns	
Fall time	t_f	—	75	—	ns	
Diode forward voltage	V_{DF}	—	-1.1	—	V	$I_F = -15A, V_{GS} = 0$
Reverse recovery time	t_{rr}	—	70	—	ns	$I_F = -15A, V_{GS} = 0$ $dI_F/dt = 50A/μs$

Note: 4. Pulse measurement

87 ns delay from off to on

200 ns delay from on to off

2SK2869 (N-channel)

Electrical Characteristics						
(Ta=25°C)						
Item		Min	Typ	Max		
Drain-source destruction voltage	$V_{(BR)DSS}$	60	—	—	V	$I_D = 10mA, V_{GS} = 0$
Gate-source destruction voltage	$V_{(BR)GSS}$	±20	—	—	V	$I_D = ±100μA, V_{DS} = 0$
Drain cutoff current	I_{DSS}	—	—	10	μA	$V_{DS} = 60V, V_{GS} = 0$
Gate cutoff current	I_{GSS}	—	—	±10	μA	$V_{DS} = ±16V, V_{DS} = 0$
Gate-source cutoff current	$V_{GS(off)}$	1.5	—	2.5	V	$V_{DS} = 10V, I_D = 1mA$
Forward transfer admittance	$ y_{fs} $	10	16	—	S	$I_D = 10A, V_{GS} = 10V, V_{DS} = 0$
Drain-source on-resistance	$R_{DS(on)}$	—	0.033	0.045	Ω	$I_D = 10A, V_{GS} = 10V, V_{DS} = 0$
Drain-source on-resistance	$R_{DS(on)}$	—	0.055	0.07	Ω	$I_D = 10A, V_{GS} = 4V, V_{DS} = 0$
Input capacitance	C_{iss}	—	740	—	pF	$V_{DS} = 10V, V_{GS} = 0$
Output capacitance	C_{oss}	—	380	—	pF	$f = 1MHz$
Feedback capacitance	C_{rss}	—	140	—	pF	
Turn-on delay	$t_d(on)$	—	10	—	ns	$V_{GS} = 10V, I_D = 10A$
Rise time	t_r	—	110	—	ns	$R_L = 3Ω$
Turn-off delay	$t_d(off)$	—	105	—	ns	
Fall time	t_f	—	120	—	ns	
Diode forward voltage	V_{DF}	—	1.0	—	V	$I_F = 20A, V_{GS} = 0$
Reverse recovery time	t_{rr}	—	40	—	ns	$I_F = 20A, V_{GS} = 0$ $dI_F/dt = 50A/μs$

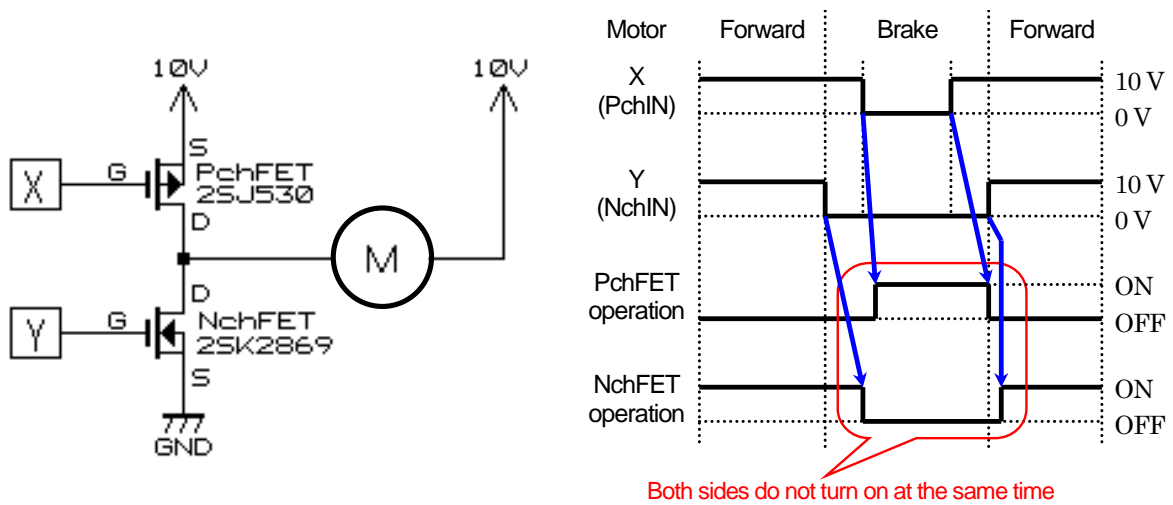
Note: 1. Pulse measurement

120 ns delay from off to on

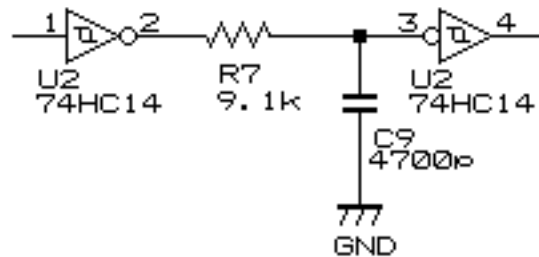
225 ns delay from on to off

3.6.7. P-Channel and N-Channel Short FETs Prevention Circuit

As a solution, instead of turning the P-channel and N-channel FETs on side A on and off at the same time as in the previous circuit diagram, we will introduce a short time shift to prevent the formation of a short circuit.



The delay is generated by an integrating circuit. There are many technical books available with information on integrating circuits, and we refer you to them if you wish to learn more. A diagram of the integrating circuit is shown below.

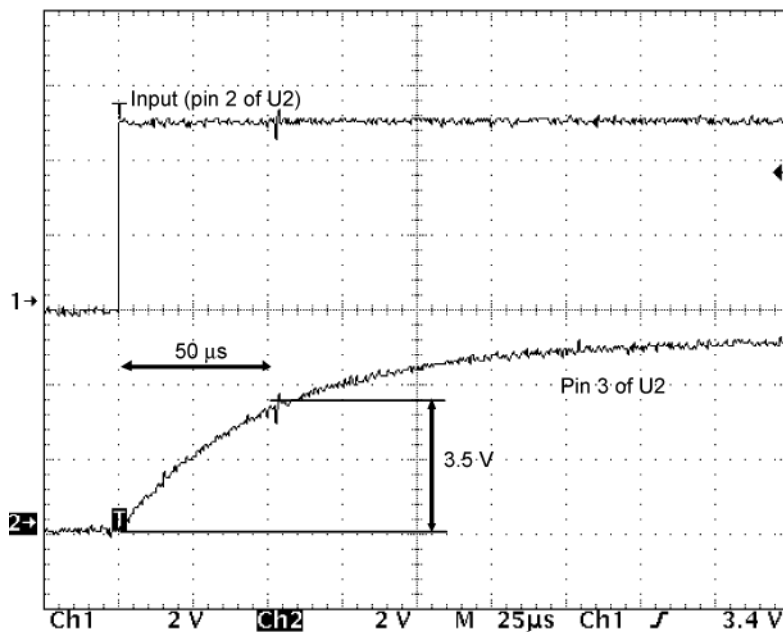


The delay time is calculated as follows:

The time constant $T = CR$ [s].

In the present case, the figures are 9.1 kΩ and 4700 pF, so the calculation is as follows:

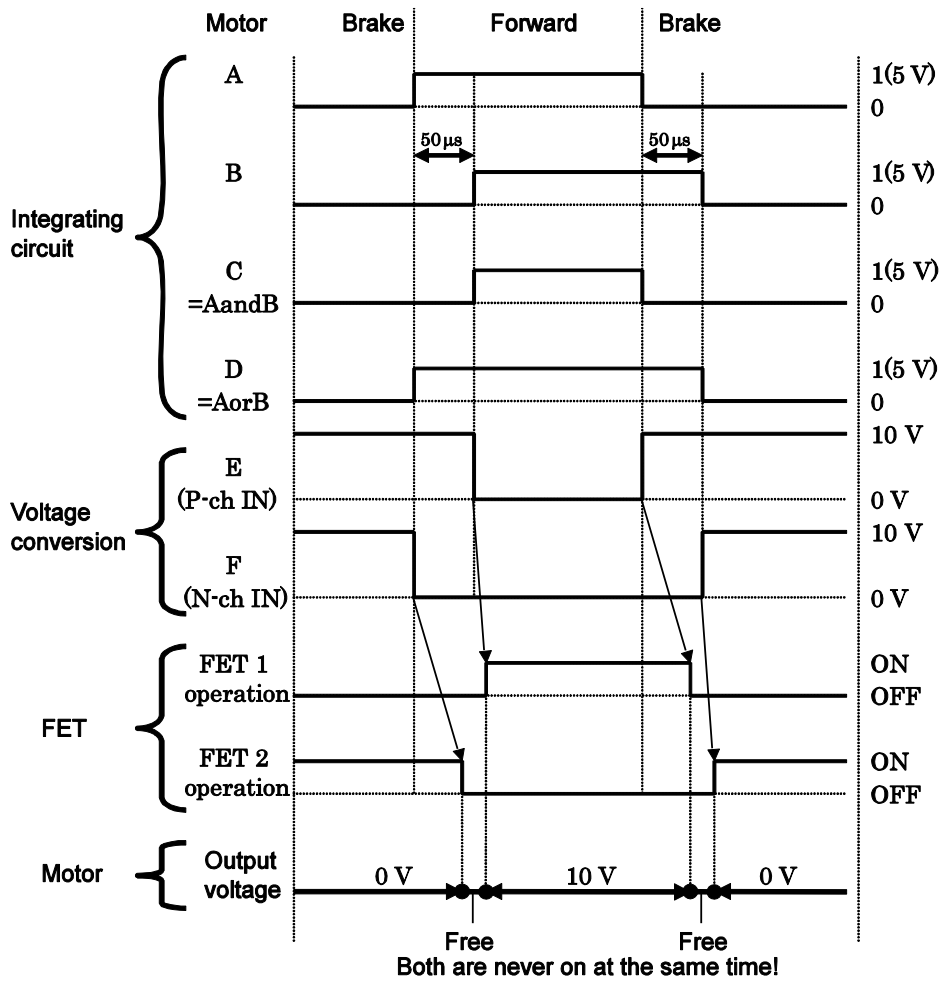
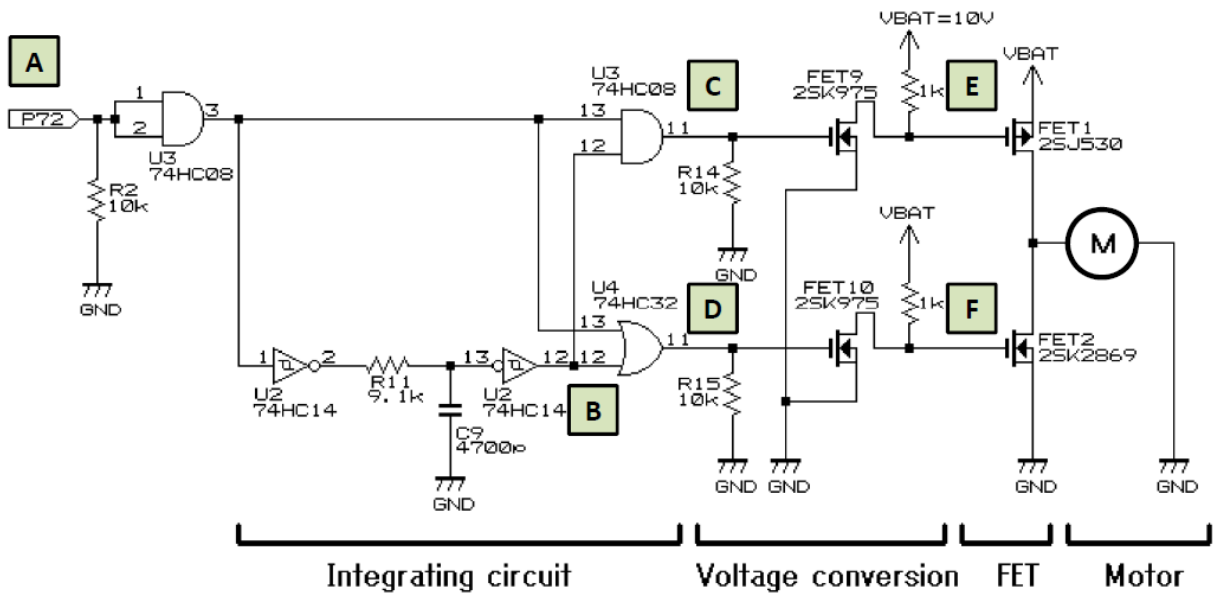
$$T = 9.1 \times 10^3 \times 4,700 \times 10^{-12} = 42.77 \text{ } [\mu\text{s}].$$



The 74HC series treats an input voltage of 3.5 V or more as 1. By measuring actual waveforms, we determined that the time required to reach 3.5 V is approximately 50 μs.

Though the maximum shift is 225 ns in the Actual Waveforms figure above, we decided to generate a delay of 50 μs with the integrating circuit. This is to accommodate delay from voltage conversion digital transistors other than FETs, as well as delay from capacitance components in the FET gates.

A circuit diagram combining an integrating circuit and the FETs is shown below.



(1) Changing from Brake to Forward Operation

1. At point **A**, a 0 signal corresponds to brake and a 1 signal to forward. The output changes from 0 (brake) to 1 (forward) at point **A**.
2. At point **B**, the integrating circuit outputs a waveform with a 50 μ s delay.
3. An A-and-B waveform is output from point **C**.
4. An A-or-B waveform is output from point **D**.
5. A signal that has been voltage converted by the digital transistor is output from point **E**. The 0 V-5 V signal from point **C** has been converted to a 10 V-0 V signal.
6. In like manner, a 10 V-0 V signal converted from the 0 V-5 V signal from point **D** is output from point **F**.
7. When the signal at point **A** changes from 0 to 1, the gate voltage of FET2 changes from 10 V to 0 V and FET2 turns off. However, it does not enter the off state immediately due to the delay. Both FET1 and FET2 are off at the point, and the motor is in the free state.
8. The gate voltage of FET1 changes from 10 V to 0 V 50 μ s after the signal at point **A** changes, and FET1 turns on. A voltage of 10 V is applied to the motor, causing it to run in the forward direction.

(2) Changing from Forward to Brake Operation

1. When the signal at point **A** changes from 1 (forward) to 0 (brake), the gate voltage of FET1 changes from 0 V to VBAT and FET1 turns off. However, it does not enter the off state immediately due to the delay. Both FET1 and FET2 are off at this point, and the motor is in the free state.
2. The gate voltage of FET2 changes from 0 V to 10 V 50 μ s after the signal at point **A** changes, and FET2 turns on. A voltage of 0 V is applied to both terminals of the motor, causing it to perform brake operation.

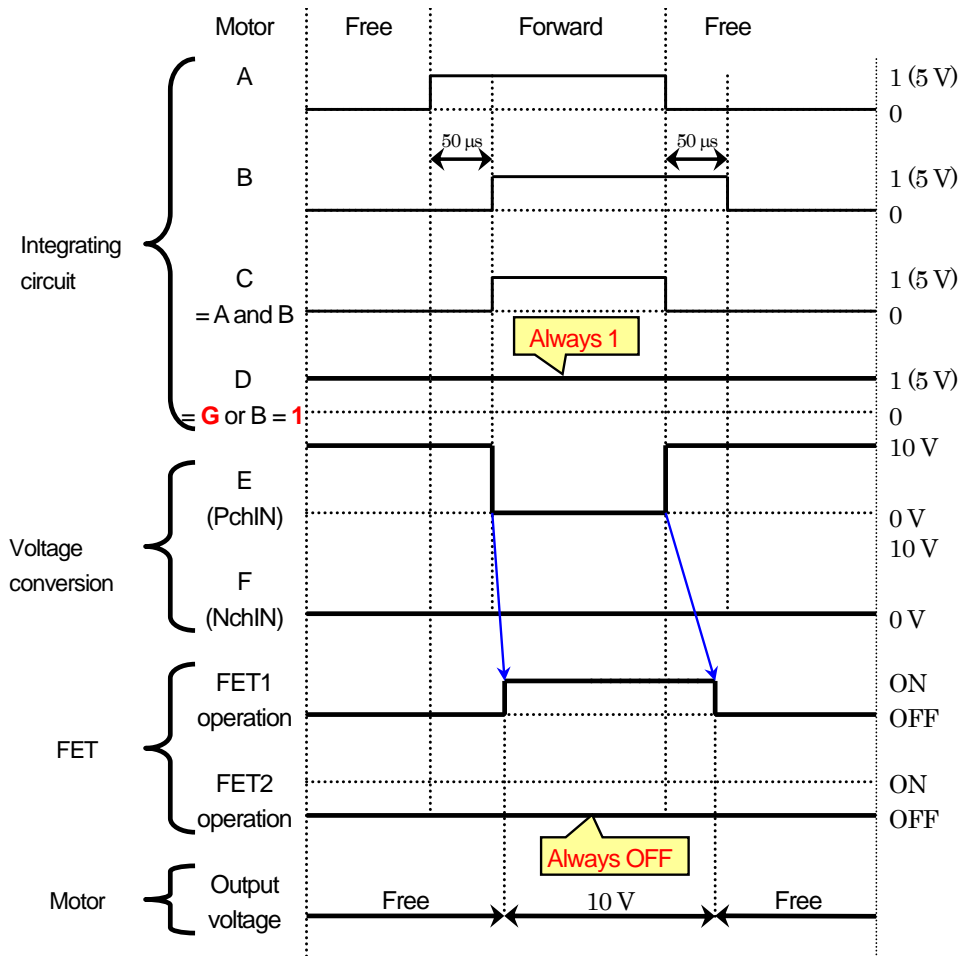
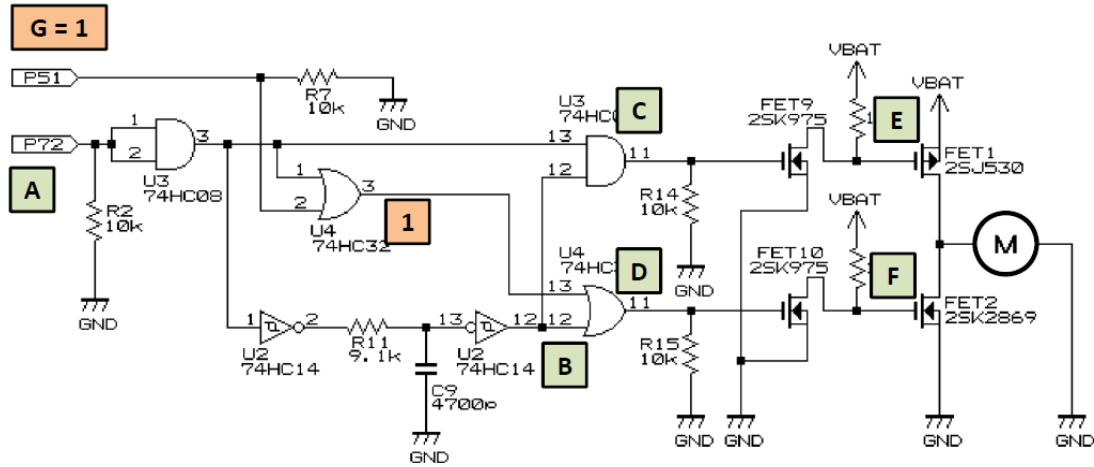
In this way, a short circuit is prevented when switching between operations by turning both FETs off for a short time and putting the motor into the free state.

Note: In this example the voltage applied to the gates is 10 V. In actual practice, the voltage matches the power supply voltage (VBAT).

3.6.8. Free Circuit

The free circuit described here is not for the purpose of preventing shorting of the P-channel and N-channel FETs. Rather, it is used to put the motors into the free or brake state when stopped.

By installing the free add-on set on the motor drive board, Ver. 5, it is possible to select between free and brake states when the motors are stopped. The state when the value of point **G** is 1 is shown below.



When the value of point **G** is 1, the value of point **D** is always 1 regardless of the states of points **A** or **B**. This means that FET2 is always off and the motor changes repeatedly between the forward and free states.

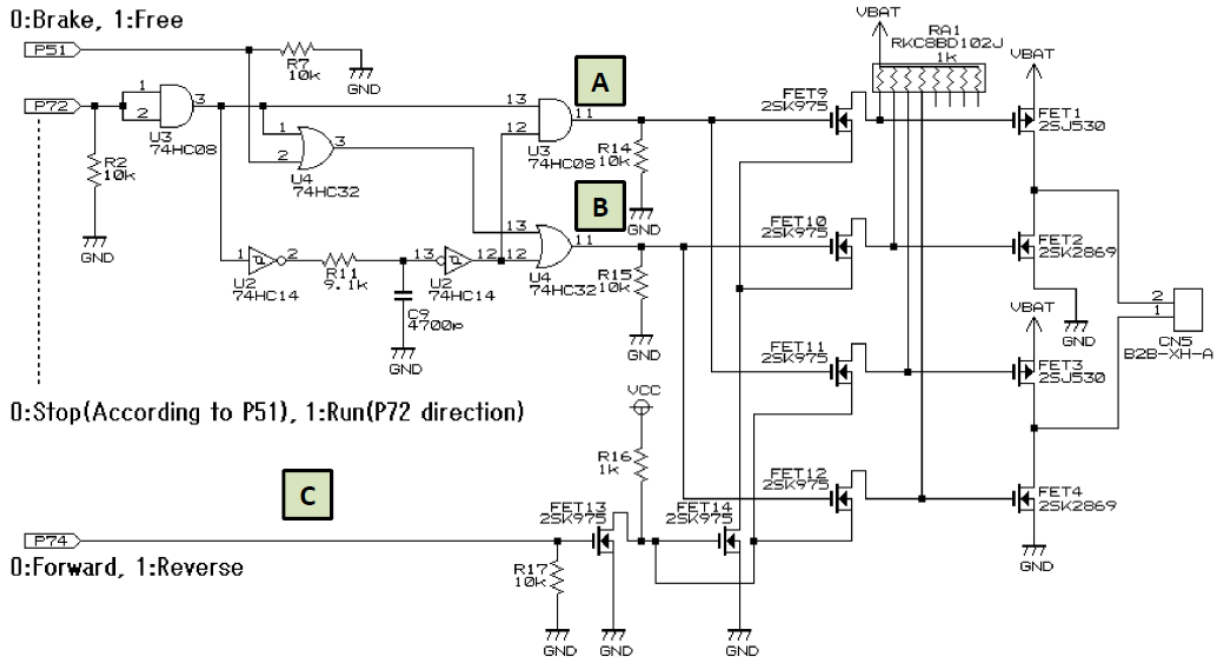
When the value of point **G** is 0, the motor changes repeatedly between the forward and brake states, as above.

3.6.9. Actual Circuit

The actual circuit configuration adds a forward/reverse switching circuit to the integrating circuit, FET circuit, and free circuit described above. The circuit configuration for the left motor is shown below. The following three pins are used.

- P72: Pin for applying PWM signal
- P74: Forward/reverse switching pin
- P51: Brake/free switching pin

(1) Circuit Diagram



(2) Direction: Forward, Stop: Signal Levels and Motor Operation in Brake State

A	B	C	FET1 gate	FET2 gate	FET3 gate	FET4 gate	Pin 2 of CN5	Pin 1 of CN5	Motor Operation
0	0	0	10 V (OFF)	10 V (ON)	10 V (OFF)	10 V (ON)	0 V	0 V	Brake
0	1		10 V (OFF)	0 V (OFF)	10 V (OFF)	10 V (ON)	Free (Open)	0 V	Free
1	1		0 V (ON)	0 V (OFF)	10 V (OFF)	10 V (ON)	10 V	0 V	Forward
0	1		10 V (OFF)	0 V (OFF)	10 V (OFF)	10 V (ON)	Free (Open)	0 V	Free
0	0		10 V (OFF)	10 V (ON)	10 V (OFF)	10 V (ON)	0 V	0 V	Brake

Note: A, B, C: 0 = 0 V, 1 = 5 V

(3) Direction: Reverse, Stop: Signal Levels and Motor Operation in Brake State

A	B	C	FET1 gate	FET2 gate	FET3 gate	FET4 gate	Pin 2 of CN5	Pin 1 of CN5	Motor Operation
0	0	1	10 V (OFF)	10 V (ON)	10 V (OFF)	10 V (ON)	0 V	0 V	Brake
0	1		10 V (OFF)	10 V (ON)	10 V (OFF)	0 V (OFF)	0 V	Free (Open)	Free
1	1		10 V (OFF)	10 V (ON)	0 V (ON)	0 V (OFF)	0 V	10 V	Reverse
0	1		10 V (OFF)	10 V (ON)	10 V (OFF)	0 V (OFF)	0 V	Free (Open)	Free
0	0		10 V (OFF)	10 V (ON)	10 V (OFF)	10 V (ON)	0 V	0 V	Brake

(4) Direction: Forward, Stop: Signal Levels and Motor Operation in Free State

A	B	C	FET1 gate	FET2 gate	FET3 gate	FET4 gate	Pin 2 of CN5	Pin 1 of CN5	Motor Operation
0	1	0	10 V (OFF)	0 V (OFF)	10 V (OFF)	10 V (ON)	Free (Open)	0 V	Free
0	1		10 V (OFF)	0 V (OFF)	10 V (OFF)	10 V (ON)	Free (Open)	0 V	Free
1	1		0 V (ON)	0 V (OFF)	10 V (OFF)	10 V (ON)	10 V	0 V	Forward
0	1		10 V (OFF)	0 V (OFF)	10 V (OFF)	10 V (ON)	Free (Open)	0 V	Free
0	1		10 V (OFF)	0 V (OFF)	10 V (OFF)	10 V (ON)	Free (Open)	0 V	Free

3.6.10. Operation of Left Motor

The left motor is controlled by three pins: P74, P72, and P51. If the free add-on set is not installed, the value of P5_1 is always 0.

Left Motor Direction P74	Left Motor PWM P72	Left Motor Stop Operation P51	Motor Operation
0	PWM	0	PWM = 1: forward, PWM = 0: brake
0	PWM	1	PWM = 1: forward, PWM = 0: free
1	PWM	0	PWM = 1: reverse, PWM = 0: brake
1	PWM	1	PWM = 1: reverse, PWM = 0: free

To operate the left motor in the forward and brake states, set P74 to 0 and P51 to 0 and input a PWM waveform on P72. The left motor will run forward according to the PWM ratio. For example, when the PWM ratio is 0% the motor will be stopped, when PWM ratio is 50% the motor will run forward at 50% voltage, and when the PWM ratio is 100% the motor will run forward at 100% voltage. In this case the motor is in the brake state when stopped.

3.6.11. Operation of Right Motor

The right motor is controlled by three pins: P75, P73, and P50. If the free add-on set is not installed, the value of P50 is always 0.

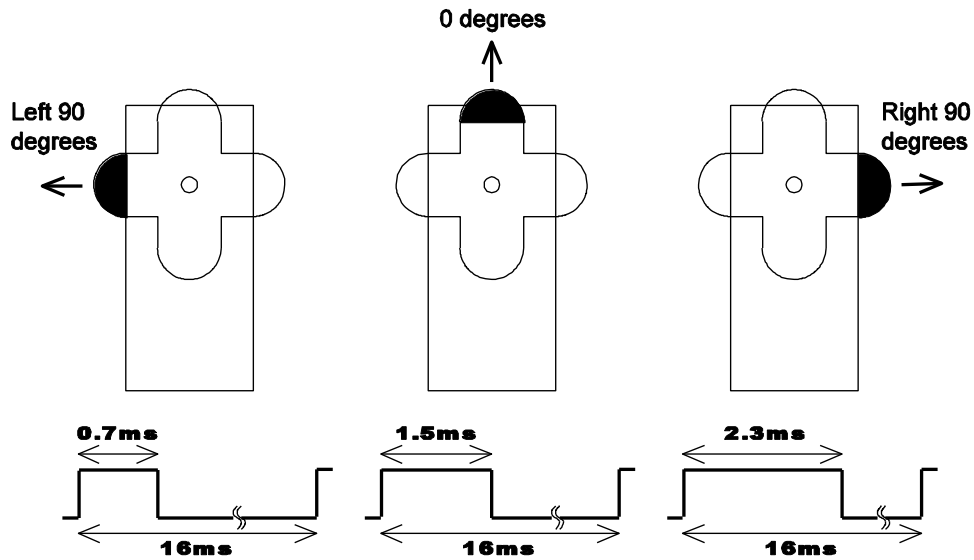
Right Motor Direction P75	Right Motor PWM P73	Right Motor Stop Operation P50	Motor Operation
0	PWM	0	PWM = 1: forward, PWM = 0: brake
0	PWM	1	PWM = 1: forward, PWM = 0: free
1	PWM	0	PWM = 1: reverse, PWM = 0: brake
1	PWM	1	PWM = 1: reverse, PWM = 0: free

To operate the right motor in the forward and free states, set P75 to 0 and P50 to 1 and input a PWM waveform on P73. The right motor will run forward according to the PWM ratio. For example, when the PWM ratio is 0% the motor will be stopped, when PWM ratio is 50% the motor will run forward at 50% voltage, and when the PWM ratio is 100% the motor will run forward at 100% voltage. In this case the motor is in the free state when stopped.

3.7. Servo Control

3.7.1. Operating Principle

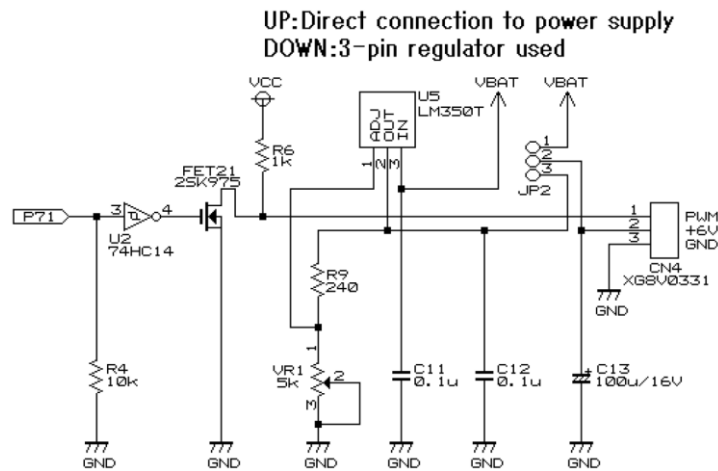
Pulses with a cycle of 16 ms are applied to the servo, and the servo angle is determined by the pulse on-width. There is some variation among servo manufacturers and individual devices in the correspondence between the servo turn angle and the pulse on-width, but generally speaking the correspondence is roughly shown below.



- The cycle is 16 ms.
- The Centre position corresponds to a pulse on-width of 1.5 ms, and a change of ± 0.8 ms produces a change in the servo angle of ± 90 degrees.

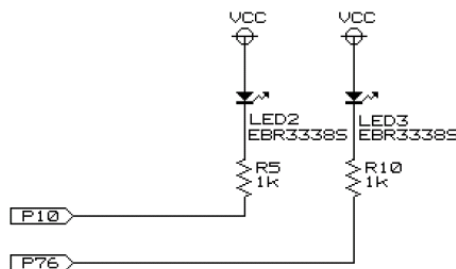
The PWM signals for servo control are generated in the reset-synchronized PWM mode of the RX62T MCU.

3.7.2. Circuit



1. The PWM signal is output on bit 1 of port 7. The on-width is changed by changing the value of MTU3.TGRD in the software.
2. A transistor with an internal resistor between the port and pin 1 of the servo acts as a buffer. If bit 1 of port 7 and pin 1 of the servo were connected directly, the MCU port could be destroyed if, for example, the power supply were accidentally connected to pin 1 or if noise were introduced. This would be fatal. In contrast, the transistor with internal resistor can be replaced easily if it is destroyed.
3. Pin 2 connects to the servo's power supply. If the motor drive power supply uses four or fewer batteries, short JP2 and the pin above it for a direct connection to the power supply. A motor drive power supply voltage higher than that produced by four batteries exceeds the rating of the servo, so it is necessary in this case to use the LM350 3-pin regulator, which has a 3 A current flow, to fix the voltage at 6 V. In this case, short JP2 and the pin below it.

3.8. LED Control

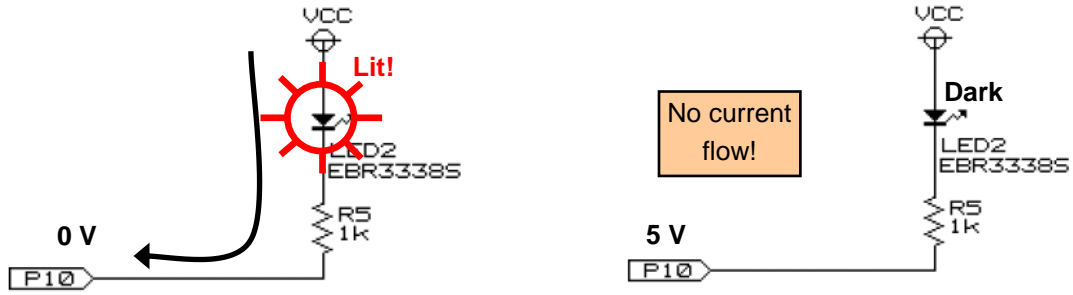


Three LEDs are mounted on the motor drive board. Of these, two can be turned on and off by the MCU. The cathode of each LED is connected directly to a port of the MCU. The current limiting resistance is 1 kΩ. A current of 20 mA can be input to the EBR3338S with a forward voltage of 1.7 V. The current limiting resistance is calculated as follows:

$$\begin{aligned} \text{Resistance} &= (\text{power supply voltage} - \text{voltage applied to LED}) / \text{current to be input to the LED} \\ &= (5 - 1.7) / 0.02 \\ &= 165 \Omega \end{aligned}$$

In practice, a 1 kΩ resistor is connected to reduce battery current consumption and limit the current flowing through the port. The current is calculated as follows:

$$\begin{aligned} \text{Current} &= (\text{power supply voltage} - \text{voltage applied to LED}) / \text{resistance} \\ &= (5 - 1.7) / 1,000 = 3.3 \text{ [mA]} \end{aligned}$$

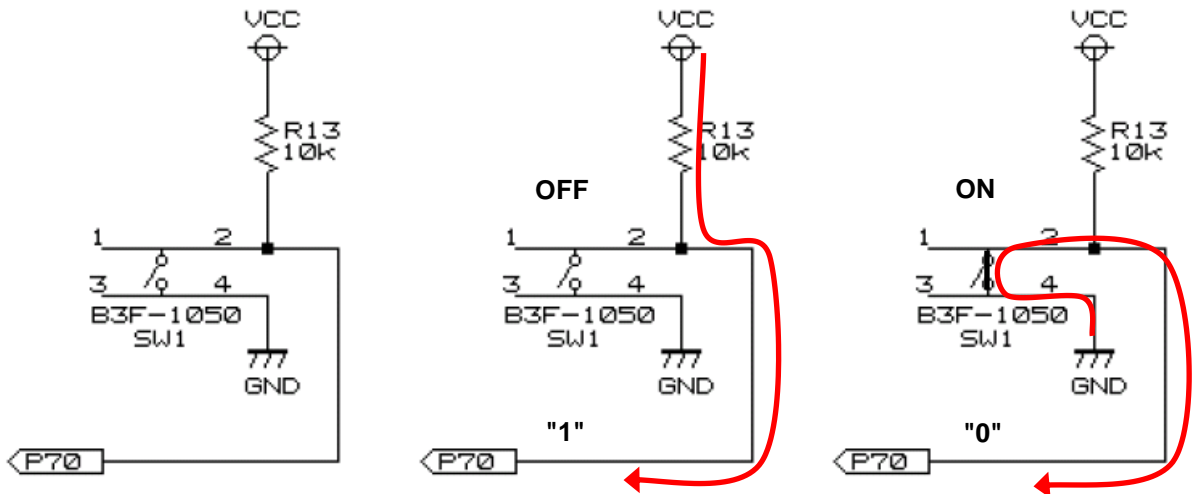


When 0 is output to P10, the voltage on the LED cathode side becomes 0 V, current flows, and the LED lights.

When 1 is output to P10, the voltage on the LED cathode side becomes 5 V, the potential difference between the two terminals of the LED is 0 V, and the LED does not light.

3.9. Pushbutton Control

One pushbutton is mounted on the motor drive board.



The pushbutton is pulled up by a 10 kΩ resistor and is connected to bit 0 of port 7.

When the pushbutton is not depressed, 1 is input to P70 via the pull-up resistor.

When the pushbutton is depressed, 0 is input to P70 via the ground (GND).




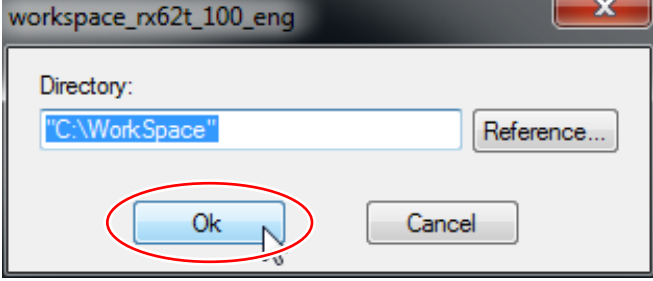
4. Sample Programs

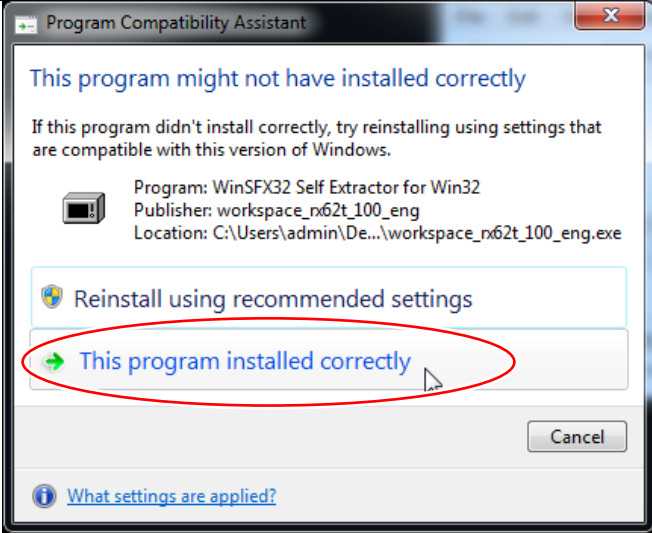
4.1. Program Development Environment

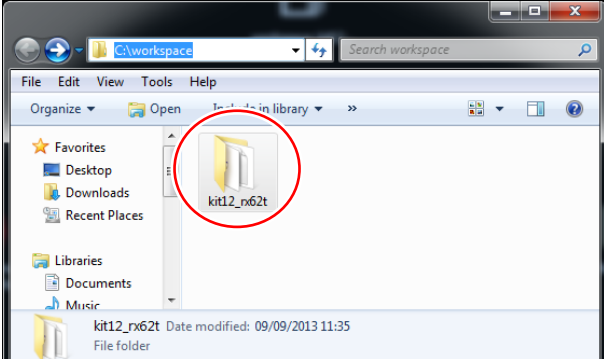
The Renesas integrated development environment is used for program development. For instructions on installing and using the Renesas integrated development environment, see *Renesas Integrated Development Environment Operation Manual (Version for RX62T)*.

4.2. Installing the Sample Programs


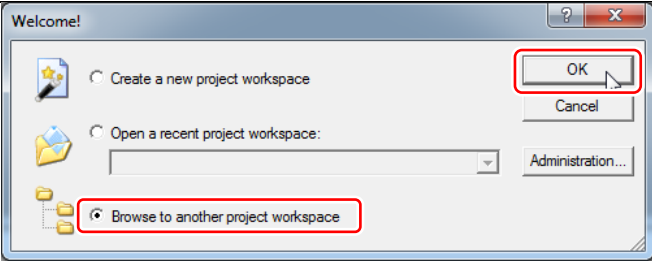
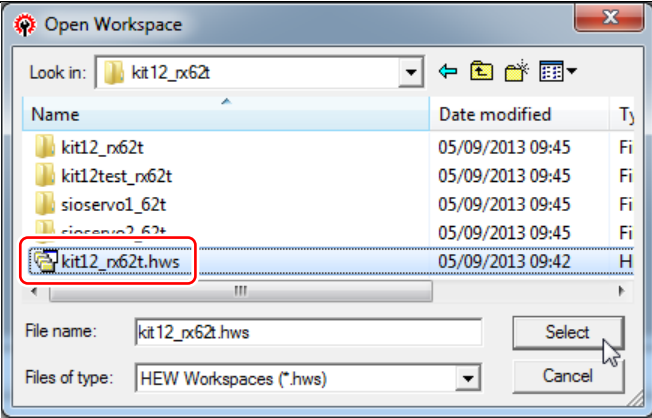
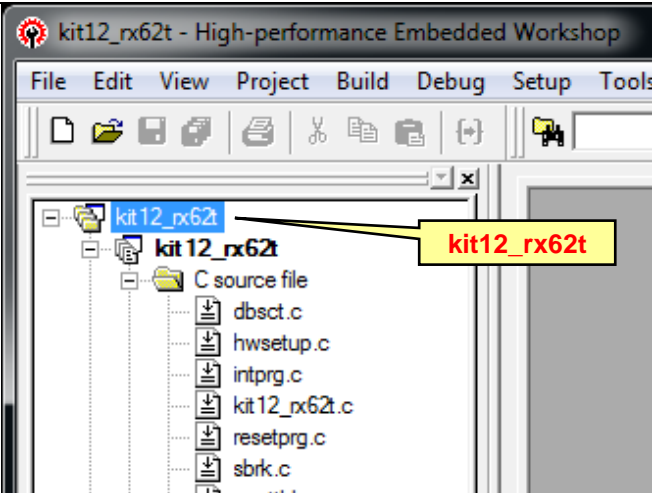
Note: Continue with step 3, if you have CD-R for this seminar.

1		<p>Get the Sample Program (workspace_rx62t_100_eng.exe) from the Renesas site.</p> <p>Renesas Electronics http://www.renesas.com/company_info/carrally/</p> <p>Click Download</p>
2		<p>Download workspace_rx62t_100_eng.exe</p>
3		<p>Run workspace_rx62t_100_eng.exe.</p> <p>Please execute "workspace_rx62t_100_eng.exe" in the following folder, if you have CD-R for this seminar. "CD-R drive:¥ 04-Programs"</p>
4		<p>The installed file is in "C:¥WorkSpace".</p> <p>Click OK.</p>

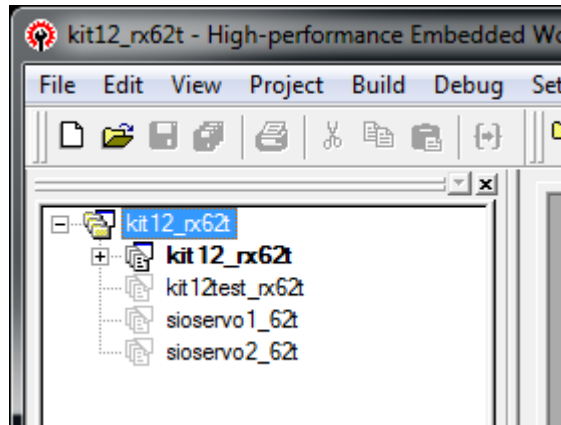
5		Installation has been completed. Click This program installed correctly .
---	---	---

6		Open The "C:\workspace" folder There is the operation test program at the folder "kit12_rx62t".
---	--	--

4.3. Opening the kit12_rx62t Workspace

1		<p>Launch the Renesas integrated development environment.</p>
2		<p>Select Browse to another project workspace.</p>
3		<p>Select kit12_rx62t.hws from the C:\¥Workspace¥kit12_rx62t folder.</p>
4		<p>The kit12_rx62t workspace opens.</p>

4.4. Project



One project is registered in the kit12_rx62t workspace.

Project	Contents
kit12_rx62t	This is the MCU car running program. The explanation of this program starts in the next section of this manual.
kit12test_rx62t	This is a program for testing the components of the completed MCU car, such as the motor drive board and sensor board, to see if they operate correctly. For details, please refer to "Operation Test Manual MCU Car Kit, Ver.5.1 (RX62T Version)".
sioservo1_62t	This is the program for adjusting the servo centre. For details, please refer to "6. Adjusting the Servo Centre and Maximum Turn Angle".
sioservo2_62t	This is the program for determining the maximum turning angle of the servo. For details, please refer to "6. Adjusting the Servo Centre and Maximum Turn Angle".

5. Program Explanation – kit12_rx62t.c

5.1. Program Code Listing

A code listing of the program that controls the MCU car by using the RX62T MCU appears below.

```

1 : /*******/
2 : /* Supported Microcontroller:RX62T */
3 : /* File: kit12_rx62t.c */
4 : /* File Contents: MCU Car Trace Basic Program(RX62T version) */
5 : /* Version number: Ver. 1.00 */
6 : /* Date: 2013.09.01 */
7 : /* Copyright: Renesas Micom Car Rally Secretariat */
8 : /*******/
9 : /*
10 : This program supports the following boards:
11 : * RMC-RX62T board
12 : * Sensor board Ver. 5
13 : * Motor drive board Ver. 5
14 : */
15 :
16 : /*=====*/
17 : /* Include */
18 : /*=====*/
19 : #include "iodefine.h"
20 :
21 : /*=====*/
22 : /* Symbol definitions */
23 : /*=====*/
24 :
25 : /* Constant settings */
26 : #define PWM_CYCLE 24575 /* Motor PWM period (16ms) */
27 : #define SERVO_CENTER 2300 /* Servo center value */
28 : #define HANDLE_STEP 13 /* 1 degree value */
29 :
30 : /* Masked value settings X:masked (disabled) 0:not masked (enabled) */
31 : #define MASK2_2 0x66 /* X 0 0 X X 0 0 X */
32 : #define MASK2_0 0x60 /* X 0 0 X X X X X */
33 : #define MASK0_2 0x06 /* X X X X X 0 0 X */
34 : #define MASK3_3 0xe7 /* 0 0 0 X X 0 0 0 */
35 : #define MASK0_3 0x07 /* X X X X X 0 0 0 */
36 : #define MASK3_0 0xe0 /* 0 0 0 X X X X X */
37 : #define MASK4_0 0xf0 /* 0 0 0 0 X X X X */
38 : #define MASK0_4 0x0f /* X X X X 0 0 0 0 */
39 : #define MASK4_4 0xff /* 0 0 0 0 0 0 0 0 */
40 :
41 : /*=====*/
42 : /* Prototype declarations */
43 : /*=====*/
44 : void init(void);
45 : void timer( unsigned long timer_set );
46 : unsigned char sensor_inp( unsigned char mask );
47 : unsigned char startbar_get( void );
48 : int check_crossline( void );
49 : int check_rightline( void );
50 : int check_leftline( void );
51 : unsigned char dipsw_get( void );
52 : unsigned char buttonsw_get( void );
53 : unsigned char pushsw_get( void );
54 : void led_out_m( unsigned char led );
55 : void led_out( unsigned char led );
56 : void motor( int accele_l, int accele_r );
57 : void handle( int angle );
58 :
59 : /*=====*/
60 : /* Global variable declarations */
61 : /*=====*/
62 : unsigned long cnt0;
63 : unsigned long cnt1;
64 : int pattern;
65 :
66 : /*******/
67 : /* Main program */
68 : /*******/
69 : void main(void)
70 : {

```



```

71 :      /* Initialize MCU functions */
72 :      init();
73 :
74 :      /* Initialize micom car state */
75 :      handle( 0 );
76 :      motor( 0, 0 );
77 :
78 :      while( 1 ) {
79 :          switch( pattern ) {
80 :
81 :              /******
82 :              Pattern-related
83 :              0: wait for switch input
84 :              1: check if start bar is open
85 :              11: normal trace
86 :              12: check end of large turn to right
87 :              13: check end of large turn to left
88 :              21: processing at 1st cross line
89 :              22: read but ignore 2nd time
90 :              23: trace, crank detection after cross line
91 :              31: left crank clearing processing ? wait until stable
92 :              32: left crank clearing processing ? check end of turn
93 :              41: right crank clearing processing ? wait until stable
94 :              42: right crank clearing processing ? check end of turn
95 :              51: processing at 1st right half line detection
96 :              52: read but ignore 2nd line
97 :              53: trace after right half line detection
98 :              54: right lane change end check
99 :              61: processing at 1st left half line detection
100 :             62: read but ignore 2nd line
101 :             63: trace after left half line detection
102 :             64: left lane change end check
103 :             *****/
104 :
105 :             case 0:
106 :                 /* Wait for switch input */
107 :                 if( pushsw_get() ) {
108 :                     pattern = 1;
109 :                     cnt1 = 0;
110 :                     break;
111 :                 }
112 :                 if( cnt1 < 100 ) {           /* LED flashing processing   */
113 :                     led_out( 0x1 );
114 :                 } else if( cnt1 < 200 ) {
115 :                     led_out( 0x2 );
116 :                 } else {
117 :                     cnt1 = 0;
118 :                 }
119 :                 break;
120 :
121 :             case 1:
122 :                 /* Check if start bar is open */
123 :                 if( !startbar_get() ) {
124 :                     /* Start!! */
125 :                     led_out( 0x0 );
126 :                     pattern = 11;
127 :                     cnt1 = 0;
128 :                     break;
129 :                 }
130 :                 if( cnt1 < 50 ) {           /* LED flashing processing   */
131 :                     led_out( 0x1 );
132 :                 } else if( cnt1 < 100 ) {
133 :                     led_out( 0x2 );
134 :                 } else {
135 :                     cnt1 = 0;
136 :                 }
137 :                 break;
138 :
139 :             case 11:
140 :                 /* Normal trace */
141 :                 if( check_crossline() ) { /* Cross line check           */
142 :                     pattern = 21;
143 :                     break;
144 :                 }
145 :                 if( check_rightline() ) { /* Right half line detection check */
146 :                     pattern = 51;
147 :                     break;
148 :                 }
149 :                 if( check_leftline() ) { /* Left half line detection check */
150 :                     pattern = 61;

```

```

151 :         break;
152 :     }
153 :     switch( sensor_inp(MASK3_3) ) {
154 :         case 0x00:
155 :             /* Center -> straight */
156 :             handle( 0 );
157 :             motor( 100 , 100 );
158 :             break;
159 :
160 :         case 0x04:
161 :             /* Slight amount left of center -> slight turn to right */
162 :             handle( 5 );
163 :             motor( 100 , 100 );
164 :             break;
165 :
166 :         case 0x06:
167 :             /* Small amount left of center -> small turn to right */
168 :             handle( 10 );
169 :             motor( 80 , 67 );
170 :             break;
171 :
172 :         case 0x07:
173 :             /* Medium amount left of center -> medium turn to right */
174 :             handle( 15 );
175 :             motor( 50 , 38 );
176 :             break;
177 :
178 :         case 0x03:
179 :             /* Large amount left of center -> large turn to right */
180 :             handle( 25 );
181 :             motor( 30 , 19 );
182 :             pattern = 12;
183 :             break;
184 :
185 :         case 0x20:
186 :             /* Slight amount right of center -> slight turn to left */
187 :             handle( -5 );
188 :             motor( 100 , 100 );
189 :             break;
190 :
191 :         case 0x60:
192 :             /* Small amount right of center -> small turn to left */
193 :             handle( -10 );
194 :             motor( 67 , 80 );
195 :             break;
196 :
197 :         case 0xe0:
198 :             /* Medium amount right of center -> medium turn to left */
199 :             handle( -15 );
200 :             motor( 38 , 50 );
201 :             break;
202 :
203 :         case 0xc0:
204 :             /* Large amount right of center -> large turn to left */
205 :             handle( -25 );
206 :             motor( 19 , 30 );
207 :             pattern = 13;
208 :             break;
209 :
210 :         default:
211 :             break;
212 :     }
213 :     break;
214 :
215 : case 12:
216 :     /* Check end of large turn to right */
217 :     if( check_crossline() ) { /* Cross line check during large turn */
218 :         pattern = 21;
219 :         break;
220 :     }
221 :     if( check_rightline() ) { /* Right half line detection check */
222 :         pattern = 51;
223 :         break;
224 :     }
225 :     if( check_leftline() ) { /* Left half line detection check */
226 :         pattern = 61;
227 :         break;
228 :     }
229 :     if( sensor_inp(MASK3_3) == 0x06 ) {
230 :         pattern = 11;

```

```

231 :     }
232 :     break;
233 :
234 :     case 13:
235 :         /* Check end of large turn to left */
236 :         if( check_crossline() ) { /* Cross line check during large turn */
237 :             pattern = 21;
238 :             break;
239 :         }
240 :         if( check_rightline() ) { /* Right half line detection check */
241 :             pattern = 51;
242 :             break;
243 :         }
244 :         if( check_leftline() ) { /* Left half line detection check */
245 :             pattern = 61;
246 :             break;
247 :         }
248 :         if( sensor_inp(MASK3_3) == 0x60 ) {
249 :             pattern = 11;
250 :         }
251 :         break;
252 :
253 :     case 21:
254 :         /* Processing at 1st cross line */
255 :         led_out( 0x3 );
256 :         handle( 0 );
257 :         motor( 0 , 0 );
258 :         pattern = 22;
259 :         cnt1 = 0;
260 :         break;
261 :
262 :     case 22:
263 :         /* Read but ignore 2nd line */
264 :         if( cnt1 > 100 ){
265 :             pattern = 23;
266 :             cnt1 = 0;
267 :         }
268 :         break;
269 :
270 :     case 23:
271 :         /* Trace, crank detection after cross line */
272 :         if( sensor_inp(MASK4_4)==0xf8 ) {
273 :             /* Left crank determined -> to left crank clearing processing */
274 :             led_out( 0x1 );
275 :             handle( -38 );
276 :             motor( 10 , 50 );
277 :             pattern = 31;
278 :             cnt1 = 0;
279 :             break;
280 :         }
281 :         if( sensor_inp(MASK4_4)==0x1f ) {
282 :             /* Right crank determined -> to right crank clearing processing */
283 :             led_out( 0x2 );
284 :             handle( 38 );
285 :             motor( 50 , 10 );
286 :             pattern = 41;
287 :             cnt1 = 0;
288 :             break;
289 :         }
290 :         switch( sensor_inp(MASK3_3) ) {
291 :             case 0x00:
292 :                 /* Center -> straight */
293 :                 handle( 0 );
294 :                 motor( 40 , 40 );
295 :                 break;
296 :             case 0x04:
297 :             case 0x06:
298 :             case 0x07:
299 :             case 0x03:
300 :                 /* Left of center -> turn to right */
301 :                 handle( 8 );
302 :                 motor( 40 , 35 );
303 :                 break;
304 :             case 0x20:
305 :             case 0x60:
306 :             case 0xe0:
307 :             case 0xc0:
308 :                 /* Right of center -> turn to left */
309 :                 handle( -8 );
310 :                 motor( 35 , 40 );

```

```

311 :             break;
312 :         }
313 :     break;
314 :
315 :     case 31:
316 :         /* Left crank clearing processing ? wait until stable */
317 :         if( cnt1 > 200 ) {
318 :             pattern = 32;
319 :             cnt1 = 0;
320 :         }
321 :         break;
322 :
323 :     case 32:
324 :         /* Left crank clearing processing ? check end of turn */
325 :         if( sensor_inp(MASK3_3) == 0x60 ) {
326 :             led_out( 0x0 );
327 :             pattern = 11;
328 :             cnt1 = 0;
329 :         }
330 :         break;
331 :
332 :     case 41:
333 :         /* Right crank clearing processing ? wait until stable */
334 :         if( cnt1 > 200 ) {
335 :             pattern = 42;
336 :             cnt1 = 0;
337 :         }
338 :         break;
339 :
340 :     case 42:
341 :         /* Right crank clearing processing ? check end of turn */
342 :         if( sensor_inp(MASK3_3) == 0x06 ) {
343 :             led_out( 0x0 );
344 :             pattern = 11;
345 :             cnt1 = 0;
346 :         }
347 :         break;
348 :
349 :     case 51:
350 :         /* Processing at 1st right half line detection */
351 :         led_out( 0x2 );
352 :         handle( 0 );
353 :         motor( 0 , 0 );
354 :         pattern = 52;
355 :         cnt1 = 0;
356 :         break;
357 :
358 :     case 52:
359 :         /* Read but ignore 2nd time */
360 :         if( cnt1 > 100 ){
361 :             pattern = 53;
362 :             cnt1 = 0;
363 :         }
364 :         break;
365 :
366 :     case 53:
367 :         /* Trace, lane change after right half line detection */
368 :         if( sensor_inp(MASK4_4) == 0x00 ) {
369 :             handle( 15 );
370 :             motor( 40 , 31 );
371 :             pattern = 54;
372 :             cnt1 = 0;
373 :             break;
374 :         }
375 :         switch( sensor_inp(MASK3_3) ) {
376 :             case 0x00:
377 :                 /* Center -> straight */
378 :                 handle( 0 );
379 :                 motor( 40 , 40 );
380 :                 break;
381 :             case 0x04:
382 :             case 0x06:
383 :             case 0x07:
384 :             case 0x03:
385 :                 /* Left of center -> turn to right */
386 :                 handle( 8 );
387 :                 motor( 40 , 35 );
388 :                 break;
389 :             case 0x20:
390 :             case 0x60:

```

```

391 :             case 0xe0:
392 :             case 0xc0:
393 :                 /* Right of center -> turn to left */
394 :                 handle( -8 );
395 :                 motor( 35 , 40 );
396 :                 break;
397 :             default:
398 :                 break;
399 :         }
400 :         break;
401 :
402 :     case 54:
403 :         /* Right lane change end check */
404 :         if( sensor_inp( MASK4_4 ) == 0x3c ) {
405 :             led_out( 0x0 );
406 :             pattern = 11;
407 :             cnt1 = 0;
408 :         }
409 :         break;
410 :
411 :     case 61:
412 :         /* Processing at 1st left half line detection */
413 :         led_out( 0x1 );
414 :         handle( 0 );
415 :         motor( 0 , 0 );
416 :         pattern = 62;
417 :         cnt1 = 0;
418 :         break;
419 :
420 :     case 62:
421 :         /* Read but ignore 2nd time */
422 :         if( cnt1 > 100 ){
423 :             pattern = 63;
424 :             cnt1 = 0;
425 :         }
426 :         break;
427 :
428 :     case 63:
429 :         /* Trace, lane change after left half line detection */
430 :         if( sensor_inp(MASK4_4) == 0x00 ) {
431 :             handle( -15 );
432 :             motor( 31 , 40 );
433 :             pattern = 64;
434 :             cnt1 = 0;
435 :             break;
436 :         }
437 :         switch( sensor_inp(MASK3_3) ) {
438 :             case 0x00:
439 :                 /* Center -> straight */
440 :                 handle( 0 );
441 :                 motor( 40 , 40 );
442 :                 break;
443 :             case 0x04:
444 :             case 0x06:
445 :             case 0x07:
446 :             case 0x03:
447 :                 /* Left of center -> turn to right */
448 :                 handle( 8 );
449 :                 motor( 40 , 35 );
450 :                 break;
451 :             case 0x20:
452 :             case 0x60:
453 :             case 0xe0:
454 :             case 0xc0:
455 :                 /* Right of center -> turn to left */
456 :                 handle( -8 );
457 :                 motor( 35 , 40 );
458 :                 break;
459 :             default:
460 :                 break;
461 :         }
462 :         break;
463 :
464 :     case 64:
465 :         /* Left lane change end check */
466 :         if( sensor_inp( MASK4_4 ) == 0x3c ) {
467 :             led_out( 0x0 );
468 :             pattern = 11;
469 :             cnt1 = 0;
470 :         }

```

```

471 :         break;
472 :
473 :     default:
474 :         /* If neither, return to standby state */
475 :         pattern = 0;
476 :         break;
477 :     }
478 : }
479 : }
480 :
481 : /*****
482 : /* RX62T Initialization */
483 : *****/
484 : void init(void)
485 : {
486 :     // System Clock
487 :     SYSTEM.SCKCR.BIT.ICK = 0;           //12.288*8=98.304MHz
488 :     SYSTEM.SCKCR.BIT.PCK = 1;         //12.288*4=49.152MHz
489 :
490 :     // Port I/O Settings
491 :     PORT1.DDR.BYTE = 0x03;             //P10:LED2 in motor drive board
492 :
493 :     PORT2.DR.BYTE = 0x08;
494 :     PORT2.DDR.BYTE = 0x1b;             //P24:SDCARD_CLK(o)
495 :                                         //P23:SDCARD_DI(o)
496 :                                         //P22:SDCARD_DO(i)
497 :                                         //CN:P21-P20
498 :     PORT3.DR.BYTE = 0x01;
499 :     PORT3.DDR.BYTE = 0x0f;            //CN:P33-P31
500 :                                         //P30:SDCARD_CS(o)
501 :     //PORT4:input                       //sensor input
502 :     //PORT5:input
503 :     //PORT6:input
504 :
505 :     PORT7.DDR.BYTE = 0x7e;             //P76:LED3 in motor drive board
506 :                                         //P75:forward reverse signal(right motor)
507 :                                         //P74:forward reverse signal(left motor)
508 :                                         //P73:PWM(right motor)
509 :                                         //P72:PWM(left motor)
510 :                                         //P71:PWM(servo motor)
511 :                                         //P70:Push-button in motor drive board
512 :     PORT8.DDR.BYTE = 0x07;             //CN:P82-P80
513 :     PORT9.DDR.BYTE = 0x7f;             //CN:P96-P90
514 :     PORTA.DR.BYTE = 0x0f;             //CN:PA5-PA4
515 :                                         //PA3:LED3(o)
516 :                                         //PA2:LED2(o)
517 :                                         //PA1:LED1(o)
518 :                                         //PA0:LEDO(o)
519 :     PORTA.DDR.BYTE = 0x3f;             //CN:PA5-PA0
520 :     PORTB.DDR.BYTE = 0xff;            //CN:PB7-PB0
521 :     PORTD.DDR.BYTE = 0x0f;            //PD7:TRST#(i)
522 :                                         //PD5:TDI(i)
523 :                                         //PD4:TCK(i)
524 :                                         //PD3:TDO(o)
525 :                                         //CN:PD2-PD0
526 :     PORTE.DDR.BYTE = 0x1b;            //PE5:SW(i)
527 :                                         //CN:PE4-PE0
528 :
529 :     // Compare match timer
530 :     MSTP_CMT0 = 0;                     //CMT Release module stop state
531 :     MSTP_CMT2 = 0;                     //CMT Release module stop state
532 :
533 :     ICU.IPR[0x04].BYTE = 0x0f;         //CMT0_CMI0 Priority of interrupts
534 :     ICU.IER[0x03].BIT.IEN4 = 1;       //CMT0_CMI0 Permission for interrupt
535 :     CMT0.CMSTRO.WORD = 0x0000;         //CMT0, CMT1 Stop counting
536 :     CMT0.CMCR.WORD = 0x00C3;           //PCLK/512
537 :     CMT0.CMCNT = 0;
538 :     CMT0.CMCOR = 96;                   //1ms/(1/(49.152MHz/512))
539 :     CMT0.CMSTRO.WORD = 0x0003;         //CMT0, CMT1 Start counting
540 :
541 :     // MTU3_3 MTU3_4 PWM mode synchronized by RESET
542 :     MSTP_MTU = 0;                       //Release module stop state
543 :     MTU.TSTRA.BYTE = 0x00;             //MTU Stop counting
544 :
545 :     MTU3.TCR.BYTE = 0x23;               //ILCK/64(651.04ns)
546 :     MTU3.TCNT = MTU4.TCNT = 0;         //MTU3, MTU4TCNT clear
547 :     MTU3.TGRA = MTU3.TGRC = PWM_CYCLE; //cycle(16ms)
548 :     MTU3.TGRB = MTU3.TGRD = SERVO_CENTER; //PWM(servo motor)
549 :     MTU4.TGRA = MTU4.TGRC = 0;         //PWM(left motor)
550 :     MTU4.TGRB = MTU4.TGRD = 0;         //PWM(right motor)

```

```

551 :     MTU.TOCR1A.BYTE = 0x40;           //Selection of output level
552 :     MTU3.TMDR.BYTE = 0x38;         //TGRC,TGRD buffer function
553 :                                     //PWM mode synchronized by RESET
554 :     MTU4.TMDR.BYTE = 0x00;         //Set 0 to exclude MTU3 effects
555 :     MTU.TOERA.BYTE = 0xc7;         //MTU3TGRB,MTU4TGRA,MTU4TGRB permission for output
556 :
557 :     MTU.TSTRA.BYTE = 0x40;         //MTU0,MTU3 count function
558 : }
559 :
560 : /*****
561 : /* Interrupt                                     */
562 : /*****
563 : #pragma interrupt Excep_CMT0_CMI0(vect=28)
564 : void Excep_CMT0_CMI0(void)
565 : {
566 :     cnt0++;
567 :     cnt1++;
568 : }
569 :
570 : /*****
571 : /* Timer unit                                     */
572 : /* Arguments: timer value, 1 = 1 ms             */
573 : /*****
574 : void timer( unsigned long timer_set )
575 : {
576 :     cnt0 = 0;
577 :     while( cnt0 < timer_set );
578 : }
579 :
580 : /*****
581 : /* Sensor state detection                       */
582 : /* Arguments:      masked values                */
583 : /* Return values:  sensor value                */
584 : /*****
585 : unsigned char sensor_inp( unsigned char mask )
586 : {
587 :     unsigned char sensor;
588 :
589 :     sensor = ~PORT4.PORT.BYTE;
590 :
591 :     sensor &= mask;
592 :
593 :     return sensor;
594 : }
595 :
596 : /*****
597 : /* Read start bar detection sensor              */
598 : /* Return values: Sensor value, ON (bar present):1,
599 : /*               OFF (no bar present):0        */
600 : /*****
601 : unsigned char startbar_get( void )
602 : {
603 :     unsigned char b;
604 :
605 :     b = ~PORT4.PORT.BIT.B0 & 0x01;    /* Read start bar signal    */
606 :
607 :     return b;
608 : }
609 :
610 : /*****
611 : /* Cross line detection processing              */
612 : /* Return values: 0: no cross line, 1: cross line
613 : /*               */
614 : int check_crossline( void )
615 : {
616 :     unsigned char b;
617 :     int ret;
618 :
619 :     ret = 0;
620 :     b = sensor_inp(MASK3_3);
621 :     if( b==0xe7 ) {
622 :         ret = 1;
623 :     }
624 :     return ret;
625 : }
626 :
627 : /*****
628 : /* Right half line detection processing         */
629 : /* Return values: 0: not detected, 1: detected */
630 : /*****

```

```

631 : int check_rightline( void )
632 : {
633 :     unsigned char b;
634 :     int ret;
635 :
636 :     ret = 0;
637 :     b = sensor_inp(MASK4_4);
638 :     if( b==0x1f ) {
639 :         ret = 1;
640 :     }
641 :     return ret;
642 : }
643 :
644 : /*****
645 : /* Left half line detection processing */
646 : /* Return values: 0: not detected, 1: detected */
647 : *****/
648 : int check_leftline( void )
649 : {
650 :     unsigned char b;
651 :     int ret;
652 :
653 :     ret = 0;
654 :     b = sensor_inp(MASK4_4);
655 :     if( b==0xf8 ) {
656 :         ret = 1;
657 :     }
658 :     return ret;
659 : }
660 :
661 : /*****
662 : /* DIP switch value read */
663 : /* Return values: Switch value, 0 to 15 */
664 : *****/
665 : unsigned char dipsw_get( void )
666 : {
667 :     unsigned char sw, d0, d1, d2, d3;
668 :
669 :     d0 = ( PORT6.PORT.BIT.B3 & 0x01 ); /* P63~P60 read */
670 :     d1 = ( PORT6.PORT.BIT.B2 & 0x01 ) << 1;
671 :     d2 = ( PORT6.PORT.BIT.B1 & 0x01 ) << 2;
672 :     d3 = ( PORT6.PORT.BIT.B0 & 0x01 ) << 3;
673 :     sw = d0 | d1 | d2 | d3;
674 :
675 :     return sw;
676 : }
677 :
678 : /*****
679 : /* Push-button in MCU board value read */
680 : /* Return values: Switch value, ON: 1, OFF: 0 */
681 : *****/
682 : unsigned char buttonsw_get( void )
683 : {
684 :     unsigned char sw;
685 :
686 :     sw = ~PORTE.PORT.BIT.B5 & 0x01; /* Read ports with switches */
687 :
688 :     return sw;
689 : }
690 :
691 : /*****
692 : /* Push-button in motor drive board value read */
693 : /* Return values: Switch value, ON: 1, OFF: 0 */
694 : *****/
695 : unsigned char pushsw_get( void )
696 : {
697 :     unsigned char sw;
698 :
699 :     sw = ~PORT7.PORT.BIT.B0 & 0x01; /* Read ports with switches */
700 :
701 :     return sw;
702 : }
703 :
704 : /*****
705 : /* LED control in MCU board */
706 : /* Arguments: Switch value, LED0: bit 0, LED1: bit 1. 0: dark, 1: lit */
707 : /* */
708 : *****/
709 : void led_out_m( unsigned char led )
710 : {

```



```

711 :     led = ~led;
712 :     PORTA.DR.BYTE = led & 0x0f;
713 : }
714 :
715 : /*****
716 : /* LED control in motor drive board */
717 : /* Arguments: Switch value, LED0: bit 0, LED1: bit 1. 0: dark, 1: lit */
718 : /* Example: 0x3 -> LED1: ON, LED0: ON, 0x2 -> LED1: ON, LED0: OFF */
719 : *****/
720 : void led_out( unsigned char led )
721 : {
722 :     led = ~led;
723 :     PORT7.DR.BIT.B6 = led & 0x01;
724 :     PORT1.DR.BIT.B0 = ( led >> 1 ) & 0x01;
725 : }
726 :
727 : /*****
728 : /* Motor speed control */
729 : /* Arguments: Left motor: -100 to 100, Right motor: -100 to 100 */
730 : /* Here, 0 is stopped, 100 is forward, and -100 is reverse. */
731 : /* Return value: None */
732 : *****/
733 : void motor( int accele_l, int accele_r )
734 : {
735 :     int sw_data;
736 :
737 :     sw_data = dipsw_get() + 5;
738 :     accele_l = accele_l * sw_data / 20;
739 :     accele_r = accele_r * sw_data / 20;
740 :
741 :     /* Left Motor Control */
742 :     if( accele_l >= 0 ) {
743 :         PORT7.DR.BYTE &= 0xef;
744 :         MTU4.TGRC = (long)( PWM_CYCLE - 1 ) * accele_l / 100;
745 :     } else {
746 :         PORT7.DR.BYTE |= 0x10;
747 :         MTU4.TGRC = (long)( PWM_CYCLE - 1 ) * ( -accele_l ) / 100;
748 :     }
749 :
750 :     /* Right Motor Control */
751 :     if( accele_r >= 0 ) {
752 :         PORT7.DR.BYTE &= 0xdf;
753 :         MTU4.TGRD = (long)( PWM_CYCLE - 1 ) * accele_r / 100;
754 :     } else {
755 :         PORT7.DR.BYTE |= 0x20;
756 :         MTU4.TGRD = (long)( PWM_CYCLE - 1 ) * ( -accele_r ) / 100;
757 :     }
758 : }
759 :
760 : /*****
761 : /* Servo steering operation */
762 : /* Arguments: servo operation angle: -90 to 90 */
763 : /* -90: 90-degree turn to left, 0: straight, */
764 : /* 90: 90-degree turn to right */
765 : *****/
766 : void handle( int angle )
767 : {
768 :     /* When the servo move from left to right in reverse, replace "-" with "+". */
769 :     MTU3.TGRD = SERVO_CENTER - angle * HANDLE_STEP;
770 : }
771 :
772 : /*****
773 : /* end of file */
774 : *****/

```

5.2. Differences between programs for kit07_rx62t.c and kit12_rx62t.c

The points of difference between kit07_rx62t.c and kit12_rx62t.c, are listed in the table below.

	kit07_rx62t.c (for use Sensor Board Ver.4)	kit12_rx62t.c (for use Sensor Board Ver.5)
sensor_inp function	<pre> unsigned char sensor_inp(unsigned char mask) { unsigned char sensor; sensor = ~PORT4.PORT.BYTE; sensor &= 0xef; if(sensor & 0x08) sensor = 0x10; sensor &= mask; return sensor; } </pre>	<pre> unsigned char sensor_inp(unsigned char mask) { unsigned char sensor; sensor = ~PORT4.PORT.BYTE; // delete this line // delete this line sensor &= mask; return sensor; } </pre>
startbar_get function	<pre> unsigned char startbar_get(void) { unsigned char b; b = ~PORT4.PORT.BIT.B4 & 0x01; return b; } </pre>	<pre> unsigned char startbar_get(void) { unsigned char b; b = ~ PORT4.PORT.BIT.B0 & 0x01; return b; } </pre>

5.3. On-Chip Peripheral Functions of RX62T MCU Used by the Program

The on-chip peripheral modules used for control on the RMC-RX62T board (RX62T MCU) included in the MCU Car Rally Kit, Ver. 5.1, are listed below.

Items	On-Chip Peripheral Module of RMC-RX62T used to Control MCU Car
Generating interrupts at 1 ms intervals 1ms	CMT
Control of left motor, right motor, and servo	Reset-synchronized PWM mode using MTU3
Rotary encoder* (pulse count)	MTU0

Note: * These are not covered in *Program Explanation Manual* (this manual).

5.4. Program Explanation

5.4.1. Start

```

1 : /*****
2 : /* Supported Microcontroller:RX62T */
3 : /* File: kit12_rx62t.c */
4 : /* File Contents: MCU Car Trace Basic Program(RX62T version) */
5 : /* Version number: Ver.1.00 */
6 : /* Date: 2013.09.01 */
7 : /* Copyright: Renesas Micom Car Rally Secretariat */
8 : *****/

```

First comes the comments section. The beginning of a comment is designated by /* and the end of a comment by */. All characters from the beginning to the end of a comment are ignored by the compiler. Comment lines are used to include notes about the program.

5.4.2. Including External Files

```

16 : /*=====*/
17 : /* Include */
18 : /*=====*/
19 : #include "iodefine.h"

```

The #include statement is used to include (call) an external file.


File Name	Description
iodefine.h	This file defines registers used to control the on-chip peripheral functions of RX62T.

5.4.3. Symbol Definitions

```

21 : /*=====*/
22 : /* Symbol definitions */
23 : /*=====*/
24 :
25 : /* Constant settings */
26 : #define PWM_CYCLE 24575 /* Motor PWM period (16ms) */
27 : #define SERVO_CENTER 2300 /* Servo center value */
28 : #define HANDLE_STEP 13 /* 1 degree value */
29 :
30 : /* Masked value settings X:masked (disabled) 0:not masked (enabled) */
31 : #define MASK2_2 0x66 /* X 0 0 X X 0 0 X */
32 : #define MASK2_0 0x60 /* X 0 0 X X X X X */
33 : #define MASK0_2 0x06 /* X X X X X 0 0 X */
34 : #define MASK3_3 0xe7 /* 0 0 0 X X 0 0 0 */
35 : #define MASK0_3 0x07 /* X X X X X 0 0 0 */
36 : #define MASK3_0 0xe0 /* 0 0 0 X X X X X */
37 : #define MASK4_0 0xf0 /* 0 0 0 0 X X X X */
38 : #define MASK0_4 0x0f /* X X X X 0 0 0 0 */
39 : #define MASK4_4 0xff /* 0 0 0 0 0 0 0 0 */

```

<p>PWM_CYCLE</p>	<p>PWM_CYCLE sets the PWM cycle for the signals applied to the right motor, left motor, and servo. Here it is set to a PWM cycle of 16 ms. The value is calculated as follows:</p> $\begin{aligned} \text{PWM_CYCLE} &= \text{desired PWM cycle} / \text{MTU3_3 timer general register C (MTU3.TGRC) count source} - 1 \\ &= 16 \text{ ms} / 651.04 \text{ ns} - 1 \\ &= (16 \times 10^{-3}) / (651.04 \times 10^{-9}) - 1 \\ &= 24576 - 1 = \mathbf{24575} \end{aligned}$ <p>For a detailed explanation, see the discussion of reset-synchronized PWM mode.</p>
<p>SERVO_CENTER</p>	<p>SERVO_CENTER sets the value at which the servo angle is 0 degrees (pointing straight ahead). A standard servo will point forward when a 1.5 [ms] pulse width is applied. Therefore, the pulse on width is set at 1.5 ms. The SERVO_CENTER setting value is calculated as follows:</p> $\begin{aligned} \text{SERVO_CENTER} &= \text{pulse on width} / \text{MTU3_3 timer general register C (MTU3.TGRC) count source select bit setting} - 1 \\ &= 1.5 \text{ ms} / 651.04 \text{ ns} - 1 \\ &= (1.5 \times 10^{-3}) / (651.04 \times 10^{-9}) - 1 \\ &= 2304 \end{aligned}$ <p>Calculated servo centre is 2304. In this sample program, 2300 is used for servo centre. However, the actual servo centre value is slightly different for every MCU car because of factors such as variation among individual servos and the way the grooves in the holes in the servo horn match up. This is analogous to the way that everyone’s fingerprints are different. For this reason, this value needs to be changed for each MCU car to adjust the servo angle such that the car runs in a straight line when the software specifies a turning angle of 0 degrees.</p> <div style="text-align: right;">  <p>▲ Servo horn</p> </div>
<p>HANDLE_STEP</p>	<p>HANDLE_STEP sets a value equivalent to 1 degree of servo movement. A 0.7 ms PWM on width causes the servo to turn 90 degrees to the left, and a 2.3 ms on width causes it to turn 90 degrees to the right. If we divide the difference between these two by 180, we can obtain the value equivalent to 1 degree.</p> <ul style="list-style-type: none"> On width of 90 degrees left $\begin{aligned} \text{MTU3.TGRD} + \text{PWM waveform on width} / \text{MTU3_3 timer counter count source} - 1 \\ &= (0.7 \times 10^{-3}) / (651.04 \times 10^{-9}) - 1 \\ &= 1075 - 1 = 1074 \end{aligned}$ On width of 90 degrees right $\begin{aligned} \text{MTU3.TGRD} + \text{PWM waveform on width} / \text{MTU3_3 timer counter count source} - 1 \\ &= (2.3 \times 10^{-3}) / (651.04 \times 10^{-9}) - 1 \\ &= 3532 - 1 = 3531 \end{aligned}$ Value equivalent to 1 degree $(\text{Right} - \text{left}) / 180 = (3531 - 1074) / 180 = 13.65 \approx 13$ <p>Therefore, the value of HANDLE_STEP is defined as 13. Change this value to adjust the value equivalent to 1 degree of servo movement.</p>
<p>MASK2_2 MASK2_0 MASK0_2 MASK3_3 MASK0_3 MASK3_0 MASK4_0 MASK0_4 MASK4_4</p>	<p>The sensor_inp function defines common mask values used when masking sensor values. These values are defined in the format MASK + A + _ (underscore) + B.</p> <ul style="list-style-type: none"> A: Of the four sensors on the left, A sensors are valid (unmasked). B: Of the four sensors on the right, B sensors are valid (unmasked). The other sensors are masked. <p>For details, see 6.4.12, sensor_inp Function.</p>

5.4.4. Prototype Declarations

```

41 : /*=====*/
42 : /* Prototype declarations          */
43 : /*=====*/
44 : void init(void);
45 : void timer( unsigned long timer_set );
46 : unsigned char sensor_inp( unsigned char mask );
47 : unsigned char startbar_get( void );
48 : int check_crossline( void );
49 : int check_rightline( void );
50 : int check_leftline( void );
51 : unsigned char dipsw_get( void );
52 : unsigned char buttonsw_get( void );
53 : unsigned char pushsw_get( void );
54 : void led_out_m( unsigned char led );
55 : void led_out( unsigned char led );
56 : void motor( int accele_l, int accele_r );
57 : void handle( int angle );
58 :

```

Prototype declarations must be made before functions are used to allow checking of the types and quantity of arguments of the user-created functions. A semicolon (;) is appended after a function to indicate a function prototype.

An example prototype declaration is shown below.

```

void motor( int accele_l, int accele_r );          /* Prototype declarations */

void main( void )
{
    int a, b;

    a = 50;
    b = 100;

    motor( a, b );  • • Check to confirm that the first and second arguments are of type int as specified in the prototype declaration. If either argument is not type int, the compiler will return an error.
}

/* Motor control function */
void motor( int accele_l, int accele_r )
{
    Program
}

```

5.4.5. Global Variable Declarations

```

59 : /*=====*/
60 : /* Global variable declarations */
61 : /*=====*/
62 : unsigned long cnt0;
63 : unsigned long cnt1;
64 : int pattern;
65 :
    
```

Global variables are defined separately from functions and may be referenced by any function. By means of comparison, the usual type of variable, which is defined within a function, is called a local variable and may be referenced only within that function.

An example prototype declaration is shown below.

```

void a( void );                /* Prototype declarations */
int timer;                    /* Global variable */

void main( void )
{
    int i;

    timer = 0;
    i = 10;
    printf( "%/n" , timer );   <- 0 is displayed.
    a();
    printf( "%/n" , timer );   <- timer is a global variable,
                                so the value 20 set by function a is displayed.
    printf( "%/n" , i );      <- Function a also uses variable i, but since it is a local variable,
                                the value of variable i within function a is irrelevant.
                                The value of 10 set by this function is displayed.
}

void a( void )
{
    int i;
    i = 20;
    timer = i;
}
    
```

The program kit12_rx62t.c contains three global variable declarations.

Variable name	Type	Usage
cnt0	unsigned long	This function increments the count value by 1 at 1 ms intervals. Used by the timer function to count at 1 ms intervals. The details are described in the section covering the timer function.
cnt1	unsigned long	This function increments the count value by 1 at 1 ms intervals. This variable can be used freely by the program to measure duration. For example, it can be used to “do ○○ if 300 ms has elapsed and do □□ otherwise.” The details are described in the section covering the main function.
pattern	int	This is the pattern number. The details are described in the section covering the main function.

Under the ANSI C standard (the C language standard), un-initialised data must have an initial value of 0x00. Therefore, these variables all have a value of 0.

5.4.6. **init** Function(Clock Choice)

The **init** function initialises the on-chip peripheral functions of the RX62T MCU. The name **init** stands for “initialise.” The **init** function initialises several on-chip peripheral functions. These are described below, broken down by function.

The RX62T MCU board already has a 12.288MHz crystal oscillator.

So choose 98.304MHz (12.288 x 8) for system clock and 49.152MHz (12.288 x 4) for peripheral clock.

```

481 :  /*****/
482 :  /* RX62T Initialization */
483 :  /*****/
484 :  void init(void)
485 :  {
486 :      // System Clock
487 :      SYSTEM.SCKCR.BIT.ICK = 0;          //12.288*8=98.304MHz
488 :      SYSTEM.SCKCR.BIT.PCK = 1;        //12.288*4=49.152MHz

```

5.4.7. **init** Function (Port I/O Settings)

Next, the **init** function makes port I/O settings.

```

490 :      // Port I/O Settings
491 :      PORT1.DDR.BYTE = 0x03;           //P10:LED2 in motor drive board
492 :
493 :      PORT2.DR.BYTE = 0x08;
494 :      PORT2.DDR.BYTE = 0x1b;          //P24:SDCARD_CLK(o)
495 :                                       //P23:SDCARD_DI(o)
496 :                                       //P22:SDCARD_DO(i)
497 :                                       //CN:P21-P20
498 :      PORT3.DR.BYTE = 0x01;
499 :      PORT3.DDR.BYTE = 0x0f;          //CN:P33-P31
500 :                                       //P30:SDCARD_CS(o)
501 :      //PORT4:input                   //sensor input
502 :      //PORT5:input
503 :      //PORT6:input
504 :
505 :      PORT7.DDR.BYTE = 0x7e;          //P76:LED3 in motor drive board
506 :                                       //P75:forward reverse signal(right motor)
507 :                                       //P74:forward reverse signal(left motor)
508 :                                       //P73:PWM(right motor)
509 :                                       //P72:PWM(left motor)
510 :                                       //P71:PWM(servo motor)
511 :                                       //P70:Push-button in motor drive board
512 :      PORT8.DDR.BYTE = 0x07;          //CN:P82-P80
513 :      PORT9.DDR.BYTE = 0x7f;          //CN:P96-P90
514 :      PORTA.DR.BYTE = 0x0f;           //CN:PA5-PA4
515 :                                       //PA3:LED3(o)
516 :                                       //PA2:LED2(o)
517 :                                       //PA1:LED1(o)
518 :                                       //PA0:LED0(o)
519 :      PORTA.DDR.BYTE = 0x3f;          //CN:PA5-PA0
520 :      PORTB.DDR.BYTE = 0xff;          //CN:PB7-PB0
521 :      PORTD.DDR.BYTE = 0x0f;          //PD7:TRST#(i)
522 :                                       //PD5:TDI(i)
523 :                                       //PD4:TCK(i)
524 :                                       //PD3:TDO(o)
525 :                                       //CN:PD2-PD0
526 :      PORTE.DDR.BYTE = 0x1b;          //PE5:SW(i)
527 :                                       //CN:PE4-PE0

```

Following table shows relationship between ports of RX62T and kit car.

Port	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
1							Not connected	LED2 in motor drive board output
2				SDCARD_CLK	SDCARD_DI	SDCARD_DO	Not connected	Not connected
3					Not connected	Not connected	Not connected	SDCARD_CS
4	Sensor board course state input	Sensor board course state input	Sensor board course state input	Sensor board start bar state input	Sensor board course state input	Sensor board course state input	Sensor board course state input	Sensor board course state input
5			Not connected	Not connected	Not connected	Not connected	Not connected	Not connected
6			Not connected	Not connected	Not connected	Not connected	Not connected	Not connected
7		LED3 in motor drive board /output	forward reverse signal (right motor)	forward reverse signal (left motor)	PWM (right motor)	PWM (left motor)	PWM (servo motor)	Push-button in motor drive board
8						Not connected	Not connected	Not connected
9		Not connected	Not connected	Not connected	Not connected	Not connected	Not connected	Not connected
A			Not connected	Not connected	LED3 in MCU board output	LED2 in MCU board output	LED1 in MCU board output	LED0 in MCU board output
B	Not connected	Not connected	Not connected	Not connected	Not connected	Not connected	Not connected	Not connected
D	TRST#	TMS	TDI	TCK	TDO	Not connected	Not connected	Not connected
E			Push-button in MCU board input	Not connected	Not connected	Not connected	Not connected	Not connected

- Bits that are crossed out in the table have no pins associated with them.
- **All ports become input ports after a reset.**
- **Port4,5,6 and bit2 of portE are input only.**

According to following rules, every ports are set by PnDDR.(n=1 to 9,A,B,D and E)

[1] Pins on which signals are output are set to 1.

[2] Pins on which signals are input are set to 0.

[3] Unconnected pins should either be connected to a pull-up or pull-down resistor and set to input mode (0) or left open (not connected to anything) and set to output mode (1). Here, the latter setting is used.

[4] Bits with no associated pins (crossed out in the table) are set to 0.

Based on rules [1] to [4], the table can be rewritten with 1s and 0s as follows:

Port	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	Hexadecimal
1	0	0	0	0	0	0	1	1	0x03
2	0	0	0	1	1	0	1	1	0x1b
3	0	0	0	0	1	1	1	1	0x0f
7	0	1	1	1	1	1	1	0	0x7e
8	0	0	0	0	0	1	1	1	0x07
9	0	1	1	1	1	1	1	1	0x7f
A	0	0	1	1	1	1	1	1	0x3f
B	1	1	1	1	1	1	1	1	0xff
D	0	0	0	0	1	1	1	1	0x0f
E	0	0	0	1	1	0	1	1	0x1b

In the C language numeric values cannot be expressed in binary notation, so they must be converted to decimal or hexadecimal format. The conversion is generally from binary to hexadecimal format, since that is easier than converting to decimal format.

From above table, value of Direction Register set as follows.

Port	Direction Register	Setting value
1	PORT1.DDR.BYTE	0x03
2	PORT2.DDR.BYTE	0x1b
3	PORT3.DDR.BYTE	0x0f
7	PORT7.DDR.BYTE	0x7e
8	PORT8.DDR.BYTE	0x07
9	PORT9.DDR.BYTE	0x7f
A	PORTA.DDR.BYTE	0x3f
B	PORTB.DDR.BYTE	0xff
D	PORTD.DDR.BYTE	0x0f
E	PORTE.DDR.BYTE	0x1b

5.4.8. **init** Function (Compare Match Timer Settings)

CMT0 is used to generate an interrupt at 1 ms intervals.

529 :	// Compare match timer	
530 :	MSTP_CMT0 = 0;	//CMT Release module stop state
531 :	MSTP_CMT2 = 0;	//CMT Release module stop state
532 :		
533 :	ICU. IPR[0x04]. BYTE = 0x0f;	//CMT0_CMI0 Priority of interrupts
534 :	ICU. IER[0x03]. BIT. IEN4 = 1;	//CMT0_CMI0 Permission for interrupt
535 :	CMT. CMSTRO. WORD = 0x0000;	//CMT0, CMT1 Stop counting
536 :	CMT0. CMCR. WORD = 0x00C3;	//PCLK/512
537 :	CMT0. CMCNT = 0;	
538 :	CMT0. CMCOR = 96;	//1ms/(1/(49.152MHz/512))
539 :	CMT. CMSTRO. WORD = 0x0003;	//CMT0, CMT1 Start counting

Line 530 to Line 531	According to the state of the module stop function, the compare match timer will be set to permitted or prohibited. At the initial state, the compare match timer is set to prohibited. So change it to permitted.
Line 533	Sets priority of interrupts. In this program, maximum level (level 15) is set.
Line 534	Permission for interrupt request to CPU.
Line 535	Sets timer counting function to stop.
Line 536	Choose clock for counting up. In this program, PCLK/512 is chosen. $1/(49.152\text{MHz}/512) = 10.42 \text{ us}$
Line 537	Initialises counter to zero.
Line 538	Sets cycle of compare match. To generate interrupts every ms, set the value of CMT0.CMCOR as 96. $\text{CMT0.CMCOR} = \text{cycle}(1\text{ms}) / \text{counting clock}(10.42 \text{ us})$ $\text{CMT0.CMCOR} = (1 \times 10^{-3}) / (10.42 \times 10^{-6})$ $\text{CMT0.CMCOR} = 95.97$ Set CMT0.CMCOR = 96.
Line 539	Sets timer counting function to start.

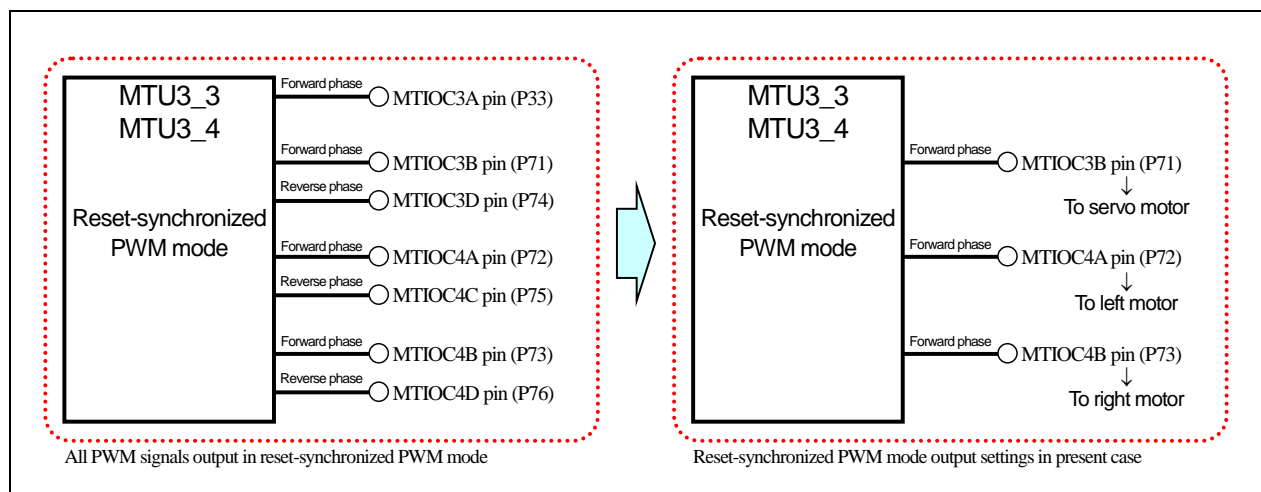
5.4.9. **init** Function (Multi-Function Timer Pulse Unit 3 Settings)

MTU3_3 and MTU3_4 are used in reset-synchronized PWM mode to output PWM signals to the left motor, right motor, and servo.

```

541 : // MTU3_3 MTU3_4 PWM mode synchronized by RESET
542 : MSTP_MTU      = 0; //Release module stop state
543 : MTU.TSTRA.BYTE = 0x00; //MTU Stop counting
544 :
545 : MTU3.TCR.BYTE = 0x23; //ILCK/64(651.04ns)
546 : MTU3.TCNT = MTU4.TCNT = 0; //MTU3,MTU4TCNT clear
547 : MTU3.TGRA = MTU3.TGRC = PWM_CYCLE; //cycle(16ms)
548 : MTU3.TGRB = MTU3.TGRD = SERVO_CENTER; //PWM(servo motor)
549 : MTU4.TGRA = MTU4.TGRC = 0; //PWM(left motor)
550 : MTU4.TGRB = MTU4.TGRD = 0; //PWM(right motor)
551 : MTU.TOCRIA.BYTE = 0x40; //Selection of output level
552 : MTU3.TMDR.BYTE = 0x38; //TGRC,TGRD buffer function
553 : //PWM mode synchronized by RESET
554 : MTU4.TMDR.BYTE = 0x00; //Set 0 to exclude MTU3 effects
555 : MTU.TOERA.BYTE = 0xc7; //MTU3TGRB,MTU4TGRA,MTU4TGRB permission
// for output
556 :
557 : MTU.TSTRA.BYTE = 0x40; //MTU0,MTU3 count function
558 : }
    
```

The output settings for reset-synchronized PWM mode in the present case are shown at right below.



5.4.10. Excep_CMT0_CMI0 Function (Interrupt Every 1 ms)

The **Excep_CMT0_CMI0** function is set by Compare Match Timer to run once every 1 ms.

```

560 : /*****/
561 : /* Interrupt */
562 : /*****/
563 : #pragma interrupt Excep_CMT0_CMI0(vect=28)
564 : void Excep_CMT0_CMI0(void)
565 : {
566 :     cnt0++;
567 :     cnt1++;
568 : }
    
```

Line 563	<p>This syntax of this line is: #pragma interrupt interrupt handler function name (vect = software interrupt number) So whenever the interrupt designated by software interrupt number occurs, interrupt handler function name is executed. From the software interrupt table we can see that the Compare Match Timer interrupt is designated as number 28. The source code uses #pragma interrupt to specify that the Excep_CMT0_CMI0 function is run when interrupt number 28 occurs.</p>
Line 564	<p>This is the function triggered by the Compare Match Timer interrupt. It is not possible to specify arguments or return values for an interrupt function. Therefore, the syntax void function name (void) must be used.</p>
Line 566	<p>Increments (+1) variable cnt0. The function is executed at 1 ms intervals, so the value of variable cnt0 increases by 1 every 1 ms.</p>
Line 567	<p>Increments (+1) variable cnt1. The function is executed at 1 ms intervals, so the value of variable cnt1 increases by 1 every 1 ms.</p>

5.4.11. timer Function (Pause)

The **timer** function is used to pause operation.

```

570 : /*****/
571 : /* Timer unit */
572 : /* Arguments: timer value, 1 = 1 ms */
573 : /*****/
574 : void timer( unsigned long timer_set )
575 : {
576 :     cnt0 = 0;
577 :     while( cnt0 < timer_set );
578 : }
    
```

(1) Using the timer Function

The usage of the **timer** function is illustrated below.

```

timer( desired pause duration );
    
```

The argument specifies the pause duration in milliseconds. An example is shown below.

```

motor( 50, 100 ); <- Several 100 μs
timer( 1000 ); <- 1000 ms
    
```

Following motor control, the **timer** function pauses operation for 1,000 ms.

(2) Program Explanation

The following explanation assumes that the function is executed as follows:

```

timer( 1000 );
    
```

Line 576	Clears cnt0 to 0.	
Line 577	1.	<p>First, the following statement is processed:</p> <pre>while(cnt0 < timer_set);</pre> <p>Variable cnt0 was cleared to 0 on line 576. Variable timer_set is an argument of the timer function. The line is therefore equivalent to:</p> <pre>while(0 < 1000);</pre> <p>The condition within the parentheses is not satisfied, so processing of the statement repeats over and over.</p>
	2.	<p>After 1 ms elapses an Excep_CMT0_CMI0 interrupt occurs. Processing of the while statement in the current line stops, and the following line in the interrupt handler is processed:</p> <pre>560 : cnt0++; <- This is equivalent to 0++, therefore the value becomes 1.</pre> <p>The value of cnt0 is now 1. Processing of the interrupt handler ends, and control returns to the line at which the interrupt occurs.</p>

Line 577	3.	Control returns to the line with the while statement: <div style="border: 1px solid black; padding: 2px; width: fit-content; margin: 5px auto;"> <code>while (1 < 1000);</code> </div> <p>The value of cnt0 is now 1 because it was incremented by the interrupt handler. Once again, the condition within the parentheses is not satisfied, and processing of the statement repeats over and over.</p>
	4.	When 2. has been processed 1,000 times, the value of cnt0 is 1,000. <div style="border: 1px solid black; padding: 2px; width: fit-content; margin: 5px auto;"> <code>while (1000 < 1000);</code> </div> <p>The condition of the while statement is satisfied. Processing of the while statement ends, and processing continues to the next line. There is no next line in the function, so the timer function ends. The Excep_CMT0_CMI0 function is executed every 1 ms when a Compare Match Timer interrupt occurs. Processing the while statement repeatedly until the value of cnt0 reaches 1,000 means that the while statement repeats over a duration of 1,000 ms. In this way, a specified duration of time can be measured by using the timer function to count the occurrences of an interrupt, which is triggered at 1 ms intervals by Compare Match Timer.</p>

5.4.12. **sensor_inp** Function (Read State of Sensors)

The **sensor_inp** function reads the state of the sensors on the sensor board.

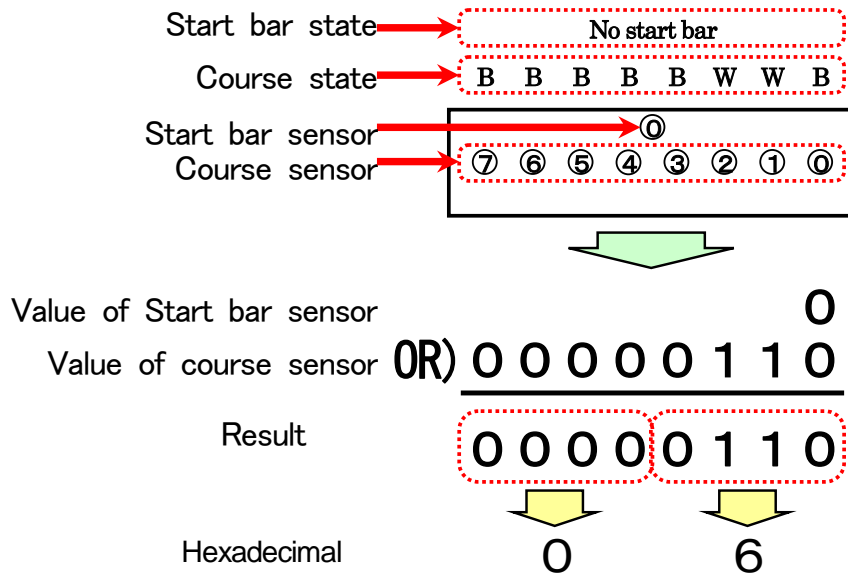
```

580 : /*****/
581 : /* Sensor state detection */
582 : /* Arguments:      masked values */
583 : /* Return values:  sensor value */
584 : /*****/
585 : unsigned char sensor_inp( unsigned char mask )
586 : {
587 :     unsigned char sensor;
588 :
589 :     sensor = ~PORT4.PORT.BYTE;
590 :
591 :     sensor &= mask;
592 :
593 :     return sensor;
594 : }
    
```

Line 587	Specifies variable sensor as type unsigned char . This variable is used in the sensor_inp function to process the states of the sensors. Sensor board data is read from port 4 with a bit width of 8 bits. Therefore, variable sensor is specified as an unsigned 8-bit char .
Line 589	Reads the sensor board data from port 4. The output from the sensors is 0 for white and 1 for black, which is confusing because it is the reverse of the human sense of sight. Therefore, a tilde (~) is used to reverse the values so that 1 represents white and 0 represents black.
Line 591	Performs an AND operation on variable sensor , which contains the sensor data processed up to line 585, and the mask value used as an argument of the sensor_inp function. This forces the value of all unneeded bits to 0.
Line 593	Returns variable sensor as a return value, ending the function.

(1) Treatment of bit0

The rightmost sensor which detects the course and the sensor which detects the start bar of the Sensor Board Ver.5 are input in an OR connection to bit 0 terminal of port 0. This is shown below.



It is unknown whether the start bar detection sensor reacted when bit 0 became “1” or whether bit 0 of the course detection sensor reacted. But the two states shown in the table below cannot happen simultaneously.

State	Picture	Description
Before start		<p>It judges</p> <p>bit0="1": There is start bar</p> <p>"0": No start bar</p> <p>because the rightmost sensor is surely black if the Sensor Board is set over the centre line.</p>
After start		<p>It judges</p> <p>bit0="1": Rightmost course :white</p> <p>"0": Rightmost course : black</p> <p>because there is no need to check the reading of the start bar sensor after the start.</p>

(3) Masking

Line 587 performs masking on the sensor data. Masking forces to 0 the value of the bits that do not require checking. It is accomplished by using an AND operation.

(a) Why Is Masking Necessary?

Since each port comprises 8 bits, **it is not possible to check bits singly**. (Actually, it is possible if we use a method called a bit field, but we will not deal with that here.) **Instead, all 8 bits are checked at the same time.** For example, if we want to determine whether or not the value of the bit corresponding to the leftmost sensor is 1, you might think we could use the following code:

```
if( Sensor value == 0x80 ) {
    /* Run this code if value of bit 7 is 1 */
}
```

But we don't know the values of bits 6 to 0. If, for example, the values of both bit 7 and bit 0 are 1, the result would be as follows:

```
if( Sensor value 0x81 == 0x80 ) {           <- Sensor value =1000 0001=0x81
    /* Run this code if value of bit 7 is 1 */
}
```

The value of bit 7 is 1, but since the value of bit 0 is also 1 the sensor value is determined not to be 0x80 and the code between the brackets is not executed. Proper checking is not possible in this case. This is why we need masking.

(b) What Masking Does

The above method does not enable us to perform proper checking because we do not know if the values of the bits other than 7 are 0 or 1. Masking lets us force the values of bits 6 to 0 to 0.

Since we know the values of bits 6 to 0 must be 0, we can check the sensor value based on this assumption. For example, we can check whether or not the value of bit 7 is 1 as follows:

```
if(sensor value with bit 7 unchanged and bits 6 to 0 cleared to 0 =0x80 ) {
    /* Run this code if value of bit 7 is 1 */
}
```

Now it is possible to determine whether the value of bit 7 is 0 or 1, and if it is 1 the code between the brackets is executed.

(c) Determining Masking Values and Performing Masking

In software we can use a logic operation known as the logical product, which is also called an AND operation. For example, if the state of the sensors is BBBWWWW, the sensor value will be 00011111. When checking bit 7 there is no need to check bits 6 to 0.

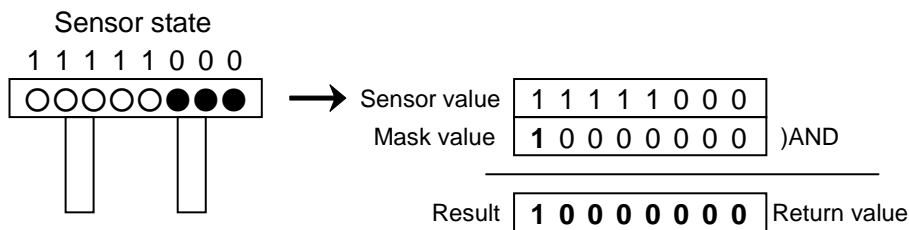
Bit	7	6	5	4	3	2	1	0
	Necessary	Unnecessary	Unnecessary	Unnecessary	Unnecessary	Unnecessary	Unnecessary	Unnecessary

To clear the unnecessary bits to 0, we set the mask value for those bits to 0 and perform an AND operation.

Consequently, in the mask value the unnecessary bits should be cleared to 0 and the necessary bits set to 1.

Bit	7	6	5	4	3	2	1	0
Mask value	1	0	0	0	0	0	0	0

We perform an AND operation on the sensor value and mask value, and then check the result against the bits to be checked (see figure below).



This can be converted into the following lines of code:

```
if( (Sensor value & 0x80) == 0x80 ) {
    /* Run this code if value of bit 7 is 1 */
}
```

We know that bits 6 to 0 have all been forcibly cleared to 0 by masking, so we can assume the value of bits 6 to 0 is 0 when making the comparison.

(d) Structure of sensor_inp Function

The **sensor_inp** function obtains sensor data and performs masking. Masking, the last step in processing the sensor value, is performed after the sensor value is processed by inversion, bit copying, etc.

```

580 : /*****
581 : /* Sensor state detection */
582 : /* Arguments:    masked values */
583 : /* Return values: sensor value */
584 : *****/
585 : unsigned char sensor_inp( unsigned char mask )
586 : {
587 :     unsigned char sensor;
588 :
589 :     sensor = ~PORT4.PORT.BYTE;
590 :
591 :     sensor &= mask;
592 :
593 :     return sensor;
594 : }
    
```

Masking is the last step in processing the sensor value.

(e) Mask Value Definition

In kit12_rx62t.c the most commonly used mask values are predefined using **define**. This is performed in lines 31 to 39 of the source code. The format used for these definitions is as follows:

```
#define MASK[A]_B      Mask value
```

Mask values are defined using the rule MASK + [A] + _ (underscore) + [B]. The meanings of these elements are as follows:

- [A]: Of the four sensors on the left, [A] sensors are valid (unmasked).
- [B]: Of the four sensors on the right, [B] sensors are valid (unmasked).
- The other sensors are masked.

The definitions used in the program are listed in the following table:

Defined Mask Character String	Mask Value	Binary	Description
MASK2_2	0x66	0110 0110	The two middle sensors on the left and the two middle sensors on the right are valid (unmasked), and the others are masked.
MASK2_0	0x60	0110 0000	The two middle sensors on the left are valid (unmasked), and the others are masked.
MASK0_2	0x06	0000 0110	The two middle sensors on the right are valid (unmasked), and the others are masked.
MASK3_3	0xe7	1110 0111	The three sensors on the left and the three sensors on the right are valid (unmasked), and the others are masked.
MASK0_3	0x07	0000 0111	The three sensors on the right are valid (unmasked), and the others are masked.
MASK3_0	0xe0	1110 0000	The three sensors on the left are valid (unmasked), and the others are masked.
MASK4_0	0xf0	1111 0000	The four sensors on the left are valid (unmasked), and the others are masked.

MASK0_4	0x0f	0000 1111	The four sensors on the right are valid (unmasked), and the others are masked.
MASK4_4	0xff	1111 1111	Four sensors on the right and four sensors on the left are valid (unmasked), and the others are masked. Note: The result of MASK4_4 is that all of the sensors are valid (unmasked). This pattern is provided because the parentheses of the sensor_inp function must contain a value. When masking is not necessary, using MASK4_4 results in a mask value of 0xff: all sensors valid (unmasked).

The mask value specified between the parentheses of the **sensor_inp** function can be set by using one of the mask character strings described above.

If none of the provided mask character strings correspond to your desired mask values, add additional definitions of your own. Alternately, you can specify the mask value directly as a numeric value, without using a mask character string.

(4) Using the sensor_inp Function

```

if( sensor_inp( [Mask value] ) == [Sensor check value] ) {
    Run this code if the expression is true.
} else {
    Run this code if the expression is not true.
}
    
```

In the **sensor_inp** function, **[mask value]** contains the value used when applying masking to the sensor value, and **[sensor check value]** contains the value after masking is applied that is used for checking. For example, the process is as follows when the sensor value is 0x1f, the mask value is MASK0_2, and the sensor check value is 0x04:







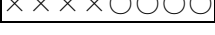
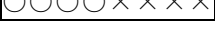
```

if( sensor_inp( [MASK0_2] ) == [0x04] ) {
    Run this code if the expression is true.
} else {
    Run this code if the expression is not true.
}
    
```

1.	<p>The sensor value is 0001 1111 and the mask value of the sensor_inp function is 0000 0110. The result of the AND operation is as follows:</p> <table style="margin-left: 40px;"> <tr> <td>Sensor value</td> <td>0001 1111</td> <td></td> </tr> <tr> <td>Mask value</td> <td>0000 0110 (AND)</td> <td></td> </tr> <tr> <td>Result</td> <td>0000 0110</td> <td>→0x06 after conversion to hexadecimal</td> </tr> </table>	Sensor value	0001 1111		Mask value	0000 0110 (AND)		Result	0000 0110	→0x06 after conversion to hexadecimal
Sensor value	0001 1111									
Mask value	0000 0110 (AND)									
Result	0000 0110	→0x06 after conversion to hexadecimal								
2.	<p>The result 0x06 and sensor check value 0x04 are compared. Since they do not match, the lines represented by “Run this code if the expression is not true” are processed.</p>									

(5) Notes

We have explained above how the **sensor_inp** function uses the same value for bits 4 and 3. However, depending on the mask value, the values of bits 4 and 3 may differ in the return value of the **sensor_inp** function. When writing program code, also pay attention to the mask value of the **sensor_inp** function.

<code>if (sensor_inp(MASK4_4) == 0x1f) {</code>		Possible
<code>}</code>		
<code>if (sensor_inp(MASK4_4) == 0x07) {</code>		Possible
<code>}</code>		
<code>if (sensor_inp(MASK4_4) == 0x0f) {</code>		0x0f not possible
<code>}</code>		
<code>if (sensor_inp(MASK4_4) == 0xf8) {</code>		Possible
<code>}</code>		
<code>if (sensor_inp(MASK4_4) == 0xe0) {</code>		Possible
<code>}</code>		
<code>if (sensor_inp(MASK4_4) == 0xf0) {</code>		0xf0 not possible
<code>}</code>		
<code>if (sensor_inp(MASK0_4) == 0x0f) {</code>		0x0f possible depending on mask value!
<code>}</code>		
<code>if (sensor_inp(MASK4_0) == 0xf0) {</code>		0xf0 possible depending on mask value!
<code>}</code>		

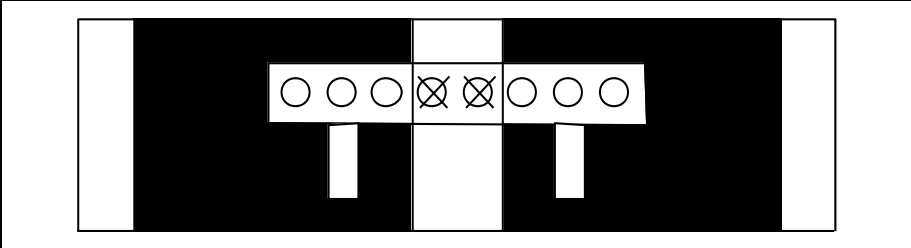
5.4.13. **check_crossline** Function (Crossline Detection)

There are two white lines on the track 500 mm to 1,000 mm before a crank. These are called cross lines. The **check_cross line** function detects these cross lines.

The return value is 1 when a cross line is detected and 0 when no cross line is detected.

```

610 : /*****/
611 : /* Cross line detection processing */
612 : /* Return values: 0: no cross line, 1: cross line */
613 : /*****/
614 : int check_crossline( void )
615 : {
616 :     unsigned char b;
617 :     int ret;
618 :
619 :     ret = 0;
620 :     b = sensor_inp(MASK3_3);
621 :     if( b==0xe7 ) {
622 :         ret = 1;
623 :     }
624 :     return ret;
625 : }
    
```

Line 619	This initialises the variable ret , which stores the return value. A value of 1 is stored in variable ret when a cross line is detected and 0 when no cross line is detected. For the time being we do not know which is correct, so we insert a value of 0, no cross line detected.
Line 620	<p>This reads the sensors and stores the result in variable b. The sensor mask value is MASK3_3 (0xe7), so a total of six sensors are read, three on the right and three on the left.</p> <p>The sensors that are read are illustrated below:</p> 

This checks whether or not the sensor state is 0xe7. A cross line is detected when the sensor value is 0xe7. This is illustrated below:

0xe7

1 1 1 0 0 1 1 1

Line 621

The **if** condition is met when the sensor value is 0xe7, and variable **ret** is set to 1. If the value is something else, the condition is not met and the value of variable **ret** remains unchanged at 0. Variable **ret** is the return value, so a value of 1 means “cross line detected” and a value of 0 means “no cross line detected.”

5.4.14. **check_rightline** function (Right Half Line Detection)

There are two right half lines on the track 500 mm to 1,000 mm before a right lane change. The **check_rightline** function detects the right half lines.

The return value is 1 when a right half line is detected and 0 when no right half line is detected.

```

627 : /*****/
628 : /* Right half line detection processing */
629 : /* Return values: 0: not detected, 1: detected */
630 : /*****/
631 : int check_rightline( void )
632 : {
633 :     unsigned char b;
634 :     int ret;
635 :
636 :     ret = 0;
637 :     b = sensor_inp(MASK4_4);
638 :     if( b==0x1f ) {
639 :         ret = 1;
640 :     }
641 :     return ret;
642 : }
    
```

Line 636	<p>This initialises the variable ret, which stores the return value. A value of 1 is stored in this variable when a right half line is detected and 0 when no right half line is detected. For the time being we do not know which is correct, so we insert a value of 0, no right half line detected.</p>
Lines 637 and 638	<p>This checks the sensor state. The mask value is MASK4_4, so all the sensors are read. A right half line is detected when the sensor value is 0x1f. This is illustrated below:</p> <div data-bbox="414 1108 1380 1624" style="text-align: center;"> </div> <p>The if condition is met when the sensor value is 0x1f, and variable ret is set to 1. If the value is something else, the condition is not met and the value of variable ret remains unchanged at 0. Variable ret is the return value, so a value of 1 means “right half line detected” and a value of 0 means “no right half line detected.”</p>

5.4.15. **check_leftline** function (Left Half Line Detection)

There are two left half lines on the track 500 mm to 1,000 mm before a left lane change. The **check_leftline** function detects the left half lines.

The return value is 1 when a left half line is detected and 0 when no left half line is detected.

```

644 : /*****/
645 : /* Left half line detection processing */
646 : /* Return values: 0: not detected, 1: detected */
647 : /*****/
648 : int check_leftline( void )
649 : {
650 :     unsigned char b;
651 :     int ret;
652 :
653 :     ret = 0;
654 :     b = sensor_inp(MASK4_4);
655 :     if( b==0xf8 ) {
656 :         ret = 1;
657 :     }
658 :     return ret;
659 : }
    
```

Line 653	<p>This initialises the variable ret, which stores the return value. A value of 1 is stored in this variable when a left half line is detected and 0 when no left half line is detected. For the time being we do not know which is correct, so we insert a value of 0, no left half line detected.</p>
Lines 654 and 655	<p>This checks the sensor state. The mask value is MASK4_4, so all the sensors are read. A left half line is detected when the sensor value is 0xf8. This is illustrated below:</p> <div data-bbox="421 1131 1339 1641" data-label="Diagram"> <p>The diagram shows a track with two half lines. The sensor output is 1 1 1 1 1 0 0 0. The sensor value is 0xf8.</p> </div> <p>The if condition is met when the sensor value is 0xf8, and variable ret is set to 1. If the value is something else, the condition is not met and the value of variable ret remains unchanged at 0. Variable ret is the return value, so a value of 1 means “left half line detected” and a value of 0 means “no left half line detected.”</p>

5.4.16. **dipsw_get** Function (Reading DIP Switches)

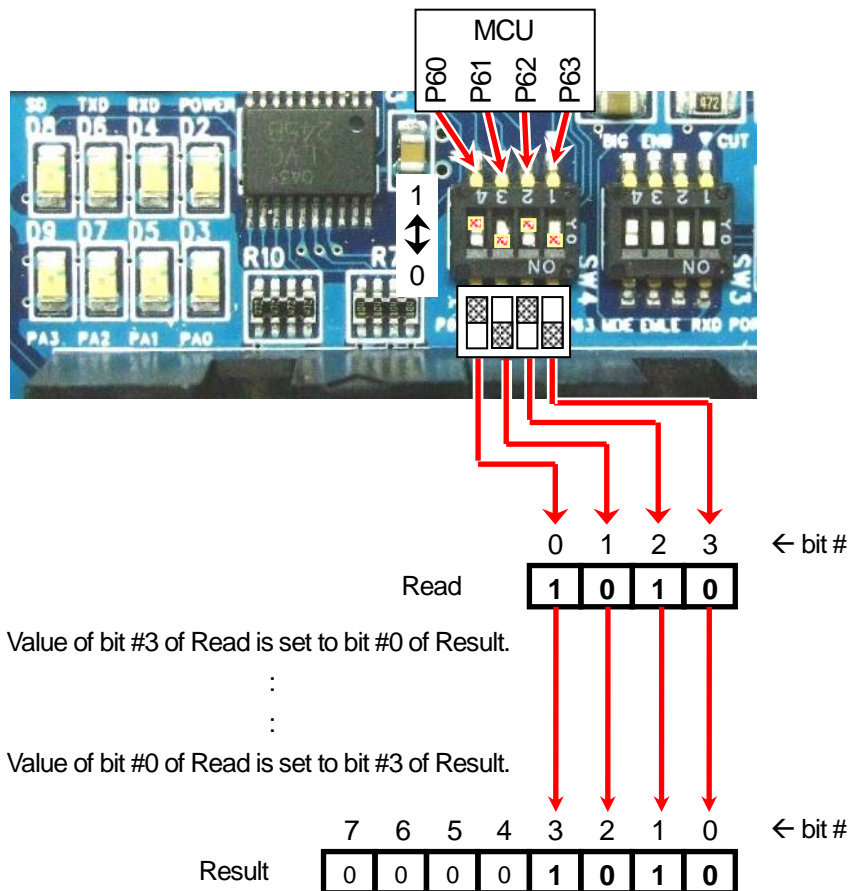
The **dipsw_get** function reads the state of the DIP switches on the RMC-RX62T board. The return value is a value between 0 and 15, according to the DIP switch value.

```

661 : /*****
662 : /* DIP switch value read
663 : /* Return values: Switch value, 0 to 15
664 : *****/
665 : unsigned char dipsw_get( void )
666 : {
667 :     unsigned char sw, d0, d1, d2, d3;
668 :
669 :     d0 = ( PORT6.PORT.BIT.B3 & 0x01 ); /* P63~P60 read
670 :     d1 = ( PORT6.PORT.BIT.B2 & 0x01 ) << 1;
671 :     d2 = ( PORT6.PORT.BIT.B1 & 0x01 ) << 2;
672 :     d3 = ( PORT6.PORT.BIT.B0 & 0x01 ) << 3;
673 :     sw = d0 | d1 | d2 | d3;
674 :
675 :     return sw;
676 : }
    
```

A 4-bit DIP switch used on the RMC-RX62T MCU board is connected to Port 6. Switch #1 of this DIP switch is connected to bit 3 of Port 6. Similarly, switch #4 is connected to bit 0 of Port 6.

The operation of the **dipsw_get** function when the DIP switch setting is **1010** (ON, OFF, ON, OFF) is illustrated below.



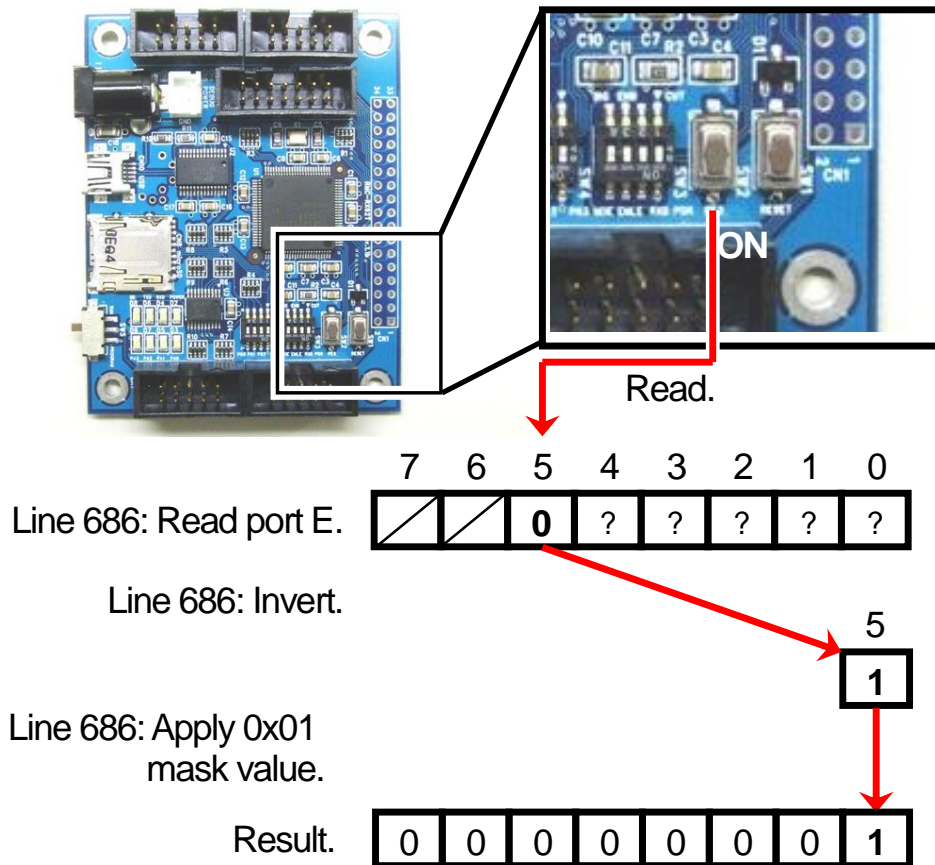
5.4.17. **buttonsw_get** Function (Reading the Pushbutton State in MCU board)

The **buttonsw_get** function reads the state of the pushbutton on the MCU board.
 The return value is 1 when the button is depressed and 0 when it is released.

```

678 : /******
679 : /* Push-button in MCU board value read
680 : /* Return values: Switch value, ON: 1, OFF: 0
681 : /******
682 : unsigned char buttonsw_get( void )
683 : {
684 :     unsigned char sw;
685 :
686 :     sw = ~PORTE.PORTE.BIT.B5 & 0x01;    /* Read ports with switches */
687 :
688 :     return sw;
689 : }
    
```

The pushbutton is connected to bit 5 of port E. The operation of the **button_get** function when the pushbutton is depressed is illustrated below.



5.4.18. **pushsw_get** Function (Reading the Pushbutton State)

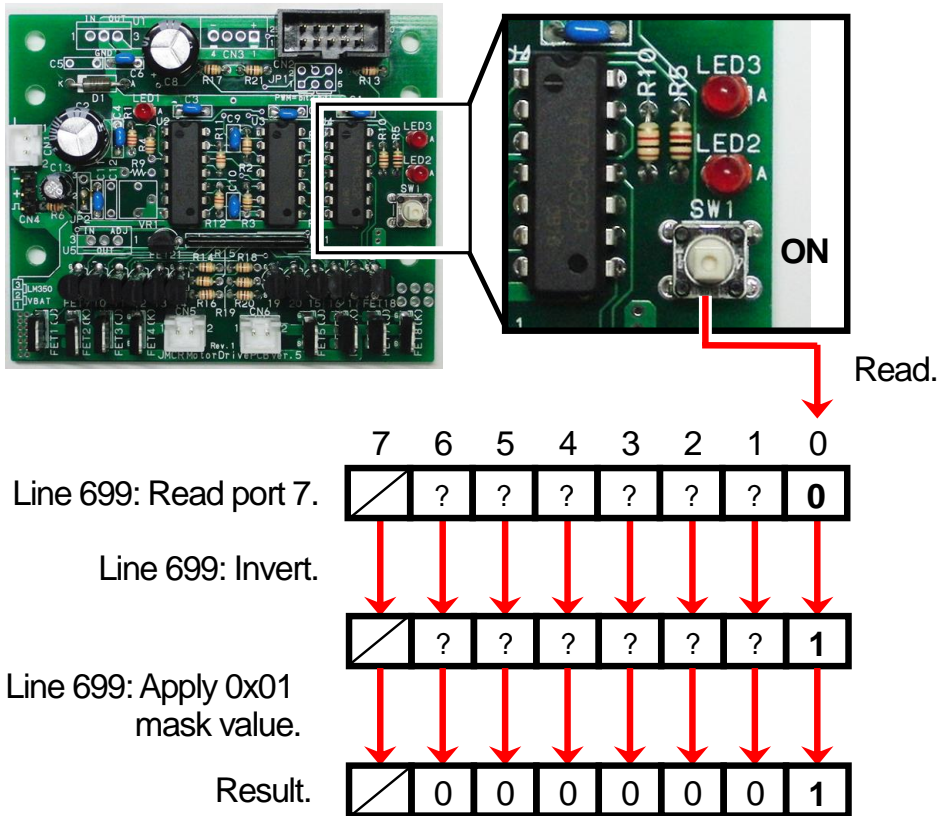
The **pushsw_get** function reads the state of the pushbutton on the motor drive board. The return value is 1 when the button is depressed and 0 when it is released.

```

691 : /*****
692 : /* Push-button in motor drive board value read
693 : /* Return values: Switch value, ON: 1, OFF: 0
694 : *****/
695 : unsigned char pushsw_get( void )
696 : {
697 :     unsigned char sw;
698 :
699 :     sw = ~PORT7.PORT.BIT.B0 & 0x01;    /* Read ports with switches */
700 :
701 :     return sw;
702 : }
    
```

The pushbutton is connected to bit 0 of port 7. Bits 6 to 1 of port 7 do not interest us, so bit operations are used to fetch the value of only the bit associated with the pushbutton.

The operation of the **pushsw_get** function when the pushbutton is depressed is illustrated below.



5.4.19. **startbar_get** Function (Reading the Start Bar Detection Sensor)

The **startbar_get** function determines whether the start bar is present (closed) or not (open). The return value is 1 when the start bar is present and 0 when it is not.

```

596 : /******  

597 : /* Read start bar detection sensor */  

598 : /* Return values: Sensor value, ON (bar present):1, */  

599 : /* OFF (no bar present):0 */  

600 : /******  

601 : unsigned char startbar_get( void )  

602 : {  

603 :     unsigned char b;  

604 :  

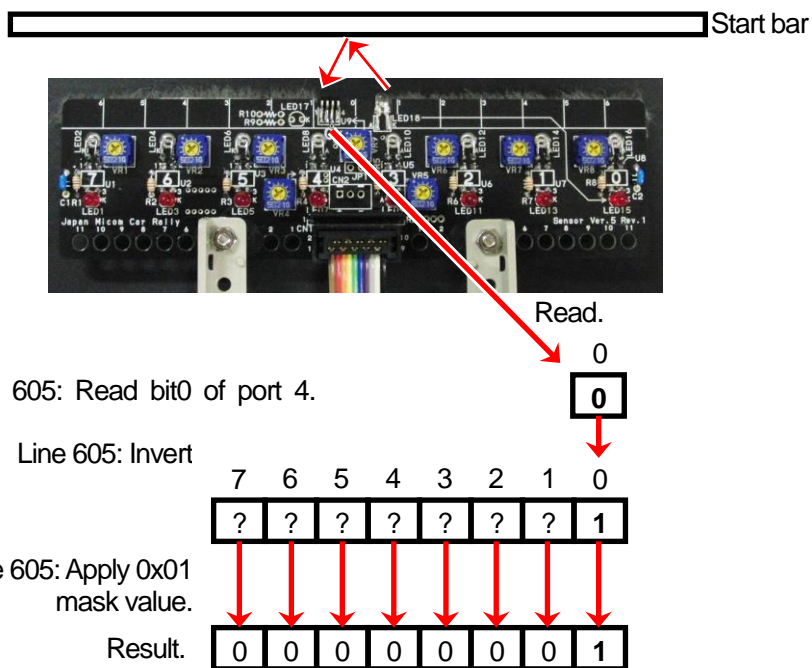
605 :     b = ~PORT4.PORT.BIT.B0 & 0x01; /* Read start bar signal */  

606 :  

607 :     return b;  

608 : }
    
```

The start bar detection sensor is connected to bit 4 of port 4. The bits other than bit 4 of port 4 do not interest us, so bit operations are used to fetch the value of only the bit associated with the sensor. The operation of the **startbar_get** function when the start bar is present is illustrated below.



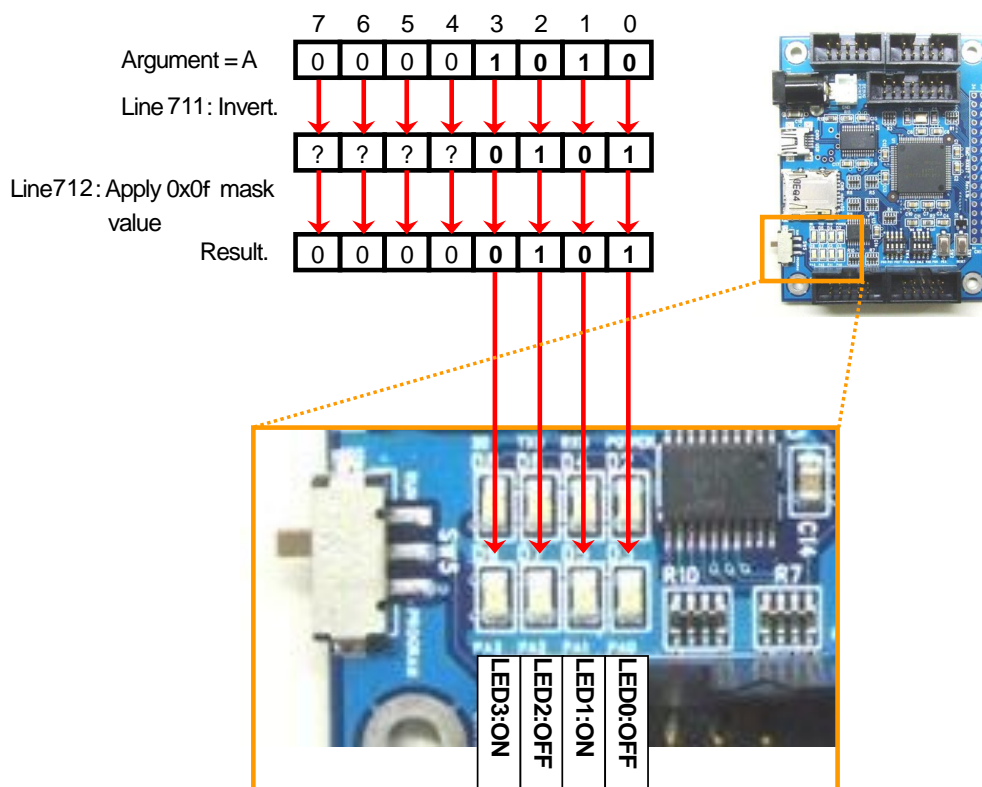
5.4.20. led_out_m Function (LED Control in MCU board)

The **led_out_m** function turns on and off LED0, LED1, LED2, and LED3 on the MCU board.

```

704 : /*****
705 : /* LED control in MCU board */
706 : /* Arguments: Switch value, LED0: bit 0, LED1: bit 1. 0: dark, 1: lit */
707 : /*
708 : /*****
709 : void led_out_m( unsigned char led )
710 : {
711 :     led = ~led;
712 :     PORTA.DR.BYTE = led & 0x0f;
713 : }
    
```

The operation of the led_out function when the value of the argument is 2 is illustrated below.



5.4.21. **led_out** Function (LED Control)

The **led_out** function turns on and off LED3 and LED2 on the motor drive board.

The correspondence between the argument supplied to the function and the illumination status of LED3 and LED2 is as follows:

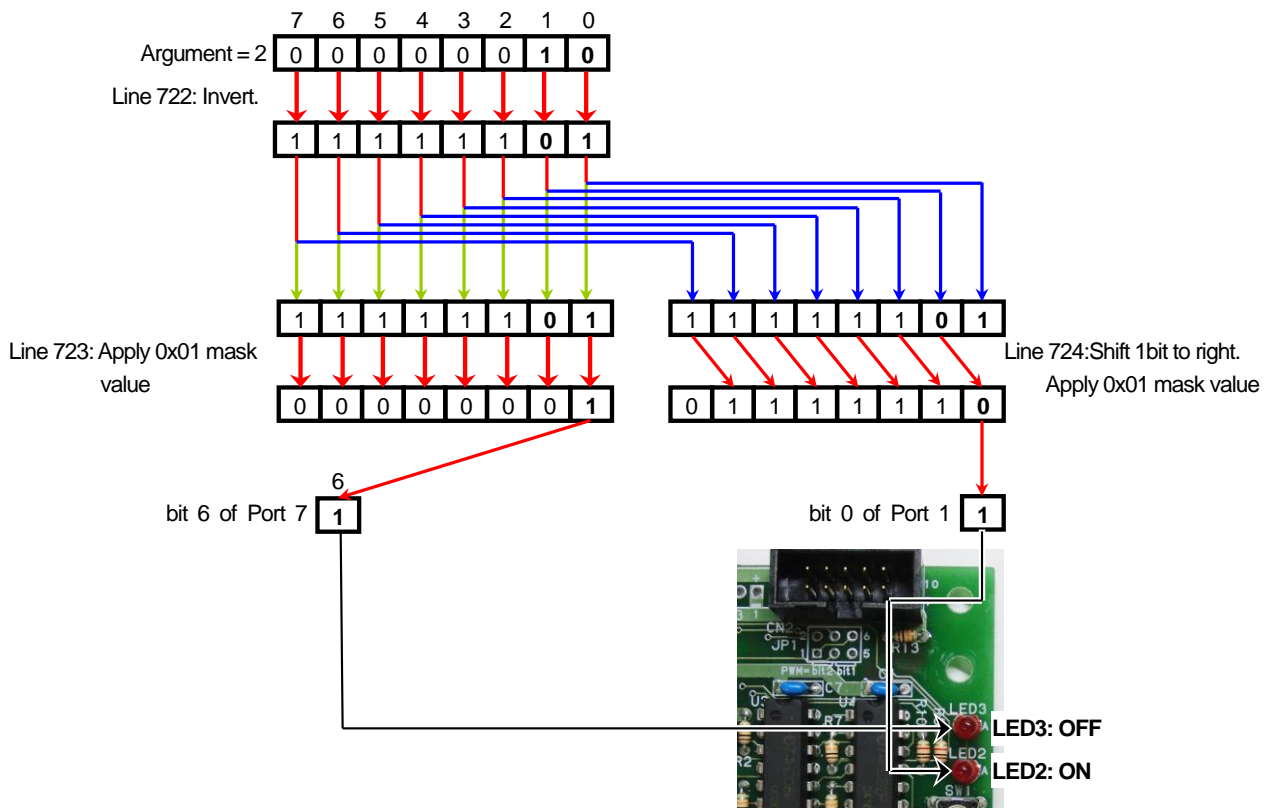
Argument	Binary	LED3	LED2
0	0 0	OFF	OFF
1	0 1	OFF	ON
2	1 0	ON	OFF
3	1 1	ON	ON

```

715 : /*****/
716 : /* LED control in motor drive board */
717 : /* Arguments: Switch value, LED0: bit 0, LED1: bit 1. 0: dark, 1: lit */
718 : /* Example: 0x3 -> LED1: ON, LED0: ON, 0x2 -> LED1: ON, LED0: OFF */
719 : /*****/
720 : void led_out( unsigned char led )
721 : {
722 :     led = ~led;
723 :     PORT7.DR.BIT.B6 = led & 0x01;
724 :     PORT1.DR.BIT.B0 = ( led >> 1 ) & 0x01;
725 : }
    
```

Be careful that LED2 should be connected to bit0 of port1 and LED3 should be connected to bit6 of port7.

The operation of the led_out function when the value of the argument is 2 is illustrated below.



5.4.22. motor Function (Motor Speed Control)

The **motor** function generates PWM output to the left and right motors. Forward and reverse operation is controlled by the sign of the argument.

```

727 : /*****/
728 : /* Motor speed control */
729 : /* Arguments: Left motor: -100 to 100, Right motor: -100 to 100 */
730 : /* Here, 0 is stopped, 100 is forward, and -100 is reverse. */
731 : /* Return value: None */
732 : /*****/
733 : void motor( int accele_l, int accele_r )
734 : {
735 :     int    sw_data;
736 :
737 :     sw_data = dipsw_get() + 5;
738 :     accele_l = accele_l * sw_data / 20;
739 :     accele_r = accele_r * sw_data / 20;
740 :
741 :     /* Left Motor Control */
742 :     if( accele_l >= 0 ) {
743 :         PORT7.DR.BYTE &= 0xef;
744 :         MTU4.TGRC = (long)( PWM_CYCLE - 1 ) * accele_l / 100;
745 :     } else {
746 :         PORT7.DR.BYTE |= 0x10;
747 :         MTU4.TGRC = (long)( PWM_CYCLE - 1 ) * ( -accele_l ) / 100;
748 :     }
749 :
750 :     /* Right Motor Control */
751 :     if( accele_r >= 0 ) {
752 :         PORT7.DR.BYTE &= 0xdf;
753 :         MTU4.TGRD = (long)( PWM_CYCLE - 1 ) * accele_r / 100;
754 :     } else {
755 :         PORT7.DR.BYTE |= 0x20;
756 :         MTU4.TGRD = (long)( PWM_CYCLE - 1 ) * ( -accele_r ) / 100;
757 :     }
758 : }

```

(1) Using the motor Function

The use of the **motor** function is described below.

```

motor (left motor PWM value, right motor PWM value );

```

The arguments are assigned the left motor PWM value and right motor PWM value, separated by a comma. The correspondence between the PWM values and motor operation are as follows:

Value	Description
-100 to -1	The motor operates in the reverse direction. A value of -100 corresponds to “reverse 100%.” The value cannot exceed -100. The setting must be an integer value.
0	The motor is stopped.
1 to 100	The motor operates in the forward direction. A value of 100 corresponds to “forward 100%.” The value cannot exceed 100. The setting must be an integer value.

The actual motor output ratios are as follows:

$$\begin{aligned} \text{PWM output to left motor} &= \text{left motor PWM value set by } \mathbf{motor} \text{ function} \times \text{DIP switch value} + 5 / 20 \\ \text{PWM output to right motor} &= \text{right motor PWM value set by } \mathbf{motor} \text{ function} \times \text{DIP switch value} + 5 / 20 \end{aligned}$$

For example, it is not actually the case that the left motor operates in the forward direction at 80% when the left motor PWM value set by the **motor** function is 80. The actual PWM ratio output to the motor differs depending on the setting of the DIP switches on the MCU board.

Let's assume that the following line of code is processed when the DIP switch setting is **1100** (12 in decimal notation):

```
motor( -70 , 100 );
```

The actual PWM values output to the motors will be as follows:

$$\begin{aligned} \text{PWM output to left motor} &= -70 \times (12 + 5) \div 20 = -70 \times 0.85 = -59.5 = -59\% \\ \text{PWM output to right motor} &= 100 \times (12 + 5) \div 20 = 100 \times 0.85 = 85\% \end{aligned}$$

The calculation result for the left motor is -59.5% , but the portion after the decimal point is discarded to produce an integer value. Thus, the PWM value output to the left motor is reverse 59% and the PWM value output to the right motor is forward 85%.

The manner in which the above is processed in practice is described below.

(2) Change to PWM Value According to DIP Switch Setting

```
737 :   sw_data = dipsw_get() + 5;           dipsw_get() = DIP switch value of 0 to 15
738 :   accele_l = accele_l * sw_data / 20;
739 :   accele_r = accele_r * sw_data / 20;
```

Line 737	This assigns a value of (DIP switch value + 5) to variable sw_data . The range of DIP switch values is 0 to 15, so variable sw_data can have a value of 5 to 20.
Line 738	<p>The PWM value ratio applied to the left motor is assigned to the variable accele_l on the left of the equal sign. The value of the variable accele_l on the right of the equal sign is the left motor PWM value set by the motor function. Thus, the PWM value applied to the left motor can be calculated as follows:</p> $\mathbf{accele_l} = \mathbf{accele_l} \text{ (left motor PWM value set by } \mathbf{motor} \text{ function)} \times \mathbf{sw_data} / 20$ <p>Variable accele_l can have a value within a range of -100 to 100.</p>
Line 739	<p>The PWM value ratio applied to the right motor is assigned to the variable accele_r on the left of the equal sign. The value of the variable accele_r on the right of the equal sign is the right motor PWM value set by the motor function. Thus, the PWM value applied to the right motor can be calculated as follows:</p> $\mathbf{accele_r} = \mathbf{accele_r} \text{ (right motor PWM value set by } \mathbf{motor} \text{ function)} \times \mathbf{sw_data} / 20$ <p>Variable accele_r can have a value within a range of -100 to 100.</p>

(3) Left Motor Control

This portion of the function controls the left motor. The left motor PWM output is on pin P72. The PWM output on pin P72 is specified by the PWM value setting in MTU3_4 timer general register A (MTU4.TGRA). However, this setting is not made directly but to MTU3_4 timer general register C (MTU4.TGRC), which functions as a buffer register, instead.

```

741 :    /* Left Motor Control */
742 :    if( accele_1 >= 0 ) {
743 :        PORT7.DR.BYTE &= 0xef;
744 :        MTU4.TGRC = (long)( PWM_CYCLE - 1 ) * accele_1 / 100;
745 :    } else {
746 :        PORT7.DR.BYTE |= 0x10;
747 :        MTU4.TGRC = (long)( PWM_CYCLE - 1 ) * ( -accele_1 ) / 100;
748 :    }
    
```

Line 742	<p>Checks whether the left motor PWM value ratio is a positive or a negative value. If positive, lines 743 and 744 are processed, if negative, lines 746 and 747.</p>																																				
Lines 743 to 744	<p>If the value is positive, lines 743 and 744 are processed. P74 is cleared to 0 and PWM is output on P72, causing the left motor to operate in the forward direction according to the PWM ratio. Line 743 performs the following bit operations and clears pin P74 to 0:</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>bit</th> <th>7</th> <th>6</th> <th>5</th> <th>4</th> <th>3</th> <th>2</th> <th>1</th> <th>0</th> </tr> </thead> <tbody> <tr> <td>Original value (port 2)</td> <td style="text-align: center;">/</td> <td style="text-align: center;">P76</td> <td style="text-align: center;">P75</td> <td style="text-align: center;">P74</td> <td style="text-align: center;">P73</td> <td style="text-align: center;">P72</td> <td style="text-align: center;">P71</td> <td style="text-align: center;">P70</td> </tr> <tr> <td>AND value</td> <td style="text-align: center;">/</td> <td style="text-align: center;">1</td> <td style="text-align: center;">1</td> <td style="text-align: center;">0</td> <td style="text-align: center;">1</td> <td style="text-align: center;">1</td> <td style="text-align: center;">1</td> <td style="text-align: center;">1</td> </tr> <tr> <td>Result</td> <td style="text-align: center;">/</td> <td style="text-align: center;">P76</td> <td style="text-align: center;">P75</td> <td style="text-align: center;">0</td> <td style="text-align: center;">P73</td> <td style="text-align: center;">P72</td> <td style="text-align: center;">P71</td> <td style="text-align: center;">P70</td> </tr> </tbody> </table> <p>Line 744 performs the calculation below and sets the PWM value in MTU3_4 timer general register C (MTU4.TGRC). If there are digits after the decimal point, they are discarded.</p> $ \begin{aligned} \text{MTU4.TGRC} &= (\text{PWM_CYCLE} - 1) \times \frac{\text{accele_1 (0 to 100)}}{100} \\ &= 24575 \times \frac{\text{accele_1 (0 to 100)}}{100} \end{aligned} $ <p>For example, when accele_1 = 80 the calculation of the value written to MTU4.TGRC is as follows:</p> $ \text{MTU4.TGRC} = 24575 \times 80/100 = 19660 $	bit	7	6	5	4	3	2	1	0	Original value (port 2)	/	P76	P75	P74	P73	P72	P71	P70	AND value	/	1	1	0	1	1	1	1	Result	/	P76	P75	0	P73	P72	P71	P70
bit	7	6	5	4	3	2	1	0																													
Original value (port 2)	/	P76	P75	P74	P73	P72	P71	P70																													
AND value	/	1	1	0	1	1	1	1																													
Result	/	P76	P75	0	P73	P72	P71	P70																													

Lines 746 to 747

If the value is negative, lines 746 and 747 are processed.
 P74 is set to 1 and PWM is output on P72, causing the left motor to operate in the reverse direction according to the PWM ratio.
 Line 746 performs the following bit operations and sets pin P74 to 1:

bit	7	6	5	4	3	2	1	0
Original value (port 2)	/	P76	P75	P74	P73	P72	P71	P70
OR value	/	0	0	1	0	0	0	0
Result	/	P76	P75	1	P73	P72	P71	P70

Line 747 performs the calculation below and sets the PWM value in timer MTU3_4 timer general register C (MTU4.TGRC). If there are digits after the decimal point, they are discarded.

$$\begin{aligned}
 \text{MTU4.TGRC} &= (\text{PWM_CYCLE} - 1) \times \frac{-\text{accele_1}(-1 \text{ to } -100)}{100} \\
 &= 24575 \times \frac{-\text{accele_1}(-1 \text{ to } -100)}{100}
 \end{aligned}$$

A key point is that the value of variable **accele_1** is a negative number. In circuit terms, setting P74 to 1 specifies reverse operation, so **accele_1** is converted to a positive number for the calculation. The conversion method is to specify **-accele_1** in the expression. For example, when **accele_1** = -50 the calculation of the value written to MUT4.TGRC is as follows:

$$\text{MTU4.TGRC} = 24575 \times \{-(-50)\}/100 = 24575 \times 50/100 = 12287.5 = 12287$$

(4) Right Motor Control

This portion of the function controls the right motor. The right motor PWM output is on pin P73. The PWM output on pin P73 is specified by the PWM value setting in MTU3_4 timer general register B (MTU4.TGRB). However, this setting is not made directly but to MTU3_4 timer general register D (MTU4.TGRD), which functions as a buffer register, instead.

```

750 :    /* Right Motor Control */
751 :    if( accele_r >= 0 ) {
752 :        PORT7.DR.BYTE &= 0xdf;
753 :        MTU4.TGRD = (long)( PWM_CYCLE - 1 ) * accele_r / 100;
754 :    } else {
755 :        PORT7.DR.BYTE |= 0x20;
756 :        MTU4.TGRD = (long)( PWM_CYCLE - 1 ) * ( -accele_r ) / 100;
757 :    }
758 : }
    
```

Line 751	Checks whether the left motor PWM value ratio is a positive or a negative value. If positive, lines 752 and 753 are processed, if negative, lines 755 and 756.																																				
Lines 752 to 753	<p>If the value is positive, lines 752 and 753 are processed. P75 is cleared to 0 and PWM is output on P73, causing the right motor to operate in the forward direction according to the PWM ratio. Line 752 performs the following bit operations and clears pin P75 to 0:</p> <table border="1" data-bbox="496 439 1262 696"> <thead> <tr> <th>bit</th> <th>7</th> <th>6</th> <th>5</th> <th>4</th> <th>3</th> <th>2</th> <th>1</th> <th>0</th> </tr> </thead> <tbody> <tr> <td>Original value (port 7)</td> <td>/</td> <td>P76</td> <td>P75</td> <td>P74</td> <td>P73</td> <td>P72</td> <td>P71</td> <td>P70</td> </tr> <tr> <td>AND value</td> <td>/</td> <td>1</td> <td>0</td> <td>1</td> <td>1</td> <td>1</td> <td>1</td> <td>1</td> </tr> <tr> <td>Result</td> <td>/</td> <td>P76</td> <td>0</td> <td>P74</td> <td>P73</td> <td>P72</td> <td>P71</td> <td>P70</td> </tr> </tbody> </table> <p>Line 753 performs the calculation below and sets the PWM value in MTU3_4 timer general register D (MTU4.TGRD). If there are digits after the decimal point, they are discarded.</p> $ \begin{aligned} \text{MTU4.TGRD} &= (\text{PWM_CYCLE} - 1) \times \frac{\text{accele_r (0 to 100)}}{100} \\ &= 24575 \times \frac{\text{accele_r (0 to 100)}}{100} \end{aligned} $ <p>For example, when accele_r = 20 the calculation of the value written to MTU4.TGRD is as follows:</p> $\text{MTU4.TGRD} = 24575 \times 20/100 = 4915$	bit	7	6	5	4	3	2	1	0	Original value (port 7)	/	P76	P75	P74	P73	P72	P71	P70	AND value	/	1	0	1	1	1	1	1	Result	/	P76	0	P74	P73	P72	P71	P70
bit	7	6	5	4	3	2	1	0																													
Original value (port 7)	/	P76	P75	P74	P73	P72	P71	P70																													
AND value	/	1	0	1	1	1	1	1																													
Result	/	P76	0	P74	P73	P72	P71	P70																													
Lines 755 to 756	<p>If the value is negative, lines 755 and 756 are processed. P75 is set to 1 and PWM is output on P73, causing the right motor to operate in the reverse direction according to the PWM ratio. Line 755 performs the following bit operations and sets pin P75 to 1:</p> <table border="1" data-bbox="496 1301 1262 1559"> <thead> <tr> <th>bit</th> <th>7</th> <th>6</th> <th>5</th> <th>4</th> <th>3</th> <th>2</th> <th>1</th> <th>0</th> </tr> </thead> <tbody> <tr> <td>Original value (port 7)</td> <td>/</td> <td>P76</td> <td>P75</td> <td>P74</td> <td>P73</td> <td>P72</td> <td>P71</td> <td>P70</td> </tr> <tr> <td>OR value</td> <td>/</td> <td>0</td> <td>1</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> <td>0</td> </tr> <tr> <td>Result</td> <td>/</td> <td>P76</td> <td>1</td> <td>P74</td> <td>P73</td> <td>P72</td> <td>P71</td> <td>P70</td> </tr> </tbody> </table> <p>Line 756 performs the calculation below and sets the PWM value in MTU3_4 timer general register D (MTU4.TGRD). If there are digits after the decimal point, they are discarded.</p> $ \begin{aligned} \text{MTU4.TGRD} &= (\text{PWM_CYCLE} - 1) \times \frac{-\text{accele_r (-1 to -100)}}{100} \\ &= 24575 \times \frac{-\text{accele_r (-1 to -100)}}{100} \end{aligned} $ <p>A key point is that the value of variable accele_r is a negative number. In circuit terms, setting P2_3 to 1 specifies reverse operation, so accele_r is converted to a positive number for the calculation. The conversion method is to specify -accele_r in the expression. For example, when accele_r = -90 the calculation of the value written to TRDGRC1 is as follows:</p> $\text{MTU4.TGRD} = 24575 \times \{-(-90)\}/100 = 24575 \times 90/100 = 22117.5 = 22117$	bit	7	6	5	4	3	2	1	0	Original value (port 7)	/	P76	P75	P74	P73	P72	P71	P70	OR value	/	0	1	0	0	0	0	0	Result	/	P76	1	P74	P73	P72	P71	P70
bit	7	6	5	4	3	2	1	0																													
Original value (port 7)	/	P76	P75	P74	P73	P72	P71	P70																													
OR value	/	0	1	0	0	0	0	0																													
Result	/	P76	1	P74	P73	P72	P71	P70																													

(5) Dip Switch Value and Motor Output

When the **motor** function setting is 100%, the DIP switch setting determines the actual output, as follows:

DIP Switch				Decimal	Calculation	Motor Ratio
P60	P61	P62	P63			
0	0	0	0	0	5/20	25%
0	0	0	1	1	6/20	30%
0	0	1	0	2	7/20	35%
0	0	1	1	3	8/20	40%
0	1	0	0	4	9/20	45%
0	1	0	1	5	10/20	50%
0	1	1	0	6	11/20	55%
0	1	1	1	7	12/20	60%
1	0	0	0	8	13/20	65%
1	0	0	1	9	14/20	70%
1	0	1	0	10	15/20	75%
1	0	1	1	11	16/20	80%
1	1	0	0	12	17/20	85%
1	1	0	1	13	18/20	90%
1	1	1	0	14	19/20	95%
1	1	1	1	15	20/20	100%

5.4.23. **handle** Function (Servo Steering Operation)

```

760 : /*****/
761 : /* Servo steering operation */
762 : /* Arguments: servo operation angle: -90 to 90 */
763 : /* -90: 90-degree turn to left, 0: straight, */
764 : /* 90: 90-degree turn to right */
765 : /*****/
766 : void handle( int angle )
767 : {
768 :     /* When the servo move from left to right in reverse, replace "-" with "+". */
769 :     MTU3.TGRD = SERVO_CENTER - angle * HANDLE_STEP;
770 : }
    
```

(1) Using the handle Function

The use of the **handle** function is described below.

```

handle( Servo angle );
    
```

The argument specifies the servo angle. The correspondence between the value and the servo angle is as follows:

Value	Description
Negative	The servo turns to the left the specified number of degrees.
0	The servo is oriented to 0 degrees (straight ahead). If the servo does not point straight ahead when the setting value is 0, the SERVO_CENTER value is off and requires adjustment.
Positive	The servo turns to the right the specified number of degrees.

Observe the following code examples:

```

handle( 0 );           0 degree
handle( 30 );         Right 30 degrees
handle( -45 );       Left 45 degrees
    
```

(2) Program Description

<pre>765 : MTU3.TGRD = SERVO_CENTER - angle * HANDLE_STEP; [1] [2] [3] [4]</pre>
--

[1]	The PWM on width setting for pin P71, which is connected to the servo, is specified by the setting of MTU3.TGRB. However, this setting is not made directly but to MTU3.TGRD, which functions as a buffer register, instead.
[2]	Value corresponding to 0 degrees.
[3]	The angle specified by the handle function is assigned to this variable.
[4]	Among of increase equivalent to 1 degree.

The examples below illustrate the calculation of the value assigned to MTU3.TGRD.

Note: In these examples, SERVO_CENTER = 2320 and HANDLE_STEP = 13.

- 0 degrees

$$\begin{aligned} \text{MTU3.TGRD} &= \text{SERVO_CENTER} - \text{angle} * \text{HANDLE_STEP} \\ &= 2320 \quad - \boxed{0} * 13 \\ &= 2320 \end{aligned}$$

- 30 degrees

$$\begin{aligned} \text{MTU3.TGRD} &= \text{SERVO_CENTER} - \text{angle} * \text{HANDLE_STEP} \\ &= 2320 \quad - \boxed{30} * 13 \\ &= 2320 \quad - 390 \\ &= 1930 \end{aligned}$$

- -45 degrees

$$\begin{aligned} \text{MTU3.TGRD} &= \text{SERVO_CENTER} - \text{angle} * \text{HANDLE_STEP} \\ &= 2320 \quad - \boxed{(-45)} * 13 \\ &= 2320 \quad - (-585) \\ &= 2905 \end{aligned}$$

5.4.24. Start

This is the main function. It is the first C language program to be called from the startup routine and run.

```

66 : /*****/
67 : /* Main program */
68 : /*****/
69 : void main(void)
70 : {
71 :     /* Initialize MCU functions */
72 :     init();
73 :
74 :     /* Initialize micom car state */
75 :     handle( 0 );
76 :     motor( 0, 0 );
    
```

Line 72	This function initialises the on-chip peripheral functions of the RX62T MCU.
Lines 75 to 76	This initialises the state of the MCU car. The servo angle is set to 0 degrees by the handle function. The left motor and right motor are set to 0% by the motor function.

5.4.25. Patterns

In kit12_rx62t.c a method called patterns is used to organize the program.

This basically involves dividing the program up into small sections. For example, there is a section for button input standby processing, one for checking whether or not a start bar is present, and so on.

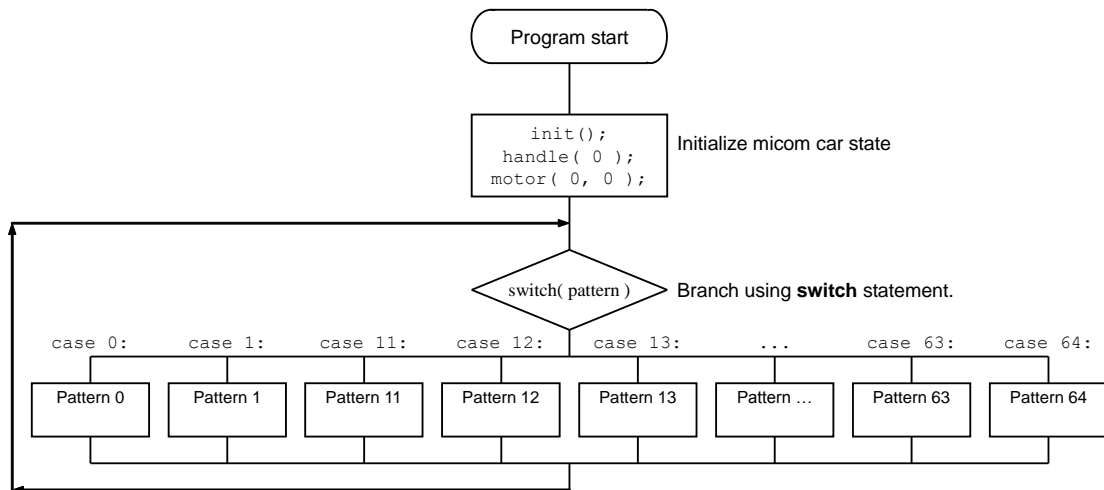
Next, a variable called **pattern** is created. This makes it possible to select the program section to be run by setting the value of the variable **pattern**.

For example, setting the value of variable **pattern** to 0 causes button input standby processing to be performed, setting variable **pattern** to 1 causes processing to check whether or not a start bar is present to be performed, and so on.

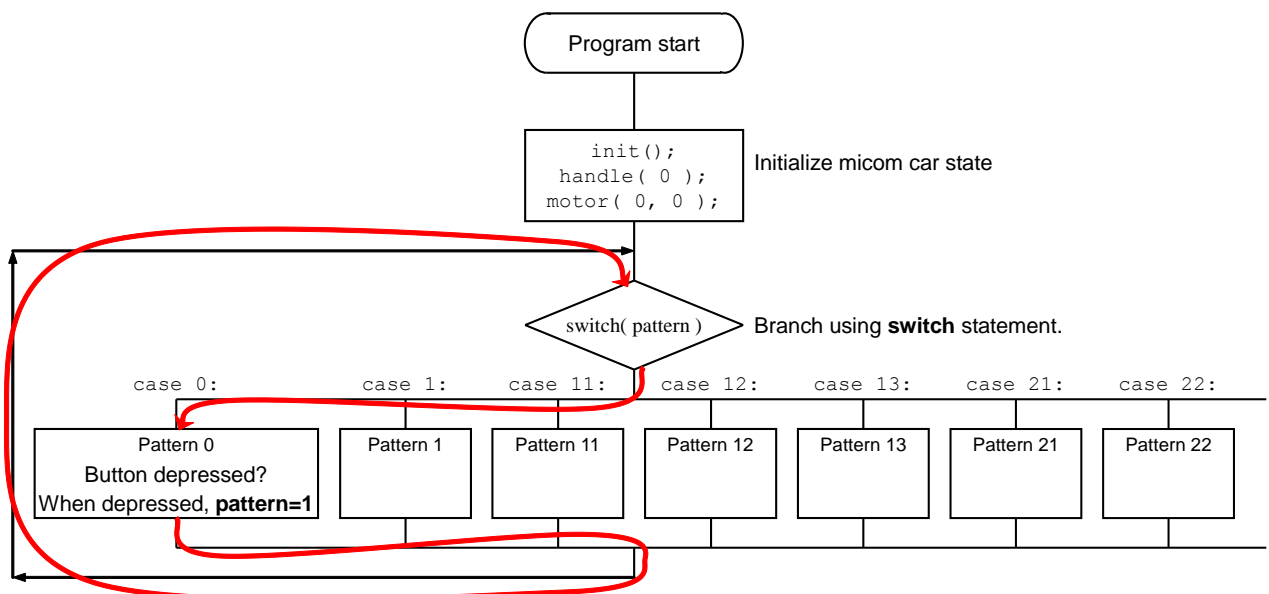
Using this method of dividing the processing into various patterns makes the program code easier to read. Such use of patterns is sometimes called **modular programming**.

5.4.26. Writing a Program

When writing a C language program with patterns, switch statements are used for branching. This is shown in the following flowchart:

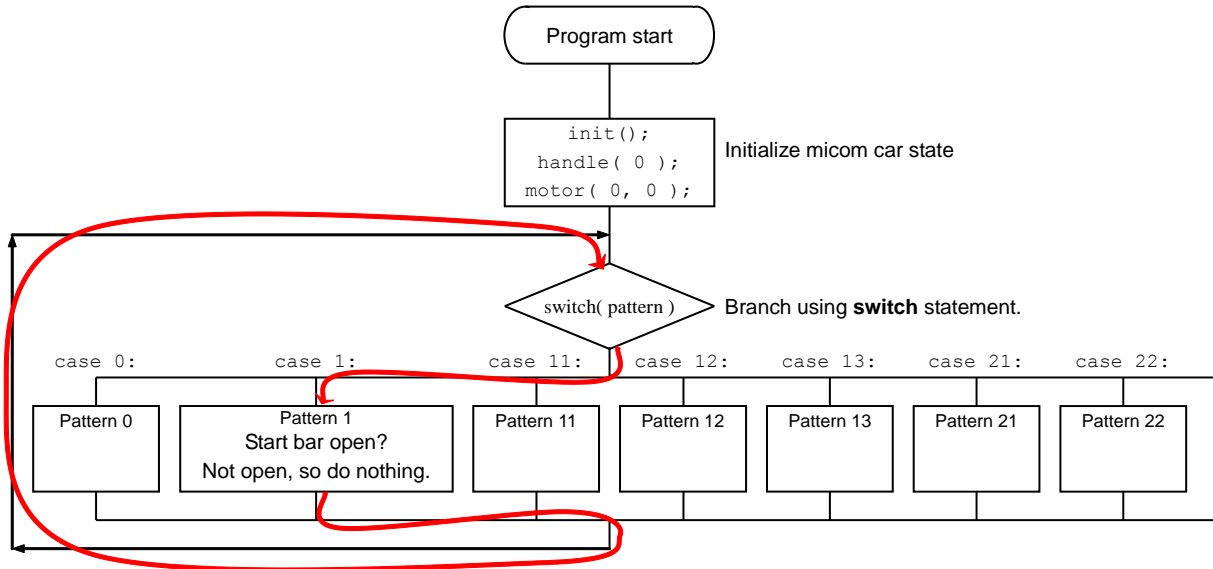


At startup, the value of variable **pattern** is 0. The **switch** statement causes execution to continue with **pattern 0**, which is the portion of the program corresponding to **case 0**. This will be described in detail later, but pattern 0 is button input standby processing. When the button is depressed, **pattern=1** is executed. This is shown in the following figure:



The value of variable **pattern** is 1 the next time the **switch** statement is executed, so the portion of the program corresponding to case 1 is run. In this program, running the portion of the program corresponding to **switch (pattern) case 1** means running **pattern 1**.

Pattern 1 is the portion of the program that checks whether or not the start bar is open. The processing flow is as follows:



As the above illustrates, the program has a modular organization. Each module performs a simple check, such as “start button depressed?” or “start bar open?”, and changes the pattern number (the value of variable **pattern**) when the condition it met.

The program code is as follows. It uses ordinary **switch** and **case** statements.

<pre> switch(pattern) { case 0: /* pattern=0 processing */ break; case 1: /* pattern=1 processing */ break; default: /* if neither */ break; } </pre>	<p>The value of pattern is compared to the values defined in the various case statements, and execution jumps to the case position whose value matches.</p> <p>Processing ends when a break statement or the end of the switch statement is encountered.</p> <p>If none of the values defined in the case statements matches, the default statement is executed. Incidentally, nothing is executed if there is no default statement.</p>
---	--

5.4.27. Pattern Descriptions

The pattern numbers used in kit12_rx62t.c, the processing performed by each pattern, and the conditions for changes between patterns are listed below.

Current pattern	Processing description	Pattern change condition
0	Wait for button input	<ul style="list-style-type: none"> When button depressed, to pattern 1
1	Check if start bar is open	<ul style="list-style-type: none"> When start bar open detected, to pattern 11
11	Normal trace	<ul style="list-style-type: none"> At large turn to right, to pattern 12 At large turn to left, to pattern 13 When crossline detected, to pattern 21 When right half line detected, to pattern 51 When left half line detected, to pattern 61
12	Check end of large turn to right	<ul style="list-style-type: none"> When large turn to right completed, to pattern 11 When crossline detected, to pattern 21 When right half line detected, to pattern 51 When left half line detected, to pattern 61
13	Check end of large turn to left	<ul style="list-style-type: none"> When large turn to left completed, to pattern 11 When crossline detected, to pattern 21 When right half line detected, to pattern 51 When left half line detected, to pattern 61
21	Processing at 1st crossline detection	<ul style="list-style-type: none"> When servo and speed settings completed, to pattern 22
22	Read but ignore 2nd time	<ul style="list-style-type: none"> After 100 ms, to pattern 23
23	Trace, crank detection after crossline	<ul style="list-style-type: none"> When left crank detected, to pattern 31 When right crank detected, to pattern 41
31	Clearing from Left crank - wait until stable	<ul style="list-style-type: none"> After 200 ms, to pattern 32
32	Clearing from Left crank - check end of turn	<ul style="list-style-type: none"> After clearing from left crank, to pattern 11
41	Clearing from Right crank - wait until stable	<ul style="list-style-type: none"> After 200 ms, to pattern 42
42	Clearing from Right crank - check end of turn	<ul style="list-style-type: none"> After clearing from right crank, to pattern 11
51	Processing at 1st right half line detection	<ul style="list-style-type: none"> When servo and speed settings completed, to pattern 52
52	Read but ignore 2nd time	<ul style="list-style-type: none"> After 100 ms, to pattern 53
53	Trace after right half line	<ul style="list-style-type: none"> If centre line disappears, turn steering wheel to right and go to pattern 54
54	Right lane change end check	<ul style="list-style-type: none"> When new centre line is at sensor centre position, to pattern 11
61	Processing at 1st left half line detection	<ul style="list-style-type: none"> When servo and speed settings completed, to pattern 62
62	Read but ignore 2nd time	<ul style="list-style-type: none"> After 100 ms, to pattern 63
63	Trace after left half line	<ul style="list-style-type: none"> If centre line disappears, turn steering wheel to left and go to pattern 64
64	Left lane change end check	<ul style="list-style-type: none"> When new centre line is at sensor centre position, to pattern 11

5.4.28. Initial while and switch when Using Patterns

```

78 :   while( 1 ) {
79 :       switch( pattern ) {
105 :         case 0:
           Pattern 0 processing
119 :           break;
120 :
121 :         case 1:
           Pattern 1 processing
137 :           break;

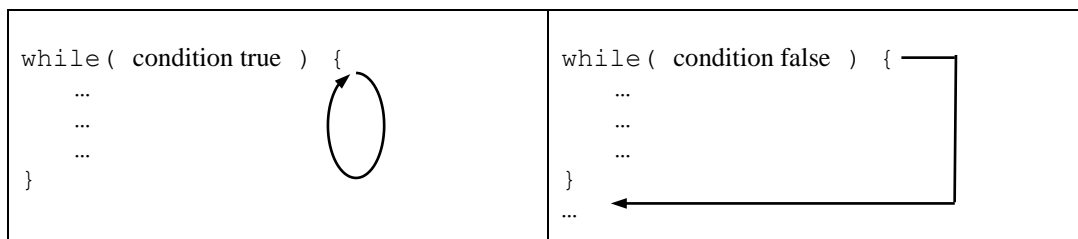
           Pattern processing

473 :         default:
474 :             /* If neither, return to standby state */
475 :             pattern = 0;
476 :             break;
477 :         }
478 :     }

```

while(1) { in line 78 and **}** in line 478 form a pair, and **switch(pattern) {** in line 79 and **}** in line 477 form a pair. Generally speaking, lines enclosed between opening and closing curly brackets { } are indented four characters to make the code easier to read. This convention is generally followed in the code listing of this program. However, the lines containing **while** and **switch** are not indented. This is to prevent complex lines from exceeding the right margin and being split into two lines in the listing, which would be more difficult to read. After all, the reason for indenting some lines is to make the code listing easier to read. Extra spaces at the beginning of lines have no effect when the code is compiled. However, if this exception is bothersome, feel free to indent lines 79 to 477 by adding four spaces at the beginning of each line.

while(condition) is a control statement that causes the code enclosed in the curly brackets { } to be executed repeatedly for as long as the condition is true, and the code following the curly brackets { } to be executed when the condition is false.



“True” and “false” are defined as follows:

	Description	Example
True	Correct, other than 0	3 < 5 3==3 1 2 3 -1 -2 -3
False	Not correct, 0	5 < 3 3==6 0

The code listing reads **while(1)**. A value of 1 is always true, so the code enclosed in the curly brackets { } is repeated infinitely. In a Windows program, for example, such an infinite loop would be a problem because it would prevent the user from quitting the application. But since our program is designed to operate a MCU car, there is no problem. Once the MCU car crosses the finish line (or runs off the course), someone can pick it up and

switch it off. On the other hand, if the MCU were to fail to complete its processing correctly and execution of the program ended, processing would continue into an area of memory containing no executable code, resulting in unpredictable behaviour. The usual approach in MCU car programs is to have the MCU repeatedly do nothing (enter an endless loop) without exiting the program or to have it transition to a low-power mode called sleep mode, in which operation stops and the MCU waits to be awakened.

5.4.29. Pattern 0: Wait For Button Input

Pattern 0 is the section of the program that checks whether or not the pushbutton has been depressed. While this checking is taking place, there is no way to know whether the program is running or not. To provide such an indication, LED0 and LED1 are illuminated alternately.

First is the section for detecting pushbutton input. The **pushsw_get** function checks the pushbutton state. It returns a value of 1 when the button is depressed, so the code enclosed in the curly brackets is executed and the value of **pattern** is set to 1.

```

105 :         case 0:
106 :             /* Wait for switch input */
107 :             if( pushsw_get() ) {           <- If button depressed (return value other than 0)...
108 :                 pattern = 1;             <- Set pattern to 1.
109 :                 cnt1 = 0;                <- Clear cnt1 to 0.
110 :                 break;                  <- End switch statement.
111 :             }

```

<p>Line 107</p>	<p>When the pushbutton is depressed, the code enclosed in the curly brackets (lines 102 to 104) is executed. Nothing is executed if the pushbutton is not depressed.</p> <p>The if statement performs a comparison. The following line compares the return value of the pushsw_get function with the value 1:</p> <pre>if(pushsw_get() == 1) {</pre> <p style="text-align: right;">If the return value of the pushsw_get function is 1...</p> <p>But our code listing reads as follows:</p> <pre>if(pushsw_get()) {</pre> <p>There is no value provided for comparison. In the C language, the meaning is as follows:</p> <pre>if(Value) { If the value is other than 0, the condition is considered true and this section is executed. } else { If the value is other than 0, the condition is considered true and this section is executed. }</pre> <p>The return value of the pushsw_get function is 1 when the button is in the depressed state and 0 when it is not depressed. Therefore, operation is as follows when the pushbutton is depressed:</p> <pre>if(1) { pushsw_get() returns a value of 1. The value is other than 0, so the code between the curly brackets is executed. }</pre>
<p>Line 108</p>	<p>The value 1 is assigned to variable pattern. The case 1: portion of the program is run the next time the switch-case statement is executed.</p>

After this are added the lines that cause the LEDs to turn on and off. First LED2 lights for 0.1 second, then LED3 lights for 0.1 second, then the sequence is repeated.

```

112 :     if( cnt1 < 100 ) {           <- Is value of cnt1 0 to 99?
113 :         led_out( 0x1 );         <- If so, light LED2 only.
114 :     } else if( cnt1 < 200 ) {    <- Is value of cnt1 100 to 199?
115 :         led_out( 0x2 );         <- If so, light LED3 only.
116 :     } else {                     <- If value of cnt1 is something else (200 or greater)...
117 :         cnt1 = 0;                 <- Clear cnt1 to 0.
118 :     }

```

Most variables, such as the variable **pattern**, do not change value once set until they are explicitly set to a new value. In kit12_rx62t.c, **variables cnt0 and cnt1 only are exceptions to this**. Variables **cnt0** and **cnt1** are each incremented (+1) every 1 ms by the **interrupt** function. This means these variables can be used to measure time.

The purpose of the **break** statement in line 113 is to end case 0.

```

78 :     while( 1 ) {
79 :         switch( pattern ) {
105 :             case 0:
Lines omitted
119 :                 break;
Lines omitted
477 :             }
478 :         }

```

[1]	The break statement in line 119 causes control to jump to the next line after line 477, which contains the closing curly bracket of the switch-case statement.
[2]	Line 478 is processed next, but since it contains the closing curly bracket of a while statement matching the opening curly bracket in line 78, control returns to line 78.
[3]	The switch-case statement starting on line 79 is processed, and control moves to a case statement according to the value of variable pattern .

Note: Not Using Variable **cnt1**

What if we used the **timer** function instead of **cnt1**?

```

if( pushsw_get() ) {
    pattern = 1;
    cnt1 = 0;
    break;
}
timer( 100 );           <- This line triggers a pause of 100 ms!!
led_out( 0x1 );
timer( 100 );           <- This line triggers a pause of 100 ms!!
led_out( 0x2 );
break;

```

This is simpler. Maybe this approach is better. But the **timer** function **does nothing but wait for a period of time to elapse**. If the pushbutton is depressed and then released while the timer function is executing, the button may no longer be in the depressed state when the **pushsw_get** function executes. The depress would not be detected in that case. In this example code the **timer** function takes 0.1 seconds to run, so you would have to depress and release the button very quickly for it not to be detected. However, if the duration was longer, say, several seconds, the periods when the button state was not being checked would be too long and it would not be possible to detect depresses reliably. **This is why variable cnt1 is used to check the time while the button state checking is taking place.**

5.4.30. Pattern 1: Check if Start Bar Is Open

Pattern 1 is the section of the program that checks whether or not the start bar is open. While this checking is taking place, there is no way to know whether the program is running or not. To provide such an indication, LED2 and LED3 are illuminated alternately.

First is the section for detecting whether the start bar is open or closed.

```

121 :         case 1:
122 :             /* Check if start bar is open */
123 :             if( !startbar_get() ) {
124 :                 /* Start!! */
125 :                 led_out( 0x0 );
126 :                 pattern = 11;
127 :                 cnt1 = 0;
128 :                 break;
129 :             }

```

<p>Line 123</p>	<p>If the start bar is open, the code enclosed in the curly brackets (lines 125 to 128) is executed. Nothing is executed if it is closed.</p> <p>The startbar_get function returns a value of 1 when the start bar is present (the sensors produce a response) and a value of 0 when the start bar is absent (the sensors produce no response). The code is analogous to that used in the pushsw_get function described earlier.</p> <pre> if(startbar_get()) { Execute this code if the start bar is present. } </pre> <p>This would mean the code between the curly brackets would be executed if the start bar were present. We actually want this code to run if the start bar is not present. Therefore, we add an exclamation mark ! to negate the statement, so now the code between the curly brackets is executed if the start bar is absent. The exclamation mark ! means negation.</p> <pre> if(! Value) { If the value is not other than 0, this section is executed. -> This section is executed if the value is 0. } else { If the value is not 0, this section is executed. -> This section is executed if the value is other than 0. } </pre> <p>This results in the following:</p> <pre> if(!startbar_get()) { If the start bar is not present, this section is executed. } </pre>
<p>Line 126</p>	<p>The value 11 is assigned to variable pattern. The case 11: portion of the program is run the next time the switch-case statement is executed.</p>
<p>Line 128</p>	<p>The break statement causes control to jump to the closing curly bracket of the switch-case statement.</p>

Next come lines that cause the LEDs to turn on and off. First LED2 lights for 0.05 seconds, then LED3 lights for 0.05 seconds, then the sequence is repeated. The flashing is faster than in pattern 1, so you know that the MCU car is waiting for the start bar to open.

<pre> 130 : 131 : 132 : 133 : 134 : 135 : 136 : 137 : </pre>	<pre> if(cnt1 < 50) { led_out(0x1); } else if(cnt1 < 100) { led_out(0x2); } else { cnt1 = 0; } break; </pre>	<pre> <- Is value of cnt1 0 to 49? <- If so, light LED0 only. <- Is value of cnt1 50 to 99? <- If so, light LED1 only. <- If value of cnt1 is something else (100 or greater)... <- Clear cnt1 to 0. </pre>
--	--	---

5.4.31. Pattern 11: Normal Trace

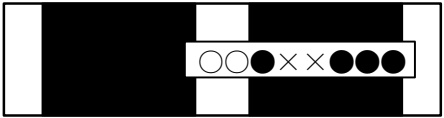
Pattern 11 is a status in which the program reads the sensor data and controls the left motor, right motor, and servo. First, let's imagine the possible sensor states. There are eight sensors, but if we tried to make use of them all we would have to deal with too much detection state data and the program code would become overly complex.

Instead, we will apply masking with **MASK3_3** and use data from a total of six sensors, three on the right and three on the left, to determine the state of the course.

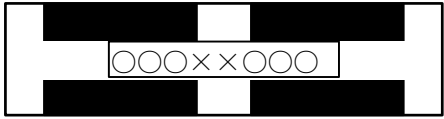
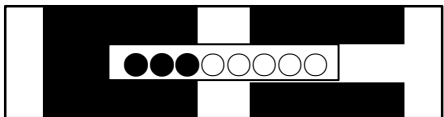
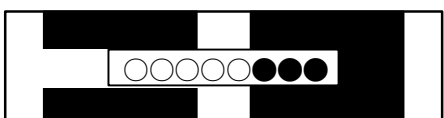
Next, let's consider the turn angle as well as the left motor and right motor PWM values. The idea is that we will keep the steering wheel pointed straight ahead and increase the speed when the sensors indicate that the MCU car is in the centre of the track. When the sensors indicate that the MCU car has deviated from the centre, we will turn the steering wheel and lower the speed of the left and right motors.

This works as follows in kit07_rx62t.c:

	Course and sensor state	Value read by sensors	Hexadecimal	Steering angle	Left motor PWM	Right motor PWM
1		00000000	0x00	0	100	100
2		00000100	0x04	5	100	100
3		00000110	0x06	10	80	67
4		00000111	0x07	15	50	38
5		00000011	0x03	25	30	19
6		00100000	0x20	-5	100	100
7		01100000	0x60	-10	67	80
8		11100000	0xe0	-15	38	50

9		11000000	0xc0	-25	19	30
---	---	----------	------	-----	----	----

The MCU car course also includes crosslines, right half lines, and left half lines. There are functions designed to detect each of these, and we will make use of them.

	Course and sensor state	Course feature and processing	Function used for checking
10	 6 sensors used	Horizontal line (crossline) ↓ When detected, to crank processing (pattern 21)	check_crossline
11	 8 sensors used	Horizontal line from centre to right edge only (right half line) ↓ When detected, to right half line processing (pattern 51)	check_rightline
12	 8 sensors used	Horizontal line from centre to left edge only (left half line) ↓ When detected, to left half line processing (pattern 61)	check_leftline

We will write the program code using the above tables as a basis.

(1) Read Sensors

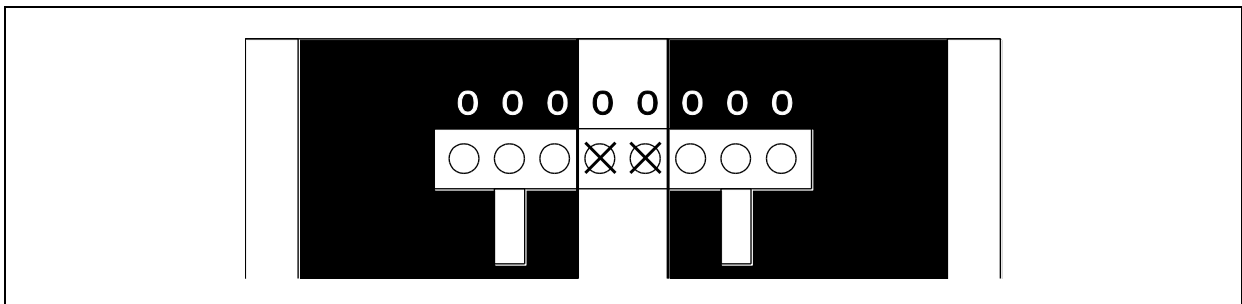
```
153 :           switch( sensor_inp(MASK3_3) ) {
```

This reads the state of the sensors. **MASK3_3** is used so that three sensors on the right and three on the left are read. A **switch-case** statement is used to branch to different locations in the program code according to the sensor state.

(2) Straight Forward

```
154 :           case 0x00:
155 :             /* Center -> straight */
156 :             handle( 0 );
157 :             motor( 100 ,100 );
158 :             break;
```

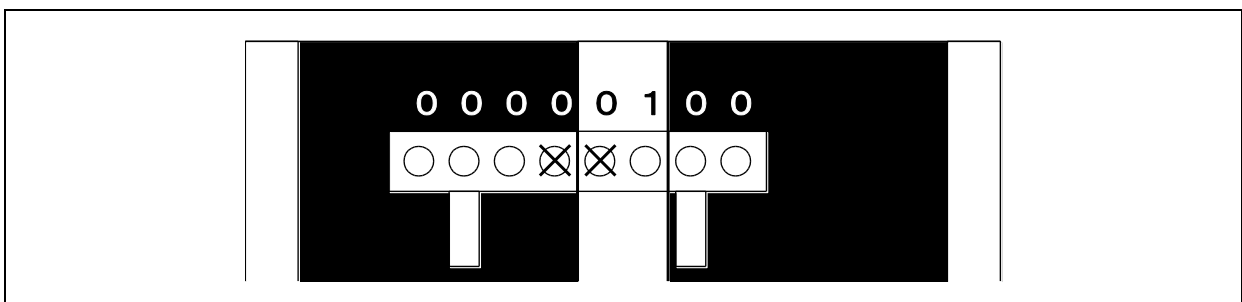
This is the state when the sensor value is 0x00. In this state the MCU car is moving straight forward, as shown in the figure below. It proceeds using the following setting values: servo angle 0 degrees, left motor 100%, the right motor 100%.



(3) Slight Amount Left of Centre

```
160 :           case 0x04:
161 :             /* Slight amount left of center -> slight turn to right */
162 :             handle( 5 );
163 :             motor( 100 ,100 );
164 :             break;
```

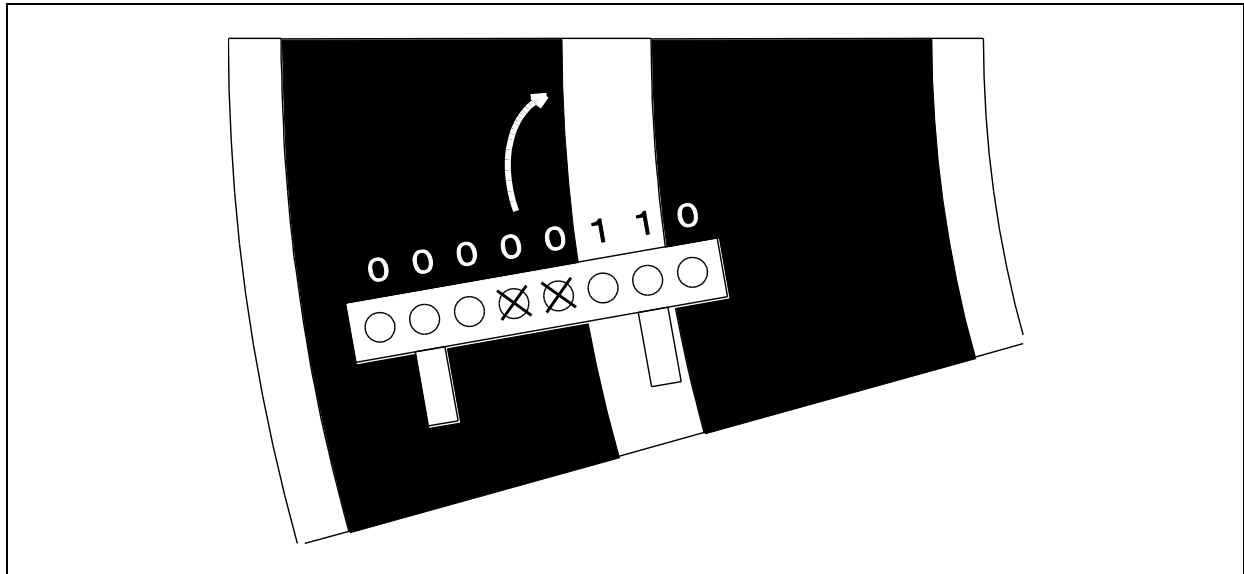
This is the state when the sensor value is 0x04. In this state the MCU car is positioned a slight amount to the left of centre, as shown in the figure below. It proceeds using the following setting values in order to move back to the centre position: servo angle 5 degrees right, left motor 100%, and right motor 100%.



(4) Small Amount Left of Centre

```
166 :           case 0x06:  
167 :             /* Small amount left of center -> small turn to right */  
168 :             handle( 10 );  
169 :             motor( 80 ,67 );  
170 :             break;
```

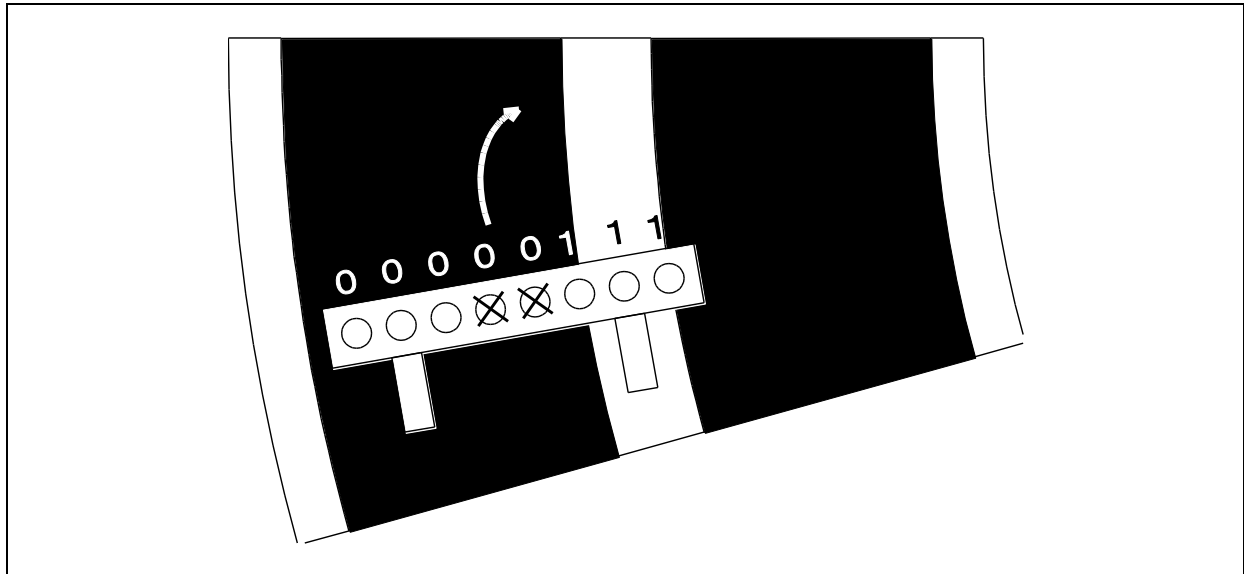
This is the state when the sensor value is 0x06. In this state the MCU car is positioned a small amount to the left of centre, as shown in the figure below. It proceeds using the following setting values and at reduced speed in order to move back to the centre position: servo angle 10 degrees right, left motor 80%, and right motor 67%.



(5) Medium Amount Left of Centre

```
172 :           case 0x07:  
173 :           /* Medium amount left of center -> medium turn to right */  
174 :           handle( 15 );  
175 :           motor( 50 , 38 );  
176 :           break;
```

This is the state when the sensor value is 0x07. In this state the MCU car is positioned a medium amount to the left of centre, as shown in the figure below. It proceeds using the following setting values and at reduced speed in order to move back to the centre position: servo angle 15 degrees right, left motor 50%, and right motor 38%.

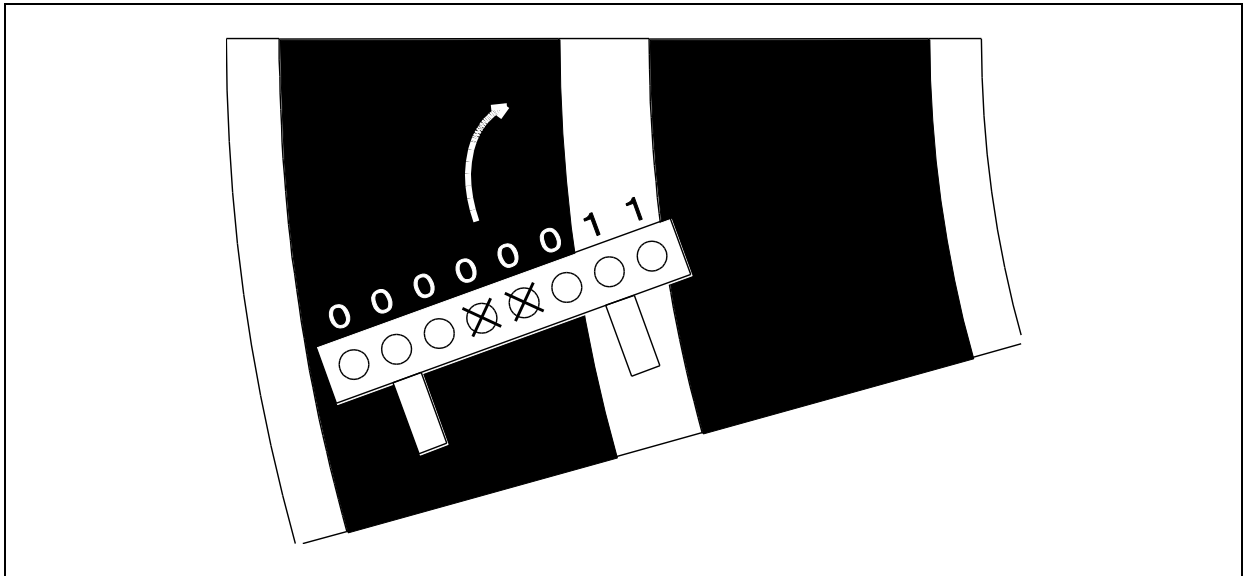


(6) Large Amount Left of Centre

```
178 :          case 0x03:  
179 :              /* Large amount left of center -> large turn to right */  
180 :              handle( 25 );  
181 :              motor( 30 , 19 );  
  
183 :              break;
```

Note: The actual program code starts from line 182. The description here is abbreviated, but details are provided later in this manual.

This is the state when the sensor value is 0x03. In this state the MCU car is positioned a large amount to the left of centre, as shown in the figure below. It proceeds using the following setting values and at reduced speed in order to move back to the centre position: servo angle 25 degrees right, left motor 30%, and right motor 19%.

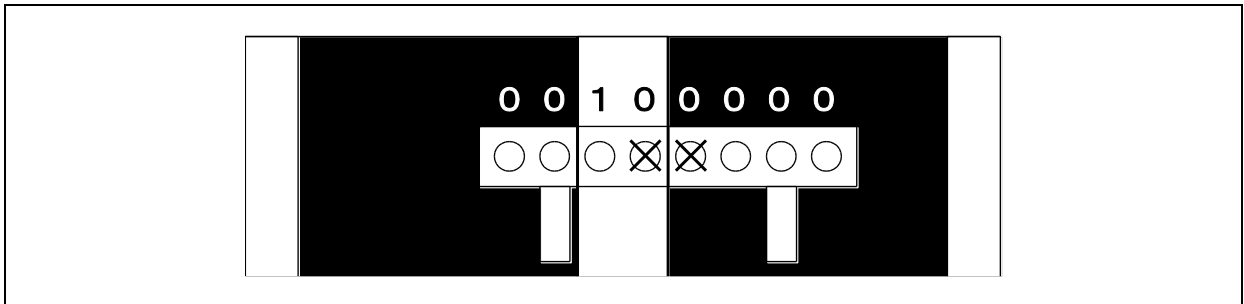


(7) Slight Amount Right of Centre

```

185 :           case 0x20:
186 :             /* Slight amount right of center -> slight turn to left */
187 :             handle( -5 );
188 :             motor( 100 ,100 );
189 :             break;
    
```

This is the state when the sensor value is 0x20. In this state the MCU car is positioned a slight amount to the right of centre, as shown in the figure below. It proceeds using the following setting values in order to move back to the centre position: servo angle 5 degrees left, left motor 100%, and right motor 100%.

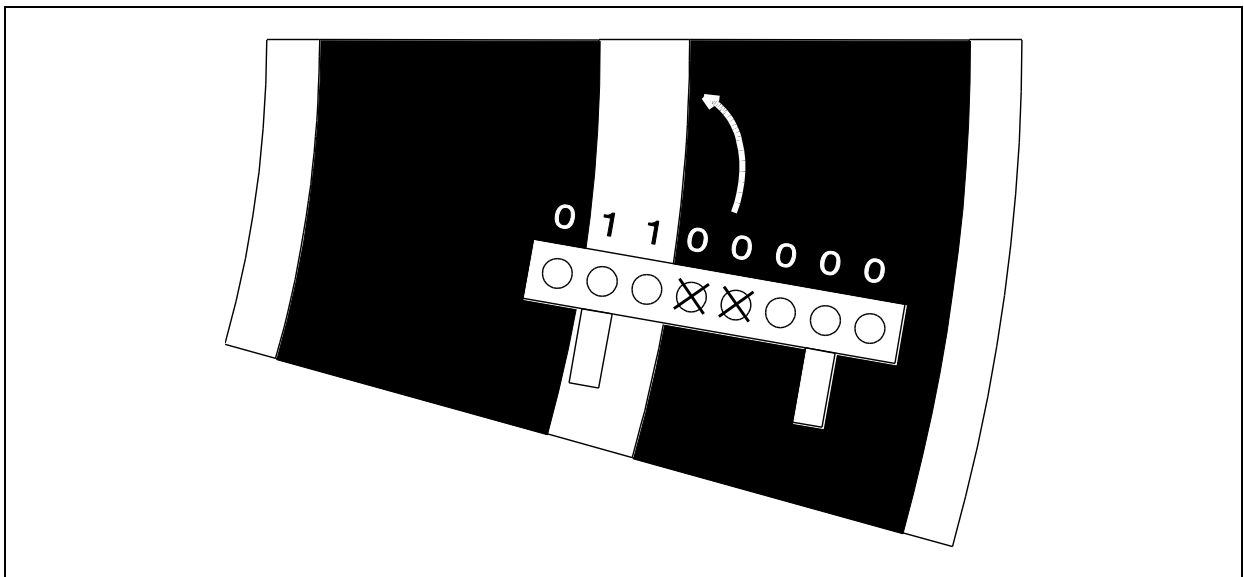


(8) Small Amount Right of Centre

```

191 :           case 0x60:
192 :             /* Small amount right of center -> small turn to left */
193 :             handle( -10 );
194 :             motor( 67 ,80 );
195 :             break;
    
```

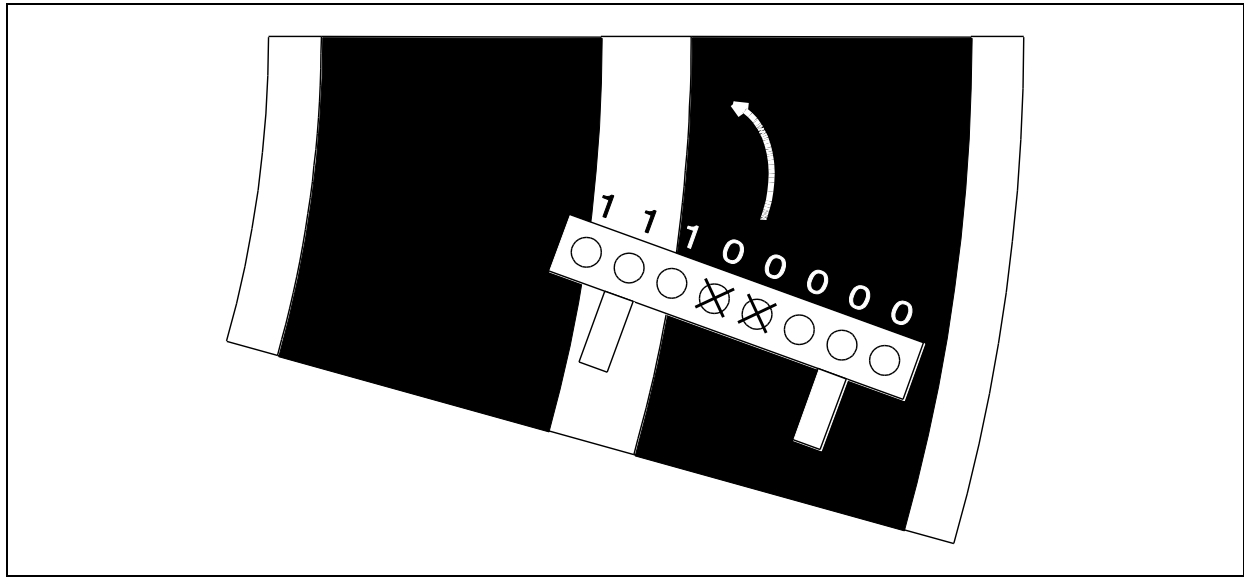
This is the state when the sensor value is 0x60. In this state the MCU car is positioned a small amount to the right of centre, as shown in the figure below. It proceeds using the following setting values and at reduced speed in order to move back to the centre position: servo angle 10 degrees left, left motor 67%, and right motor 80%.



(9) Medium Amount Right of Centre

```
197 :           case 0xe0:  
198 :             /* Medium amount right of center -> medium turn to left */  
199 :             handle( -15 );  
200 :             motor( 38 , 50 );  
201 :             break;
```

This is the state when the sensor value is 0xe0. In this state the MCU car is positioned a medium amount to the right of centre, as shown in the figure below. It proceeds using the following setting values and at reduced speed in order to move back to the centre position: servo angle 15 degrees left, left motor 38%, and right motor 50%.



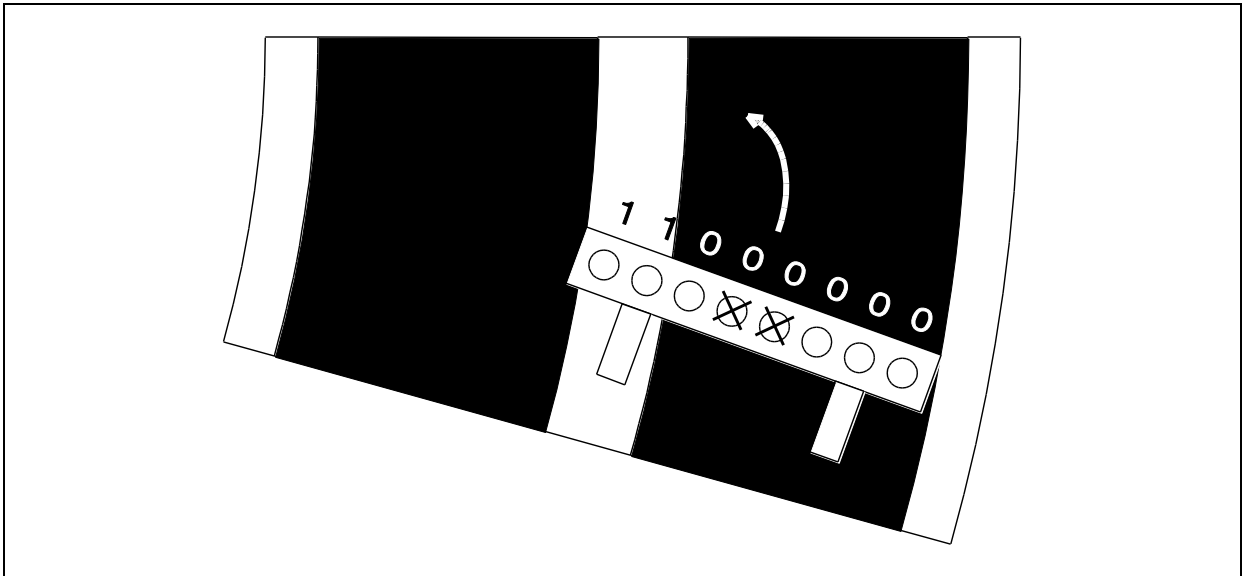
(10) Large Amount Right of Centre

```

203 :           case 0xc0:
204 :             /* Large amount right of center -> large turn to left */
205 :             handle( -25 );
206 :             motor( 19 , 30 );
207 :
208 :             break;
    
```

Note: The actual program code starts from line 207. The description here is abbreviated, but details are provided later in this manual.

This is the state when the sensor value is 0xc0. In this state the MCU car is positioned a large amount to the right of centre, as shown in the figure below. It proceeds using the following setting values and at reduced speed in order to move back to the centre position: servo angle 25 degrees left, left motor 19%, and right motor 30%.



(11) Check Crossline

```

141 :           if( check_crossline() ) { /* Cross line check          */
142 :             pattern = 21;
143 :             break;
144 :           }
    
```

The **check_crossline** function returns a value of 0 to indicate no crossline detected and 1 to indicate crossline detected. When a crossline is detected, **pattern** is set to 21 and a **break** statement is used to end processing of the **switch-case** statement. Crossline checking is important, so this code is executed before the normal trace program code.

(12) Right Half Line

```

145 :           if( check_rightline() ) { /* Right half line detection check */
146 :               pattern = 51;
147 :               break;
148 :           }

```

The **check_rightline** function returns a value of 0 to indicate no right half line detected and 1 to indicate right half line detected. When a right half line is detected, **pattern** is set to 51 and a **break** statement is used to end processing of the **switch-case** statement. Right half line checking is important, so this code is executed before the normal trace program code.

(13) Left Half Line

```

149 :           if( check_leftline() ) { /* Left half line detection check */
150 :               pattern = 61;
151 :               break;
152 :           }

```

The **check_leftline** function returns a value of 0 to indicate no left half line detected and 1 to indicate left half line detected. When a left half line is detected, **pattern** is set to 61 and a **break** statement is used to end processing of the **switch-case** statement. Left half line checking is important, so this code is executed before the normal trace program code.

(14) Other

```

210 :           default:
211 :               break;

```

When a pattern other than those described above is encountered, control jumps to this **default** section, which does nothing.

(15) Position of break Statements to Terminate Execution

A **break** statement is used to terminate execution of a **switch** statement or a **for**, **while**, or **do-while** loop. When a **break** statement is used within overlapping loops, **it only terminates one of the loops within which it is enclosed**, and control passes immediately to the outer loop. It is important to remember this point that a **break** statement **only terminates one of the loops within which it is enclosed**.

The positions exited from by **break** statements within pattern 11 are shown below. These positions differ, so it is important to examine carefully within which loop the **break** statement is used.

```

while( 1 ) {
switch( pattern ) {

    Line omitted


case 11: ← case corresponding to switch( pattern )
    /* Normal trace */
    if( check_crossline() ) {
        pattern = 21;
        break; ← break from switch( pattern ), control passes to 1
    }
    if( check_rightline() ) {
        pattern = 51;
        break; ← break from switch( pattern ), control passes to 1
    }
    if( check_leftline() ) {
        pattern = 61;
        break; ← break from switch( pattern ), control passes to 1
    }
switch( sensor_inp(MASK3_3) ) {
    case 0x00: ← case corresponding to
        /* Center -> straight */
        handle( 0 );
        speed( 100 ,100 );
        break; ← break from switch( sensor_inp(MASK3_3) ),
        control passes to 2
    case 0x04: ← case corresponding to
        /* Slight amount left of centre
        -> slight turn to right */
        handle( 5 );
        speed( 100 ,100 );
        break; ← break from switch( sensor_inp(MASK3_3) ),
        control passes to 2
    Line omitted
    default:
        break; ← default corresponding to
        switch( sensor_inp(MASK3_3) )
} 2
break; ← break from switch( sensor_inp(MASK3_3) ),
control passes to 2
    Line omitted
} 1
}

```

5.4.32. Pattern 12: Check End of Large Turn to Right

A sensor state of 0x03 indicates the largest amount of skew to left of centre. Therefore, any further move away from the centre could produce results like those shown in the figure below:

1		<p>This is the sensor state when a large amount left of centre. The sensor value is 0000 0011. When this state is encountered, the following lines of code are executed:</p> <pre>handle(25); motor(30, 19);</pre> <p>This causes a turn with the servo angle 25 degrees right, left motor 30%, and right motor 19%.</p>
2		<p>The leftward skew has increased. The sensor value is 1000 0001. No code is provided for execution in this state. Instead, the previous state is maintained. The previous state was a sensor value of 0000 0011, so the motor speed and servo angle settings for that sensor state are used.</p>
3		<p>The car has moved even further to the left. The sensor value is 1100 0000.</p>
4		<p>As shown in the figure at left, a sensor value is 1100 0000 is associated with a state in which the MCU car is right of centre. The assumption is that the steering wheel should turn to the left in this case. Therefore, the following lines of code are executed:</p> <pre>handle(-25); motor(19, 30);</pre> <p>This causes a turn with the servo angle 25 degrees right, left motor 19%, and right motor 30%.</p>

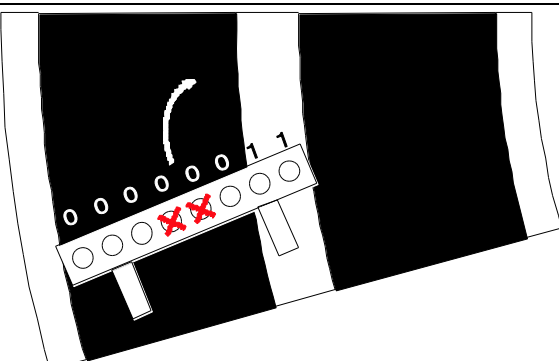
5		<p>In fact, the MCU car is far to the left of centre. Turning the steering wheel to the left will cause it to go off the track.</p>
---	---	---

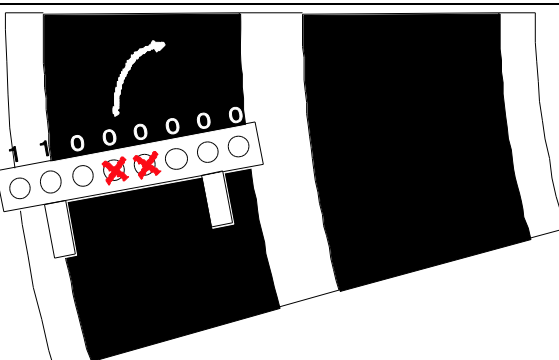
To prevent this, once a large turn to the right begins the software needs to continue turning the MCU car to the right until the sensors return to a certain state. Pattern 12 is designed to identify this “**certain state.**”

case 0x03 portion of pattern 11

```

178 :           case 0x03:
179 :             /* Large amount left of center -> large turn to right */
180 :             handle( 25 );
181 :             motor( 30 , 19 );
182 :             pattern = 12;    <-Added: Move to pattern 12.
183 :             break;
    
```

6		<p>When the sensor value is 0000 0011, control passes to pattern 12. Pattern 12 is designed to hand control back to pattern 11 once the sensor value is 0000 0110, one sensor closer to the centre. Let's see how this approach works.</p>
---	--	--

7		<p>The sensor value is now 1100 0000. Since it is not the expected value of 0000 0110, the MCU car continues to turn to the right. Previously, the program mistakenly assumed that the MCU car was too far to the right, but now that we have pattern 12 this problem does not arise.</p>
---	---	---

8		<p>The sensor value is now 0110 0000. It is still not 0000 0110, so the MCU car continues to turn to the right.</p>
---	--	---

9		<p>The sensor value is now 0000 0100. The MCU car has almost left the track, but the program continues to turn to the right because the sensor value is not 0000 0110. Nevertheless, once things get to this stage the MCU car may go off the track anyway.</p>
---	--	---

10		<p>The skew to the left has started to be reduced by a shift to the right, and now the sensor value is 0000 0110. In this state, control returns to pattern 11.</p>
----	--	--

This is the program code based on the above thinking:

```

case 12:
    /* Check end of large turn to right */
    if( sensor_inp(MASK3_3) == 0x06 ) {
        pattern = 11;
    }
    break;
    
```

But not so fast! Pattern 11 includes checks for crosslines, right half lines, and left half lines. Don't we need these in pattern 12 as well?

11		<p>The sensor value is now 0000 0011, and the crosslines are ahead. Go to pattern 12.</p>
----	--	--

12		<p>The sensor value is not 0000 0110, so continue turning to the right.</p>
----	--	--

13		<p>Crossline encountered, so we should switch to crank detection processing. But pattern 12 only checks whether or not the sensor value is 0000 0110. This means the MCU car continues on without detecting the crossline.</p> <p>As the above illustrates, it may be necessary to detect crosslines even when pattern 12 processing is taking place. The same goes for right half line and left half line checking. So we'll add these three types of checks to pattern 12 as well.</p>
----	--	---

This is the final program code:

```

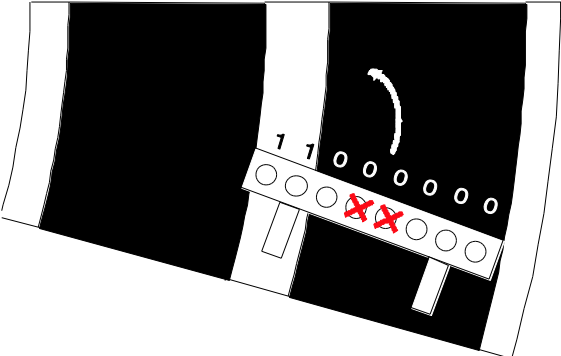
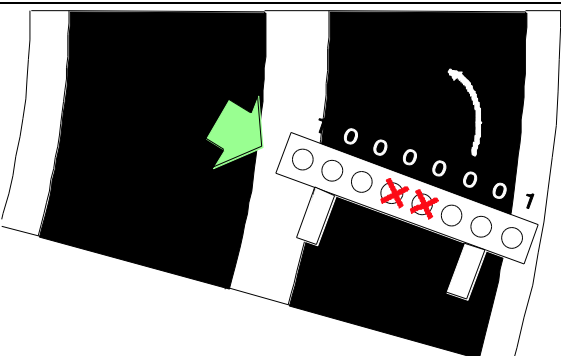
215 :         case 12:
216 :             /* Check end of large turn to right */
217 :             if( check_crossline() ) { /* Cross line check during large turn */
218 :                 pattern = 21;
219 :                 break;
220 :             }
221 :             if( check_rightline() ) { /* Right half line detection check */
222 :                 pattern = 51;
223 :                 break;
224 :             }
225 :             if( check_leftline() ) { /* Left half line detection check */
226 :                 pattern = 61;
227 :                 break;
228 :             }
229 :             if( sensor_inp(MASK3_3) == 0x06 ) {
230 :                 pattern = 11;
231 :             }
232 :             break;

```

This is the completed program code for pattern 12.

5.4.33. Pattern 13: Check End of Large Turn to Left

A sensor state of 0xC0 indicates the largest amount of skew to right of centre. Therefore, any further move away from the centre could produce results like those shown in the figure below:

1		<p>This is the sensor state when a large amount left of centre. The sensor value is 1100 0000. When this state is encountered, the following lines of code are executed:</p> <pre> handle(-25); motor(19, 30); </pre> <p>This causes a turn with the servo angle 25 degrees left, left motor 19%, and right motor 30%.</p>
2		<p>The rightward skew has increased. The sensor value is 1000 0001. No code is provided for execution in this state. Instead, the previous state is maintained. The previous state was a sensor value of 1100 0000, so the motor speed and servo angle settings for that sensor state are used.</p>

3		<p>The car has moved even further to the right. The sensor value is 0000 0011.</p>
---	--	---

4		<p>As shown in the figure at left, a sensor value is 0000 0011 is associated with a state in which the MCU car is left of centre. The assumption is that the steering wheel should turn to the right in this case. Therefore, the following lines of code are executed:</p> <pre> handle(25); motor(30, 19); </pre> <p>This causes a turn with the servo angle 25 degrees right, left motor 30%, and right motor 19%.</p>
---	--	--

5		<p>In fact, the MCU car is far to the right of centre. Turning the steering wheel to the right will cause it to go off the track.</p>
---	--	---

To prevent this, once a large turn to the left begins the software needs to continue turning the MCU car to the left until the sensors return to a certain state. Pattern 13 is designed to identify this “**certain state.**”

case 0xc0 portion of pattern 11

```

203 :             case 0xc0:
204 :                 /* Large amount right of center -> large turn to left */
205 :                 handle( -25 );
206 :                 motor( 19 , 30 );
207 :                 pattern = 13;    <- Added: Move to pattern 13.
208 :                 break;

```


6		<p>When the sensor value is 1100 0000, control passes to pattern 13. Pattern 13 is designed to hand control back to pattern 11 once the sensor value is 0110 0000, one sensor closer to the centre. Let's see how this approach works.</p>
---	--	--

7		<p>The sensor value is now 0000 0011. Since it is not the expected value of 0110 0000, the MCU car continues to turn to the left. Previously, the program mistakenly assumed that the MCU car was too far to the left, but now that we have pattern 13 this problem does not arise.</p>
---	--	---

8		<p>The sensor value is now 0000 0110. It is still not 0110 0000, so the MCU car continues to turn to the left.</p>
---	--	--

9		<p>The sensor value is now 0010 0000. The MCU car has almost left the track, but the program continues to turn to the left because the sensor value is not 0110 0000. Nevertheless, once things get to this stage the MCU car may go off the track anyway.</p>
---	--	--

10		<p>The skew to the right has started to be reduced by a shift to the left, and now the sensor value is 0110 0000. In this state, control returns to pattern 11.</p>
----	--	--

This is the program code based on the above thinking:

```

case 13:
  /* Check end of large turn to left */
  if( sensor_inp(MASK3_3) == 0x60 ) {
    pattern = 11;
  }
  break;
    
```

But not so fast! Pattern 11 includes checks for crosslines, right half lines, and left half lines. Don't we need these in pattern 13 as well?

11		<p>The sensor value is now 1100 0000, and the crosslines are ahead. Go to pattern 13.</p>
----	--	--

12		<p>The sensor value is not 0110 0000, so continue turning to the left.</p>
----	--	---

13		<p>Crossline encountered, so we should switch to crank detection processing. But pattern 13 only checks whether or not the sensor value is 01100000. This means the MCU car continues on without detecting the crossline.</p> <p>As the above illustrates, it may be necessary to detect crosslines even when pattern 13 processing is taking place. The same goes for right half line and left half line checking. So we'll add these three types of checks to pattern 13 as well.</p>
----	--	--

This is the final program code:

```

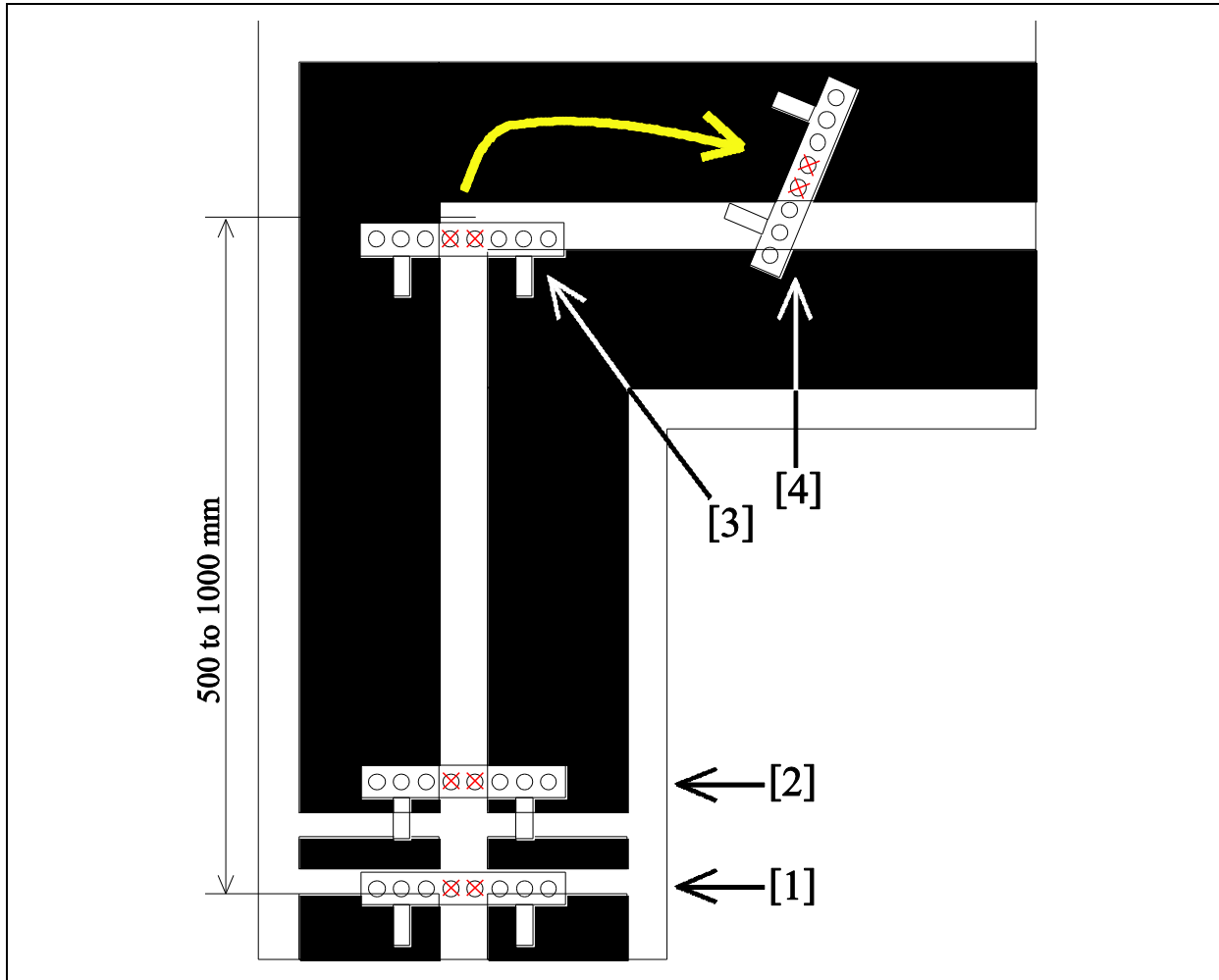
234 :         case 13:
235 :             /* Check end of large turn to left */
236 :             if( check_crossline() ) { /* Cross line check during large turn */
237 :                 pattern = 21;
238 :                 break;
239 :             }
240 :             if( check_rightline() ) { /* Right half line detection check */
241 :                 pattern = 51;
242 :                 break;
243 :             }
244 :             if( check_leftline() ) { /* Left half line detection check */
245 :                 pattern = 61;
246 :                 break;
247 :             }
248 :             if( sensor_inp(MASK3_3) == 0x60 ) {
249 :                 pattern = 11;
250 :             }
251 :             break;

```

This is the completed program code for pattern 13.

5.4.34. Crank Overview

Patterns 21 to 42 contain code related to “cranks” (right-angle turns). The figure below provides an overview of the processing:

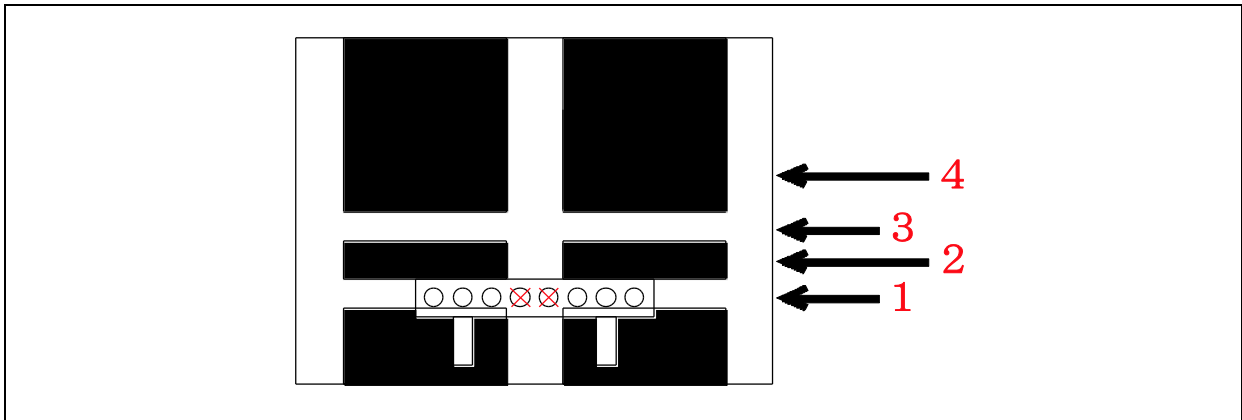


[1]	The check_crossline function detects the presence of crosslines. A crossline indicates that 500 mm to 1000 mm ahead is a right or left crank, so the MCU car must apply the brakes to reduce speed in order to navigate it successfully. In addition, the sensor data is not referenced until position [2] to ensure that the second crossline does not result in detection of erroneous sensor data.
[2]	This position is the start of the proceed slowly area. The MCU car advances straight ahead along the centre line.
[3]	When the crank is detected, the MCU car turns in the direction of the crank.
[4]	When the centre line is detected, control returns to pattern 11 and line tracing restarts.

In this way, the MCU car clears from crank. The specifics of the program code used are described below.

5.4.35. Pattern 21: Processing at 1st Crossline Detection

Control passes to pattern 21 the moment a crossline is detected. First, the MCU car passes over the crosslines. The characteristics of the portion of track from the position at which the first crossline is discovered to the position immediately after the second crossline are shown in the following figure:



- [1] First crossline
- [2] Normal track
- [3] Second crossline
- [4] Normal track, proceed slowly while tracing centre line

The track, other than the centre line, changes from white to black to white to black again by the time position [4] is reached. The program must detect these changes and respond appropriately. That sounds pretty complicated.

Let's look at this in a different way. The distance from position [1] to position [4] is about 100 mm, allowing some margin for error. (The precise distance is 70 mm: 20 mm for the first crossline + 30 mm of black area + 20 mm for the second crossline = 70 mm.) If the MCU car is positioned roughly over the centre line and continues to move forward for about 100 mm while we ignore the sensor data, we'll probably come out roughly on course. The kit car includes no mechanism for detecting distances, but we can use the timer to interrupt reading of sensor data for a specified duration. We don't know how long a duration yet because that will depend on how fast the MCU car is travelling. For the time being, let's use a pause duration of 0.1 seconds and do fine tuning later. In addition, we'll make the LEDs on the motor drive board light to indicate externally that processing of pattern 21 has started.

To summarize:

- Illuminate LED2 and LED3.
- Set steering angle to 0 degrees.
- Set PWM value of right and left motors to 0% to initiate brake operation.
- Wait 0.1 seconds.
- After 0.1 seconds elapse, go to next pattern.

This is what the program code of pattern 21 must accomplish.

```

case 21:
    /* Processing at 1st cross line */
    led_out( 0x3 );
    handle( 0 );
    speed( 0 , 0 );
    if( cnt1 > 100 ) {
        pattern = 22; /* After 0.1 seconds, to pattern 22 */
    }
    break;

```

This is the completed program code. Let's take a moment to review it. When the value of **cnt1** is 100 or greater (after 100 milliseconds have elapsed), control passes to pattern 22. For this to work as expected, the value of **cnt1** must be 0 when pattern 21 starts. For example, if the value of **cnt1** is 1000 when control passes to pattern 21, the value of **cnt1** will be judged to be 100 or greater the first time the condition is tested, and control will pass immediately to pattern 22. Execution of pattern 21 takes place only once (a duration of a few dozen μ s) rather than lasting for 0.1 seconds. We need to add another pattern. Pattern 21 will start brake operation and clear **cnt1** to 0, and pattern 22 will check whether 0.1 seconds have elapsed.

To summarize once again:

Tasks performed by pattern 21:	<ul style="list-style-type: none"> • Illuminate LED2 and LED3. • Set steering angle to 0 degrees. • Set PWM value of right and left motors to 0% to initiate brake operation. • Go to next pattern. • Clear cnt1.
Tasks performed by pattern 22:	<ul style="list-style-type: none"> • If value of cnt1 is 100 or greater, go to next pattern.

Let's rewrite the program to reflect the above changes.

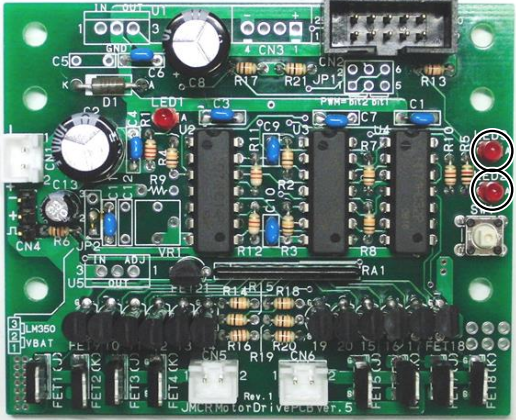
```

253 :         case 21:
254 :             /* Processing at 1st cross line */
255 :             led_out( 0x3 );
256 :             handle( 0 );
257 :             motor( 0 , 0 );
258 :             pattern = 22;
259 :             cnt1 = 0;
260 :             break;
261 :
262 :         case 22:
263 :             /* Read but ignore 2nd line */
264 :             if( cnt1 > 100 ){
265 :                 pattern = 23;
266 :                 cnt1 = 0;
267 :             }
268 :             break;

```

The portion of the program code from detection of the crossline to the start of the trace centre line area is now complete.

Hint



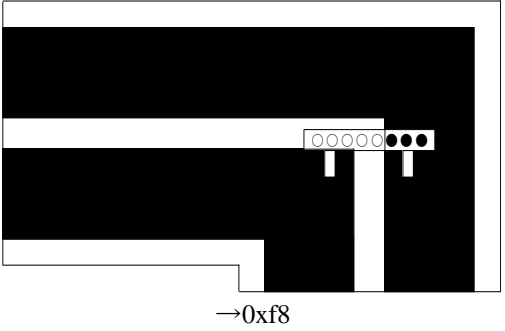
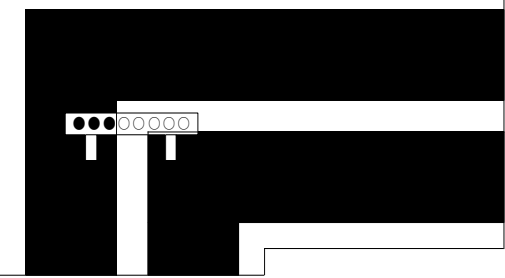
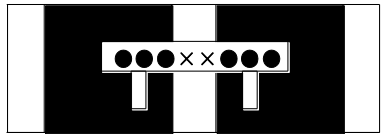
When a crossline is detected, the two LEDs on the motor drive board light. No crossline has been detected if they do not light. If crossline detection is not working properly, try unplugging the motor connectors and pushing the MCU car forward by hand.

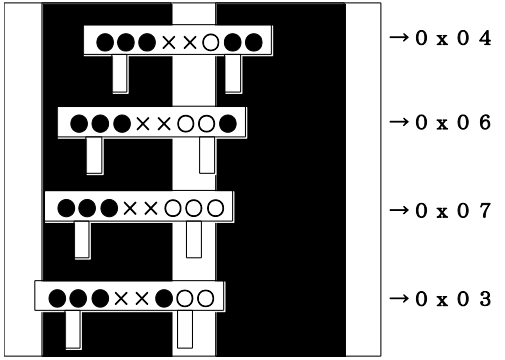
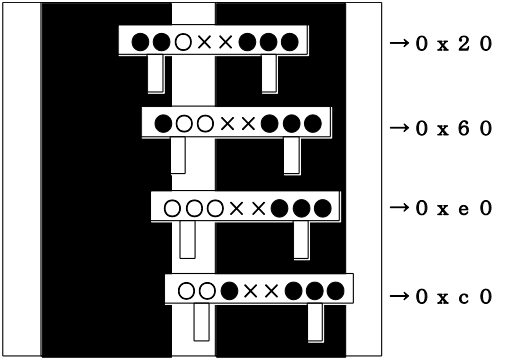
5.4.36. Pattern 23: Trace, Crank Detection After Crossline

Patterns 21 and 22 perform brake operation for 0.1 seconds after detection of the first crossline, allowing the MCU car to pass the second crossline. Pattern 23 continues the processing after this.

The MCU car is past the crosslines, so the next task is detecting the crank (right-angle turn). The MCU car must turn immediately as soon as the crank is encountered, so it is proceeding at low speed. In addition, the MCU car must continue to trace the centre line up to the crank.

We envision the present situation as shown in the following figures:

 <p style="text-align: center;">→0xf8</p> <p style="text-align: center;">(Checking with all 8 sensors)</p>	<p>At a left crank, the state of the eight sensors is 0xf8, as shown in the figure at left. The software judges a sensor state of 0xf8 as indicating a left crank.</p> <p>At this point, the MCU car will drift toward the edge and go off the track if the steering wheel is not turned all the way to the left. How many degrees of turn is this? The actual value depends on the physical characteristics of the individual MCU car, so it is necessary to confirm how far the steering wheel can turn by looking at the actual car. We will use a value of about 38 degrees.</p> <p>To accomplish a sharp left turn, we will use a left motor speed that is lower and a right motor speed that is higher. As for the actual percentages, we can't say for sure until we try it out. For the time being, we'll use settings of 10% for the left motor and 50% for the right motor. The settings can be summarized as follows:</p> <p>Steering angle: -38 degrees Left motor: 10%, Right motor: 50%</p> <p>Afterward, go to pattern 31.</p>
 <p style="text-align: center;">→0x1f</p> <p style="text-align: center;">(Checking with all 8 sensors)</p>	<p>This is a right crank. The basic approach is the same as for a left crank. The settings can be summarized as follows:</p> <p>Steering angle: 38 degrees Left motor: 50%, Right motor: 10%</p> <p>Afterward, go to pattern 41.</p>
 <p style="text-align: right;">→ 0 x 0 0</p>	<p>When proceeding straight ahead, the sensor state is 0x00. The software judges this as meaning that the MCU car is positioned over the centre line. The steering angle is 0 degrees. The problem is the PWM values of the motors. The motor PWM values must be such that the MCU car can negotiate a 90-degree turn when the crank is encountered. For the time being, we'll use a setting 40% for both motors. This can be fine tuned later when doing test runs. The settings can be summarized as follows:</p> <p>Steering angle: 0 degrees Left motor: 40%, Right motor: 40%</p>

	<p>Let's assume the MCU car has drifted left of centre.</p> <p>As little by little the MCU car drifts farther left of centre, the four resulting sensor states are as shown in the figure at left. Additional sensor states are possible when the MCU car is even farther left of centre, but we will not add more sensor states because we know that the section of track following the crosslines will be straight and further movement to the left is not likely.</p> <p>Since the MCU car is left of centre, it is necessary to turn the steering wheel to the right. If the amount of turn is too small and the amount of drift is large, the MCU car will be unable to return to the centre. If the amount of turn is too large, it will overshoot the centre and the car will end up zigzagging right and left. Fine adjusting the angle to precisely the right value is difficult. For the time being, we will use a setting of 8 degrees. We'll make the right motor PWM value lower than that of the left motor since we're turning to the right. The settings can be summarized as follows:</p> <p>Steering angle: 8 degrees Left motor: 40%, Right motor: 35%</p>
	<p>Let's assume the MCU car has drifted right of centre.</p> <p>As little by little the MCU car drifts farther right of centre, the four resulting sensor states are as shown in the figure at left. Additional sensor states are possible when the MCU car is even farther right of centre, but we will not add more sensor states because we know that the section of track following the crosslines will be straight and further movement to the right is not likely.</p> <p>Since the MCU car is right of centre, it is necessary to turn the steering wheel to the left. If the amount of turn is too small and the amount of drift is large, the MCU car will be unable to return to the centre. If the amount of turn is too large, it will overshoot the centre and the car will end up zigzagging right and left. Fine adjusting the angle to precisely the right value is difficult. For the time being, we will use a setting of -8 degrees. We'll make the left motor PWM value lower than that of the right motor since we're turning to the left. The settings can be summarized as follows:</p> <p>Steering angle: -8 degrees Left motor: 35%, Right motor: 40%</p>

It is important to remember that the all eight sensors are used for crank checking. For other checking, MASK3_3 masking is applied and the two middle sensors are not used.

The finished program code is as follows:

```

270 :         case 23:
271 :             /* Trace, crank detection after cross line */
272 :             if( sensor_inp(MASK4_4)==0xf8 ) {
273 :                 /* Left crank determined -> to left crank clearing processing */
274 :                 led_out( 0x1 );
275 :                 handle( -38 );
276 :                 motor( 10 , 50 );
277 :                 pattern = 31;
278 :                 cnt1 = 0;
279 :                 break;
280 :             }
281 :             if( sensor_inp(MASK4_4)==0x1f ) {
282 :                 /* Right crank determined -> to right crank clearing processing */
283 :                 led_out( 0x2 );
284 :                 handle( 38 );
285 :                 motor( 50 , 10 );
286 :                 pattern = 41;
287 :                 cnt1 = 0;
288 :                 break;
289 :             }
290 :             switch( sensor_inp(MASK3_3) ) {
291 :                 case 0x00:
292 :                     /* Center -> straight */
293 :                     handle( 0 );
294 :                     motor( 40 , 40 );
295 :                     break;
296 :                 case 0x04:
297 :                 case 0x06:
298 :                 case 0x07:
299 :                 case 0x03:
300 :                     /* Left of center -> turn to right */
301 :                     handle( 8 );
302 :                     motor( 40 , 35 );
303 :                     break;
304 :                 case 0x20:
305 :                 case 0x60:
306 :                 case 0xe0:
307 :                 case 0xc0:
308 :                     /* Right of center -> turn to left */
309 :                     handle( -8 );
310 :                     motor( 35 , 40 );
311 :                     break;
312 :             }
313 :             break;

```

The meaning of these consecutive case statements is: “when 0x04 or 0x06 or 0x07 or 0x03 is the case.”

The meaning of these consecutive case statements is: “when 0x20 or 0x60 or 0xe0 or 0xc0 is the case.”

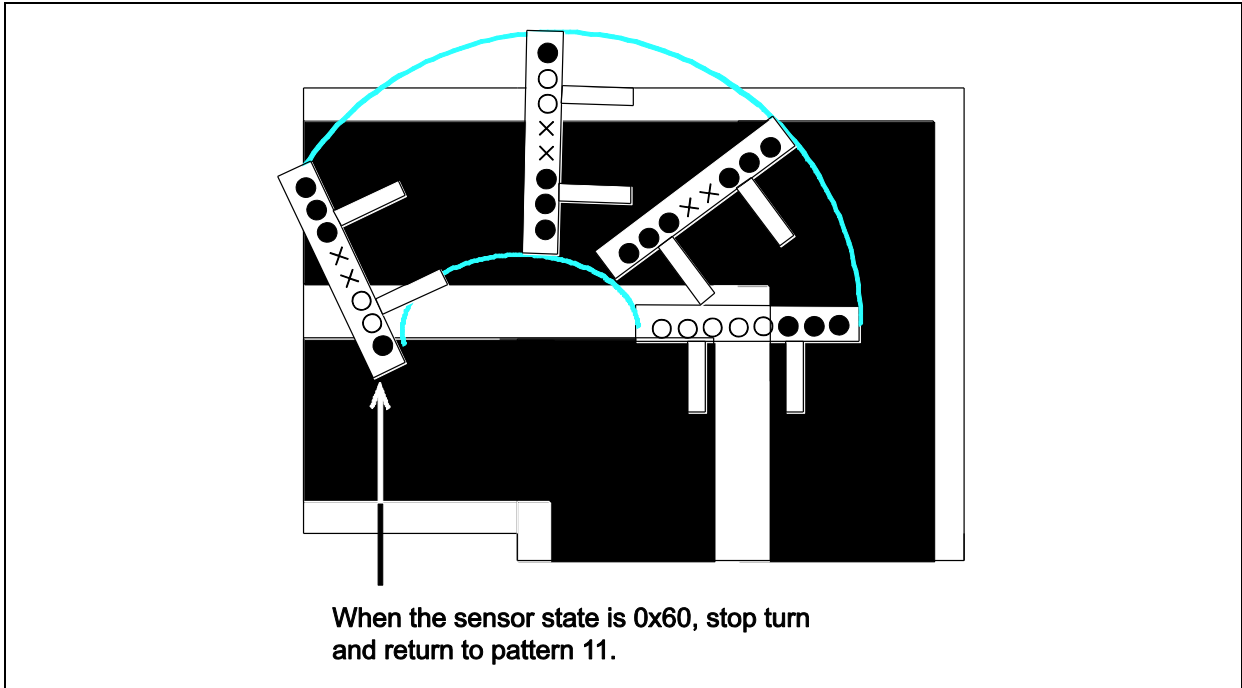
The program uses **if** statements to distinguish between left and right cranks. Also, a **switch** statement is used to branch to the appropriate **case** according to the value of **sensor_inp(MASK3_3)**.

5.4.37. Patterns 31 and 32: Clearing from Left Crank

Pattern 23 judges a value of 0xf8 from all eight sensors to indicate a left crank and starts a large turn to the left in order to clear from the crank.

The following issue arises: How long should the large left turn continue? This portion of the program is dedicated to patterns 31 and 32.

We envision the situation as shown in the following figures:



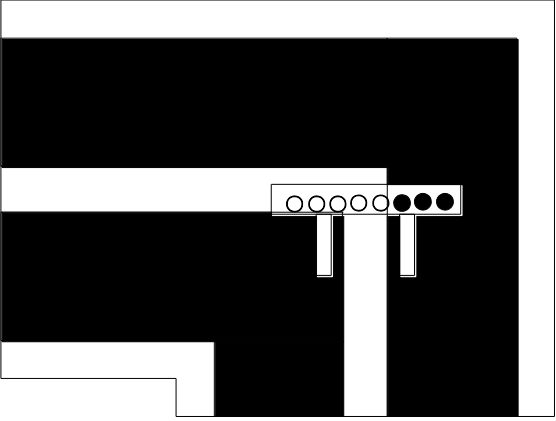
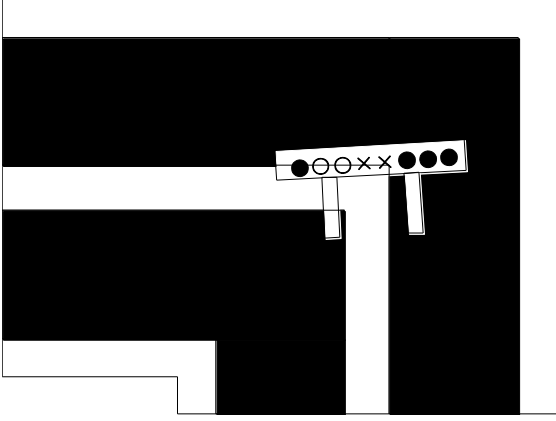
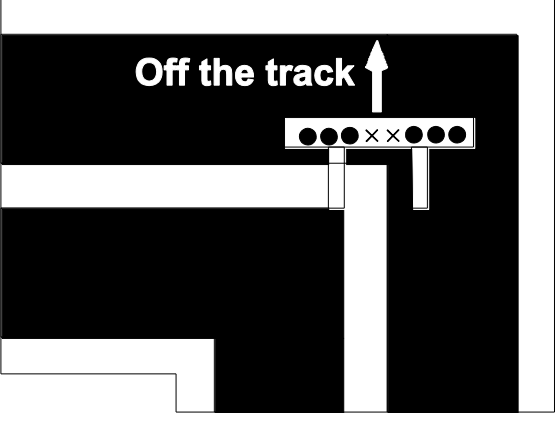
The software executes a large turn to left when a sensor state of 0xf8 occurs, but due to the speed at which the MCU car is travelling it turns gradually rather than sharply. When the sensor value is 0x60, indicating that the car is back near the centre line, the turn is judged to be finished and control returns to pattern 11.

This can be coded as follows:

```
case 31:
    if( sensor_inp(MASK3_3) == 0x60 ) {
        pattern = 11;
    }
    break;
```

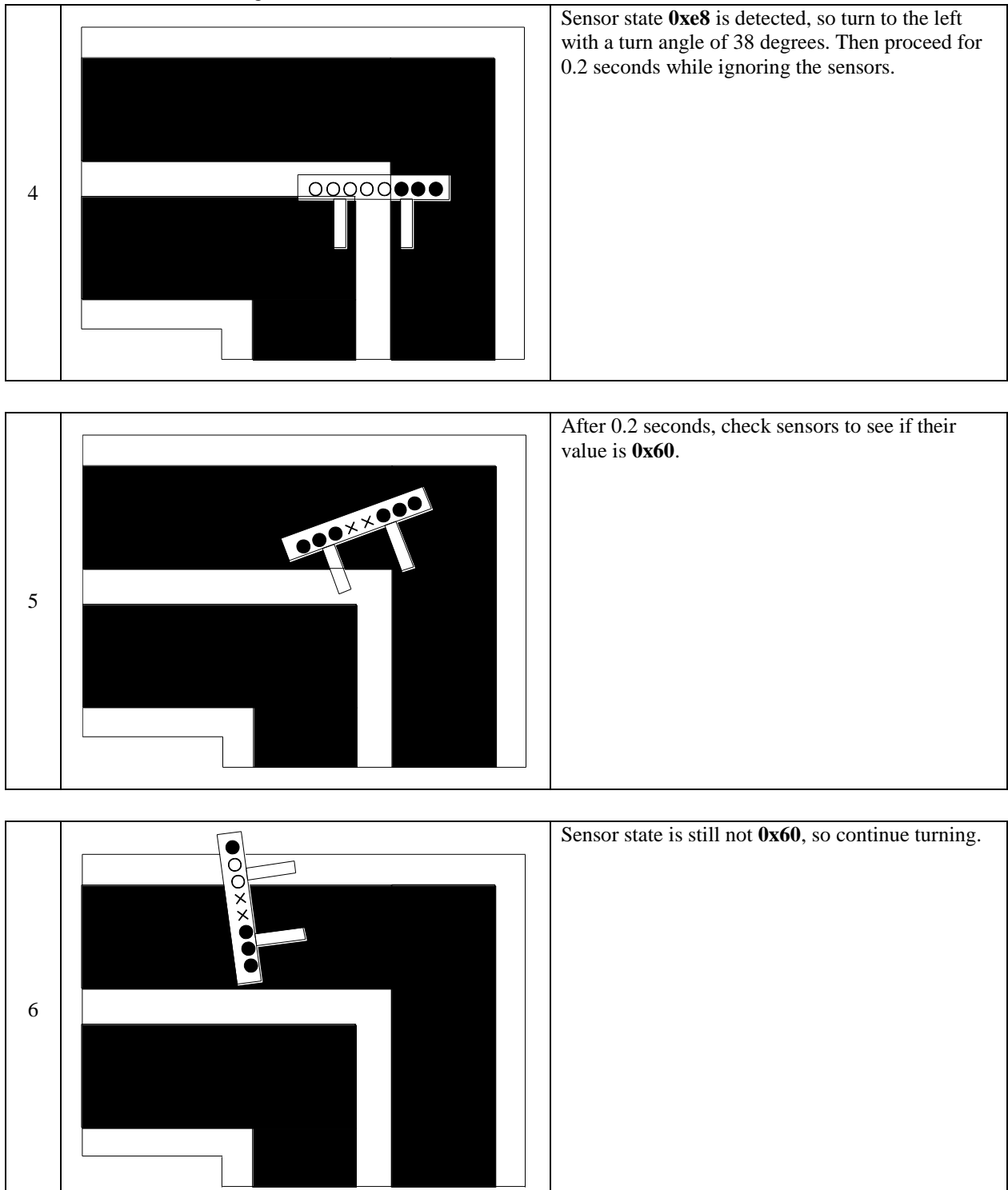
We tried actually running the MCU car using this program code. What happened was that the moment the sensors detected the left crank and the sensor state was **0xf8** the steering wheel started turning to the left. We anticipated that the turn would continue until the sensor state was **0x60**, but instead of continuing to turn, it immediately straightened out again and the MCU car ran straight through the crank and off the track. Since the MCU car moves too fast for us to see exactly what happened, we tried disconnecting the motors and servo and then slowly moving the car by hand. Careful observation reveals the sensor states to be as shown in the figures that follow.

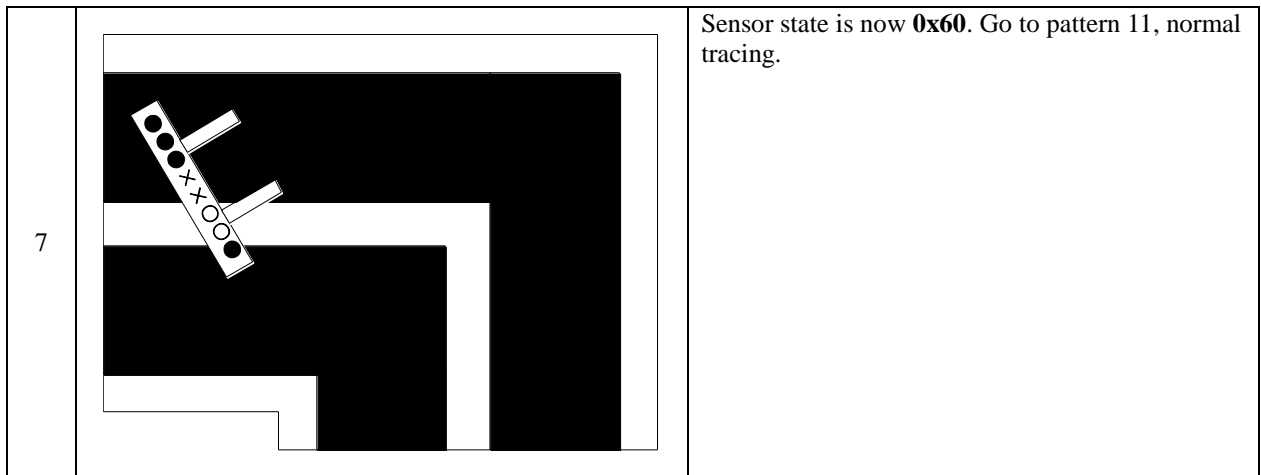
Note
When you are unsure exactly how the MCU car is moving,
we recommend disconnecting the motors and pushing it by hand to check.

1		<p>Sensor state 0xf8 is detected, so turn to the left with a turn angle of 38 degrees until the sensor state is 0x60.</p>
2		<p>At the moment the turn starts, the sensor state is 0x60 at the place where the white line changes to black. (Actually, there are white, grey, and black areas, but we will consider the grey areas to be white.) When the software recognizes the sensor state of 0x60 it passes control to pattern 11.</p>
3		<p>Pattern 11 interprets the sensor state of 0x00 to mean that the MCU car is centred over the centre line and proceeds straight ahead at turn angle 0 and motor speed 100%. The MCU car runs straight off the track.</p>

When we checked the sensors we discovered that the leftmost sensor was not optimally adjusted, and it erroneously gave a reading of 0 before the second and third sensors from the left. This caused an incorrect value of 0x60 at the change from white to black. This could be corrected by adjusting the leftmost sensor to increase its sensitivity a small amount. But we don't want the MCU car to run off the track just because of a small difference in sensor sensitivity, so we'll modify the program to correct the problem.

Come to think of it, a certain amount of time has to pass between when **0xf8** is detected and when the final sensor state of **0x60** occurs at the detection of the centre line. We can rewrite the code so that after the left crank is encountered, the motor settings are made and the sensors are ignored for 0.2 seconds. Once the portion of the track where the colour changes has been passed over, we can reactivate checking of the sensors 0.2 seconds later. We can illustrate this idea with figures as follows:





As figure 5 shows, after 0.2 seconds the sensors are past the place where the white line changes to black. After this, the MCU car can safely continue turning until the sensor state is **0x60**. This should do it. It can be coded as follows:

```

315 :         case 31:
316 :             /* Left crank clearing processing ? wait until stable */
317 :             if( cnt1 > 200 ) {
318 :                 pattern = 32;
319 :                 cnt1 = 0;
320 :             }
321 :             break;
322 :
323 :         case 32:
324 :             /* Left crank clearing processing ? check end of turn */
325 :             if( sensor_inp(MASK3_3) == 0x60 ) {
326 :                 led_out( 0x0 );
327 :                 pattern = 11;
328 :                 cnt1 = 0;
329 :             }
330 :             break;

```

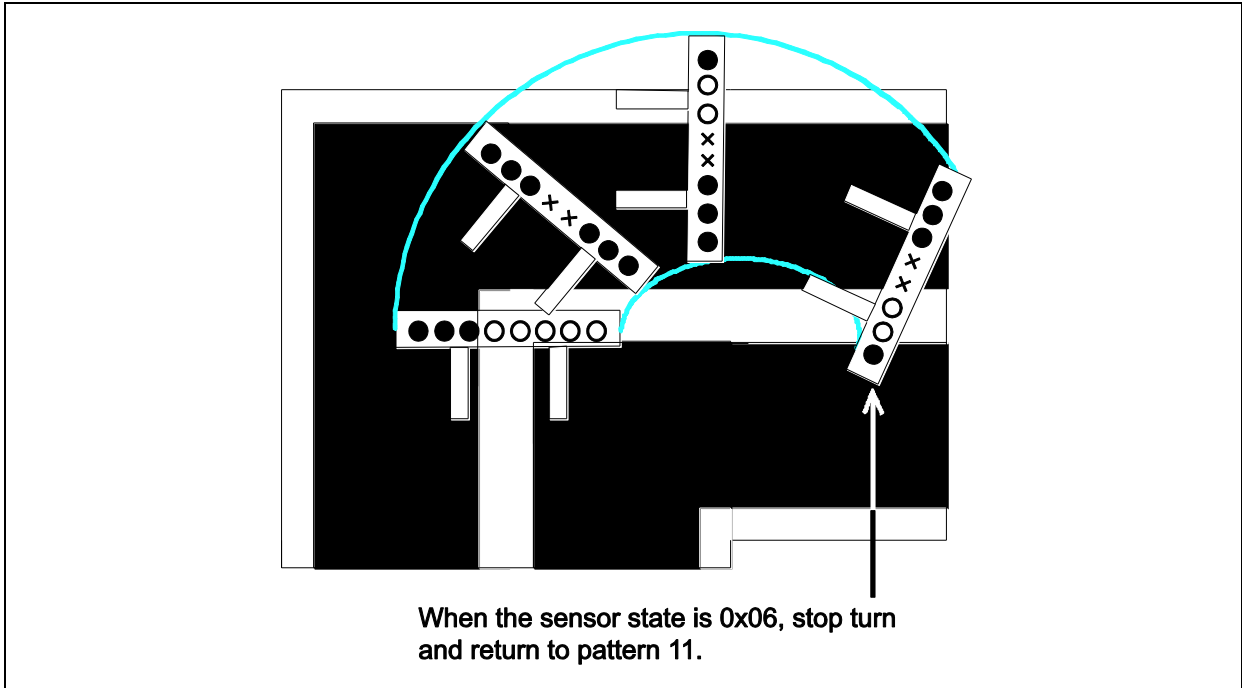
Line 311 checks if the value of **cnt1** is 200 or greater. If it is, that means 0.2 seconds have elapsed, so control passes to pattern 32. Incidentally, **cnt1** is cleared to 0 in line 272 before the jump to pattern 31.

5.4.38. Patterns 41 and 42: Right Crank Clearing Processing

Pattern 23 judges a value of 0x1F from all eight sensors to indicate a right crank and starts a large turn to the right in order to clear from the crank.

The following issue arises: How long should the large right turn continue? This portion of the program is dedicated to patterns 41 and 42.

We envision the situation as shown in the following figures:



The software executes a large turn to right when a sensor state of 0x1f occurs, but due to the speed at which the MCU car is travelling it turns gradually rather than sharply. When the sensor value is 0x06, indicating that the car is back near the centre line, the turn is judged to be finished and control returns to pattern 11.

This can be coded as follows:

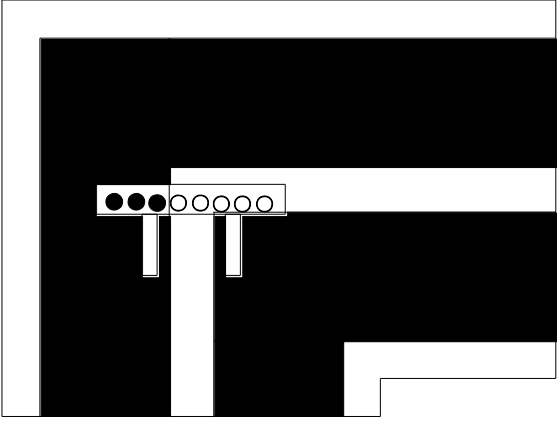
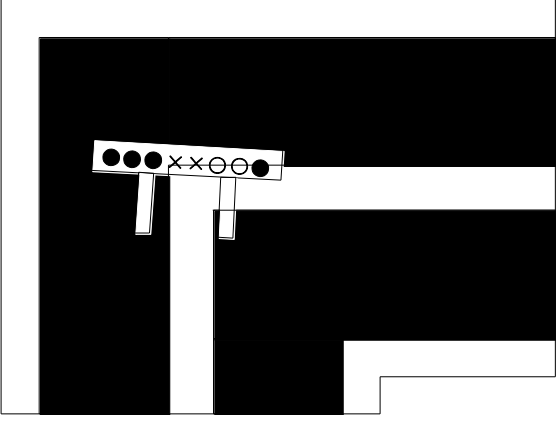
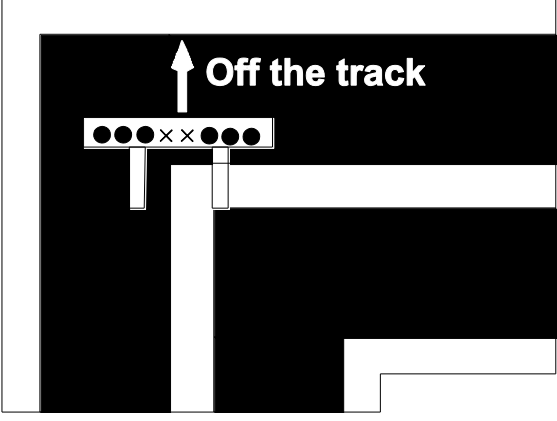
```

case 41:
    if( sensor_inp(MASK3_3) == 0x06 ) {
        pattern = 11;
    }
    break;
    
```

We tried actually running the MCU car using this program code. What happened was that the moment the sensors detected the right crank and the sensor state was 0x1f the steering wheel started turning to the right. We anticipated that the turn would continue until the sensor state was 0x06, but instead of continuing to turn, it immediately straightened out again and the MCU car ran straight through the crank and off the track. Since the MCU car moves too fast for us to see exactly what happened, we tried disconnecting the motors and servo and then slowly moving the car by hand. Careful observation reveals the sensor states to be as shown in the figures that follow.

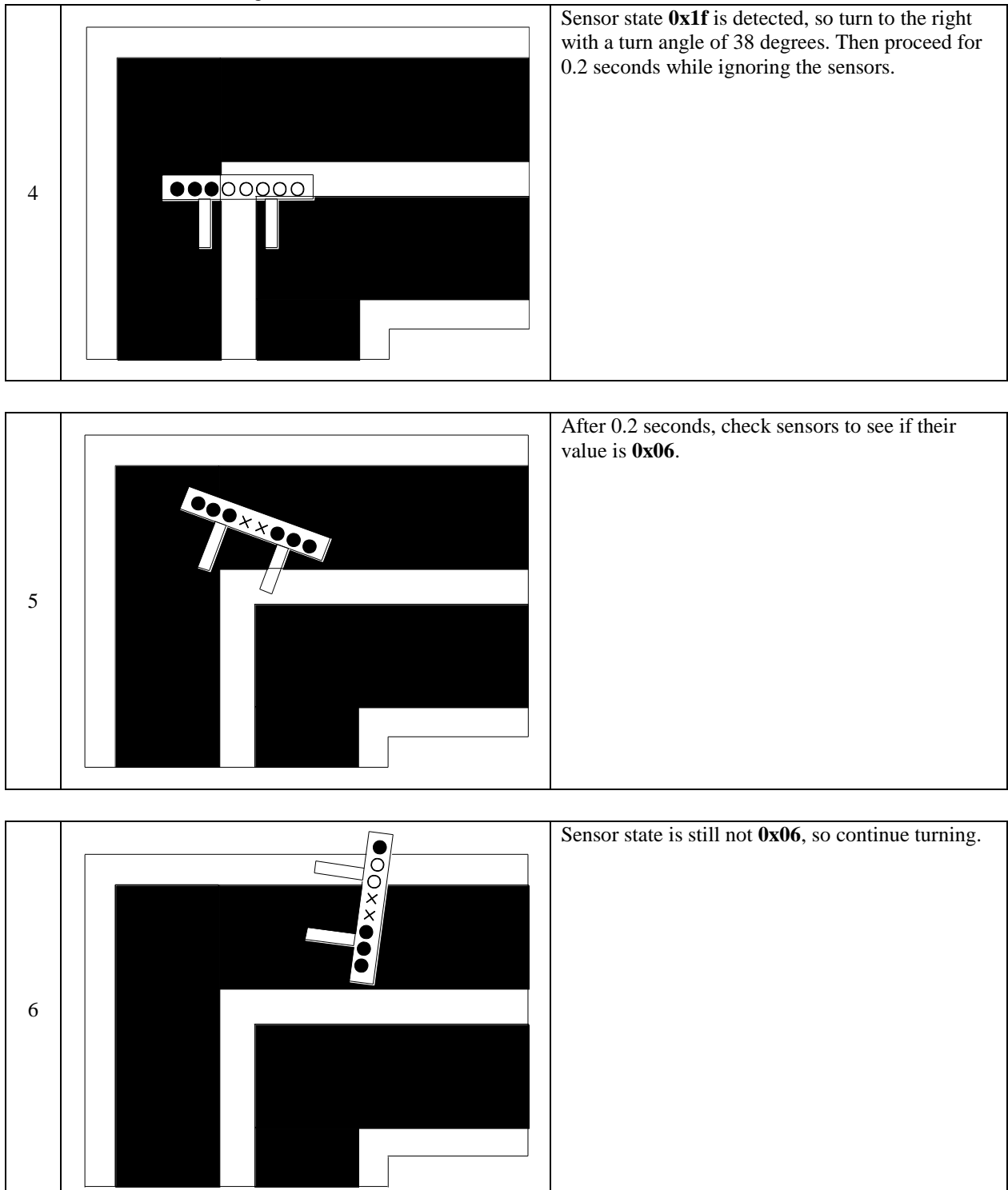
Note

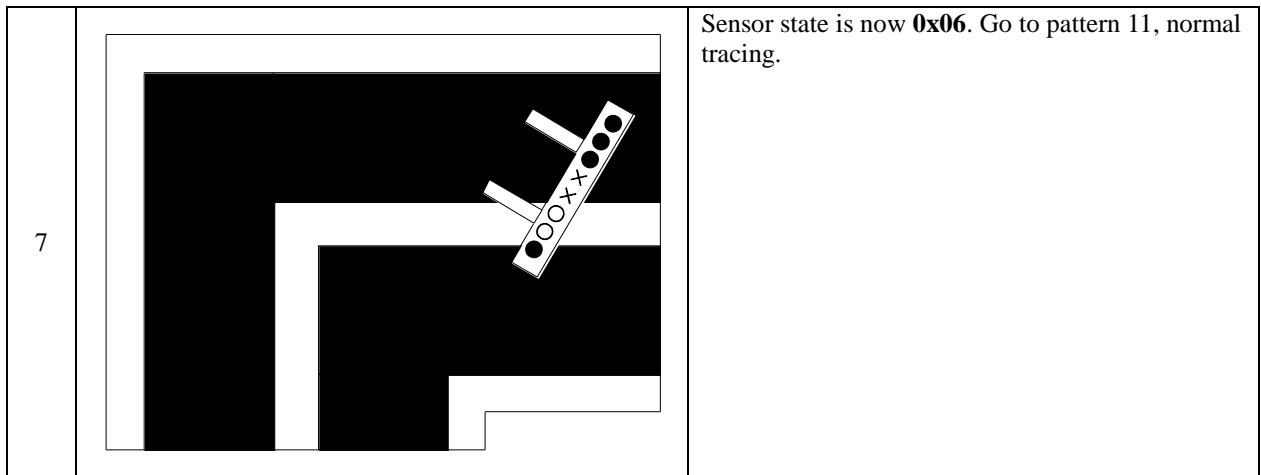
When you are unsure exactly how the MCU car is moving,
we recommend disconnecting the motors and pushing it by hand to check.

1		<p>Sensor state 0x1f is detected, so turn to the left with a turn angle of 38 degrees until the sensor state is 0x06.</p>
2		<p>At the moment the turn starts, the sensor state is 0x06 at the place where the white line changes to black. (Actually, there are white, grey, and black areas, but we will consider the grey areas to be white.) When the software recognizes the sensor state of 0x06 it passes control to pattern 11.</p>
3		<p>Pattern 11 interprets the sensor state of 0x00 to mean that the MCU car is centred over the centre line and proceeds straight ahead at turn angle 0 and motor speed 100%. The MCU car runs straight off the track.</p>

When we checked the sensors we discovered that the rightmost sensor was not optimally adjusted, and it erroneously gave a reading of 0 before the second and third sensors from the right. This caused an incorrect value of 0x06 at the change from white to black. This could be corrected by adjusting the rightmost sensor to increase its sensitivity a small amount. But we don't want the MCU car to run off the track just because of a small difference in sensor sensitivity, so we'll modify the program to correct the problem.

Come to think of it, a certain amount of time has to pass between when **0x1f** is detected and when the final sensor state of **0x06** occurs at the detection of the centre line. We can rewrite the code so that after the right crank is encountered, the motor settings are made and the sensors are ignored for 0.2 seconds. Once the portion of the track where the colour changes has been passed over, we can reactivate checking of the sensors 0.2 seconds later. We can illustrate this idea with figures as follows:





As figure 5 shows, after 0.2 seconds the sensors are past the place where the white line changes to black. After this, the MCU car can safely continue turning until the sensor state is **0x06**. This should do it. It can be coded as follows:

```

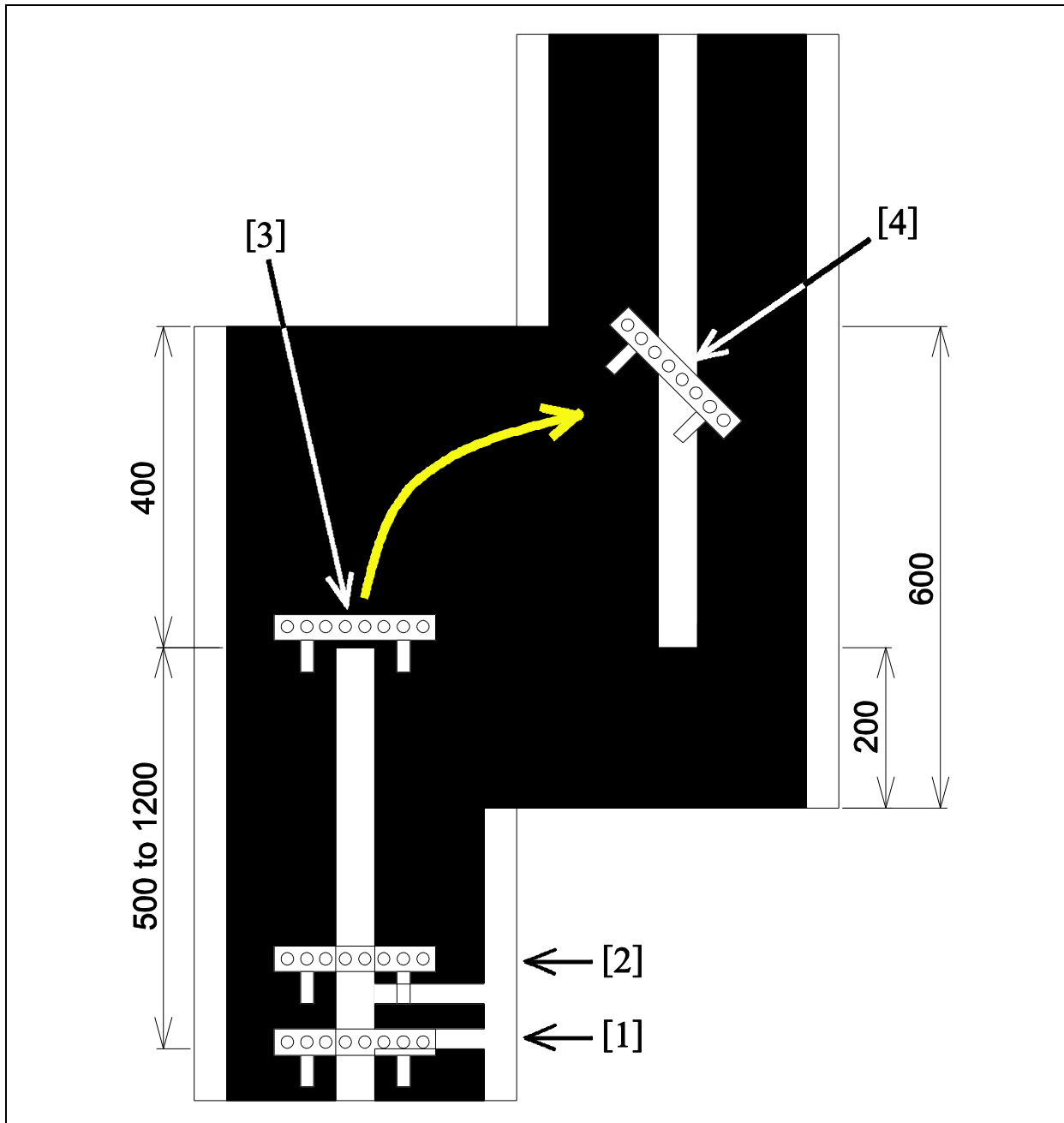
332 :         case 41:
333 :             /* Right crank clearing processing ? wait until stable */
334 :             if( cnt1 > 200 ) {
335 :                 pattern = 42;
336 :                 cnt1 = 0;
337 :             }
338 :             break;
339 :
340 :         case 42:
341 :             /* Right crank clearing processing ? check end of turn */
342 :             if( sensor_inp(MASK3_3) == 0x06 ) {
343 :                 led_out( 0x0 );
344 :                 pattern = 11;
345 :                 cnt1 = 0;
346 :             }
347 :             break;

```

Line 328 checks if the value of **cnt1** is 200 or greater. If it is, that means 0.2 seconds have elapsed, so control passes to pattern 42. Incidentally, **cnt1** is cleared to 0 in line 281 before the jump to pattern 41.

5.4.39. Right Lane Change Outline

Patterns 51 to 54 contain program code related to executing a right lane change. An outline of the processing involved is provided below:

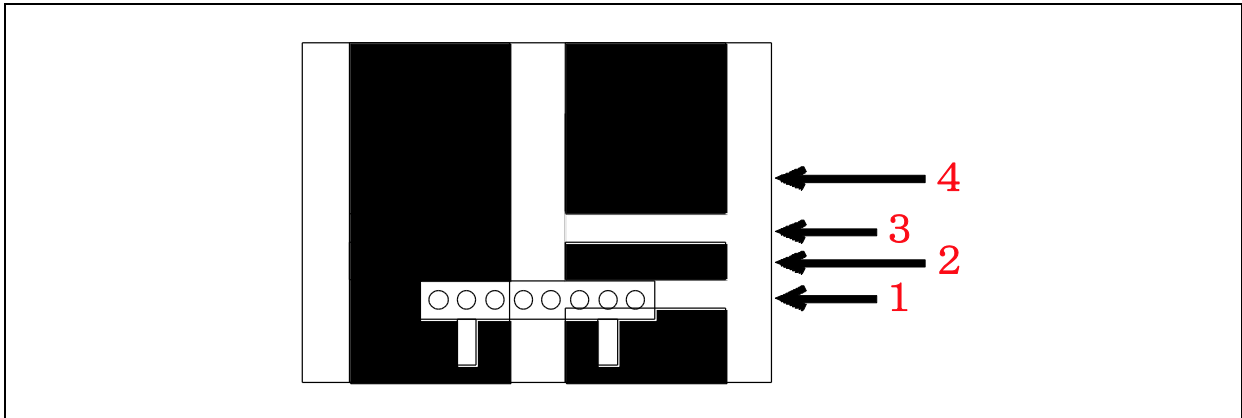


[1]	The check_rightline function detects a right half line. The MCU car must change to the right lane 500 to 1200 mm ahead, so brake operation is performed. Also, sensor input is ignored up to position [2] to prevent erroneous sensor detection at the second right half line.
[2]	The MCU car starts to proceed slowly from this point. It advances while tracing the centre line.
[3]	When the centre line ends, the steering wheel turns to the right.
[4]	When a new centre line is detected, line tracing restarts using the new centre line.

In this way, the MCU car manoeuvres right lane change. The specifics of the program code used are described below.

5.4.40. Pattern 51: Processing at 1st Right Half Line Detection

Control passes to pattern 51 the moment a right half line is detected. First, the MCU car passes over the right half lines. The characteristics of the portion of track from the position at which the first right half line is discovered to the position immediately after the second right half line are shown in the following figure:



- [1] First crossline
- [2] Normal track
- [3] Second crossline
- [4] Normal track, proceed slowly while tracing centre line

The track, other than the centre line, changes from white to black to white to black again by the time position [4] is reached. The program must detect these changes and respond appropriately. That sounds pretty complicated.

Let's look at this in a different way. The distance from position [1] to position [4] is about 100 mm, allowing some margin for error. (The precise distance is 70 mm: 20 mm for the first half line + 30 mm of black area + 20 mm for the second half line = 70 mm.) If the MCU car is positioned roughly over the centre line and continues to move forward for about 100 mm while we ignore the sensor data, we'll probably come out roughly on course. The kit car includes no mechanism for detecting distances, but we can use the timer to interrupt reading of sensor data for a specified duration. We don't know how long a duration yet because that will depend on how fast the MCU car is travelling. For the time being, let's use a pause duration of 0.1 seconds and do fine tuning later. In addition, we'll make the LEDs on the motor drive board light to indicate externally that processing of pattern 51 has started.

To summarize:

- Illuminate LED2. (This differs from the processing of crossline to make it possible to tell them apart.)
- Set steering angle to 0 degrees.
- Set PWM value of right and left motors to 0% to initiate brake operation.
- Wait 0.1 seconds.
- After 0.1 seconds elapse, go to next pattern.

This is what the program code of pattern 51 must accomplish.

```

case 51:
    /* Processing at 1st right half line detection */
    led_out( 0x2 );
    handle( 0 );
    speed( 0 , 0 );
    if( cnt1 > 100 ) {
        pattern = 52; /* After 0.1 seconds, to pattern 52 */
    }
    break;

```

This is the completed program code. Let's take a moment to review it. When the value of **cnt1** is 100 or greater (after 100 milliseconds have elapsed), control passes to pattern 52. For this to work as expected, the value of **cnt1** must be 0 when pattern 51 starts. For example, if the value of **cnt1** is 1000 when control passes to pattern 51, the value of **cnt1** will be judged to be 100 or greater the first time the condition is tested, and control will pass immediately to pattern 52. Execution of pattern 51 takes place only once (a duration of a few dozen μ s) rather than lasting for 0.1 seconds. We need to add another pattern. Pattern 51 will start brake operation and clear **cnt1** to 0, and pattern 52 will check whether 0.1 seconds have elapsed.

To summarize once again:

Tasks performed by pattern 51:	<ul style="list-style-type: none"> • Illuminate LED2. • Set steering angle to 0 degrees. • Set PWM value of right and left motors to 0% to initiate brake operation. • Go to next pattern. • Clear cnt1.
Tasks performed by pattern 52:	<ul style="list-style-type: none"> • If value of cnt1 is 100 or greater, go to next pattern.

Let's rewrite the program to reflect the above changes.

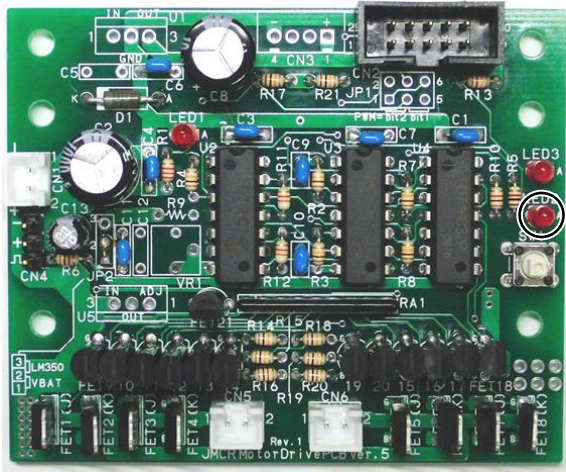
```

349 :         case 51:
350 :             /* Processing at 1st right half line detection */
351 :             led_out( 0x2 );
352 :             handle( 0 );
353 :             motor( 0 , 0 );
354 :             pattern = 52;
355 :             cnt1 = 0;
356 :             break;
357 :
358 :         case 52:
359 :             /* Read but ignore 2nd time */
360 :             if( cnt1 > 100 ){
361 :                 pattern = 53;
362 :                 cnt1 = 0;
363 :             }
364 :             break;

```

The portion of the program code from detection of the right half line to the start of the trace centre line area is now complete.

Hint



When a right half line is detected, one LED (the bottom one) on the motor drive board lights. No right half line has been detected if it does not light.

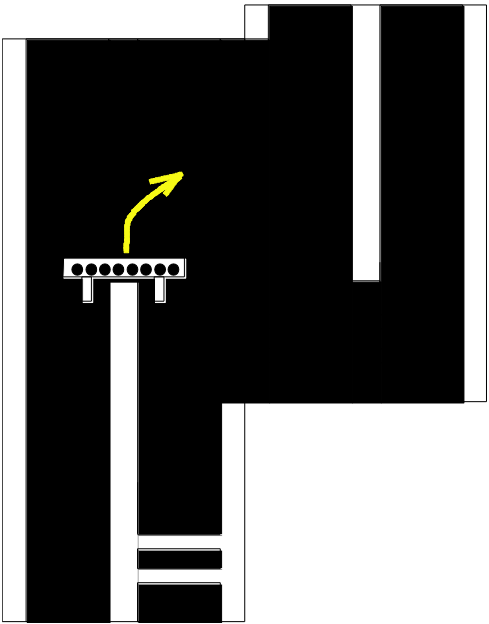
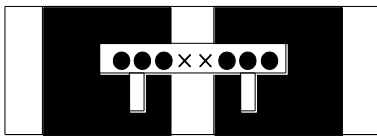
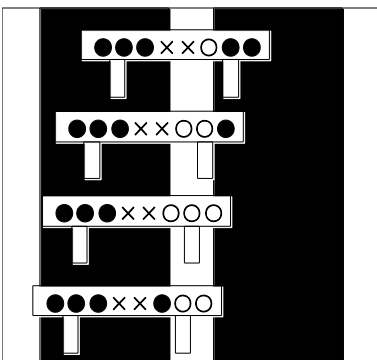
If right half line detection is not working properly, try unplugging the motor connectors and pushing the MCU car forward by hand.

5.4.41. Pattern 53: Trace after Right Half Line

Patterns 51 and 52 perform brake operation for 0.1 seconds after detection of the first right half line, allowing the MCU car to pass the second right half line. Pattern 53 continues the processing after this.

The MCU car is past the right half lines, so the next task is detecting the end of the centre line. In addition, the MCU car must continue to trace the straight section of track that leads to the end of the centre line, so normal trace operation is necessary.

We envision the present situation as shown in the following figures:

 <p style="text-align: center;">→0x00 (Checking with all 8 sensors)</p>	<p>After the right half line ends, the state of the eight sensors is 0x00, as shown in the figure at left. When this state is detected, the MCU car starts turning to the right. For a right turn, we would expect that the right motor speed should be lower and the left motor speed higher. As for the actual percentages, these will differ depending on factors such as the speed of the MCU car, wheel slippage, and the response speed of the servo. We'll have to see what happens when we try it out with the actual MCU car. For the time being, we'll use following settings, which can be modified later based on running tests.</p> <p>Steering angle: 15 degrees Left motor: 40%, Right motor: 31%</p> <p>Afterward, go to pattern 54.</p>
 <p style="text-align: right;">→ 0 x 0 0</p>	<p>When proceeding straight ahead, the sensor state is 0x00. The software judges a sensor state of 0x00 as indicating straight ahead. There can be no doubt that the steering angle must be straight forward. The problem is the PWM values of the motors. The speed associated with a particular value can only be determined in an actual test run. The speed must be sufficiently low that the MCU car can negotiate the turn when the end of the centre line is encountered. For the time being, we'll use a setting 40% for both motors. This can be fine tuned later when doing test runs. The settings can be summarized as follows:</p> <p>Steering angle: 0 degrees Left motor: 40%, Right motor: 40%</p>
 <p style="text-align: right;">→ 0 x 0 4 → 0 x 0 6 → 0 x 0 7 → 0 x 0 3</p>	<p>Let's assume the MCU car has drifted left of centre. As little by little the MCU car drifts farther left of centre, the four resulting sensor states are as shown in the figure at left. Additional sensor states are possible when the MCU car is even farther left of centre, but we will not add more sensor states because we know that the section of track following the right half lines will be straight.</p> <p>Since the MCU car is left of centre, it is necessary to turn the steering wheel to the right. If the amount of turn is too small and the amount of drift is large, the MCU car will be unable to return to the centre. If the amount of turn is too large, it will overshoot the centre and the car will end up zigzagging right and left. Fine adjusting the angle to precisely the right value is difficult. For the time being, we will use a setting of 8 degrees. The settings can be summarized as follows:</p> <p>Steering angle: 8 degrees Left motor: 40%, Right motor: 35%</p>

	<p>Let's assume the MCU car has drifted right of centre. As little by little the MCU car drifts farther right of centre, the four resulting sensor states are as shown in the figure at left. Additional sensor states are possible when the MCU car is even farther right of centre, but we will not add more sensor states because we know that the section of track following the right half lines will be straight.</p> <p>Since the MCU car is right of centre, it is necessary to turn the steering wheel to the left. If the amount of turn is too small and the amount of drift is large, the MCU car will be unable to return to the centre. If the amount of turn is too large, it will overshoot the centre and the car will end up zigzagging left and right. Fine adjusting the angle to precisely the right value is difficult. For the time being, we will use a setting of -8 degrees. The settings can be summarized as follows:</p> <p>Steering angle: -8 degrees Left motor: 35%, Right motor: 40%</p>
--	--

It is important to remember that all eight sensors are used for detecting the end of the centre line. For other checking, **MASK3_3** masking is applied and the two middle sensors are not used.

The finished program code is as follows:

```

366 :         case 53:
367 :             /* Trace, lane change after right half line detection */
368 :             if( sensor_inp(MASK4_4) == 0x00 ) {
369 :                 handle( 15 );
370 :                 motor( 40 , 31 );
371 :                 pattern = 54;
372 :                 cnt1 = 0;
373 :                 break;
374 :             }
375 :             switch( sensor_inp(MASK3_3) ) {
376 :                 case 0x00:
377 :                     /* Center -> straight */
378 :                     handle( 0 );
379 :                     motor( 40 , 40 );
380 :                     break;
381 :                 case 0x04:
382 :                 case 0x06:
383 :                 case 0x07:
384 :                 case 0x03:
385 :                     /* Left of center -> turn to right */
386 :                     handle( 8 );
387 :                     motor( 40 , 35 );
388 :                     break;
389 :                 case 0x20:
390 :                 case 0x60:
391 :                 case 0xe0:
392 :                 case 0xc0:
393 :                     /* Right of center -> turn to left */
394 :                     handle( -8 );
395 :                     motor( 35 , 40 );
396 :                     break;
397 :                 default:
398 :                     break;
399 :             }
400 :             break;

```

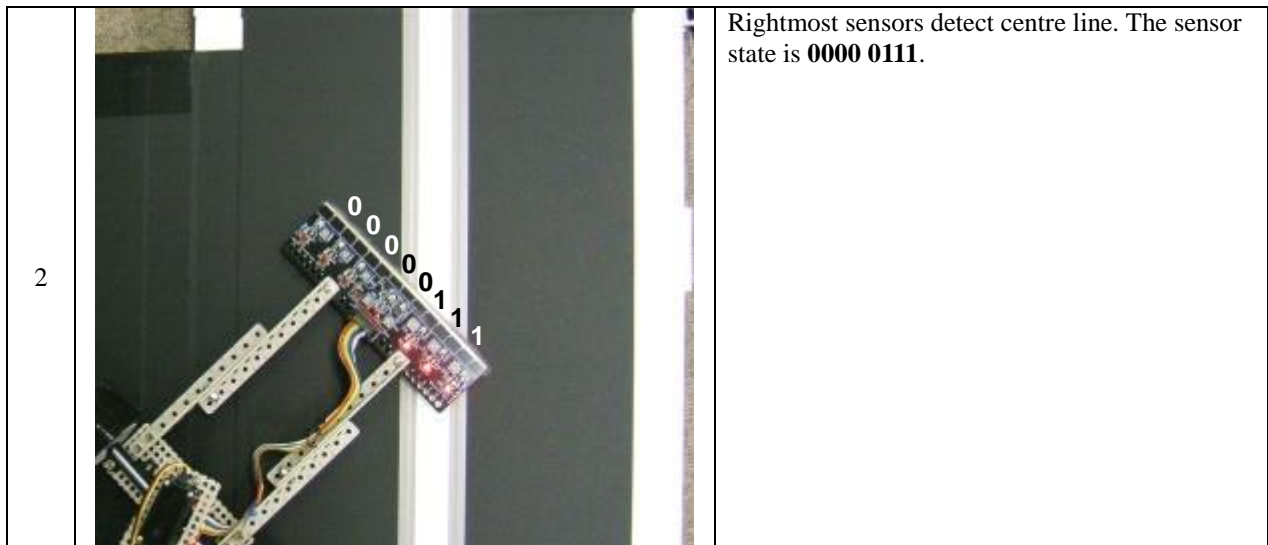
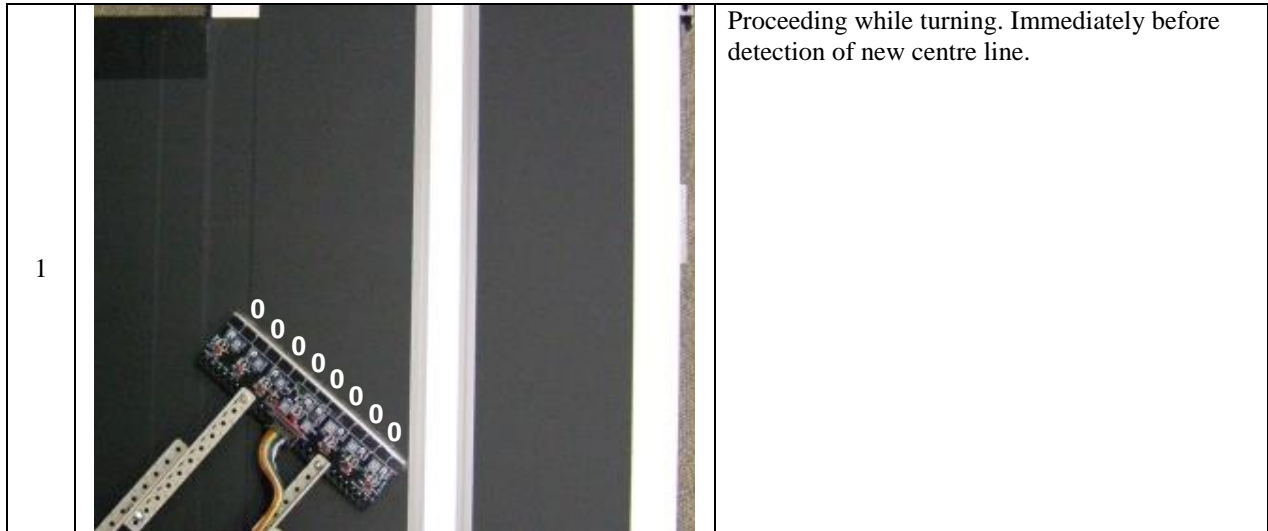
The meaning of these consecutive **case** statements is:
 “when 0x04 or 0x06 or 0x07 or 0x03 is the case.”

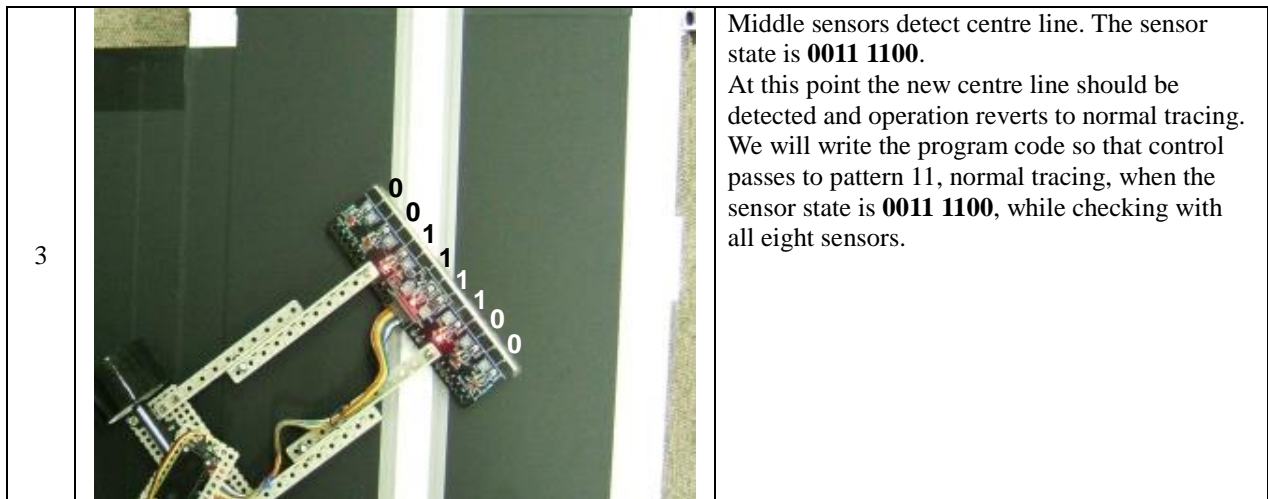
The meaning of these consecutive **case** statements is:
 “when 0x20 or 0x60 or 0xe0 or 0xc0 is the case.”

5.4.42. Pattern 54: Right Lane Change End Check

Pattern 53 determines that a right lane change should start when the eight sensors have a value of **0x00** and turns to the right with a turn angle of 15 degrees. The question then is how long the turn to the right should continue. This portion of the program code is designated as pattern 54.

Pattern 54 causes the MCU car to proceed to the new centre line on the right. Once the new centre line is found, it must trace that centre line. This will complete the processing for the right lane change operation. Now, what sort of sensor status should be interpreted as the presence of the new centre line?



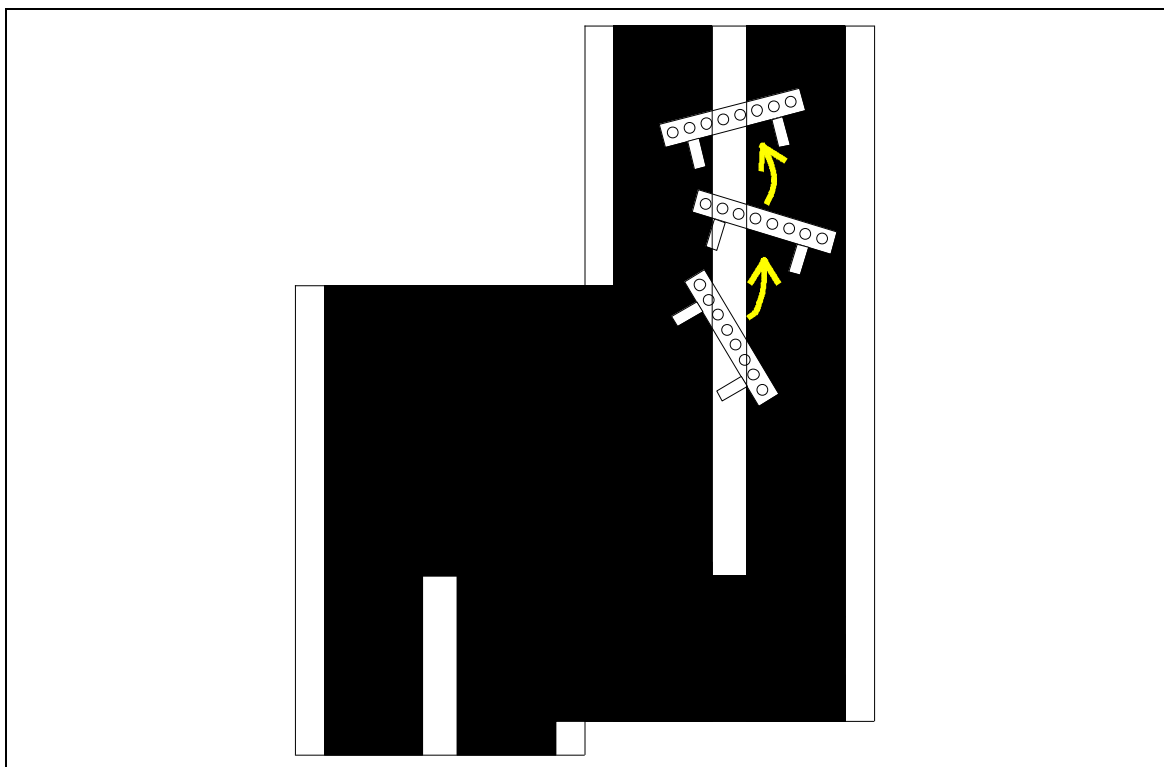


This is the resulting code, based on this thinking:

```

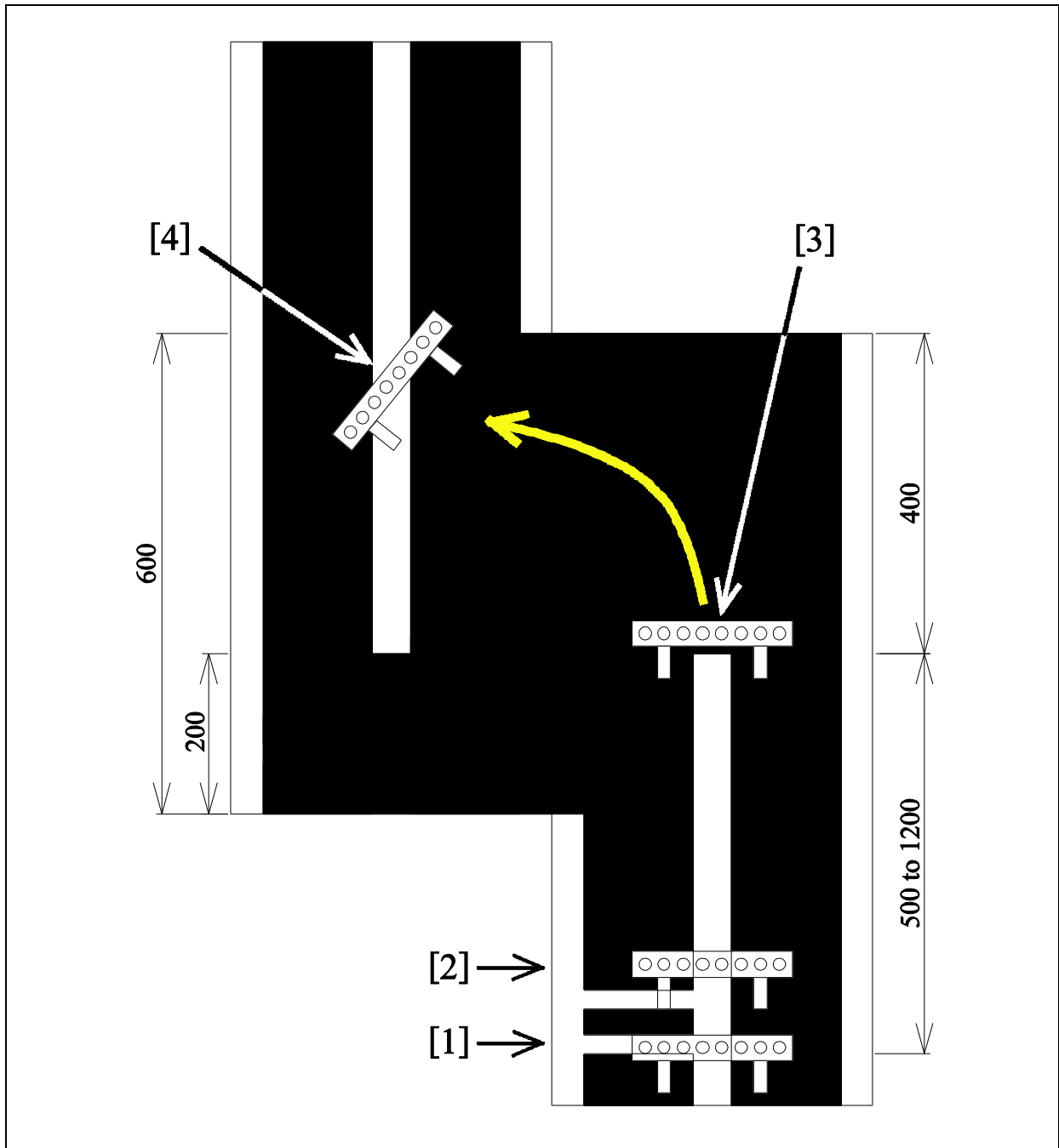
402 :         case 54:
403 :             /* Right lane change end check */
404 :             if( sensor_inp( MASK4_4 ) == 0x3c ) {
405 :                 led_out( 0x0 );
406 :                 pattern = 11;
407 :                 cnt1 = 0;
408 :             }
409 :             break;
    
```

An LED was illuminated when the right half line was detected, so after turning off the LED in line 405 control is passed to pattern 11. The situation after **0x3c** is detected and control returns to pattern 11 is shown below. The MCU car is travelling at a rightward angle when it encounters the centre line, so processing by pattern 11 brings it back to the centre of the track.



5.4.43. Left Lane Change Outline

Patterns 61 to 64 contain program code related to executing a left lane change. An outline of the processing involved is provided below:

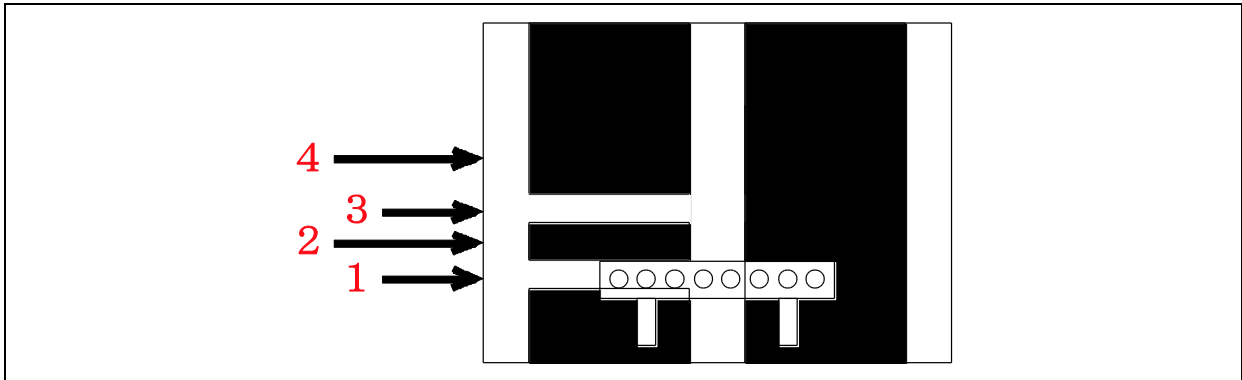


[1]	The check_leftline function detects a left half line. The MCU car must change to the left lane 500 to 1200 mm ahead, so brake operation is performed. Also, sensor input is ignored up to position [2] to prevent erroneous sensor detection at the second left half line.
[2]	The MCU car starts to proceed slowly from this point. It advances while tracing the centre line.
[3]	When the centre line ends, the steering wheel turns to the left.
[4]	When a new centre line is detected, line-tracing restarts using the new centre line.

In this way, the MCU car manoeuvres right lane change. The specifics of the program code used are described below.

5.4.44. Processing at 1st Left Half Line Detection

Control passes to pattern 61 the moment a left half line is detected. First, the MCU car passes over the left half lines. The characteristics of the portion of track from the position at which the first left half line is discovered to the position immediately after the second left half line are shown in the following figure:



- [1] First crossline
- [2] Normal track
- [3] Second crossline
- [4] Normal track, proceed slowly while tracing centre line

The track, other than the centre line, changes from white to black to white to black again by the time position [4] is reached. The program must detect these changes and respond appropriately. That sounds pretty complicated.

Let's look at this in a different way. The distance from position [1] to position [4] is about 100 mm, allowing some margin for error. (The precise distance is 70 mm: 20 mm for the first half line + 30 mm of black area + 20 mm for the second half line = 70 mm.) If the MCU car is positioned roughly over the centre line and continues to move forward for about 100 mm while we ignore the sensor data, we'll probably come out roughly on course. The kit car includes no mechanism for detecting distances, but we can use the timer to interrupt reading of sensor data for a specified duration. We don't know how long a duration yet because that will depend on how fast the MCU car is travelling. For the time being, let's use a pause duration of 0.1 seconds and do fine tuning later. In addition, we'll make the LEDs on the motor drive board light to indicate externally that processing of pattern 61 has started.

To summarize:

- Illuminate LED3. (This differs from the processing of crossline to make it possible to tell them apart.)
- Set steering angle to 0 degrees.
- Set PWM value of right and left motors to 0% to initiate brake operation.
- Wait 0.1 seconds.
- After 0.1 seconds elapse, go to next pattern.

This is what the program code of pattern 61 must accomplish.

```

case 61:
    /* Processing at 1st left half line detection */
    led_out( 0x1 );
    handle( 0 );
    speed( 0 , 0 );
    if( cnt1 > 100 ) {
        pattern = 62; /* After 0.1 seconds, to pattern 62 */
    }
    break;

```

This is the completed program code. Let's take a moment to review it. When the value of **cnt1** is 100 or greater (after 100 milliseconds have elapsed), control passes to pattern 62. For this to work as expected, the value of **cnt1** must be 0 when pattern 61 starts. For example, if the value of **cnt1** is 1000 when control passes to pattern 61, the value of **cnt1** will be judged to be 100 or greater the first time the condition is tested, and control will pass immediately to pattern 62. Execution of pattern 61 takes place only once (a duration of a few dozen μ s) rather than lasting for 0.1 seconds. We need to add another pattern. Pattern 61 will start brake operation and clear **cnt1** to 0, and pattern 62 will check whether 0.1 seconds have elapsed.

To summarize once again:

Tasks performed by pattern 61:	<ul style="list-style-type: none"> • Illuminate LED3. • Set steering angle to 0 degrees. • Set PWM value of right and left motors to 0% to initiate brake operation. • Go to next pattern. • Clear cnt1.
Tasks performed by pattern 62:	<ul style="list-style-type: none"> • If value of cnt1 is 100 or greater, go to next pattern.

Let's rewrite the program to reflect the above changes.

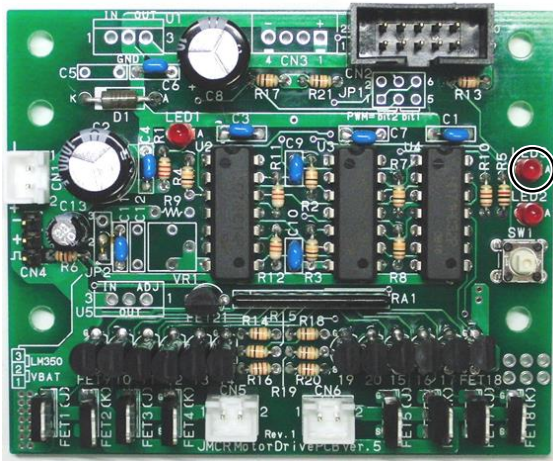
```

411 :         case 61:
412 :             /* Processing at 1st left half line detection */
413 :             led_out( 0x1 );
414 :             handle( 0 );
415 :             motor( 0 , 0 );
416 :             pattern = 62;
417 :             cnt1 = 0;
418 :             break;
419 :
420 :         case 62:
421 :             /* Read but ignore 2nd time */
422 :             if( cnt1 > 100 ){
423 :                 pattern = 63;
424 :                 cnt1 = 0;
425 :             }
426 :             break;

```

The portion of the program code from detection of the left half line to the start of the trace centre line area is now complete.

Hint



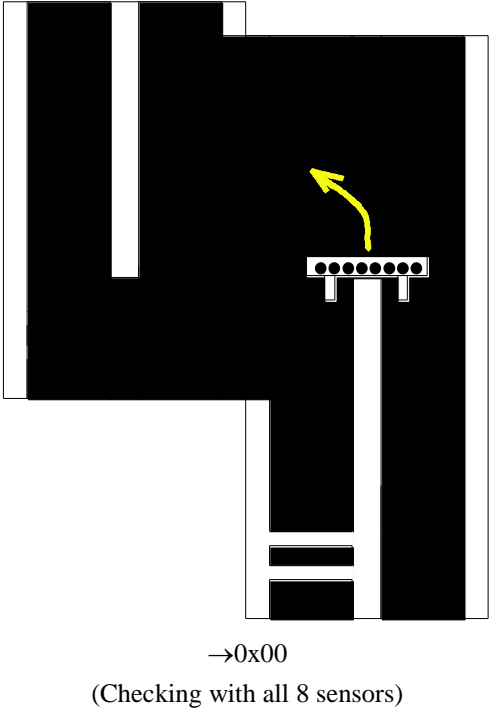
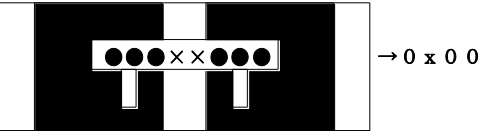
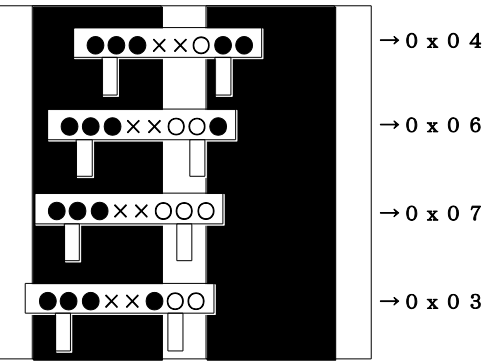
When a left half line is detected, one LED (the top one) on the motor drive board lights. No left half line has been detected if it does not light. If left half line detection is not working properly, try unplugging the motor connectors and pushing the MCU car forward by hand.

5.4.45. Pattern 63: Trace after Left Half Line

Patterns 61 and 62 perform brake operation for 0.1 seconds after detection of the first left half line, allowing the MCU car to pass the second left half line. Pattern 63 continues the processing after this.

The MCU car is past the left half lines, so the next task is detecting the end of the centre line. In addition, the MCU car must continue to trace the straight section of track that leads to the end of the centre line, so normal trace operation is necessary.

We envision the present situation as shown in the following figures:

 <p>→0x00 (Checking with all 8 sensors)</p>	<p>After the left half line ends, the state of the eight sensors is 0x00, as shown in the figure at left. When this state is detected, the MCU car starts turning to the left. For a left turn, we would expect that the left motor speed should be lower and the right motor speed higher. As for the actual percentages, these will differ depending on factors such as the speed of the MCU car, wheel slippage, and the response speed of the servo. We'll have to see what happens when we try it out with the actual MCU car. For the time being, we'll use following settings, which can be modified later based on running tests.</p> <p>Steering angle: -15 degrees Left motor: 31%, Right motor: 40%</p> <p>Afterward, go to pattern 64.</p>
 <p>→ 0 x 0 0</p>	<p>When proceeding straight ahead, the sensor state is 0x00. The software judges a sensor state of 0x00 as indicating straight ahead. There can be no doubt that the steering angle must be straight forward. The problem is the PWM values of the motors. The speed associated with a particular value can only be determined in an actual test run. The speed must be sufficiently low that the MCU car can negotiate the turn when the end of the centre line is encountered. For the time being, we'll use a setting 40% for both motors. This can be fine tuned later when doing test runs. The settings can be summarized as follows:</p> <p>Steering angle: 0 degrees Left motor: 40%, Right motor: 40%</p>
 <p>→ 0 x 0 4 → 0 x 0 6 → 0 x 0 7 → 0 x 0 3</p>	<p>Let's assume the MCU car has drifted left of centre. As little by little the MCU car drifts farther left of centre, the four resulting sensor states are as shown in the figure at left. Additional sensor states are possible when the MCU car is even farther left of centre, but we will not add more sensor states because we know that the section of track following the right half lines will be straight.</p> <p>Since the MCU car is left of centre, it is necessary to turn the steering wheel to the right. If the amount of turn is too small and the amount of drift is large, the MCU car will be unable to return to the centre. If the amount of turn is too large, it will overshoot the centre and the car will end up zigzagging right and left. Fine adjusting the angle to precisely the right value is difficult. For the time being, we will use a setting of 8 degrees. The settings can be summarized as follows:</p> <p>Steering angle: 8 degrees Left motor: 40%, Right motor: 35%</p>

	<p>→ 0 x 2 0</p> <p>→ 0 x 6 0</p> <p>→ 0 x e 0</p> <p>→ 0 x c 0</p>	<p>Let's assume the MCU car has drifted right of centre. As little by little the MCU car drifts farther right of centre, the four resulting sensor states are as shown in the figure at left. Additional sensor states are possible when the MCU car is even farther right of centre, but we will not add more sensor states because we know that the section of track following the right half lines will be straight.</p>
		<p>Since the MCU car is right of centre, it is necessary to turn the steering wheel to the left. If the amount of turn is too small and the amount of drift is large, the MCU car will be unable to return to the centre. If the amount of turn is too large, it will overshoot the centre and the car will end up zigzagging left and right. Fine adjusting the angle to precisely the right value is difficult. For the time being, we will use a setting of -8 degrees. The settings can be summarized as follows:</p> <p>Steering angle: -8 degrees Left motor: 35%, Right motor: 40%</p>

It is important to remember that all eight sensors are used for detecting the end of the centre line. For other checking, **MASK3_3** masking is applied and the two middle sensors are not used.

The finished program code is as follows:

```

428 :         case 63:
429 :             /* Trace, lane change after left half line detection */
430 :             if( sensor_inp(MASK4_4) == 0x00 ) {
431 :                 handle( -15 );
432 :                 motor( 31 , 40 );
433 :                 pattern = 64;
434 :                 cnt1 = 0;
435 :                 break;
436 :             }
437 :             switch( sensor_inp(MASK3_3) ) {
438 :                 case 0x00:
439 :                     /* Center -> straight */
440 :                     handle( 0 );
441 :                     motor( 40 , 40 );
442 :                     break;
443 :                 case 0x04:
444 :                 case 0x06:
445 :                 case 0x07:
446 :                 case 0x03:
447 :                     /* Left of center -> turn to right */
448 :                     handle( 8 );
449 :                     motor( 40 , 35 );
450 :                     break;
451 :                 case 0x20:
452 :                 case 0x60:
453 :                 case 0xe0:
454 :                 case 0xc0:
455 :                     /* Right of center -> turn to left */
456 :                     handle( -8 );
457 :                     motor( 35 , 40 );
458 :                     break;
459 :                 default:
460 :                     break;
461 :             }
462 :             break;

```

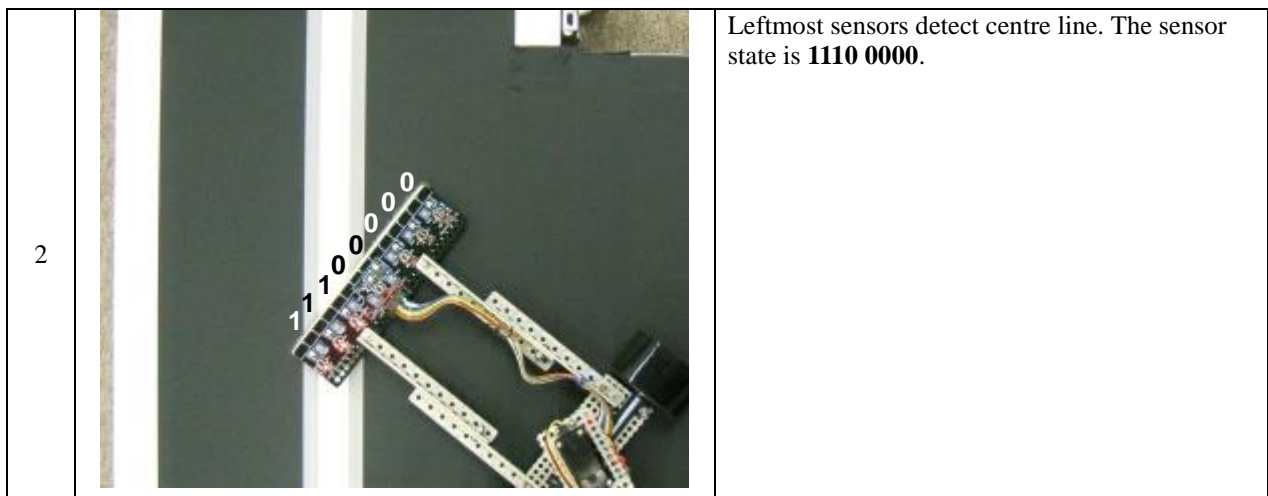
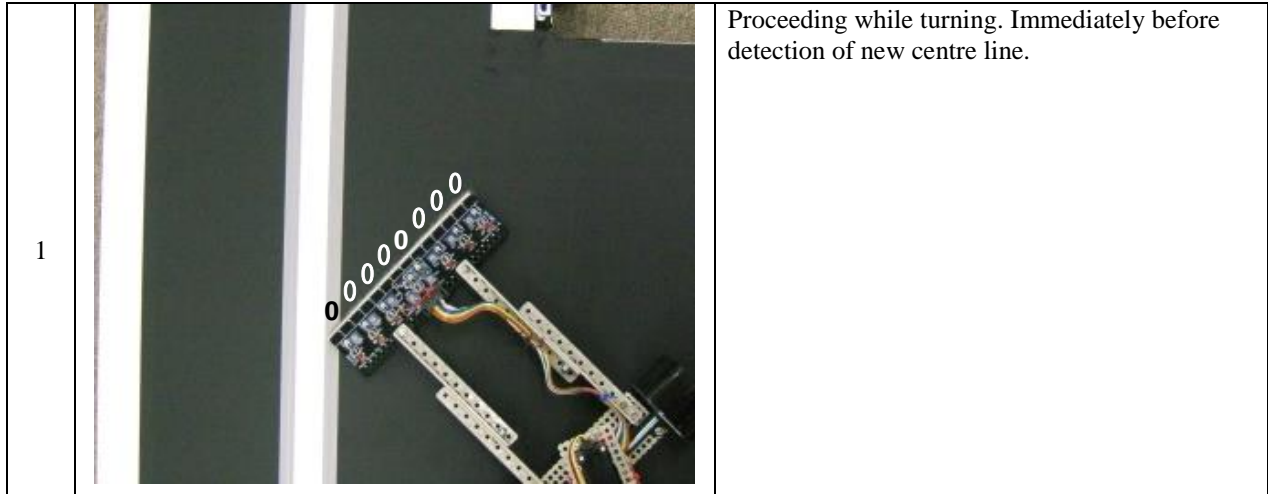
The meaning of these consecutive **case** statements is: "when 0x04 or 0x06 or 0x07 or 0x03 is the case."

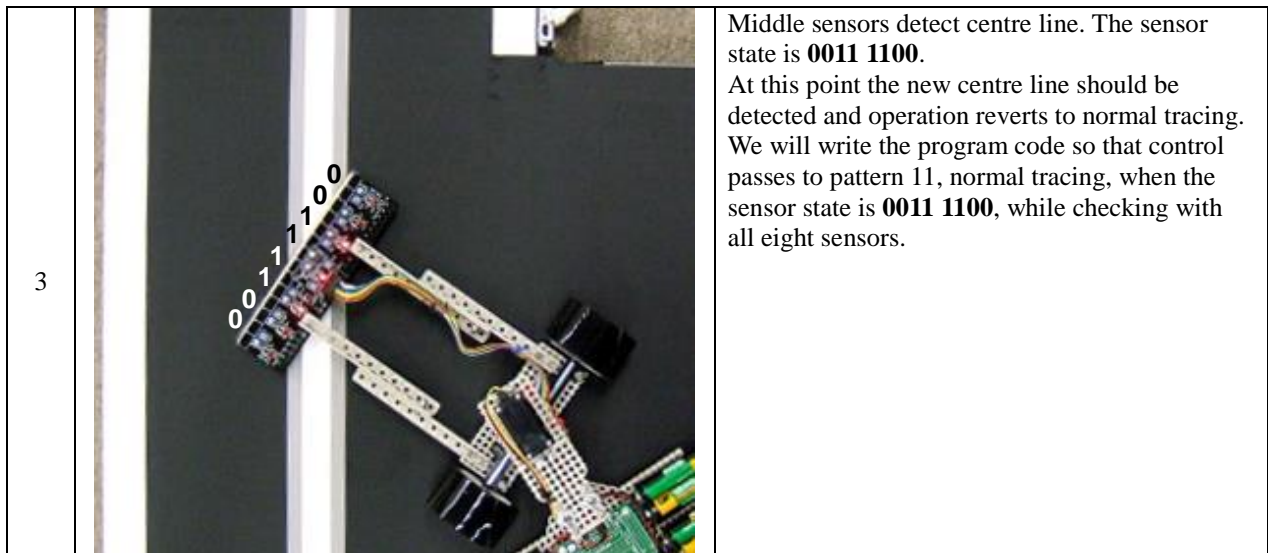
The meaning of these consecutive **case** statements is: "when 0x20 or 0x60 or 0xe0 or 0xc0 is the case."

5.4.46. Pattern 64: Left Lane Change End Check

Pattern 63 determines that a left lane change should start when the eight sensors have a value of 0x00 and turns to the left with a turn angle of 15 degrees. The question then is how long the turn to the left should continue. This portion of the program code is designated as pattern 64.

Pattern 64 causes the MCU car to proceed to the new centre line on the left. Once the new centre line is found, it must trace that centre line. This will complete the processing for the left lane change operation. Now, what sort of sensor status should be interpreted as the presence of the new centre line?



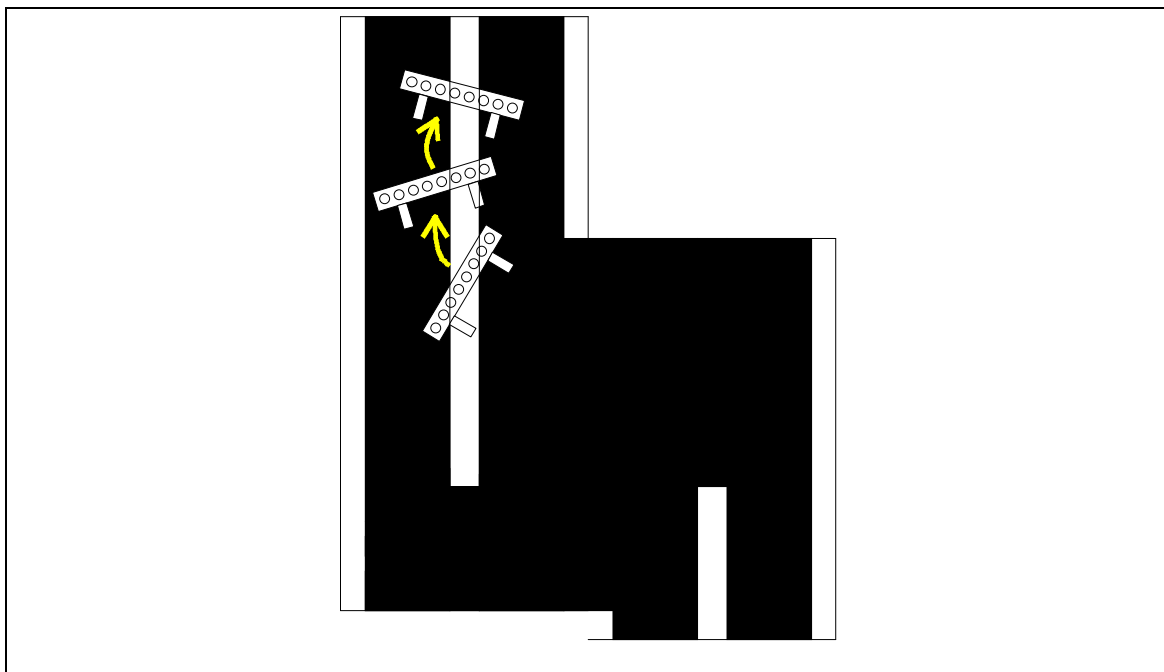


This is the resulting code, based on this thinking:

```

464 :         case 64:
465 :             /* Left lane change end check */
466 :             if( sensor_inp( MASK4_4 ) == 0x3c ) {
467 :                 led_out( 0x0 );
468 :                 pattern = 11;
469 :                 cnt1 = 0;
470 :             }
471 :             break;
    
```

An LED was illuminated when the left half line was detected, so after turning off the LED in line 467 control is passed to pattern 11. The situation after **0x3c** is detected and control returns to pattern 11 is shown below. The MCU car is travelling at a leftward angle when it encounters the centre line, so processing by pattern 11 brings it back to the centre of the track.

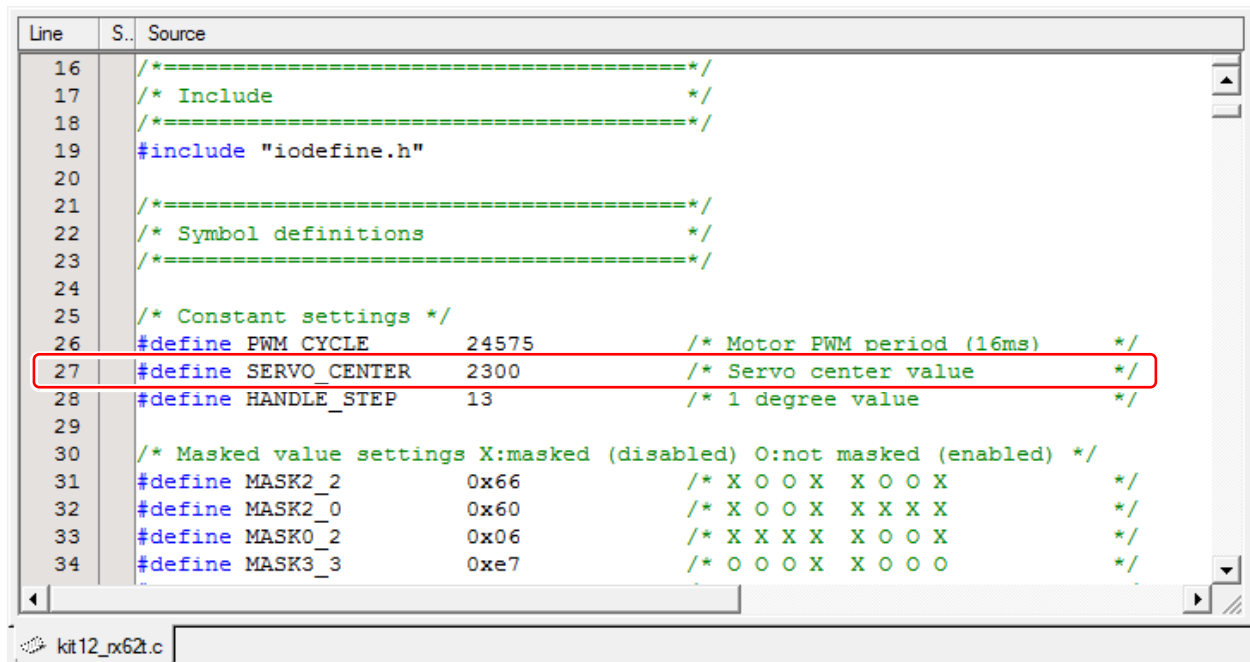


6. Adjusting the Servo Centre and Maximum Turn Angle

6.1. Outline

After writing kit12_rx62t to the MCU and powering on the MCU car, the steering angle will not be exactly 0 degrees (straight ahead) in most cases. Just as each person's fingerprints are different from those of everyone else, each servo has a different numerical value that translates into "straight ahead."

This section explains how to adjust the servo centre. The servo centre value appears on line 27 of kit12_rx62t.c



```

Line  S.. Source
16  /*=====*/
17  /* Include */
18  /*=====*/
19  #include "iodefine.h"
20
21  /*=====*/
22  /* Symbol definitions */
23  /*=====*/
24
25  /* Constant settings */
26  #define PWM_CYCLE 24575 /* Motor PWM period (16ms) */
27  #define SERVO_CENTER 2300 /* Servo center value */
28  #define HANDLE_STEP 13 /* 1 degree value */
29
30  /* Masked value settings X:masked (disabled) O:not masked (enabled) */
31  #define MASK2_2 0x66 /* X O O X X O O X */
32  #define MASK2_0 0x60 /* X O O X X X X X */
33  #define MASK0_2 0x06 /* X X X X X O O X */
34  #define MASK3_3 0xe7 /* O O O X X O O O */
  
```

To adjust it so that the servo is centred correctly, it is generally necessary to repeat the following steps several times:

- Adjust the value based on the amount of skew (13 is equivalent to 1 degree, and the servo position moves to the left when the value is decreased and to the right when it is increased).
- Build the program.
- Write the program to the RMC-RX62T board.
- Check the 0-degree position.
- Adjust again if it is not straight ahead.

Therefore connect a PC and a MCU car with communications cable.

Adjusting is OK to do following steps.

- Adjust the centre of the servo and find the value of 0 degree while using the keyboard of the PC.
- Write the value to the program.
- Build the program
- Write the program to the RMC-RX62T board.



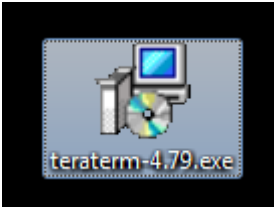
It is straighter than before. This section explains how to adjust the following using the keyboard of the PC.

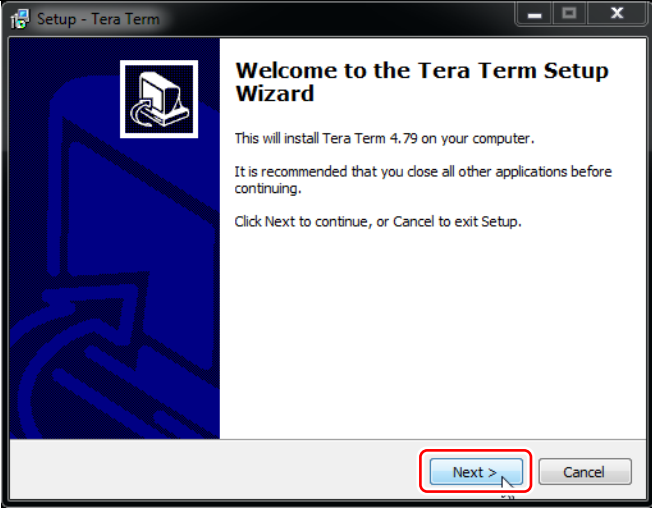
- ① Adjust the value of the servo centre simply (implementation at sioservo1_62t project).
- ② Find the right maximum turning angle: how many degrees the car can turn to the right (implementation at sioservo2_62t project).
- ③ Find the left maximum turning angle how many degrees the car can turn to the left (implementation at sioservo2_62t project).

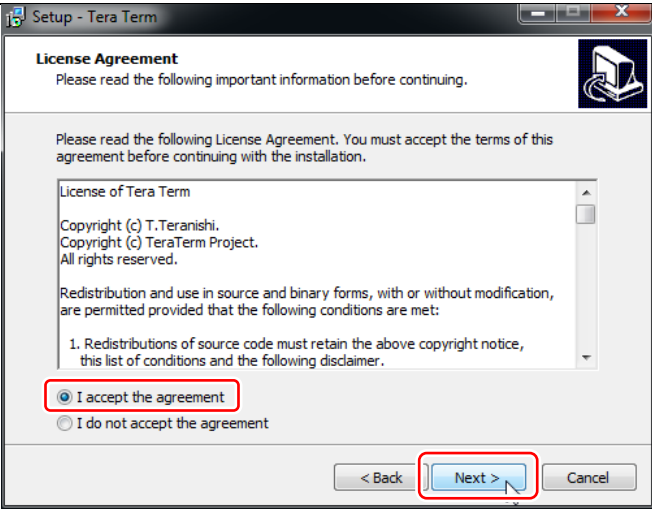
6.2. Install the communication program Tera Term

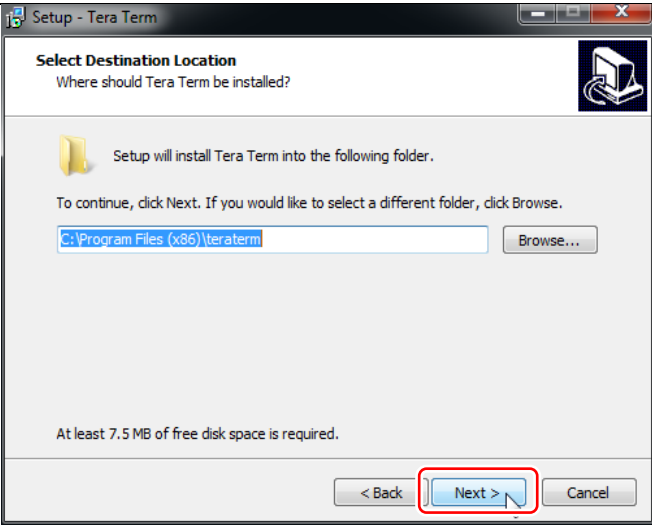
Perform communication with MCU using free communication soft **Tera Term**. This section explains how to install it. If you already have installed it, there is no need to install again.

Note: Continue with step 3, if you have CD-R for this seminar.

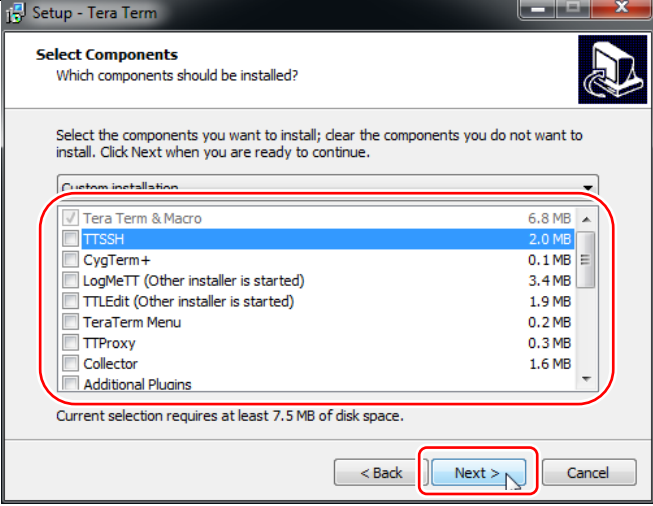
<p>1</p>		<p>First download the software. Open the following URL in a Web browser</p> <p>http://www.forest.impress.co.jp/library/software/utf8teraterm/</p> <p>Note: Download from the site named “windows forest”, or a similar site.</p>
<p>2</p>		<p>Click DOWNLOAD and the download the file.</p>
<p>3</p>		<p>Launch the teraterm-4.79.exe.</p> <p>Please execute "teraterm-4.79.exe" in the following folder, if you have CD-R for this seminar. "CD-R drive:¥01-Softwares"</p> <p>Note: The number 4.79 is different depending on the version.</p>

4		Click Next > .
---	---	--------------------------

5		Click accept after reading the agreement. Click Next > .
---	--	--

6		Click Next > .
---	---	--------------------------

7



Choosing the components.
Uncheck all of the extra components, they are not needed.

Click **Next >**.

8



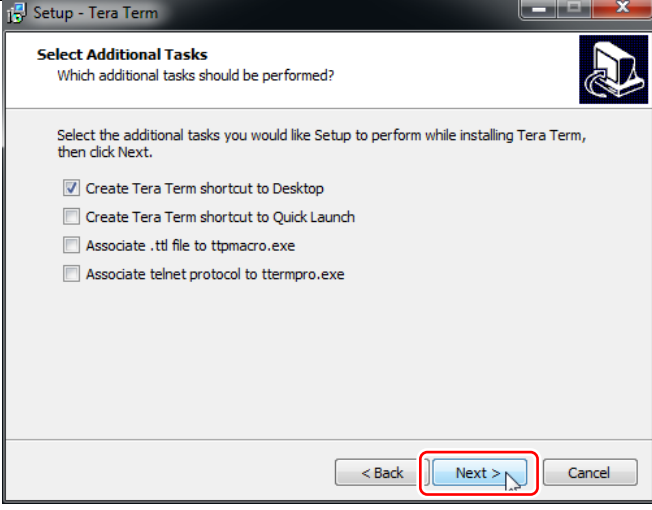
Select **English**.

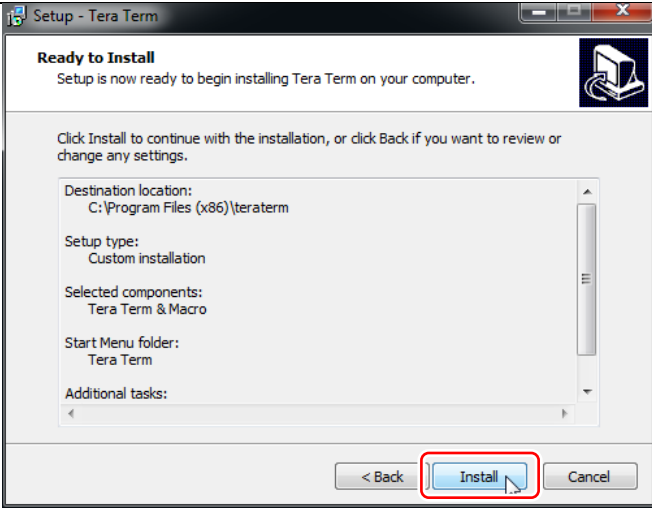
Click **Next >**.

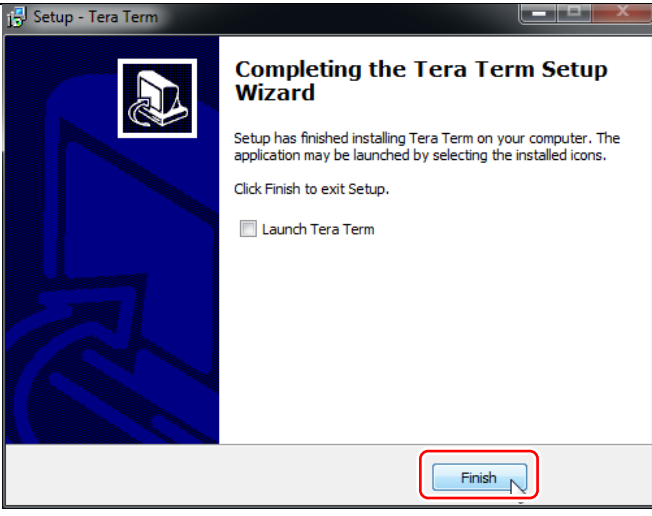
9




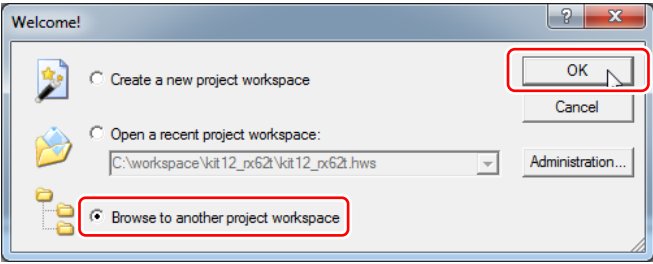
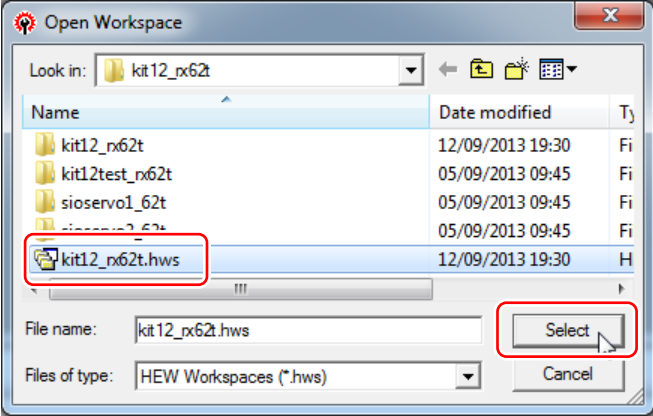
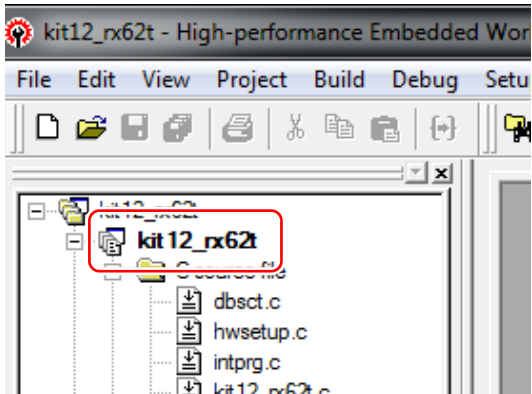
Click **Next >**.

10		<p>Select the additional task (no need to change normally).</p> <p>Click Next >.</p>
----	---	--

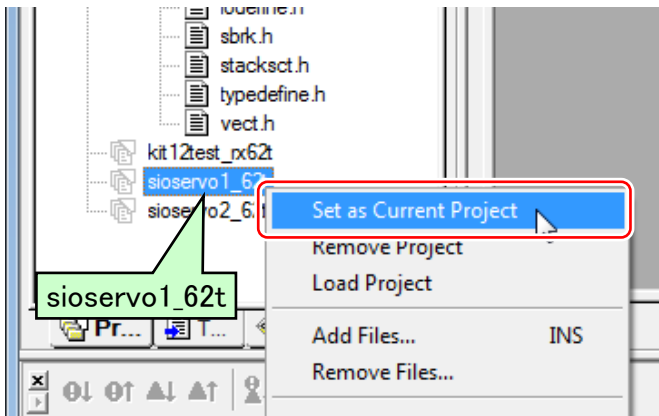
11		<p>Click Install.</p>
----	--	------------------------------

12		<p>Click Finish to close the installer.</p>
----	---	--

6.3. Adjusting the Servo Centre

1		<p>Launch the Renesas integrated development environment.</p>
2		<p>Select Browse to another project workspace.</p> <p>Click OK.</p>
3		<p>Select kit12_rx62t.hws from the C:\Workspace\kit12_rx62t folder.</p>
4		<p>The kit12_rx62t workspace opens.</p>

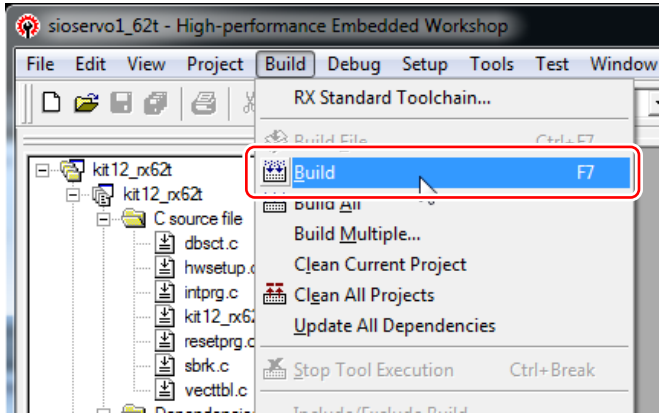
5



The screenshot shows a project browser window with a tree view containing files like 'sbrk.h', 'stacksect.h', 'typedefine.h', and 'vect.h'. Below these are project folders 'kit12test_rx62t', 'sioservo1_62t', and 'sioservo2_62t'. A context menu is open over 'sioservo1_62t', with 'Set as Current Project' highlighted in blue. A red box highlights this menu item. A green callout box points to the 'sioservo1_62t' folder in the tree.

Set project **sioservo1_62t** as the current project.

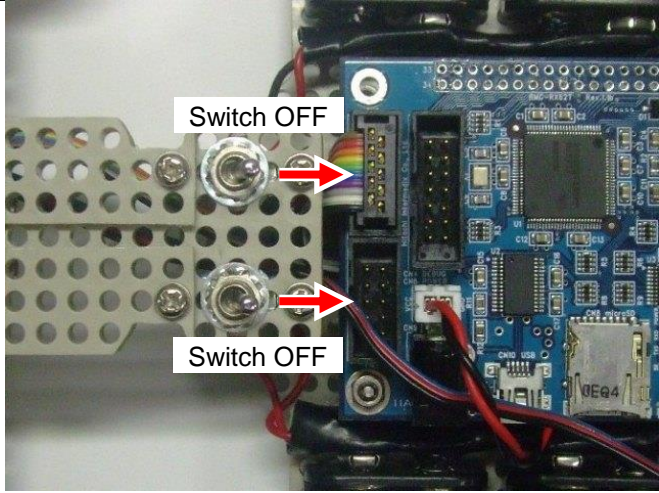
6



The screenshot shows the IDE's 'Build' menu. The menu items include 'Build', 'Build Multiple...', 'Clean Current Project', 'Clean All Projects', 'Update All Dependencies', and 'Stop Tool Execution'. The 'Build' item is highlighted in blue and has a red box around it. The keyboard shortcut 'F7' is visible next to it.

Select **Build** from the **Build** menu.
This generates a MOT file.

7



The photograph shows the MCU car kit's internal components. Two power switches are visible on a breadboard. Red arrows point from the text 'Switch OFF' to each of the switches.

Move the two power switches of the MCU car to the off position.

8		<p>[1] Connect PC and MCU board by USB cable.</p> <p>[2] Turn on RXD and EMLE of SW3.</p> <p>[3] Turn SW5 to PROGRAM. SW5 must change states while the power is off.</p>
---	--	--

9		<p>Move the two power switches of the MCU car to the on position.</p>
---	--	---

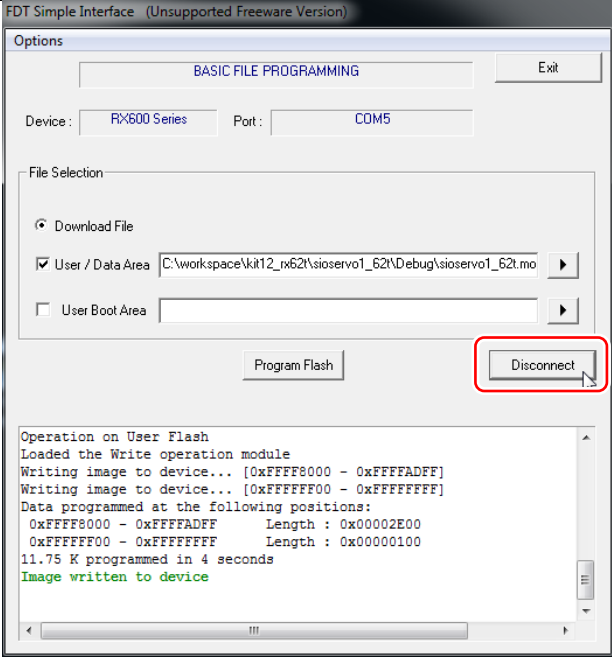
10		<p>Launch the Flash Development Toolkit 4.09 Basic.</p>
----	--	--

11		<p>Check User/Data Area. Then click the triangle on the far right and click Browse.</p>
----	--	---

12		<p>Open the sioservo1_62t.mot file.</p> <p>The sioservo1_62t.mot file is found in the below folder. "C:\WorkSpace\kit12_rx62t\sioservo1_62t\Debug"</p> <p>Click Open.</p>
----	--	---

13		<p>Click Program Flash. Then program writing will begin.</p>
----	--	---

14



Options

BASIC FILE PROGRAMMING

Device: RX600 Series Port: COM5

File Selection

Download File

User / Data Area C:\workspace\kit12_rx62t\sioservo1_62t\Debug\sioservo1_62t.mo

User Boot Area

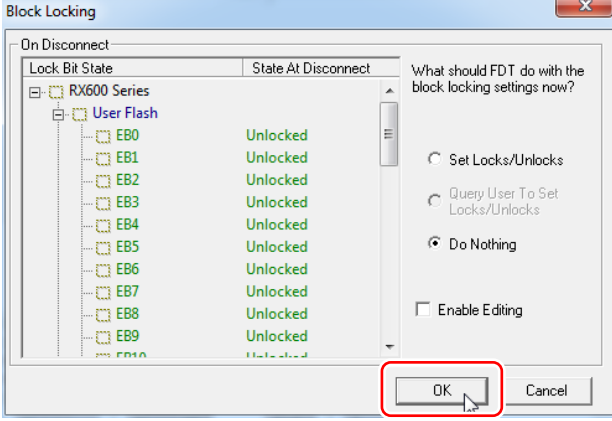
Program Flash

Disconnect

Operation on User Flash
 Loaded the Write operation module
 Writing image to device... [0xFFFF8000 - 0xFFFFADFF]
 Writing image to device... [0xFFFFF00 - 0xFFFFFFFF]
 Data programmed at the following positions:
 0xFFFF8000 - 0xFFFFADFF Length : 0x00002E00
 0xFFFFF00 - 0xFFFFFFFF Length : 0x00000100
 11.75 K programmed in 4 seconds
 Image written to device

After programming has finished, click Disconnect.

15



Block Locking

On Disconnect

Lock Bit State	State At Disconnect
RX600 Series	
User Flash	
EB0	Unlocked
EB1	Unlocked
EB2	Unlocked
EB3	Unlocked
EB4	Unlocked
EB5	Unlocked
EB6	Unlocked
EB7	Unlocked
EB8	Unlocked
EB9	Unlocked

What should FDT do with the block locking settings now?

Set Locks/Unlocks

Query User To Set Locks/Unlocks

Do Nothing

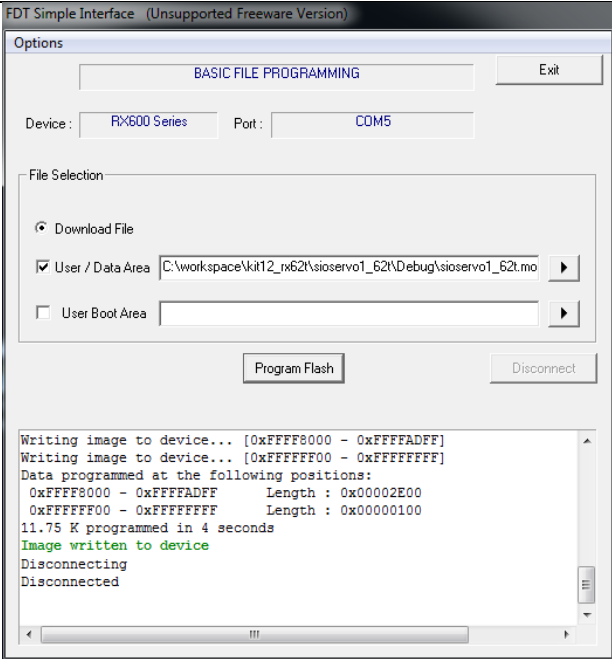
Enable Editing

OK

Cancel

Click OK.

16



Options

BASIC FILE PROGRAMMING

Device: RX600 Series Port: COM5

File Selection

Download File

User / Data Area C:\workspace\kit12_rx62t\sioservo1_62t\Debug\sioservo1_62t.mo

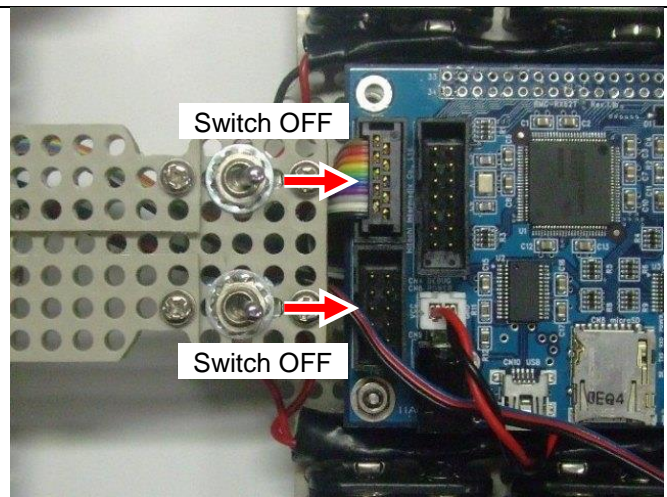
User Boot Area

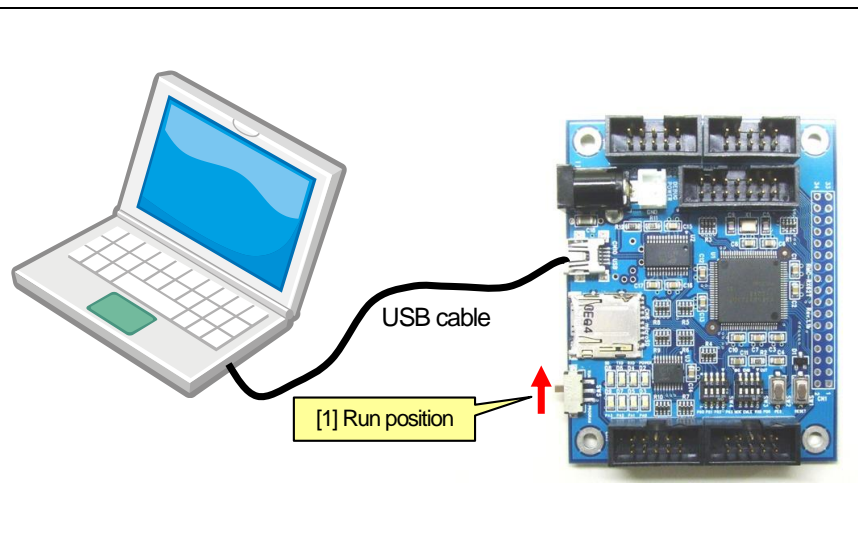
Program Flash

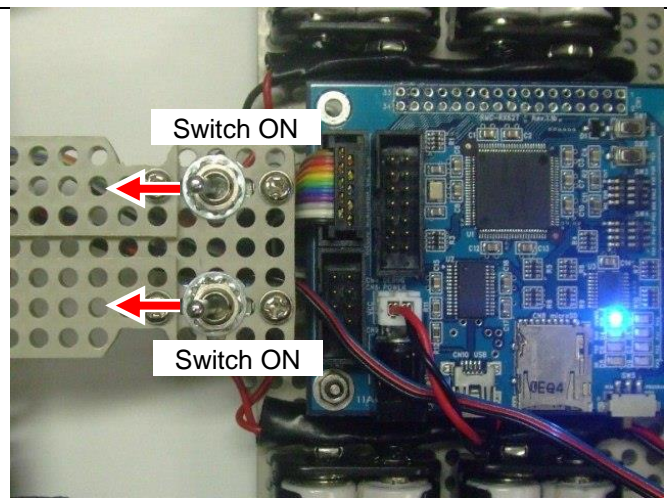
Disconnect


Writing image to device... [0xFFFF8000 - 0xFFFFADFF]
 Writing image to device... [0xFFFFF00 - 0xFFFFFFFF]
 Data programmed at the following positions:
 0xFFFF8000 - 0xFFFFADFF Length : 0x00002E00
 0xFFFFF00 - 0xFFFFFFFF Length : 0x00000100
 11.75 K programmed in 4 seconds
 Image written to device
 Disconnecting
 Disconnected

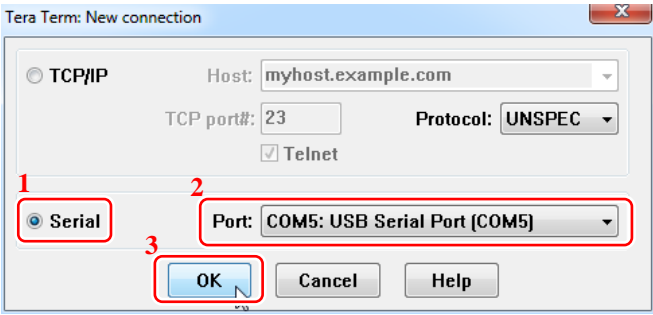
Program writing completed.

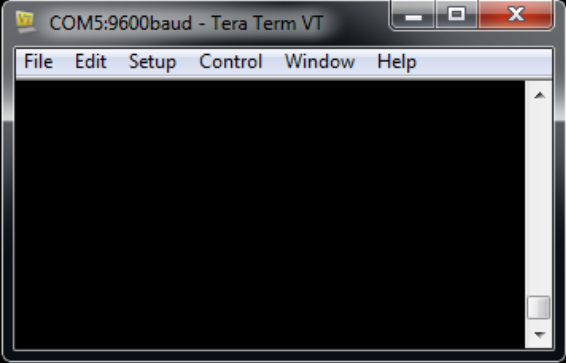
17		Move the two power switches of the MCU car to the off position.
----	---	---

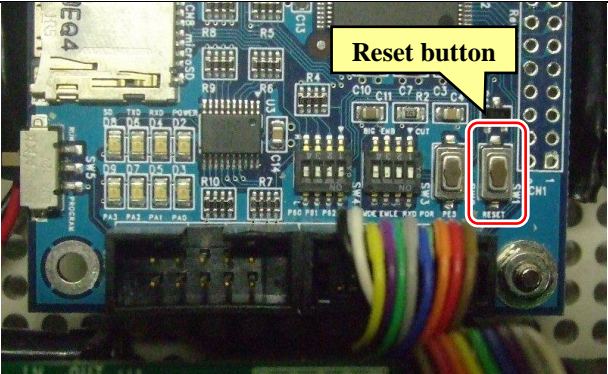
18		[1] Turn SW5 to RUN. Note: keep on having connected the USB cable.
----	---	---

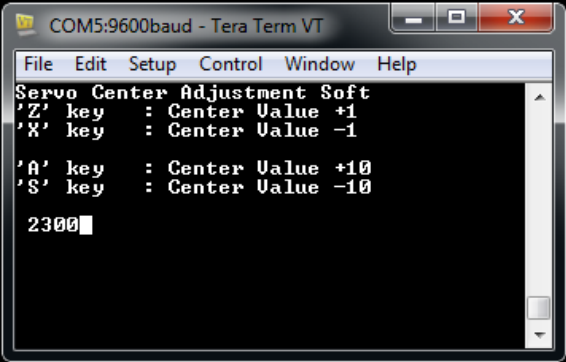
19		Move the two power switches of the MCU car to the on position.
----	---	--

20		Launch the Tera Term .
----	---	-------------------------------

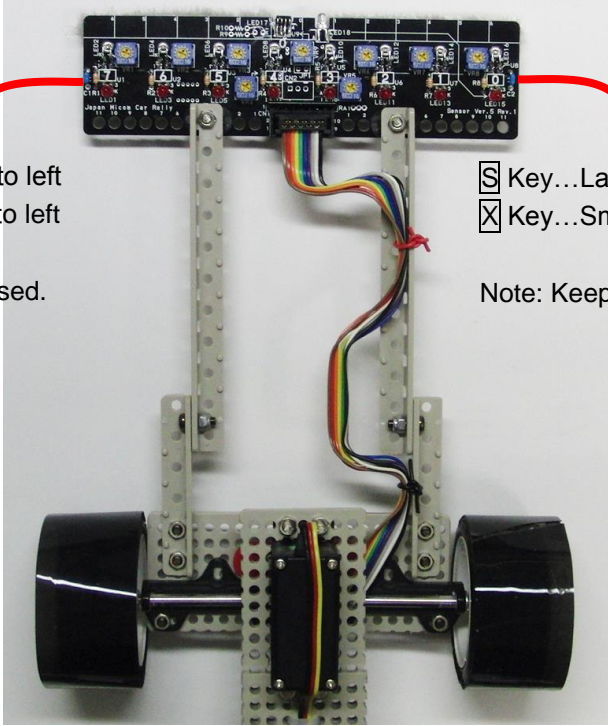
21		<p>[1] A new connection window appears. Select Serial.</p> <p>[2] For Port select the number with the USB Serial Port indication or the number of the currently connected serial port.</p> <p>[3] Click OK.</p>
----	---	---

22		The Tera Term terminal window appears.
----	--	--

23		Push the reset button of RMC-RX62T board.
----	---	---

24		<p>A message like the one shown on the left appears on the screen when the MCU car is powered on.</p> <p>The number 2300 at the bottom is the current servo centre value.</p>
----	---	--

25



A Key...Large amount to left
 Z Key...Small amount to left

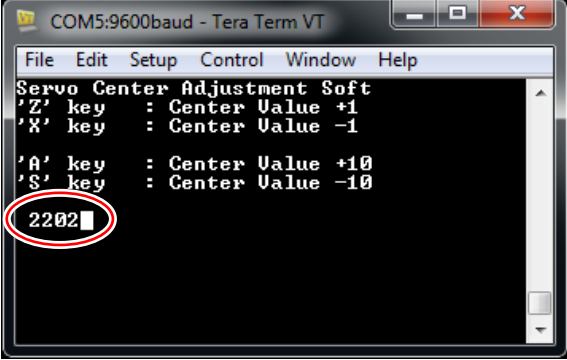
Note: Keep key depressed.

S Key...Large amount to right
 X Key...Small amount to right

Note: Keep key depressed.

Pressing and holding down the A, S, Z, or X key causes the servo to move as indicated. **Use the keys to adjust the servo angle so that it is pointed straight.**

26



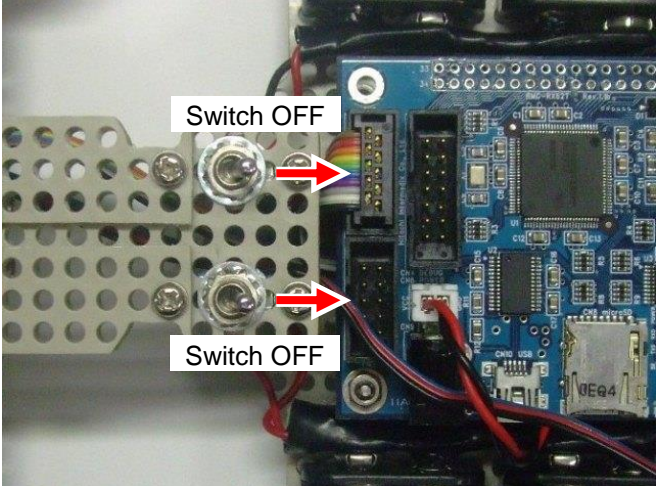
```

COM5:9600baud - Tera Term VT
File Edit Setup Control Window Help
Servo Center Adjustment Soft
'Z' key : Center Value +1
'X' key : Center Value -1
'A' key : Center Value +10
'S' key : Center Value -10
2202
    
```

Once the servo has been adjusted to the straight-ahead position, check the number displayed in the Tera Term screen. In this case, it is 2202. This is the MCU car's servo centre (**SERVO_CENTER**) value. Write it down for later reference.

Next, we will proceed to the `sioservo2_62t` project.
Close Tera Term.

27



Move the two power switches of the MCU car to the off position.

6.4. Determining the Maximum Turning Angle of the Servo

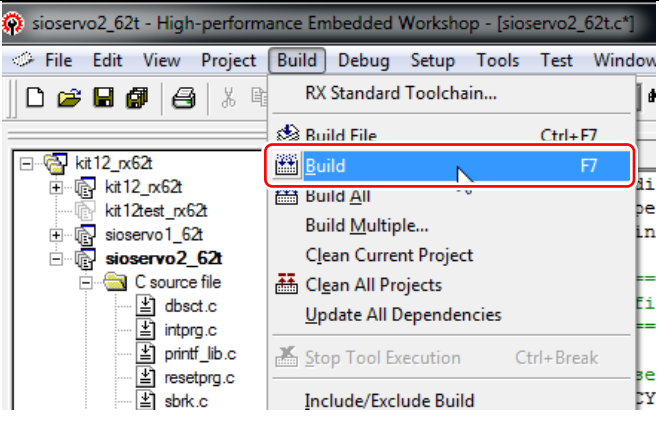
Next, we will determine the maximum turning angle of the servo.


1		<p>Set sioservo2_62t as the current project.</p>
---	--	---

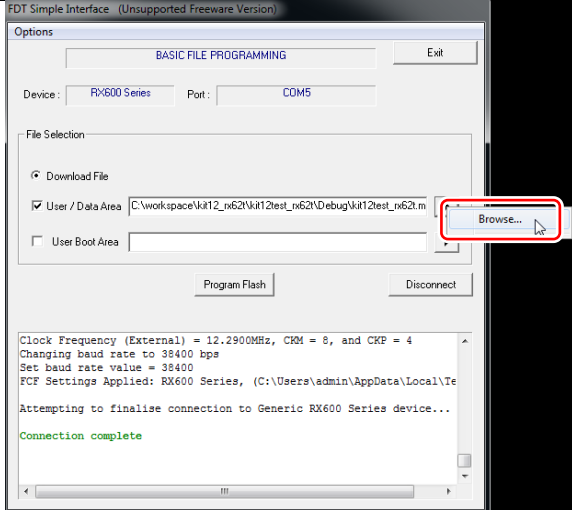
2		<p>Double click sioservo2_62t.c to open the editor window.</p>
---	--	---

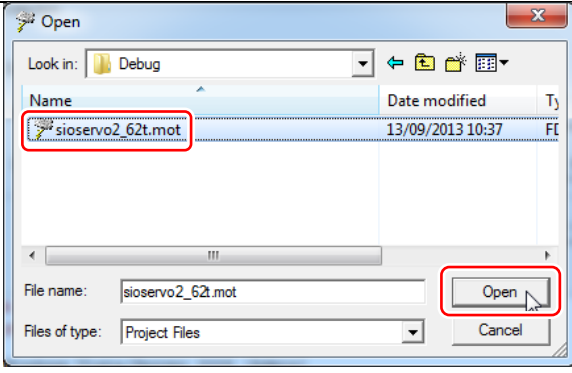
3	<pre> 32 #include "printf_lib.h" // 33 34 /*=====*/ 35 /* Symbol definitions */ 36 /*=====*/ 37 38 /* Constant settings */ 39 #define PWM_CYCLE 24575 // 40 #define SERVO_CENTER 2300 // 41 #define HANDLE_STEP 13 // 42 43 /*=====*/ 44 /* Prototype declaration */ 45 /*=====*/ 46 void init(void); 47 </pre>	<p>Line 40 contains the following definition: SERVO_CENTER 2300</p>
---	---	---

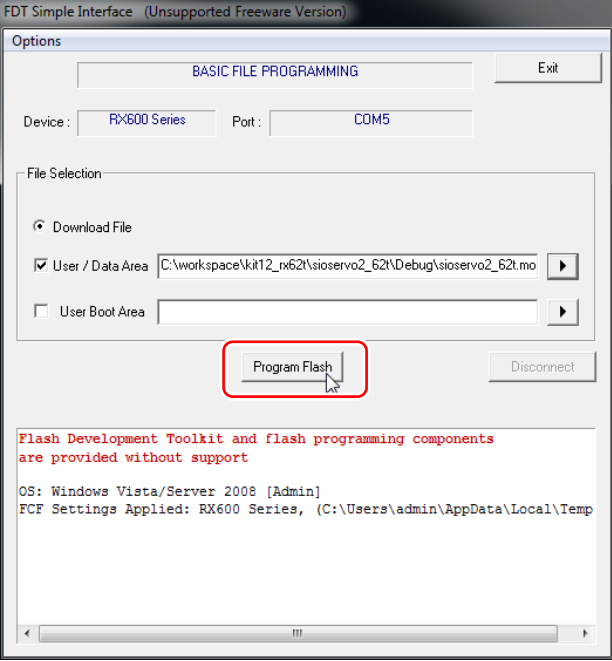
4	<pre> 31 #include "typedefine.h" 32 #include "printf_lib.h" // 33 34 /*=====*/ 35 /* Symbol definitions */ 36 /*=====*/ 37 38 /* Constant settings */ 39 #define PWM_CYCLE 24575 // 40 #define SERVO_CENTER 2202 // 41 #define HANDLE_STEP 13 // 42 43 /*=====*/ 44 /* Prototype declaration */ 45 /*=====*/ 46 void init(void); </pre>	<p>Replace this value with the servo centre value identified earlier. The screenshot shows a value of 2202, as an example.</p>
---	---	---

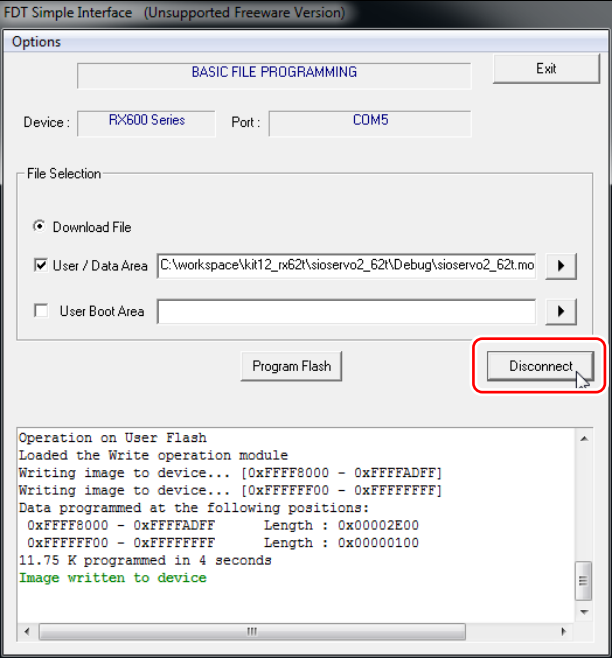
5	 <p>The screenshot shows the 'Build' menu in the IDE. The 'Build' option is highlighted with a red rectangle. The menu also includes options like 'Build All', 'Build Multiple...', 'Clean Current Project', 'Clean All Projects', 'Update All Dependencies', 'Stop Tool Execution', and 'Include/Exclude Build'.</p>	<p>Select Build from the Build menu to generate a MOT file.</p>
---	--	---

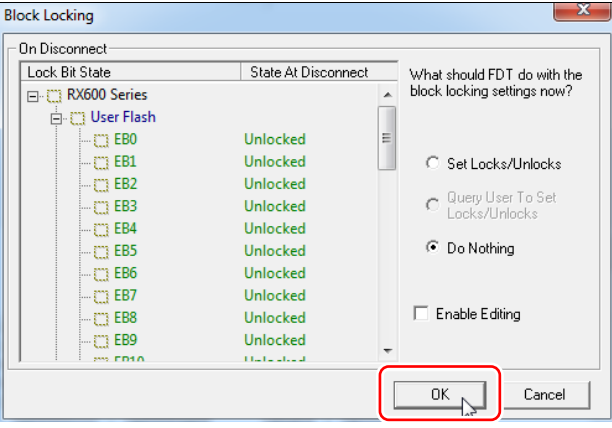
6	 <p>The logo for the Flash Development Toolkit 4.09 Basic, featuring a cloud and a lightning bolt icon.</p>	<p>Launch the Flash Development Toolkit 4.09 Basic.</p>
---	--	--

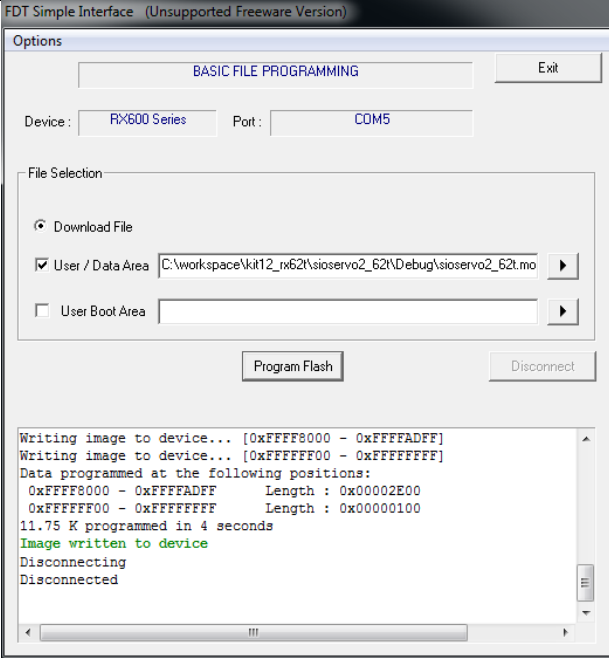
7	 <p>The screenshot shows the 'Options' dialog box in the FDT Simple Interface. The 'User / Data Area' checkbox is checked, and the 'Browse...' button next to it is highlighted with a red rectangle. The dialog also shows device settings and a log window at the bottom.</p>	<p>Check User/Data Area. Then click the triangle on the far right and click Browse.</p>
---	---	---


8	 <p>The screenshot shows a Windows File Explorer window. The file 'sioservo2_62t.mot' is selected and highlighted with a red rectangle. The 'Open' button at the bottom right is also highlighted with a red rectangle.</p>	<p>Open the sioservo2_62t.mot file.</p> <p>The sioservo2_62t.mot file is found in the below folder. "C:\¥Workspace¥kit12_rx62t¥sioservo2_62t¥Debug"</p> <p>Click Open.</p>
---	--	--

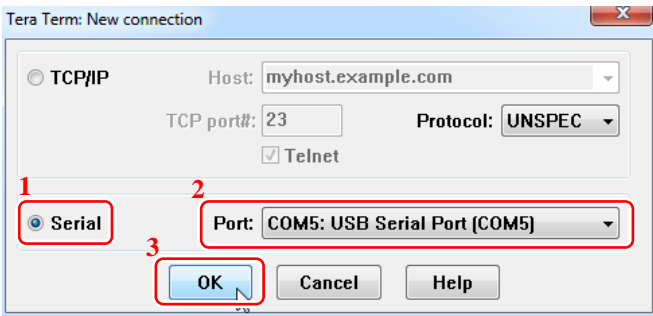
9	 <p>The screenshot shows the 'FDT Simple Interface' window with the 'Options' tab selected. Under 'BASIC FILE PROGRAMMING', the 'Device' is set to 'RX600 Series' and the 'Port' is 'COM5'. In the 'File Selection' section, the 'Download File' checkbox is checked, and the file path is 'C:\workspace\kit12_rx62t\sioservo2_62t\Debug\sioservo2_62t.mo'. The 'Program Flash' button is highlighted with a red box.</p>	<p>Click Program Flash. Then program writing will begin.</p>
---	--	---

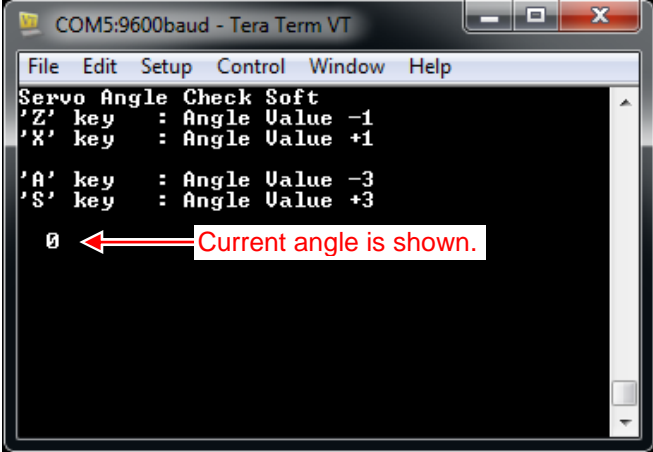
10	 <p>The screenshot shows the 'FDT Simple Interface' window after programming. The 'Program Flash' button is no longer highlighted. The 'Disconnect' button is highlighted with a red box. The status window at the bottom shows the following text: Operation on User Flash Loaded the Write operation module Writing image to device... [0xFFFF8000 - 0xFFFFADFF] Writing image to device... [0xFFFFFFFF00 - 0xFFFFFFFF] Data programmed at the following positions: 0xFFFF8000 - 0xFFFFADFF Length : 0x00002E00 0xFFFFFFFF00 - 0xFFFFFFFF Length : 0x00000100 11.75 K programmed in 4 seconds Image written to device</p>	<p>After programming has finished, click Disconnect.</p>
----	---	---

11	 <p>The screenshot shows the 'Block Locking' dialog box. It has a tree view on the left showing 'Lock Bit State' for 'RX600 Series' and 'User Flash'. The 'State At Disconnect' column shows 'Unlocked' for all bits (EB0-EB9). On the right, under 'What should FDT do with the block locking settings now?', the 'Do Nothing' radio button is selected. The 'OK' button is highlighted with a red box.</p>	<p>Click OK.</p>
----	---	-------------------------

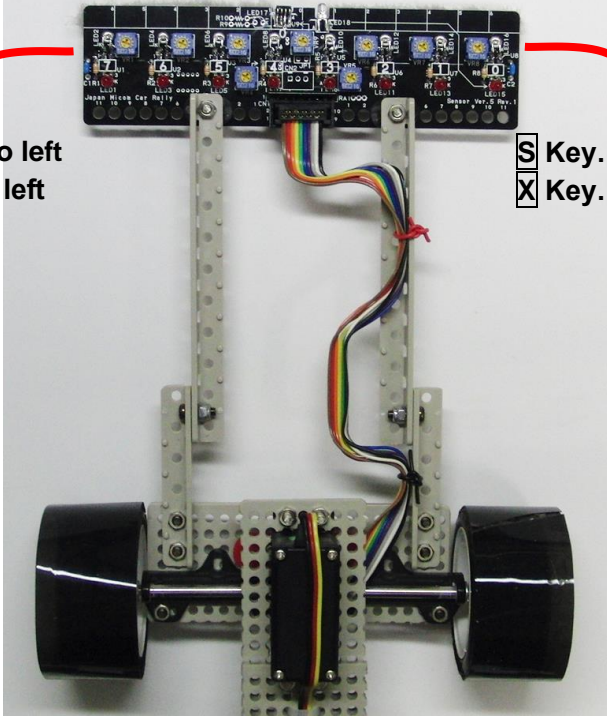
12		<p>Program writing completed.</p> <p>After program writing, following execute.</p> <ol style="list-style-type: none"> [1] Move the two power switches of the MCU car to the off position. [2] Turn SW5 to RUN. <p>Note: keep the USB cable connected.</p>
----	---	---

13		<ol style="list-style-type: none"> [1] Switch on the two power switches of the MCU car. [2] Launch the Tera Term.
----	--	--

14		<ol style="list-style-type: none"> [1] A new connection window appears. Select Serial. [2] For Port select the number with the USB Serial Port indication or the number of the currently connected serial port. [3] Click OK.
----	---	--

15		<p>When you power on the MCU car, the message shown at left is displayed. If this message does not appear, check the cable connection, the batteries of the MCU car, the position of the write switch on the MCU board, the number of the communication port, and that the program you wrote to the MCU is actually <code>sioservo2_62t.mot</code> from the project <code>sioservo2_62t</code>.</p>
----	---	---

16

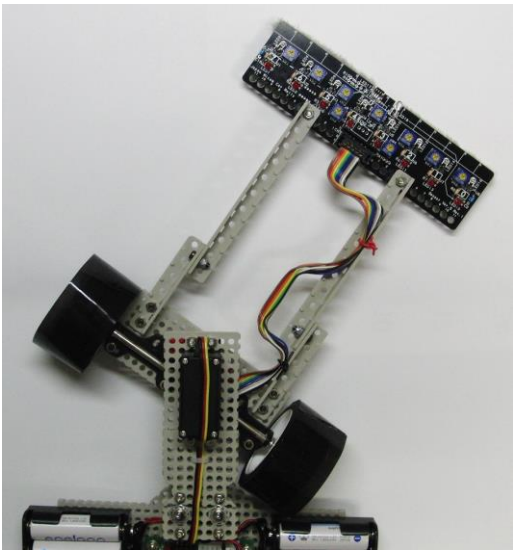


A Key...3 degrees to left
Z Key...1 degree to left

S Key...3 degrees to right
X Key...1 degree to right

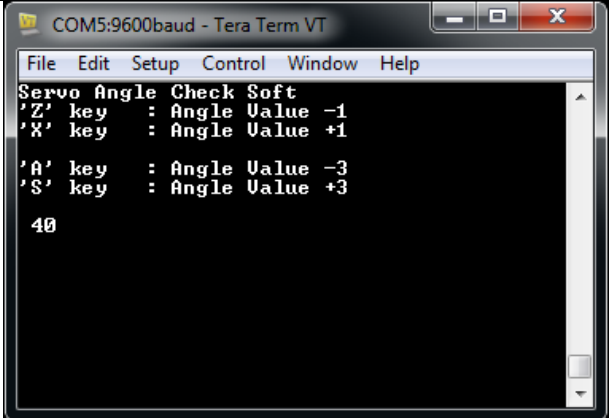
Pressing the A, S, Z, or X key causes the servo to move as indicated. **See how far to the right you can turn the steering wheel. Then do the same for turning to the left.**

17

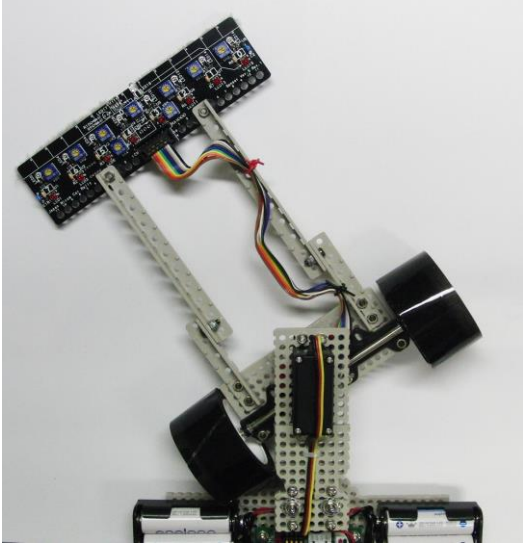


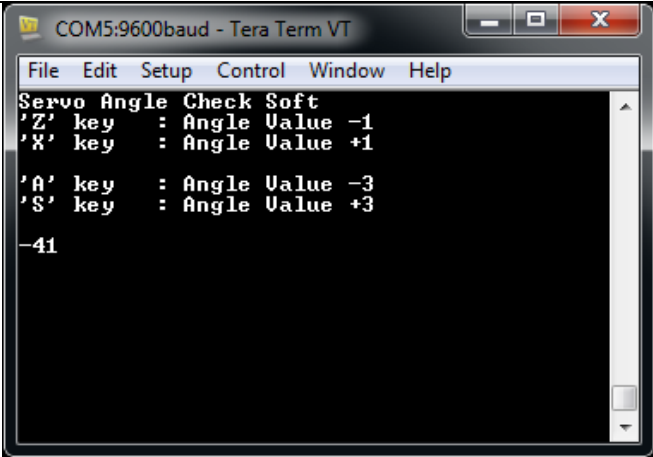
First, use the S and X keys to find the limit on the right. Confirm that the wheels really turn when you press the keys. If the wheel is about to touch the chassis, press the Z key to reduce the turn angle by a small amount.

18



Tera Term displays a numerical value. This is the number of degrees to the right the steering wheel is currently turned. **In this case, the turn angle is 40 degrees.** However, that the maximum right turn angle is 40 does not necessarily mean that the maximum left turn angle is -40. Make sure to test both right and left.

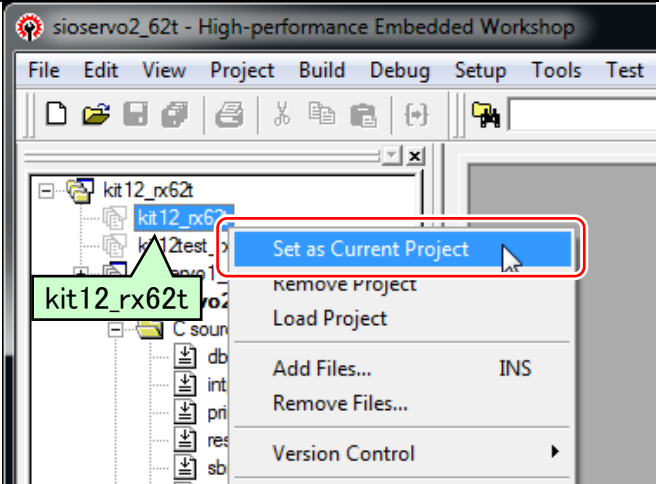
19		<p>Now use the A and Z keys to find the limit on the left. Once again, confirm that the wheels really turn when you press the keys. If the wheel is about to touch the chassis, press the X key to reduce the turn angle by a small amount.</p>
----	---	---

20		<p>Tera Term displays a numerical value. This is the number of degrees to the left the steering wheel is currently turned. In this case, the turn angle is -41 degrees.</p>
----	--	--

6.5. Overwriting the kit12_62t.c Program Code

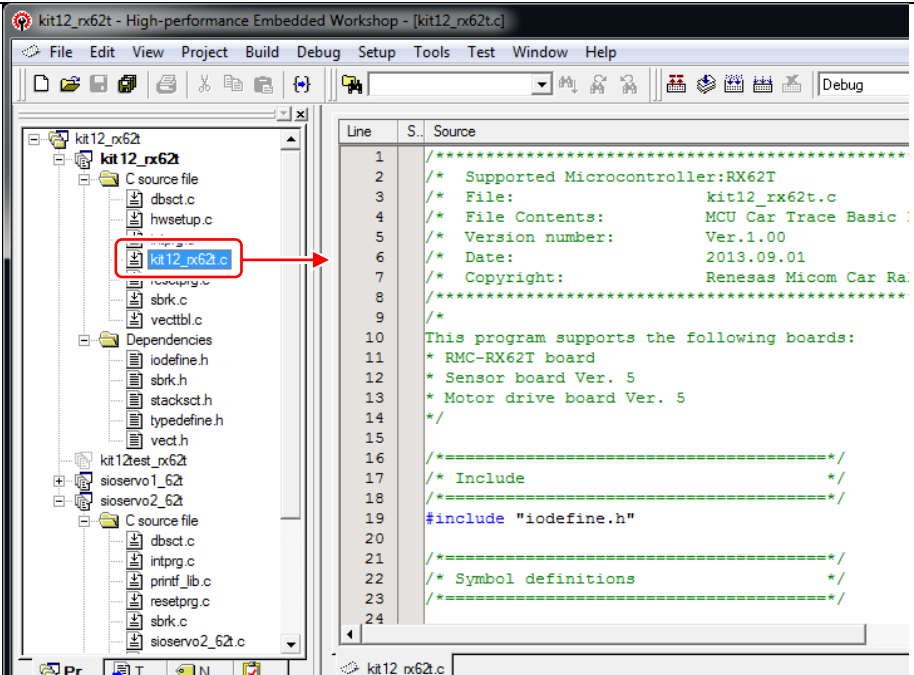
The **sioservo1_62t** and **sioservo2_62t** projects enabled us to determine three numerical values. We must now plug these numerical values into **kit12_rx62t.c**, the program that controls the operation of the MCU car. The source file is part of the **kit12_rx62t** project.

1



Set project **kit12_rx62t** as the current project.

2



Double click **Kit12_rx62t.c** to open the editor window.

Change the three values indicated below to match your own MCU car.

Description	Line number to be modified in kit12_rx62t.c	Kit standard value	Value from present example
Servo centre	Line 27	2300	2202
Max. turn angle left	Line 275	-38	-41
Max. turn angle right	Line 284	38	40

[1]Servo centre

3

```

21  /*=====*/
22  /* Symbol definitions */
23  /*=====*/
24
25  /* Constant settings */
26  #define DWM_CYCLE 24575 /* Motor PWM period (16ms) */
27  #define SERVO_CENTER 2300 /* Servo center value */
28  #define HANDLE_STEP 13 /* 1 degree value */
    
```

[2]Max. turn angle left

```

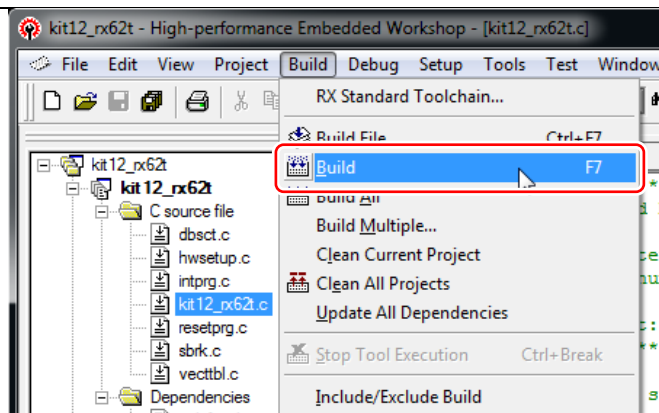
270          case 23:
271              /* Trace, crank detection after cross line */
272              if( sensor_inp(MASK4_4)==0xf8 ) {
273                  /* Left crank determined -> to left crank clearing processing
274                  led_out( 0x1 );
275                  handle( -38 );
276                  motor( 10, 50 );
277                  pattern = 31;
278                  cnt1 = 0;
279                  break;
280              }
    
```

[3]Max. turn angle right

```

281          if( sensor_inp(MASK4_4)==0x1f ) {
282              /* Right crank determined -> to right crank clearing processing
283              led_out( 0x2 );
284              handle( 38 );
285              motor( 50, 10 );
286              pattern = 41;
287              cnt1 = 0;
288              break;
289          }
    
```

4



Select **Build** from the **Build** menu to generate a MOT file.

Launch the FDT, and please transfer a program "kit12_rx62t.mot"

Now kit12_rx62t.c has been adjusted and written to the MCU. Now let's try out the MCU car on the course!

7. Hints on Modifying the Program

7.1. Outline

After building the **kit12_rx62t** program, write it to the RMC-RX62T board and try running the MCU car. Even if no modifications are made to the sample program code (except for changing values such as **SERVO_CENTER** to match the unique characteristics of the MCU car), in most cases the MCU car will probably go off the track at some point.

The explanation up to this point is based on the following assumptions about the course conditions:

- All the sensors respond in the same manner when the colour of the track changes from white to black.
- The MCU car is travelling straight when it encounters crosslines and half lines.
- When travelling straight ahead, the MCU car is tracing roughly the centre of the track.

But in most cases, conditions such as the following occur:

- The response of the sensors is uneven.
- The MCU car encounters crosslines and half lines at something of an angle.
- The MCU car is skewed to the right or left side of the track even when it is travelling straight ahead.

These factors can cause the MCU car to go off the track.

The explanation that follows is based on careful observation of the conditions under which MCU cars go off the track. The results of these observations are presented from the next section onward.

In each of the discussions that follows, the following three topics are treated as a set:

(a) Description of Problem

How did the MCU car go off the track?

(b) Analysis Findings

Why did the MCU car go off the track?

(c) Example Solution

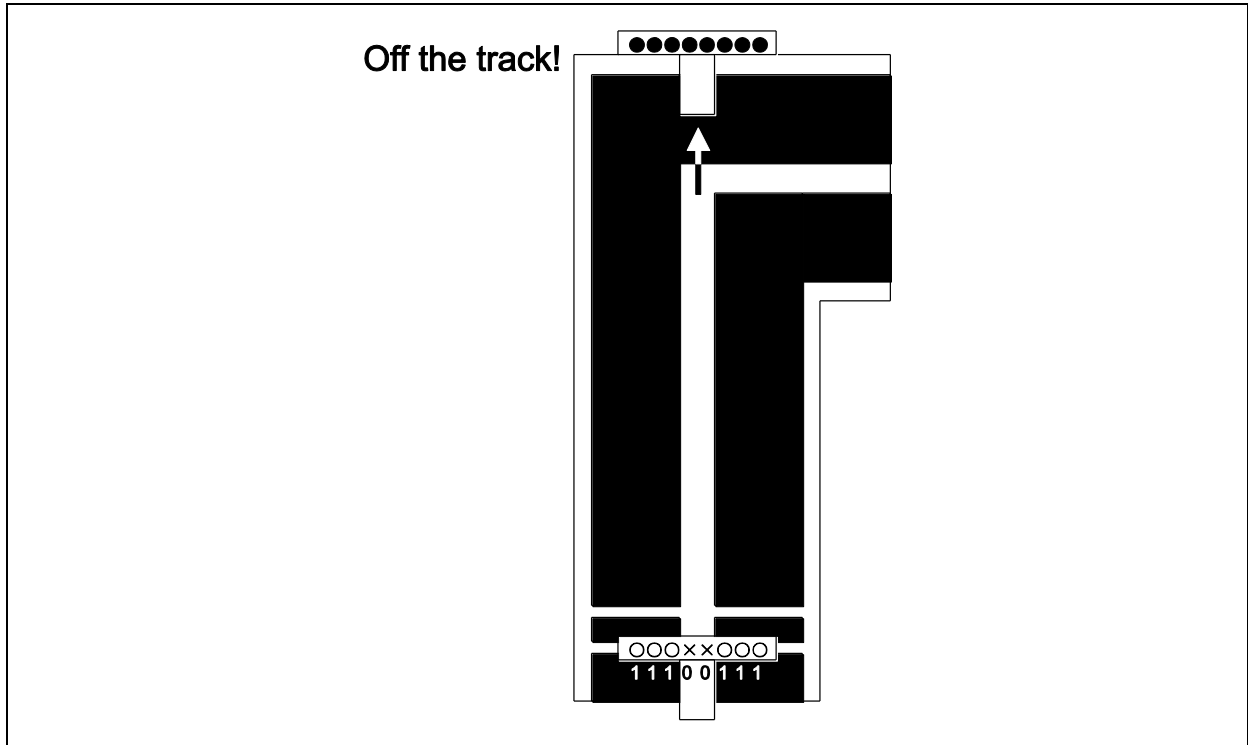
How can the problem be corrected?

7.2. Examples of the MCU Car Going Off the Track

7.2.1. Crossline Not Detected Correctly

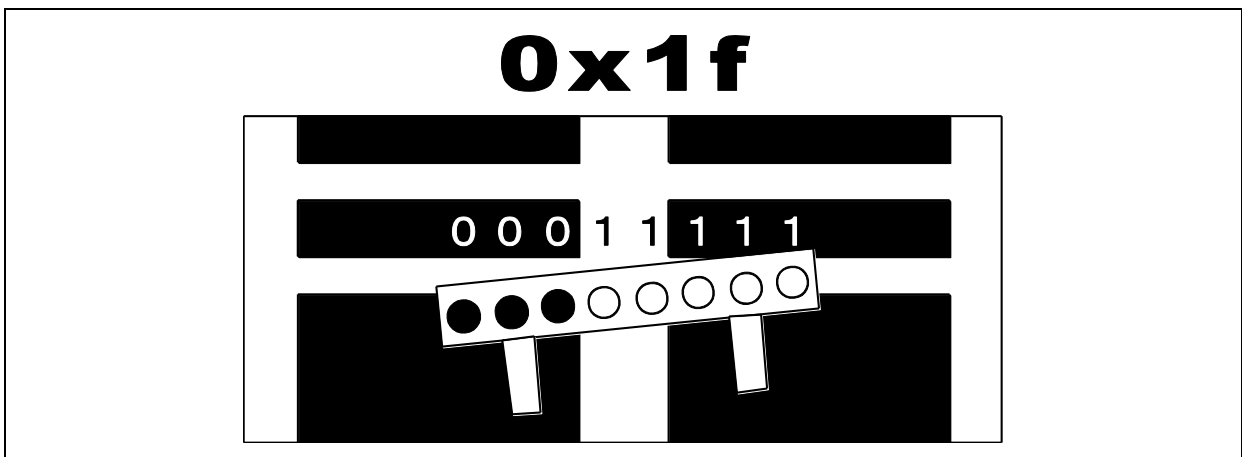
(a) Description of Problem

After the crosslines were encountered, the MCU car failed to turn at the crank and instead continued straight off the track.

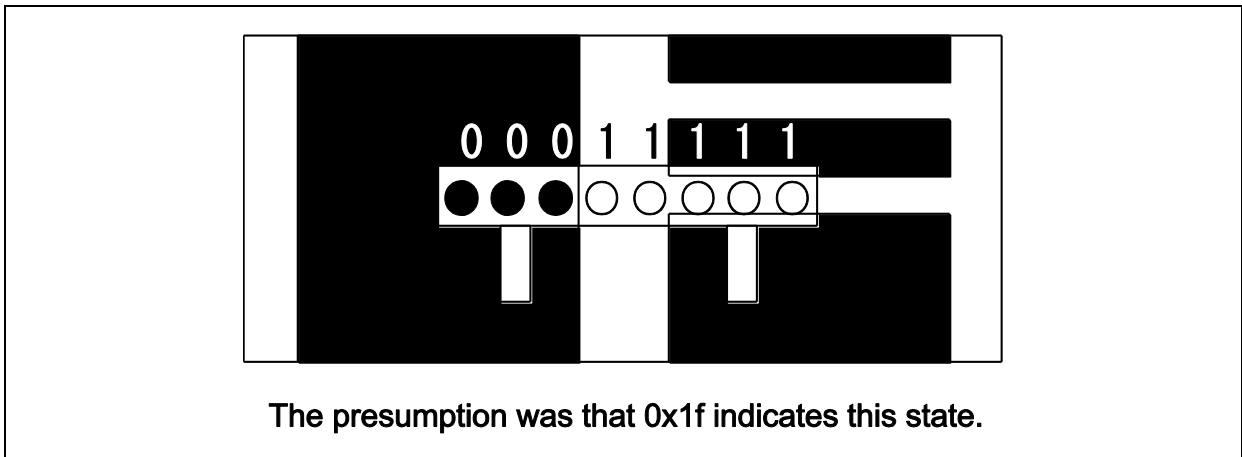


(b) Analysis Findings

Collection and analysis of the running data showed that at the moment of crossline detection, the sensor state was 0x1f rather than the anticipated value of 0xe7. (See following figure.)



A sensor response of 0x1f seems familiar. During right half line detection, using all eight sensors, a value of 0x1f is interpreted as indicating a right half line.



Thus, the sensor data matched that for right half line detection, even though the actual course feature was a crossline, and the result was a malfunction.

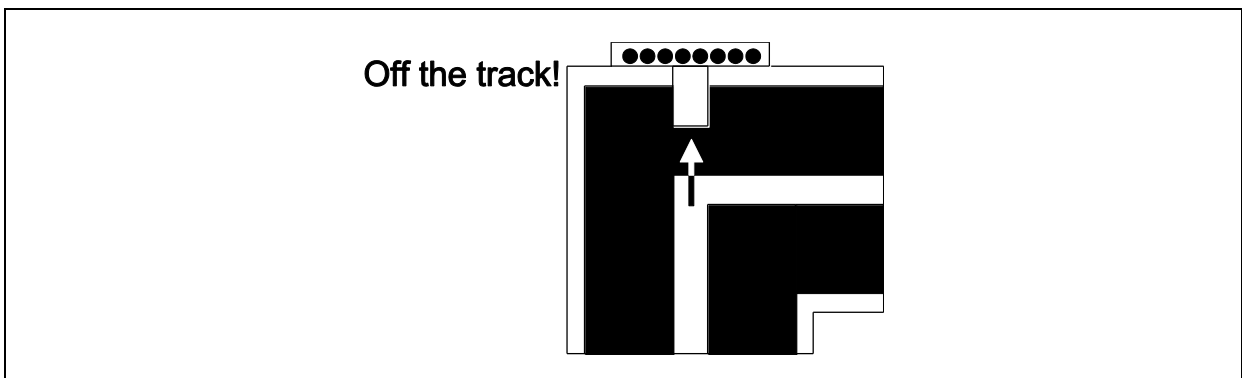
(c) Example Solution

The program code is pretty hopeless in cases where the sensors are oriented at an angle. Even if the servo centre value is aligned properly, it seems that only the sensors on one side or the other register at the moment when the line is encountered. As a possible solution, sensor checking could continue for a short time even when a right half line is detected. Then **the judgment could be changed to “crossline”** in cases where this is appropriate. Left half line detection would be analogous.

7.2.2. Crank Not Detected Correctly

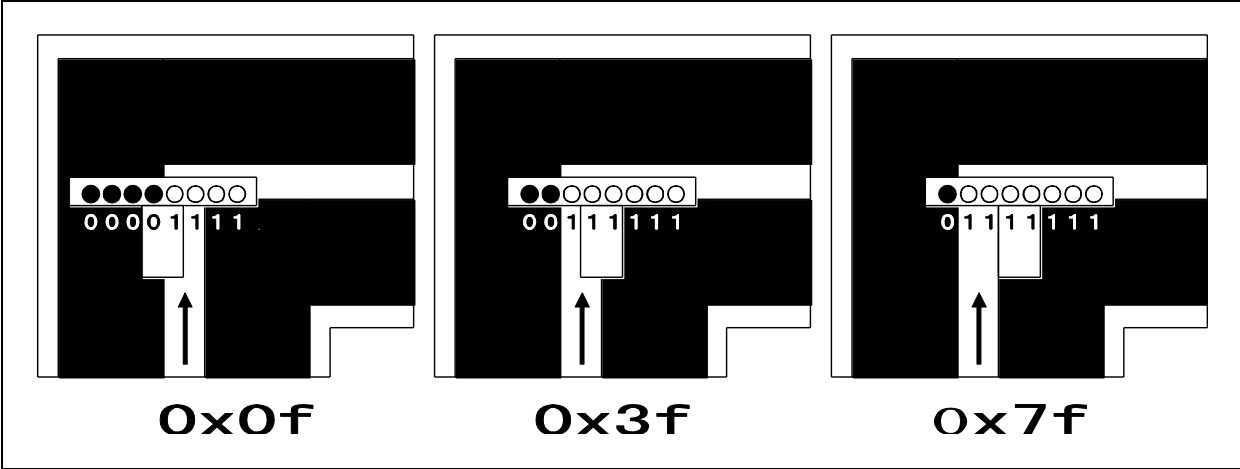
(a) Description of Problem

At the crank, the MCU car failed to turn and instead continued straight off the track. Two LEDs were lit on the motor drive board, so it would seem that pattern 23 was being processed.



(b) Analysis Findings

Collection and analysis of the running data showed that at the moment of right crank detection, the sensor state was 0x0f, 0x3f, or 0x7f rather than the anticipated value of 0x1f. (See following figure.)



Thus, even though a right crank was encountered, the sensor data did not match the anticipated sensor state for a right crank and the MCU car proceeded straight off the track.

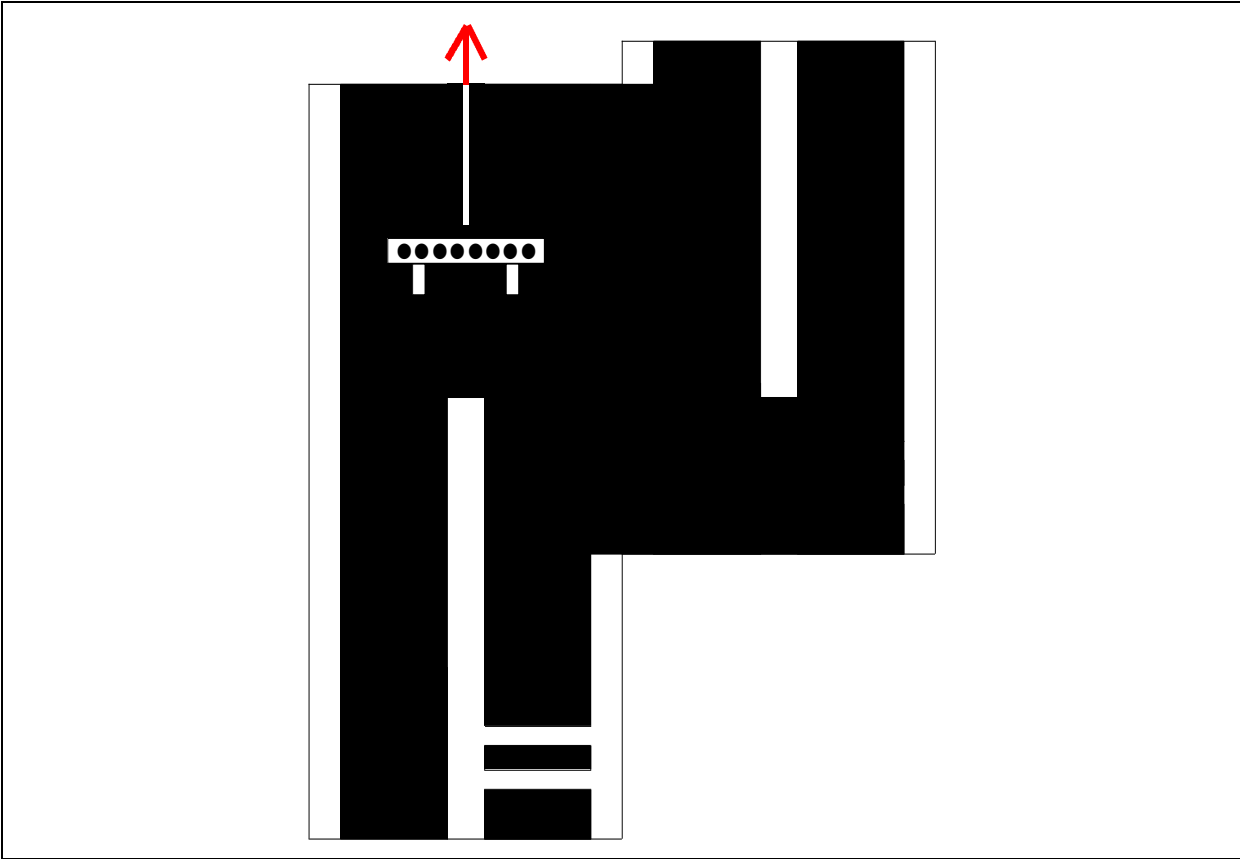
(c) Example Solution

In the program code supplied with the kit, the only sensor state detected as a right crank is 0x1f. In fact, actually encountering a right crank sometimes produces states such as 0x0f, 0x3f, or 0x7f. Therefore, these sensor states should be added as also indicating a right crank. A similar malfunction is likely at left cranks as well, so analogous sensor states should be added for left crank detection.

7.2.3. Half Line Not Detected Correctly

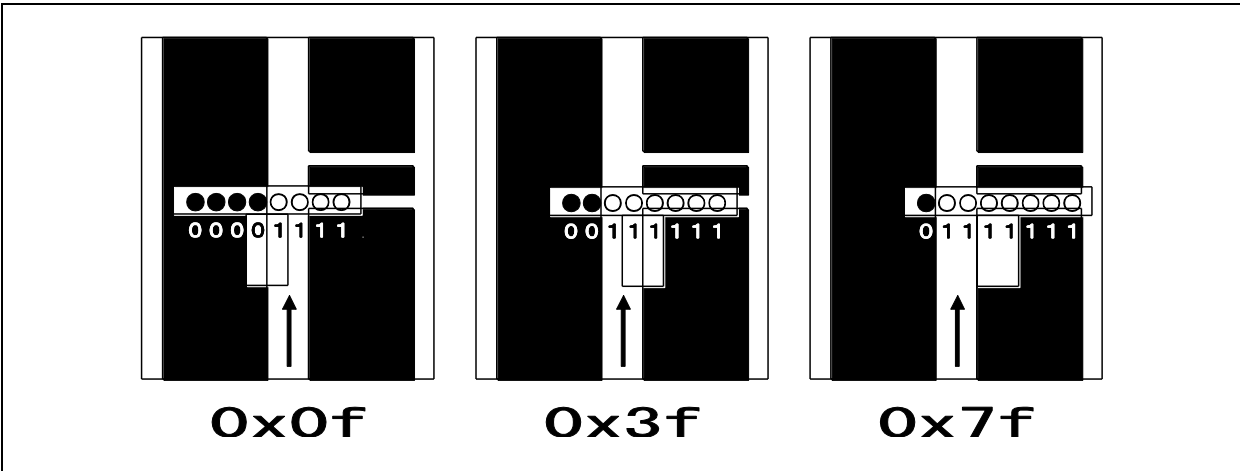
(a) Description of Problem

At a right lane change, the MCU car continued straight ahead and off the track.



(b) Analysis Findings

Collection and analysis of the running data showed that at the moment of right half line detection, the sensor state was 0x0f, 0x3f, or 0x7f rather than the anticipated value of 0x1f. (See following figure.)



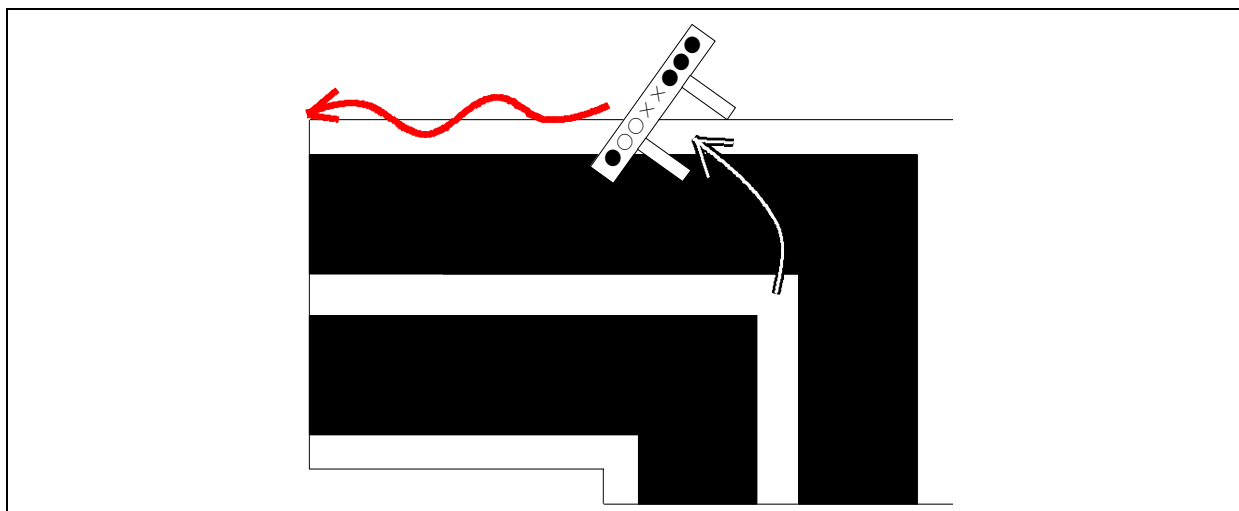
(c) Example Solution

In the program code supplied with the kit, the only sensor state detected as a right half line is 0x1f. In fact, actually encountering a right half line sometimes produces states such as 0x0f, 0x3f, or 0x7f. Therefore, these sensor states should be added as also indicating a right half line. A similar malfunction is likely at left half lines as well, so analogous sensor states should be added for left half line detection.

7.2.4. After Clearing from Crank, MCU Car Mistakes Outer White Line for Center Line and Goes off Track

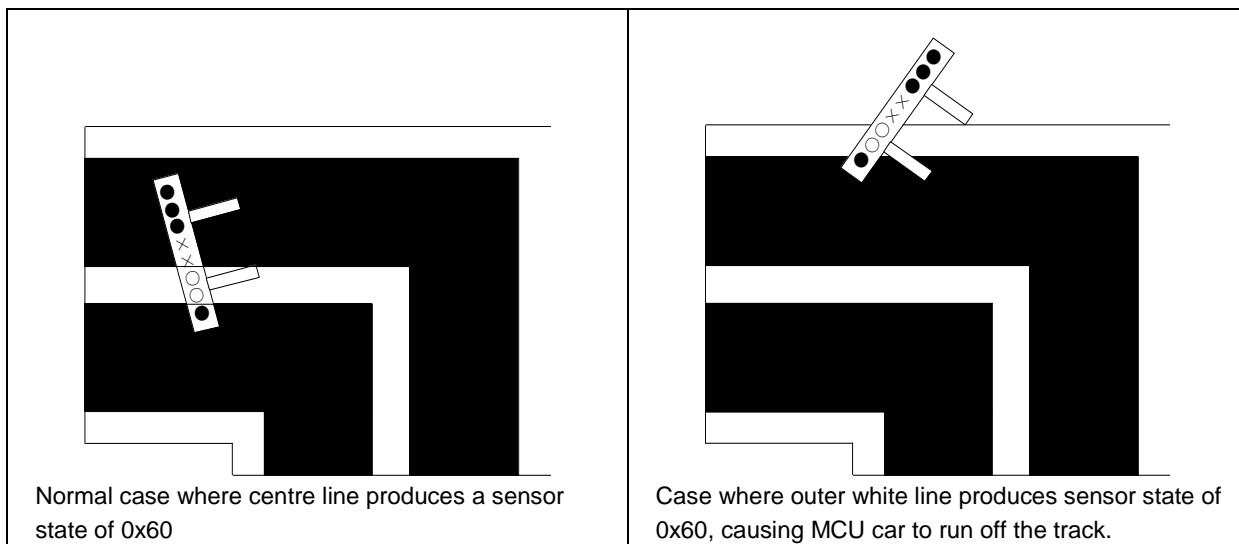
(a) Description of Problem

When a left crank was detected, the steering wheel turned to the left. After turning for a while in a somewhat wide arc, the MCU car started to trace the outer white line and eventually went off the track.



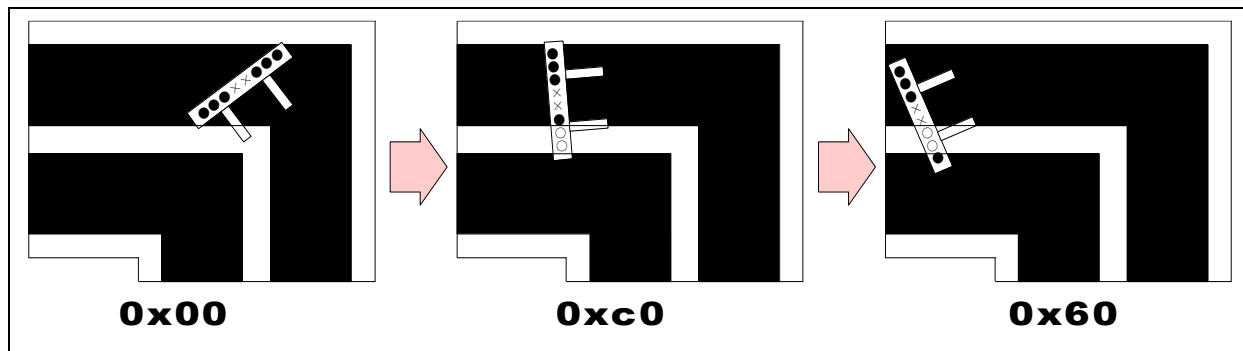
(b) Analysis Findings

Let's assume the settings when a left crank is encountered are left 38 degrees, left motor 10%, right motor 50%. When will the MCU car finish the turn and return to the normal pattern under these conditions? In the sample program code, it is when the sensor state is 0x60. This is assumed to occur when the centre line is detected, as in the figure at left below. But if the MCU car is moving too fast, it may not turn tightly enough and end up at the outer edge of the track, resulting in the outer white line producing a sensor state of 0x60, as in the figure at right below. This will cause the MCU car to return to the normal pattern and run off the track.

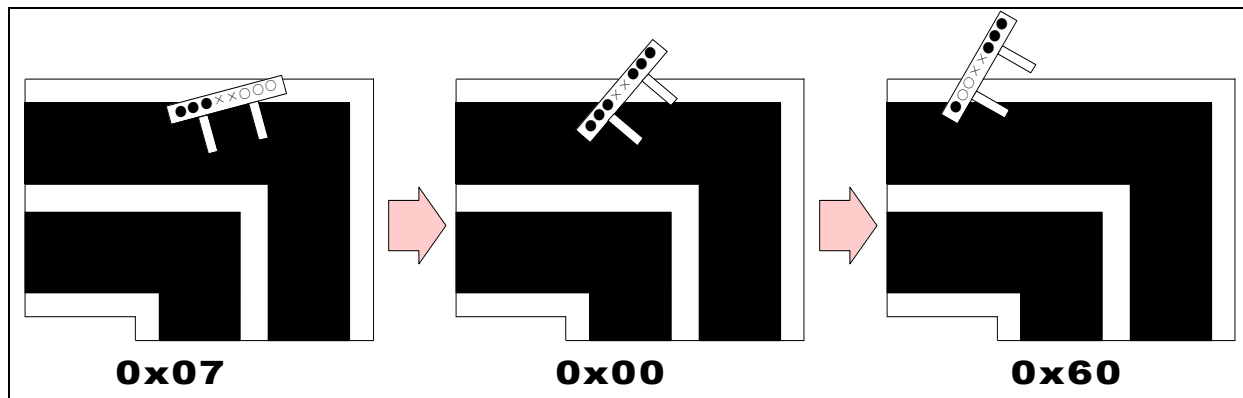


(c) Example Solution

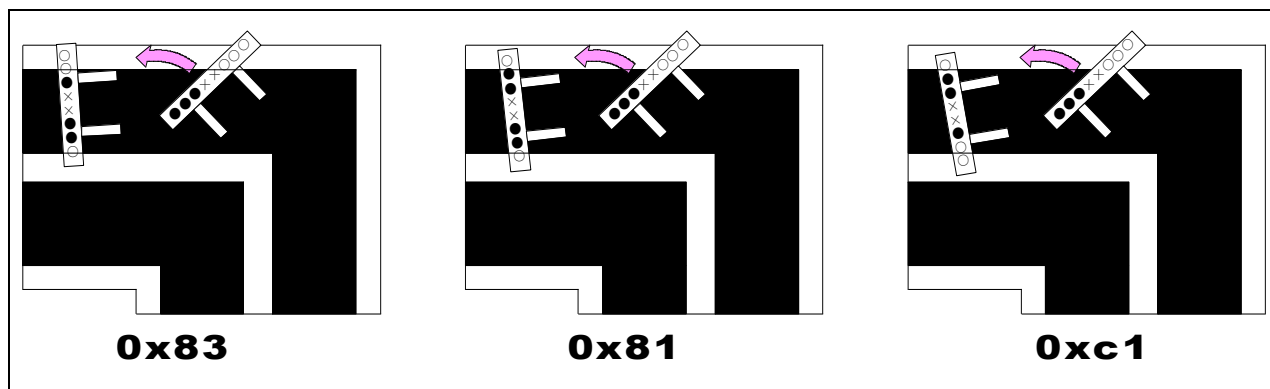
Let's review the changes in the sensor state when the centre line is correctly detected. The sensor state changes from 0x00 to 0xc0 to 0x60, finally returning to normal running operation. (See following figure.)



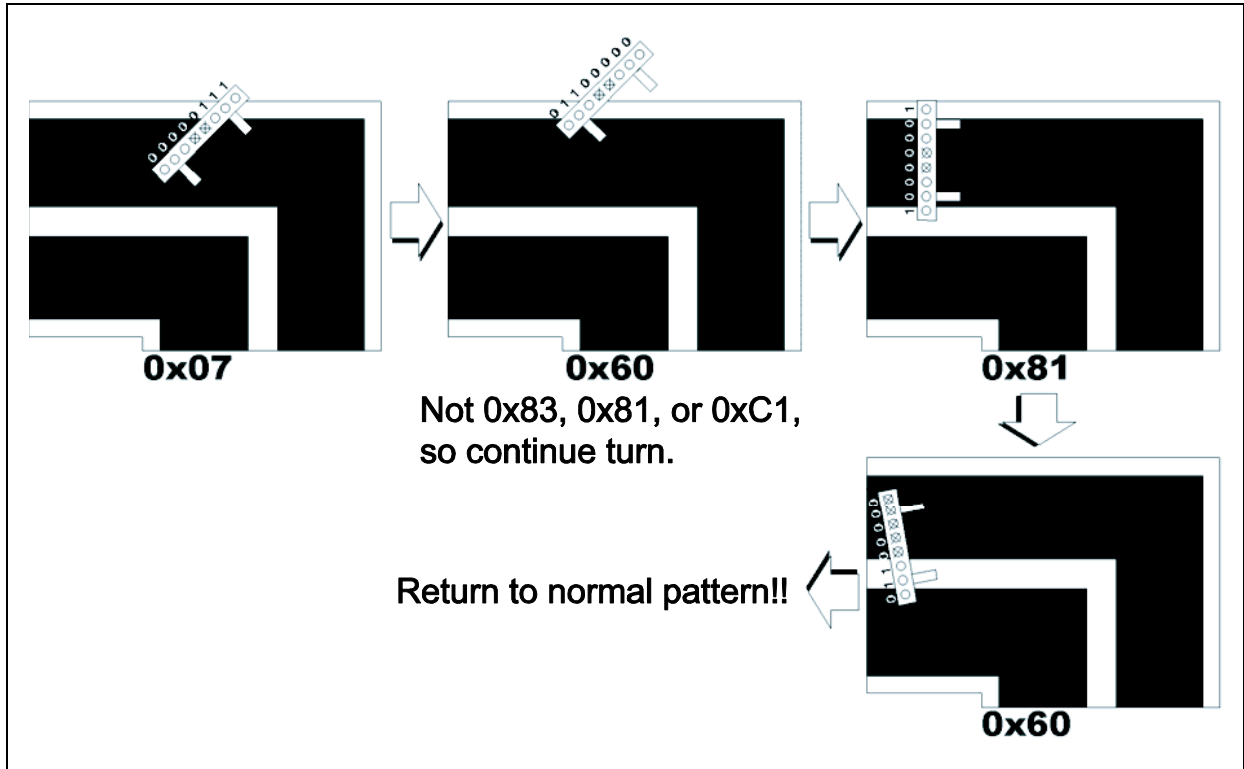
Now let's review the changes in the sensor state when the outer white line is incorrectly detected as the centre line. The sensor state changes from 0x07 to 0x00 to 0x60, finally returning to normal running operation. (See following figure.)



A comparison of the two sequences shows that the malfunction occurs when a sensor state of **0x07** is followed by **0x60**. So how about modifying the program code so that when a sensor state of 0x07 is encountered, turning continues until the state changes to 0x83, 0x81, or 0xc1. (See following figure.)



Let's try a simulation. When a sensor state of 0x07 occurs, turning continues until the state changes to 0x83, 0x81, or 0xc1. This means turning continues even if a sensor state of 0x60 occurs. Previously, control would return to the normal pattern at this point and the MCU car would end up going off the track. Now the turn continues until a sensor state of 0x81 occurs, after which the program checks for a state of 0x60. When the state changes to 0x60, it is judged to be the centre line and control returns to the normal pattern.

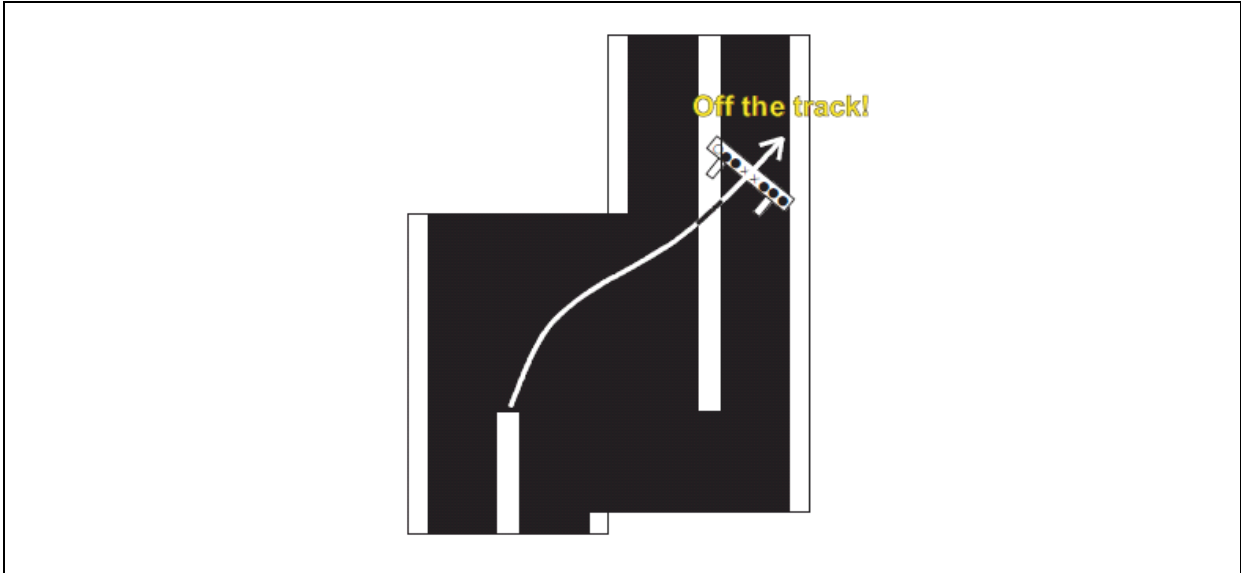


Try modifying the program code based on this idea. This will eliminate cases where malfunctions result from the outer white line being mistakenly detected as the centre line. The right crank is analogous.

7.2.5. End of Lane Change Not Detected Correctly

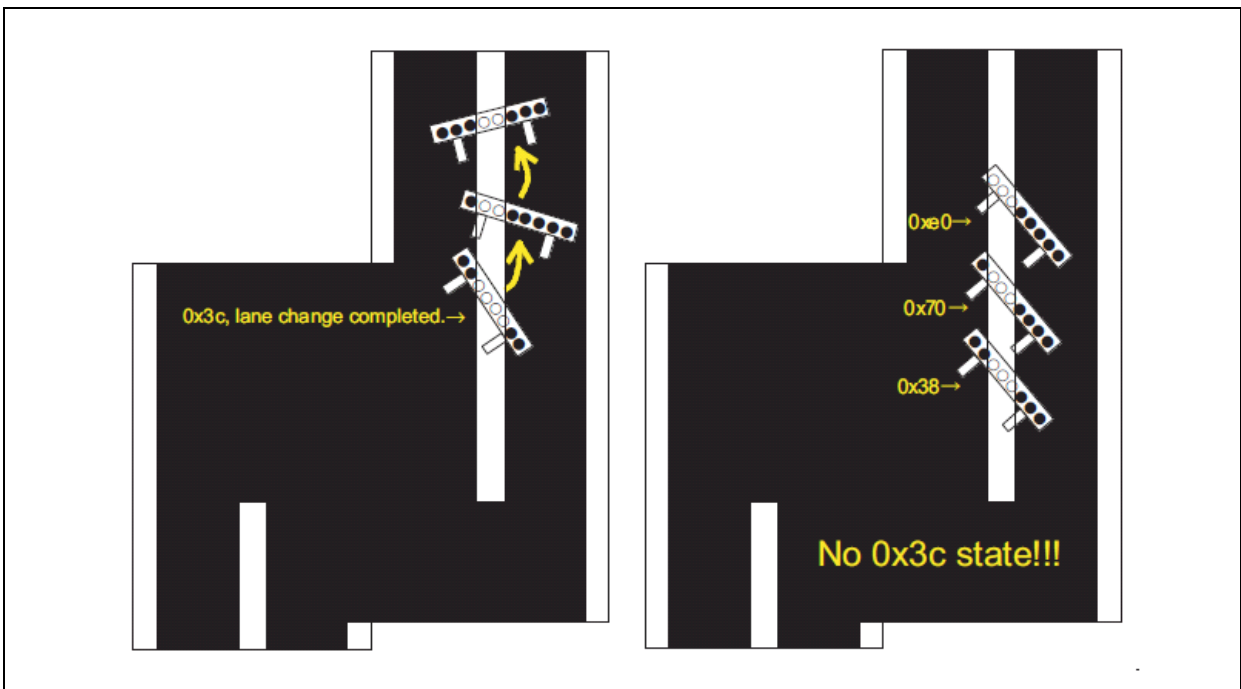
(a) Description of Problem

A right lane change was detected and the MCU car turned to the right. Then, instead of detecting the new centre line and beginning to trace it, the MCU car continued past it and went off the track. (See following figure.)



(b) Analysis Findings

An analysis of the sensor states shows a sequence of 0x38 to 0x70 to 0xe0 when the new centre line was detected. In the sample program code, the right lane change is determined to have completed when the sensor state is 0x3c, checking with all eight sensors (left figure below). But depending on the angle at which the MCU car is proceeding, a sensor state of 0x3c may never occur (right figure below).



(c) Example Solution

To the sensor state 0x3c indicating detection of a new centre line, add the other sensor states identified in the analysis findings. In addition, a variety of other modifications come to mind, such as changing the sensor detection state to a completely different value or switching the servo to a shallower steering angle once the centre line is detected and then proceeding. Try out several different approaches to determine which enables a stable (and quicker) lane change manoeuvre.

7.3. Conclusion

This applies to every case, but to use crossline detection as an example, let's say a sensor state of □□ is used.

- A sensor state of □□ indicates a crossline.
- However, a sensor state of □□ can also occur at a half line.
- As a result, misdetection occurs and the MCU car goes off the track.

In other words, a sensor state match occurs in a place where the program (you) did not expect it, causing the MCU car to go off the track.

The key thing when devising solutions is to discover suitable sensor states for specific situations on your own.

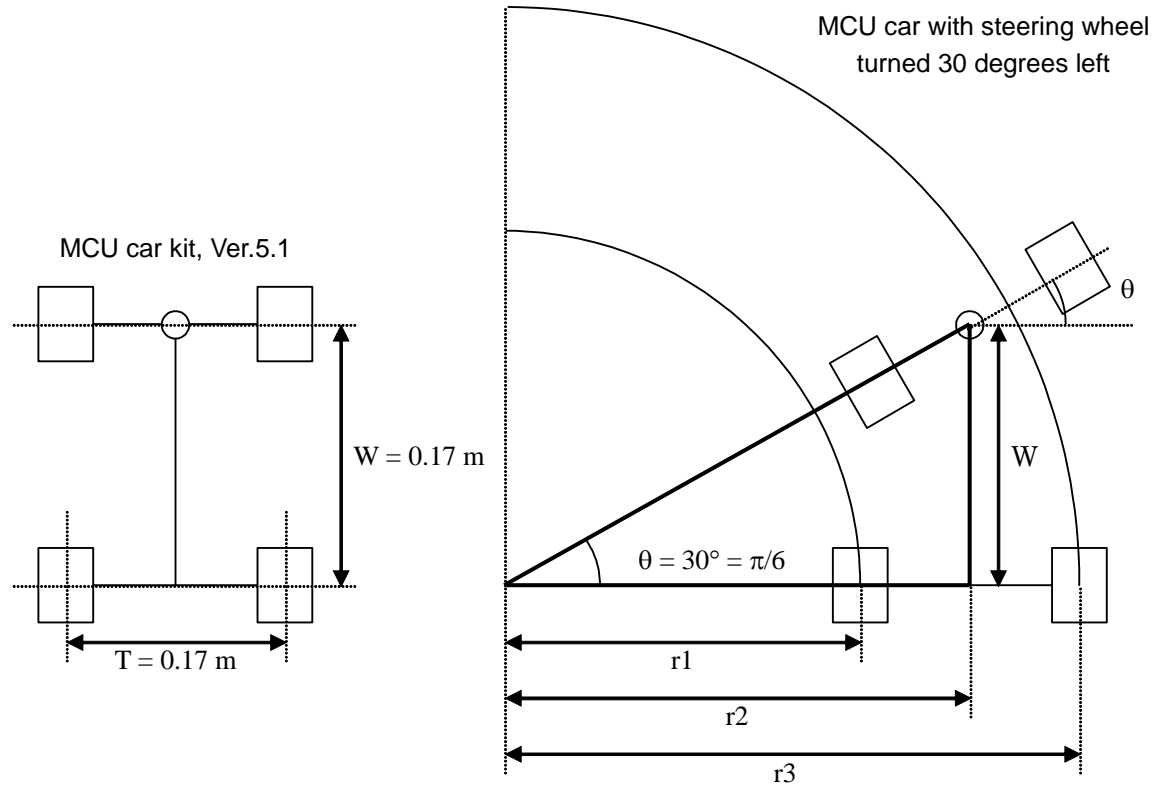
- A sensor state of △△ indicates a crossline.
- A sensor state of △△ does not occur under other circumstances.
- Therefore, there is no danger of malfunction!!

Several example problems are illustrated above, but you may still encounter cases where the MCU car goes off the track after applying the suggestions provided. When this happens, don't just accept it as bad luck. Thoroughly investigate possible causes (involving both hardware and software) and devise countermeasures. The secret to completing the course at the competition is a commitment to patient and steady problem solving, one step at a time.

8. Calculating the Left-Right Motor Speed Differential

8.1. Calculation Method

When the steering wheel is turned, the inside and outside wheels turn at different speeds. The method of calculating this speed differential is described below:



T = tread: The distance from the centre line of the left and right wheels. This is 0.17 [m] in the case of the kit.

W = wheelbase: The distance from the front to the rear wheels. This is 0.17 [m] in the case of the kit.

As shown in the figure, the following triangle is formed between the base r_2 , the height W , and the angle θ :

$$\tan\theta = W / r_2$$

We know the angle θ and W , so we can calculate r_2 as follows:

$$r_2 = W / \tan\theta = 0.17 / \tan(\pi / 6) = 0.294 \text{ [m]}$$

The radius r_1 of the inside wheel is as follows:

$$r_1 = r_2 - T / 2 = 0.294 - 0.085 = 0.209$$

The radius r_3 of the outside wheel is as follows:

$$r_3 = r_2 + T / 2 = 0.294 + 0.085 = 0.379$$

Therefore, if the rotational speed of the outside wheel is 100, that of the inside wheel is:

$$r_1 / r_3 \times 100 = 0.209 / 0.379 \times 100 = 55$$

When the steering wheel is turned 30 degrees left,
the rotational speed of the left wheel is 55 relative to a right wheel rotational speed of 100.

The following code can be used to ensure that the left and right wheels turn at a speed at which no loss occurs:

```
handle( -30 );  
motor( 55, 100 );
```