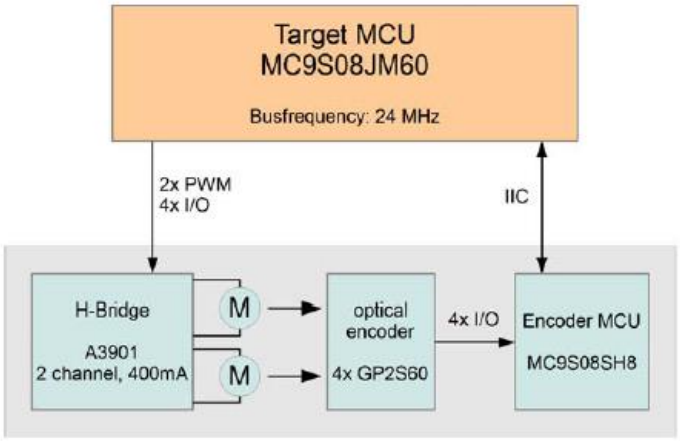
8: IIC-Bussystem / PID

Sie verstehen das IIC-Busprotokoll und können das IIC-Controller Modul des MC9S08JM60 zur Kommunikation mit anderen IIC-Busteilnehmern einsetzen.

1. Geschwindigkeitsmessung im MC-Car

Abbildung 1 zeigt das Prinzip der Motoransteuerung und Geschwindigkeitsmessung durch die Traget MCU. Details zur Adressierung der Encoder MCU über den IIC-Bus finden Sie im Encoder_Datenblatt.pdf.



2. IIC-Bussystem

Legen Sie in CW ein neues C-Projekt mit Copy/Paste an und nutzen Sie das gegebene File main.c als Hauptdatei. Fügen Sie in der Bibliothek MC_Library zu den Verzeichnissen Lib_Headers bzw. Lib_Sources die gegebenen Files error.h, i2c.h und quad.h bzw. quad_temp.c und i2c_temp.c zu.

a) Analysieren Sie den gegebenen Code und implementieren Sie im File i2c_temp.c die IICInitialisierung sowie die beiden zusammengesetzten IIC-Funktionen unter Verwendung der IICGrundfunktionen (siehe CW Task-View).
   Testen Sie Ihre Implementierung in dem Sie die Räder des MC-Cars drehen (von Hand oder über die Fernbedienung) und im Debugger die Werte der globalen Variable speed beobachten.
b) Implementieren Sie dann im File quad_temp.c die beiden Funktionen zum Auslesen und Rücksetzen der Tick-Counter des Encoders.
   Testen Sie Ihre Implementierung in dem Sie die Räder des MC-Cars langsam drehen und im Debugger die Werte der globalen Variable ticks beobachten.

i2c.c

[…]

```
//---------------------------------------------------------------------------------------------
// I 2 C   -   Z U S A M M E N G E S E T Z T E   F U N K T I O N E N
//---------------------------------------------------------------------------------------------


/**
 * Prüft, ob ein I2C-Device antwortet.
 *
 * @param [in] ucAdr
 *      die I2C-Adresse des gewünschten Gerätes
 *
 * @return
 *      EC_SUCCESS          falls der Slave immer mit ACK geantwortet hat
 *      EC_I2C_NO_ANSWER    falls I2C-Gerät nicht geantwortet hat
 */
tError i2cTest(uint8 adr)
{
  tError result;

  result = i2cStart(adr, FALSE);
  if (result != EC_SUCCESS) return result;

  i2cStop();
  return EC_SUCCESS;
}


 /**
 * Liest Daten von einem I2C-Gerät dass zusätzlich ein
 * Befehls-Byte benötigt.
 *
 * @param [in] adr
 *      die I2C-Adresse des gewünschten Gerätes
 * @param [in] cmd
 *      der gewünschte Befehl (Laut Datenblatt vom Slave)
 * @param [out] data
 *      die zu sendenden Daten
 * @param [in] length
 *      die Anzahl Bytes, die gesendet werden.
 *
 * @return
 *      EC_SUCCESS          falls der Slave immer mit ACK geantwortet hat
 *      EC_I2C_NAK          falls kein ACK empf. wurde.
 *      EC_I2C_NO_ANSWER    falls I2C-Gerät nicht geantwortet hat
 */
tError i2cReadCmdData(uint8 adr, uint8 cmd, uint8 *data, uint8 length)
{
  tError result;

  result = i2cStart(adr, TRUE);             // adr senden -> WRITE
  if (result != EC_SUCCESS) return result;  // bei einem Error abbrechen

  result = i2cSendData(&cmd, 1);            // cmd senden
  if (result != EC_SUCCESS) return result;  // bei einem Error abbrechen

  result = i2cRepeatedStart(adr, FALSE);    // adr senden -> READ
  if (result != EC_SUCCESS) return result;  // bei einem Error abbrechen

  i2cReceiveData(data, length);             // Daten empfangen (i2cStop wird automatisch aufgerufen)

  return EC_SUCCESS;                        // Success
}
```

```
/**
 * Sendet Daten zu einem I2C-Gerät, das zusätzlich ein
 * Befehls-Byte benötigt.
 *
 * @param [in] adr
 *      die I2C-Adresse des gewünschten Gerätes
 * @param [in] cmd
 *      der gewünschte Befehl (Laut Datenblatt vom Slave)
 * @param [in] data
 *      die zu sendenden Daten
 * @param [in] length
 *      die Anzahl Bytes, die gesendet werden.
 *
 * @return
 *      EC_SUCCESS          falls der Slave immer mit ACK geantwortet hat
 *      EC_I2C_NAK          falls kein ACK empf. wurde.
 *      EC_I2C_NO_ANSWER    falls I2C-Gerät nicht geantwortet hat
 */
tError i2cWriteCmdData(uint8 adr, uint8 cmd, uint8 *data, uint8 length)
{
  tError result;

  result = i2cStart(adr, TRUE);             // adr senden (write)
  if (result != EC_SUCCESS) return result;  // bei einem Error abbrechen

  result = i2cSendData(&cmd, 1);            // cmd senden
  if (result != EC_SUCCESS) return result;  // bei einem Error abbrechen

  result = i2cSendData(data, length);       // data senden
  if (result != EC_SUCCESS) return result;  // bei einem Error abbrechen

  i2cStop();                                // Stop i2C (free i2c Bus)
  return EC_SUCCESS;                        // Success
}


/**
 * Initialisiert den I2C-Bus
 */
void i2cInit()
{
  // enable I2C with 400 kHz SCL clock frequency
  // Bus speed = 24MHz
  // 24'000 / 400 => 60 -> 2 * 30
  IICF_MULT = 0x01;     // 0x01 mul = 02
  IICF_ICR  = 0x05;     // 0x05 icr = 30

  IICC_IICEN = 1;       // I2C einschalten
}
```

quad.c

```
#include "platform.h"
#include "quad.h"
#include "i2c.h"


#define I2C_QUAD_ADR            0x54

#define QUAD_STATUS_CONTROL     0
#define QUAD_SPEED              1
#define QUAD_TICKS              5
#define QUAD_ERRORS             9
#define QUAD_CARRIER_MODULO    11


typedef union                           // Register 00: control/status
{
  unsigned char Byte;
  struct
  {
    tQuadMode encMode           :2;     // Bit[0-1]: Encoder Mode: 0=off, 1=Encoder, 2=Calibration
    uint8 encResetTicks         :1;     // Bit[2]: 1 = resets ticks left & right
    uint8 encResetError         :1;     // Bit[3]: 1 = resets error counter
    uint8 carrierEnable         :1;     // Bit[4]: 0=off, 1=on
    uint8 oledEnable            :1;     // Bit[5]: 0=off, 1=on
    uint8 reserved              :1;     // -
    uint8 power                 :1;     // Bit[7]: 1=power off immediately
  } Bits;
} tQuadStatusControlReg;


typedef union
{
  struct
  {
    tQuadStatusControlReg statusControl;
    int16 speedL;
    int16 speedR;
    int16 ticksL;
    int16 ticksR;
    uint8 errorL;
    uint8 errorR;
    uint16 carrierModulo;
  } tFields;

  char data[sizeof(tFields)];
} tQuadReg;



/**
 * Returns the operating mode
 *
 * @return
 *    a tQuadMode enum value.
 */
tQuadMode quadGetMode(void)
{
  tQuadStatusControlReg sc;
  tError result = i2cReadCmdData(I2C_QUAD_ADR, QUAD_STATUS_CONTROL, &sc.Byte,
sizeof(tQuadStatusControlReg));

  if (result == EC_SUCCESS) return sc.Bits.encMode;
  return qmUnkownError;
}
```

```
/**
 * Sets the operating mode:
 * - 0 => Off (power saving mode)
 * - 1 => normal operating mode
 * - 2 => calibration mode
 * - 3 => reserved
 *
 * @param [in] mode
 *    a pointer to a tQuadMode structure.
 *
 * @return
 *    EC_SUCCESS if the command was executed sucessfully
 *    EC_I2C_NAK if the device answered with an NAK
 *    EC_I2C_NO_ANSWER if the device did not answer (wrong address?)
 */
tError quadSetMode(tQuadMode mode)
{
  tQuadStatusControlReg sc;
  tError result = i2cReadCmdData(I2C_QUAD_ADR, QUAD_STATUS_CONTROL, &sc.Byte,
sizeof(tQuadStatusControlReg));

  if (result == EC_SUCCESS)
  {
    sc.Bits.encMode = (uint8)mode;
    result = i2cWriteCmdData(I2C_QUAD_ADR, QUAD_STATUS_CONTROL, (char*)&sc,
sizeof(tQuadStatusControlReg));
  }
  return result;
}


/**
 * Returns the speed of the left and right wheel.
 * The Speed is in mm/sec.
 *
 * @param [out] speed
 *    a pointer to a tQuadSpeed structure that includes both speed values
 *
 * @return
 *    EC_SUCCESS if the command was executed sucessfully
 *    EC_I2C_NAK if the device answered with an NAK
 *    EC_I2C_NO_ANSWER if the device did not answer (wrong address?)
 */
tError quadGetSpeed(pQuadSpeed speed)
{
  return i2cReadCmdData(I2C_QUAD_ADR, QUAD_SPEED, (char*)speed, sizeof(tQuadSpeed));
}


/**
 * Returns both ticks counter (left and right)
 *
 * @param [out] ticks
 *    a pointer to a tQuadTicks structure that includes both counters
 *
 * @return
 *    EC_SUCCESS if the command was executed sucessfully
 *    EC_I2C_NAK if the device answered with an NAK
 *    EC_I2C_NO_ANSWER if the device did not answer (wrong address?)
 */
tError quadGetTicks(pQuadTicks ticks)
{
  return i2cReadCmdData(I2C_QUAD_ADR, QUAD_TICKS, (char*)ticks, sizeof(tQuadTicks));
}
```

```
/**
 * Resets both ticks counter to zero (left and right).
 *
 * @return
 *    EC_SUCCESS if the command was executed sucessfully
 *    EC_I2C_NAK if the device answered with an NAK
 *    EC_I2C_NO_ANSWER if the device did not answer (wrong address?)
 */
tError quadResetTicks(void)
{
  tQuadStatusControlReg sc;

  tError result = i2cReadCmdData(I2C_QUAD_ADR, QUAD_STATUS_CONTROL, &sc.Byte,
sizeof(tQuadStatusControlReg));

  if (result == EC_SUCCESS)
  {
    sc.Bits.encResetTicks = TRUE;
    result = i2cWriteCmdData(I2C_QUAD_ADR, QUAD_STATUS_CONTROL, (char*)&sc,
sizeof(tQuadStatusControlReg));
  }

  return result;
}


/**
 * Returns both ticks error counter (left and right)
 *
 * @param [out] errors
 *    a pointer to a tQuadErrors structure that includes both error counters
 *
 * @return
 *    EC_SUCCESS if the command was executed sucessfully
 *    EC_I2C_NAK if the device answered with an NAK
 *    EC_I2C_NO_ANSWER if the device did not answer (wrong address?)
 */
tError quadGetErrors(pQuadErrors errors)
{
  return i2cReadCmdData(I2C_QUAD_ADR, QUAD_ERRORS, (char*)errors, sizeof(tQuadErrors));
}


/**
 * Resets both ticks error counter to zero.
 *
 * @return
 *    EC_SUCCESS if the command was executed sucessfully
 *    EC_I2C_NAK if the device answered with an NAK
 *    EC_I2C_NO_ANSWER if the device did not answer (wrong address?)
 */
tError quadResetErrors(void)
{
  tQuadStatusControlReg sc;

  tError result = i2cReadCmdData(I2C_QUAD_ADR, QUAD_STATUS_CONTROL, &sc.Byte,
sizeof(tQuadStatusControlReg));

  if (result == EC_SUCCESS)
  {
    sc.Bits.encResetError = TRUE;
    result = i2cWriteCmdData(I2C_QUAD_ADR, QUAD_STATUS_CONTROL, &sc.Byte,
sizeof(tQuadStatusControlReg));
  }

  return result;
}
```