

Microcontrollers

ApNote

AP0820

☒ : Additional file
AP082001.EXE available

8-Bit C500 Family

The CAN Controller in the C515C

The **C**ontroller **A**rea **N**etwork (CAN) module which has been implemented in the Siemens C515C microcontroller allows communication between several stations (CAN nodes). This document describes the CAN functionality, the initialisation and the use of the CAN module. Furthermore, examples concerning interrupt generation and error handling will be given.

Author : Dr. Jens Barrenscheen / HL MC PD Microcontroller Product Definition

Contents	Page
1 Principle of this CAN application	3
2 CAN Initialisation	4
3 Definition of a Message Object	5
4 Interrupt Handling	7
5 Sending a Message	8
6 Error Handling	9
7 Busoff State and CAN Re-Initialisation	10
Appendix	13
A CANREG.H	13
B INTC515C.H	17
C REGC515C.H	17

AP0820 ApNote - Revision History		
Actual Revision : 12.96		Previous Revision : none (Original Version)
Page of actual Rev.	Page of prev.Rel.	Subjects (changes since last release)

1 Principle of this CAN application

The on-chip CAN-controller provides all features of **FULL-CAN** controllers, such as message management and acceptance filtering (by input masks) in order to minimise CPU load. The **BASIC-CAN** functionality with only one message object (nr. 15) is available, too. This device supports the standard CAN protocol (**specification 2.0 A, 11 bit identifier**), as well as the extended CAN protocol (**specification 2.0 B active, 29 bit identifier**).

All CAN nodes are connected in parallel to the two-wire CAN-bus (CAN_H and CAN_L). The C515C is connected to an external CAN-bus transceiver by the signal TxDC and RxDC, the principle is shown in **figure 1**.

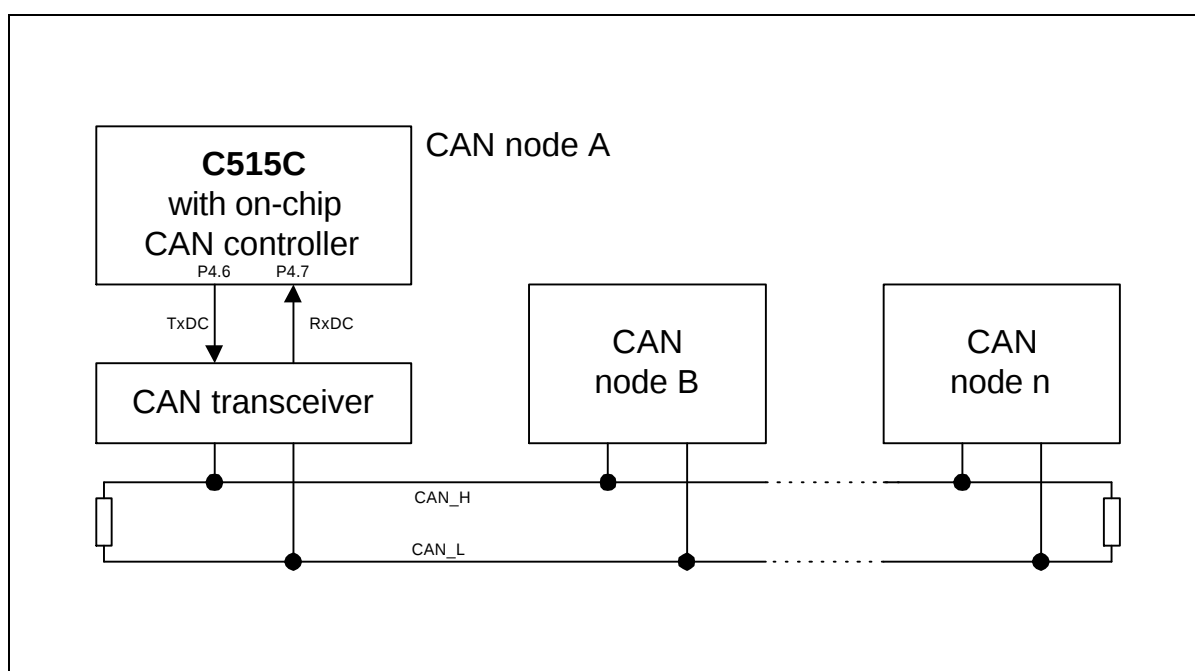


Figure 1 :
Connection of the Siemens C515C microcontroller to the CAN bus

Fifteen different message objects can be used independently by the Siemens C515C microcontroller. Each one has its own specific identifier; **two message objects with identical identifiers are not allowed !**

Two different types of objects can be defined, transmit objects and receive objects. Transmit objects contain data with the data length programmable from 1 to 8 bytes. They are transmitted as soon as the CAN bus is idle after setting of the corresponding transmit request flag. Receive objects are used to store data of incoming transmit objects with matching identifier. A transmission request concerning receive objects causes the transmission of a remote frame in order to request data transfer from another CAN node with identical identifier.

2 CAN Initialisation

The initialisation of the CAN module can be made as follows: All CAN registers are located in the on-chip XRAM memory, from address F700_H up to F7FF_H. An access to this special memory space requires the modification of register XPAGE to the value of F7_H. Furthermore, bits XMAP0 and XMAP1 in register SYSCON must be forced to "0".

The initialisation of the CAN controller begins with setting the bits CCE and INIT in the Control Register CR (F700_H) to "1". This instruction (CCE=1) enables the modification of the CAN Bit Timing Registers BTR0, located at F704_H, and BTR1 at F705_H in order to program the desired bit timing and baudrate. Bit CCE should be reset to "0" after the access to these registers to avoid erroneous modification of the internal timing. The following values have been tested:

Table 1 :
Programming of the CAN Bit Timing Registers

Baudrate	125 kBaud	250 kBaud	500 kBaud	1 MBaud
BTR0	C4 _H	C1 _H	C0 _H	80 _H
BTR1	49 _H	6B _H	6B _H	25 _H

This description only focuses on the application of complete message identifiers where no masking is done. Therefore, the Global Mask Registers GMS0, GMS1, UGML0, UGML1, LGML0 and LGML1 (F709_H .. F70B_H) contain the value of FF_H.

The CAN controller provides three different types of interrupt sources:

- status interrupts
- error interrupts
- message specific interrupts

The first type is generated by a status change of the CAN module, which is indicated in the CAN Status Register SR (F701_H). This can be a successful transmission (TXOK is set) or reception (RXOK is set) of any message object, or the occurrence of an error (see LEC bitfield). These interrupt sources can be enabled by setting bit SIE in the CAN Control Register CR, which is located at address F700_H.

The second type are error interrupts, which can be enabled by the bit EIE. They are generated after the change of the flags EWRN or BOFF in the CAN Status Register.

Interrupts of the third type are generated by each message object after successful transmission or reception. This function can be enabled by setting to „1“ bits TXIE and/or RXIE in the lowbyte of the corresponding CAN Message Control Register MCR0_n (F7n0h), with n being the number of the corresponding message object (1..15).

Bit IE in the CAN Control Register globally enables (IE=1) or disables all interrupt sources of the CAN module. The internal structure of the C515C requires setting bits ECAN (register IEN2) and EAL (register IEN0) to „1“ in order to service an interrupt request.

Before the end of the CAN initialisation sequence where bit INIT is reset to „0“, all message objects must be completely initialised or declared as not valid (MSGVAL="0" in the corresponding CAN Message Control Register) in order to avoid erroneous data transfers.

This structure leads to the following CAN initialisation procedure :

```

unsigned char    IE = 1;      // all CAN interrupts enabled
unsigned char    SIE = 0;    // status interrupts disabled
unsigned char    EIE = 0;    // error interrupts disabled
//.....
void can_init (void)
{SYSCON &= 0xFC;             // XMAP0=0, XMAP1=0
  XPAGE = 0xF7;              // CAN memory space in XRAM

  GMS0 = 0xFF; GMS1 = 0xFF;  // global mask short
  UGML0 = 0xFF; UGML1 = 0xFF; // global mask long
  LGML0 = 0xFF; LGML1 = 0xFF;
  UMLM0 = 0xFF; UMLM1 = 0xFF; // last message mask
  LMLM0 = 0xFF; LMLM1 = 0xFF;

  CR = 0x41;                 // INIT=1 and CCE=1 (enable access baudrate)
  BTR0 = 0xC1; BTR1 = 0x6B;  // access bit timing registers: 10 MHz, 250 kBaud
  CR = 0x01;                 // CCE=0 (disable access baudrate)
  SR = 0xE7;                 // clear TXOK and RXOK

  MCR0_1 = 0x55;             // message 1 not valid
  ...
  MCR0_15 = 0x55;            // message 15 not valid

  IEN2 = 2;                  // enable CAN interrupt (ECAN=1)
  CR = 0;                    // INIT=0
  if ( IE) CR |= 0x02;        // enable global can_int
  if (SIE) CR |= 0x04;        // enable can_status_int
  if (EIE) CR |= 0x08;        // enable can_error_int
}

```

3 Definition of a Message Object

The complete definition of a message object includes the determination of its identifier in the corresponding CAN Arbitration Registers UAR0_n, UAR1_n, LAR0_n and LAR1_n (F7n2_H..F7n5_H). In the case of transmit objects, the desired data bytes have to be written to the addresses from F7n7_H (DB0_n) up to F7nE_H (DB7_n) in the message object.

The interrupt enable bits TXIE and RXIE are application specific and can be set in the corresponding CAN Message Control Register. They enable the interrupt generation on a successful message transfer. Furthermore, the remaining flags INTPND and RMTPNPND are set to „0“ to get defined starting conditions. In order to avoid a CAN action on a message object which is currently accessed by the CPU, bit CPUUPD has to be set to “1” before the CPU works on the data of this message object.

The CAN Message Configuration Register MCFG_n is located at address F7n6. It is used to define the character of the message object, such as the direction of the data transfer by bit DIR, or the data length by bit field DLC. Receive objects (DIR is set to “0”) contain no direct data, so their data length is 0. The data length of transmit objects (DIR is set to “1”) can be defined from 1 to 8 bytes. Furthermore, bit XTD determines whether an extended identifier of 29 bits (XTD=“1”), or a standard identifier of 11 bits (XTD=“0”) is used (see **table 2**).

After the last CPU access, the bit CPUUPD has to be cleared, as well as the bit NEWDAT. This bit is set after a CAN action on these data. Then the message object can be declared valid by setting to “1” bit MSGVAL, because the CAN module only works on valid message objects.

Table 2 :
Programming of the Message Configuration Register

Object	Direction	Data length	Protocol	Address	Value
1	transmit	1	standard	F716 _H	18 _H
2	receive	0	standard	F726 _H	00 _H
7	transmit	8	extended	F776 _H	8C _H
11	receive	0	extended	F7B6 _H	04 _H

Table 3 :
Impact of bit DIR

	Transmission of this message object generates ...	If a data frame with matching identifier is received ...	If a remote frame with matching identifier is received ...
Bit DIR = "0" Receive Object (receives data frames, transmits remote frames)	... a remote frame. The corresponding data frame is stored in this MO on reception.	... the data frame is stored.	... the remote frame is NOT answered.
Bit DIR = "1" Transmit Object (transmits data frames, receives remote frames)	... a data frame.	... the data frame is NOT stored.	... the remote frame is answered by the corresponding data frame

A message object initialisation procedure (objects 1 and 2, as given in the example) can be programmed as follows:

```
//.....
// definition message 1, standard transmission frame
void def_1 (void)
{MCR1_1 = 0xFB;           // CPUUPD=1
  UAR0_1 = 0x11; UAR1_1 = 0x20; // identifier 00010001 001
  MCFG_1 = 0x18;           // frame definition
  DB0_1 = 0x5A;           // data byte 0 : 5Ah
  MCR0_1 = 0xA5;           // MSGVAL=1, TXIE=1, RXIE=0, INTPND=0
  MCR1_1 = 0x55;           // RMPND=0, TXRQ=0, CPUUPD=0, NEWDAT=0
}

//.....
// definition message 2, standard remote frame
void def_2 (void)
{MCR1_2 = 0xFB;           // CPUUPD=1
  UAR0_2 = 0x22; UAR1_2 = 0x20; // identifier 00100010 001
  MCFG_2 = 0x00;           // frame definition
  MCR0_2 = 0xA9;           // MSGVAL=1, TXIE=1, RXIE=1, INTPND=0
  MCR1_2 = 0x55;           // RMPND=0, TXRQ=0, CPUUPD=0, NEWDAT=0
}
```

4 Interrupt Handling

As many different interrupt sources can generate only one global CAN interrupt request, their internal structure must be taken into account in the interrupt service procedure. The INTID code in the CAN Interrupt Register IR (F702_H) indicates which source has activated the request.

The status interrupt (if enabled by SIE) and the error interrupt (if enabled by EIE) cause the interrupt with the highest priority, which is indicated by the INTID value of "1". In the case of a status change due to a successful message transfer, one of the flags TXOK or RXOK in the CAN Status Register is set to "1". An erroneous message transfer is indicated by the LEC bit field. In the case of an error interrupt, at least one of the error flags EWRN and BOFF has changed. The CAN Status Register must be read in the interrupt service procedure in order to identify the interrupt source and to reset the pending interrupt request! The flags in this register must then all be cleared by software.

An INTID code of "2..16" indicates a message specific transmit (if TXIE="1") or receive (if RXIE="1") interrupt. A successful message transfer sets the corresponding bit INTPND to "1", which must be cleared by software to reset this interrupt request.

The priority of the internal CAN interrupt sources decreases with an increasing INTID code. This structure must also be taken into account for the identification of the interrupt source. For example, a successful transmission of only one message object can cause two independent interrupt requests if bit SIE and the corresponding bit TXIE have been set to "1". While the status interrupt (highest priority) is serviced and bit INTPND of this message object is not cleared, the message interrupt stays still pending. This will generate a second interrupt request (message specific) due to the same action. Only an INTID code of "0" indicates that all requested interrupts have been correctly serviced. A standard CAN interrupt procedure can be programmed as follows:

```
//.....
// CAN interrupt
void int_can (void) interrupt CANI
{unsigned char status, intreg;
  while (intreg = IR)
  {status = SR; SR = 0;          // read and reset CAN status
   switch (intreg)
   {case 1:                      // status and error interrupt
     if (SIE)                    // status interrupts
     {if (status & 0x08) {...} // transmit interrupt
      if (status & 0x10) {...} // receive interrupt
      if (status & 0x07) {...} // erroneous transfer
     }
     if (EIE)                    // error interrupts
     {if (status & 0x40) {...} // EWRN has changed
      if (status & 0x80) {...} // BUSOFF has changed
     }
     break;
   case 3:                      // message 1 interrupt
     MCR0_1 = 0xFD;             // reset INTPND
     if (status & 0x08) {...} // transmit interrupt
     if (status & 0x10) {...} // receive interrupt
     break;
   case 4:                      // message 2 interrupt
     MCR0_2 = 0xFD;             // reset INTPND
     if (status & 0x08) {...} // transmit interrupt
     if (status & 0x10) {...} // receive interrupt
     break;
   }
 } }
```

A valid message object can be sent by the CAN controller by setting bit TXRQ in the corresponding CAN Message Control Register. This can be done in the following way:

At the end of each transferred message (transmit or remote frame), the receiving CAN nodes on the bus answer with a dominant (“0”) acknowledge signal (Ack) to indicate successful message transfer. After the transmission of a remote frame from node A, node B (transmission object with matching identifier) answers by sending the requested data frame. These actions can generate status interrupts concerning flags TXOK and RXOK, as shown in **figure 2** for node A.



If several TXRQ bits are activated at the same time, the CAN controller starts sending the object with the highest priority (lowest number). A transmission request is only serviced if the corresponding message object is valid (MSGVAL="1") and not currently accessed by the CPU (CPUUPD="0"). In the case of bit TXRQ being set to „1“ while the bit CPUUPD is still „1“, the transmission request is taken into account after bit CPUUPD has been cleared and therefore need not to be repeated by the CPU.

6 Error Handling

After the detection of a transfer error, the sending CAN node immediately (one bit time) stops the transmission of the current message object and sends an error frame consisting of 6 dominant bits. They are detected by all other nodes, which then answer by the emission of another 6 dominant bits. After this sequence, the bus remains at logic "1" level for 11 bits, before the CAN controller automatically restarts the transmission of the disturbed message object.

An example of a short disturbance with only one erroneous bit is shown in fig. 3. In this case, the error counter of the sending CAN node is incremented by 8. It is decremented by 1 after each successful message transfer. An interrupt can be generated by each transfer error if bit SIE has been set to "1". The code in the LEC bitfield contains information about the error type. In the given example (see **figure 3**), a status interrupt is generated due to a Bit1Error. The status interrupt which is shown in **figure 3** is generated after the successful message transfer and the TxOK bit has been tested.

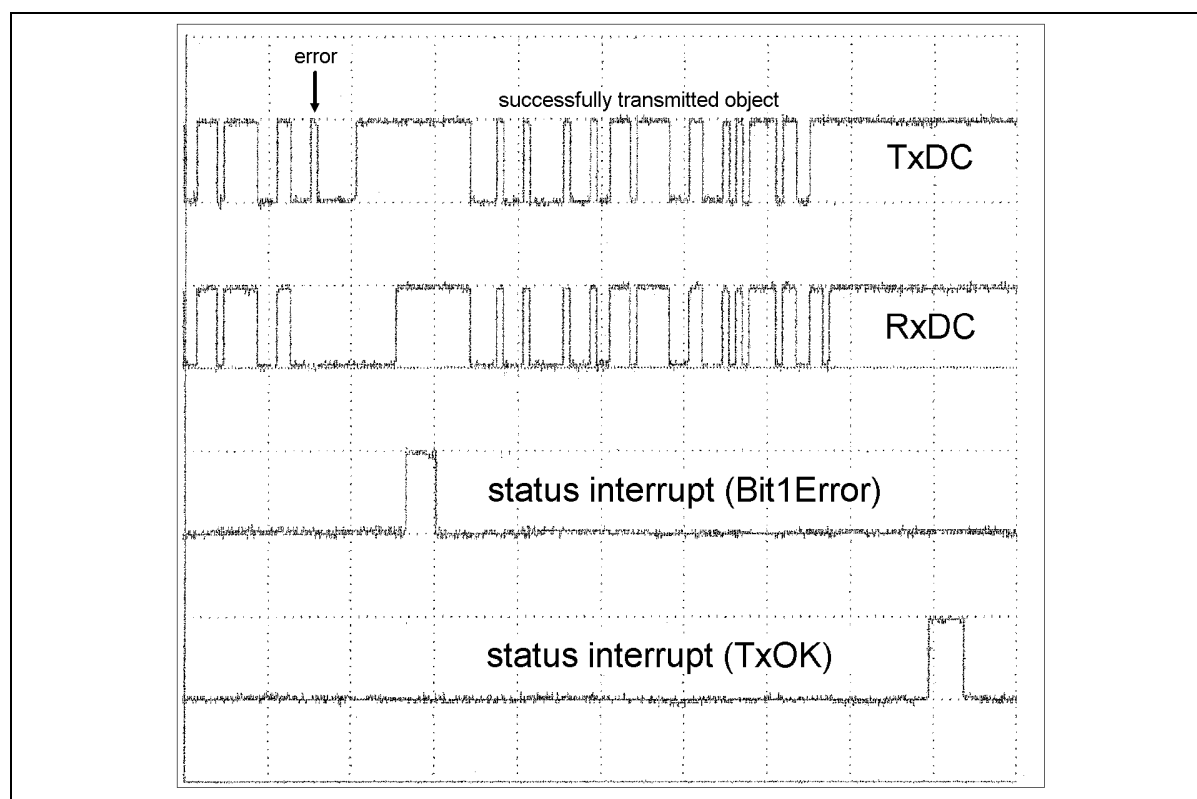


Figure 3 :
Status interrupt generation due to a Bit1Error (50µs/div)

7 Busoff State and CAN Re-Initialisation

Each transfer error causes the incrementation of one of the two error counters (transmit error counter and receive error counter). In the case of longer disturbances, they are also incremented each time a sequence of 8 erroneous bits occurs. When one of the counter reaches the value of 96, the bit EWRN is set to "1".

If more errors occur and one error counter reaches 255, the CAN controller stops all actions on the bus and goes into the busoff state. This is indicated by the BOFF flag changing to "1", which is shown **figure 4**.

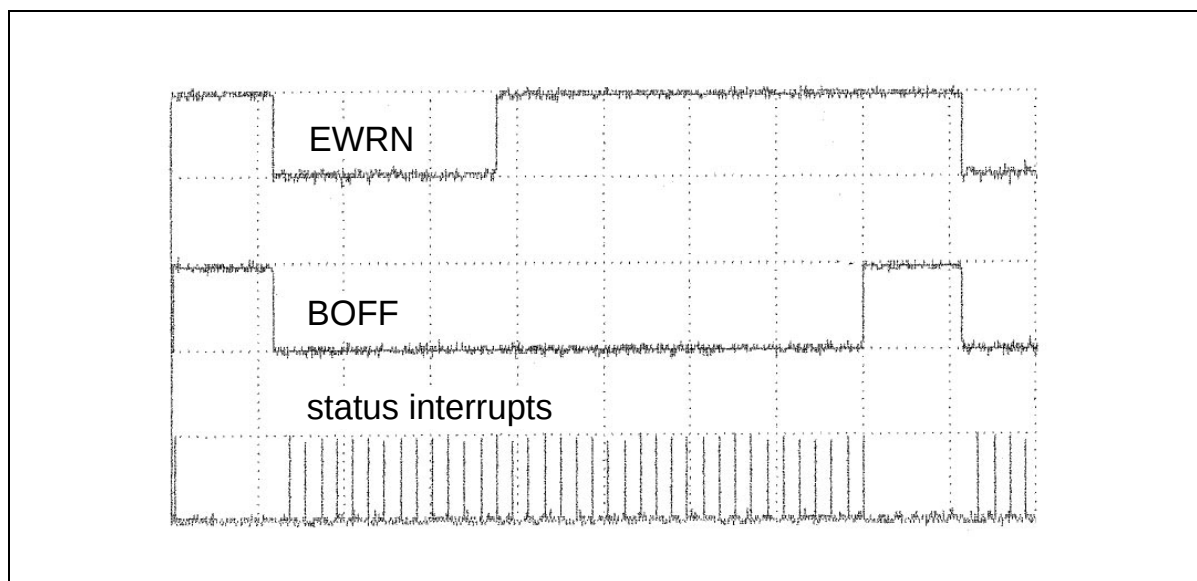


Figure 4 :
EWRN, BOFF and CAN Re-Initialisation (5ms/div)

An error interrupt is generated if the bit EIE is set and either EWRN or BOFF change the status. The re-initialisation of the CAN module to ensure its normal functionality can be achieved by clearing bit INIT, which is set automatically when entering busoff state. During busoff recovery the LEC bitfield contains the code representing a Bit0Error each time a sequence of 11 recessive bits has been monitored. As this code can generate a status interrupt, the SIE bit should be cleared in the busoff state in order to avoid higher CPU load. The end of the busoff state can be detected by using error interrupts (EIE="1") and the bit SIE can then be set to the desired interrupt mode.

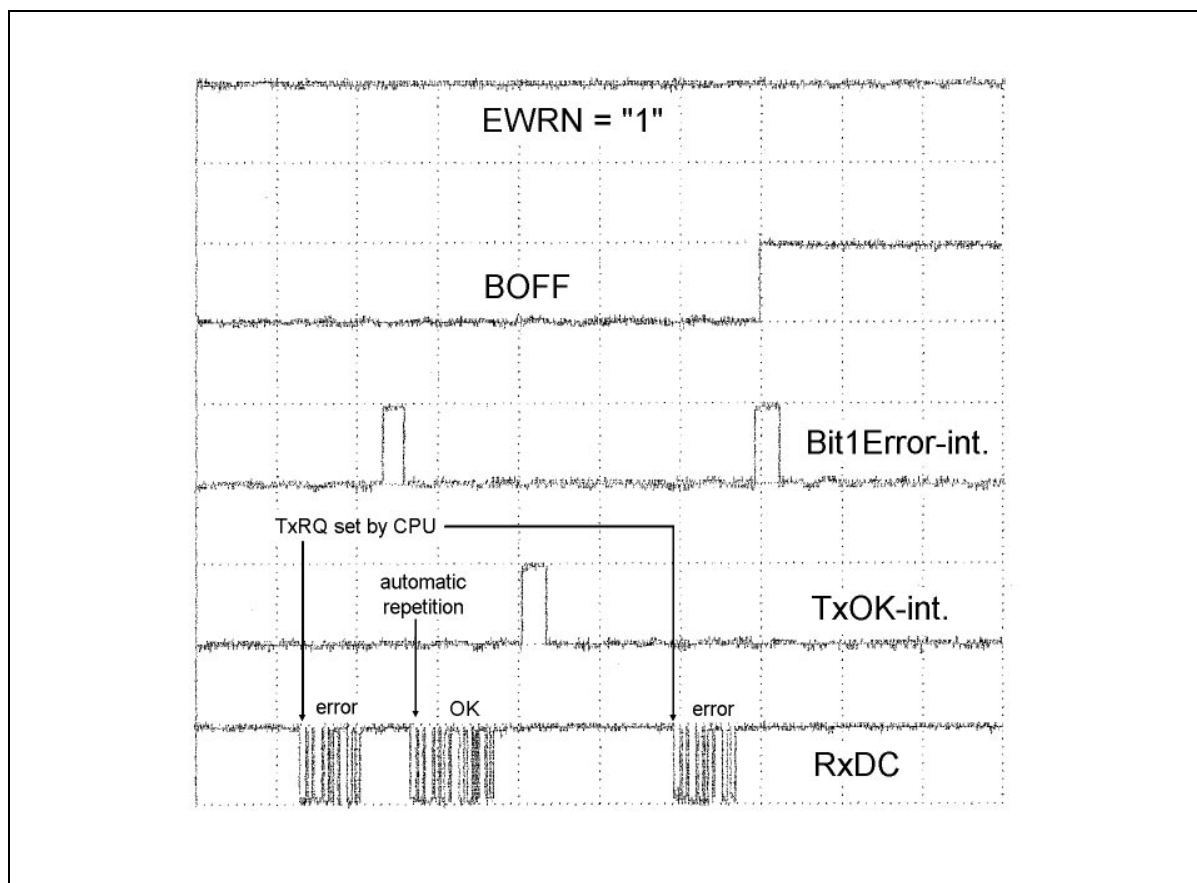


Figure 5 :
Message transfer just before busoff state (200µs/div)

If the CAN controller stops all actions on the bus and goes busoff, the disturbed message can not be repeated immediately, see fig. 5. After successful busoff recovery (started by resetting bit INIT), it is automatically repeated without involving the CPU if the corresponding bit TxRQ remains set, see **figure 6**. This functionality avoids the loss of messages due to temporary transfer errors. If the automatic repetition of the last erroneous message after busoff recovery is not desired, it can be disabled by resetting the corresponding bit TxRQ by software.

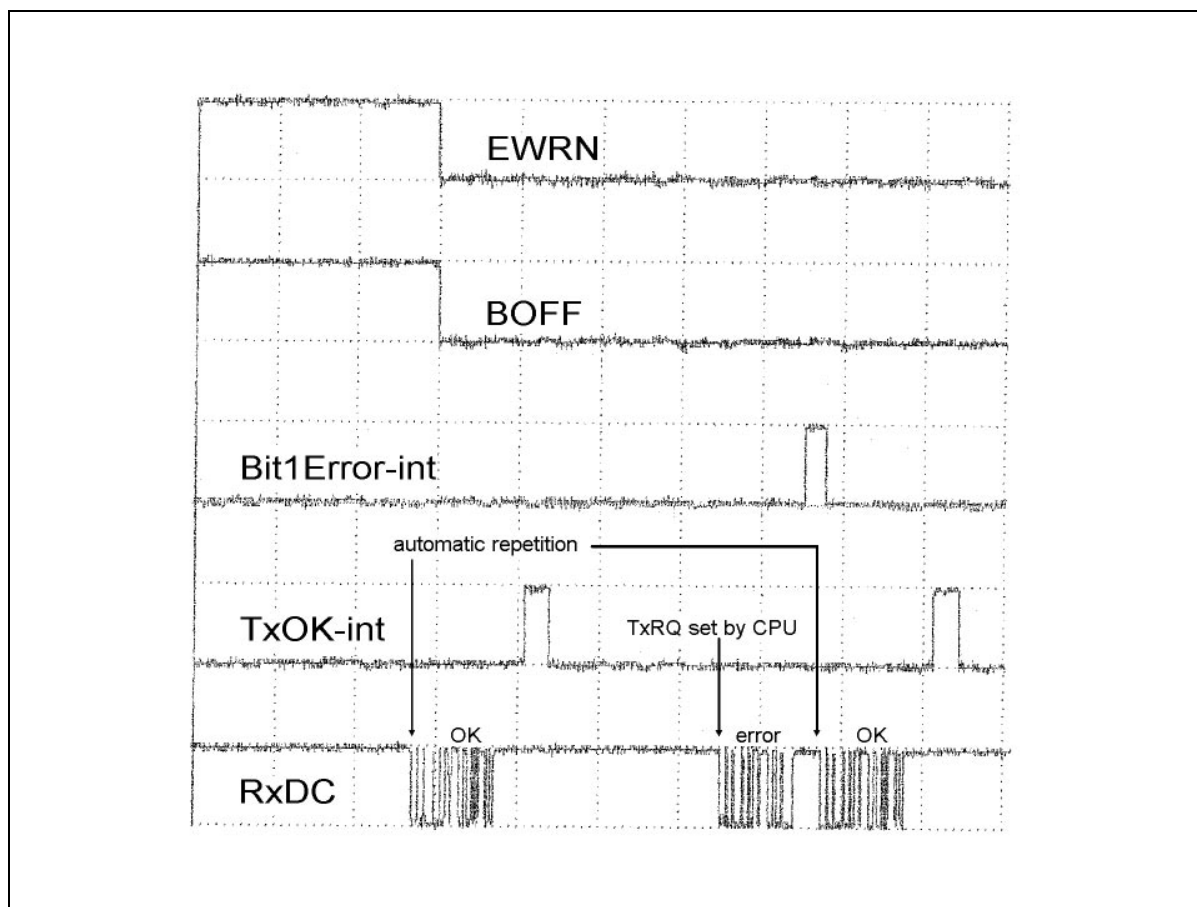


Figure 6 :
Automatic message repetition after busoff state (200µs/div)

The Siemens C515C microcontroller with on-chip CAN module provides all features of a FULL-CAN controller. This includes the use of up to 15 independent message objects with standard or extended identifiers, which can be transferred with a maximal baudrate of 1 Mbaud. Thanks to the multitude of different CAN interrupt sources, all information which are necessary for normal data transfer and error handling are directly available. For these reasons, the C515C can easily be used for data transfer and treatment in CAN bus systems.

Appendix**A CANREG.H**

```
extern unsigned char pdata canreg[256];
```

```
/* **** */
/* CAN Register Declaration */
/* CAN Control Registers */

#define CR          canreg[ 0]
#define SR          canreg[ 1]
#define IR          canreg[ 2]
#define BTR0        canreg[ 4]
#define BTR1        canreg[ 5]
#define GMS0        canreg[ 6]
#define GMS1        canreg[ 7]
#define UGML0       canreg[ 8]
#define UGML1       canreg[ 9]
#define LGML0       canreg[10]
#define LGML1       canreg[11]
#define UMLM0       canreg[12]
#define UMLM1       canreg[13]
#define LMLM0       canreg[14]
#define LMLM1       canreg[15]

/* CAN Message 1 Registers */
#define MCR0_M1     canreg[16]
#define MCR1_M1     canreg[17]
#define UAR0_M1     canreg[18]
#define UAR1_M1     canreg[19]
#define LAR0_M1     canreg[20]
#define LAR1_M1     canreg[21]
#define MCFG_M1     canreg[22]
#define DB0_M1      canreg[23]
#define DB1_M1      canreg[24]
#define DB2_M1      canreg[25]
#define DB3_M1      canreg[26]
#define DB4_M1      canreg[27]
#define DB5_M1      canreg[28]
#define DB6_M1      canreg[29]
#define DB7_M1      canreg[30]

/* CAN Message 2 Registers */
#define MCR0_M2     canreg[32]
#define MCR1_M2     canreg[33]
#define UAR0_M2     canreg[34]
#define UAR1_M2     canreg[35]
#define LAR0_M2     canreg[36]
#define LAR1_M2     canreg[37]
#define MCFG_M2     canreg[38]
#define DB0_M2      canreg[39]
#define DB1_M2      canreg[40]
#define DB2_M2      canreg[41]
#define DB3_M2      canreg[42]
#define DB4_M2      canreg[43]
#define DB5_M2      canreg[44]
#define DB6_M2      canreg[45]
#define DB7_M2      canreg[46]

/* CAN Message 3 Registers */
#define MCR0_M3     canreg[48]
#define MCR1_M3     canreg[49]
#define UAR0_M3     canreg[50]
#define UAR1_M3     canreg[51]
#define LAR0_M3     canreg[52]
#define LAR1_M3     canreg[53]
```

```
#define MCFG_M3      canreg[ 54]
#define DB0_M3       canreg[ 55]
#define DB1_M3       canreg[ 56]
#define DB2_M3       canreg[ 57]
#define DB3_M3       canreg[ 58]
#define DB4_M3       canreg[ 59]
#define DB5_M3       canreg[ 60]
#define DB6_M3       canreg[ 61]
#define DB7_M3       canreg[ 62]
```

```
/* CAN Message 4 Registers */
#define MCR0_M4      canreg[ 64]
#define MCR1_M4      canreg[ 65]
#define UAR0_M4      canreg[ 66]
#define UAR1_M4      canreg[ 67]
#define LAR0_M4      canreg[ 68]
#define LAR1_M4      canreg[ 69]
#define MCFG_M4      canreg[ 70]
#define DB0_M4       canreg[ 71]
#define DB1_M4       canreg[ 72]
#define DB2_M4       canreg[ 73]
#define DB3_M4       canreg[ 74]
#define DB4_M4       canreg[ 75]
#define DB5_M4       canreg[ 76]
#define DB6_M4       canreg[ 77]
#define DB7_M4       canreg[ 78]
```

```
/* CAN Message 5 Registers */
#define MCR0_M5      canreg[ 80]
#define MCR1_M5      canreg[ 81]
#define UAR0_M5      canreg[ 82]
#define UAR1_M5      canreg[ 83]
#define LAR0_M5      canreg[ 84]
#define LAR1_M5      canreg[ 85]
#define MCFG_M5      canreg[ 86]
#define DB0_M5       canreg[ 87]
#define DB1_M5       canreg[ 88]
#define DB2_M5       canreg[ 89]
#define DB3_M5       canreg[ 90]
#define DB4_M5       canreg[ 91]
#define DB5_M5       canreg[ 92]
#define DB6_M5       canreg[ 93]
#define DB7_M5       canreg[ 94]
```

```
/* CAN Message 6 Registers */
#define MCR0_M6      canreg[ 96]
#define MCR1_M6      canreg[ 97]
#define UAR0_M6      canreg[ 98]
#define UAR1_M6      canreg[ 99]
#define LAR0_M6      canreg[100]
#define LAR1_M6      canreg[101]
#define MCFG_M6      canreg[102]
#define DB0_M6       canreg[103]
#define DB1_M6       canreg[104]
#define DB2_M6       canreg[105]
#define DB3_M6       canreg[106]
#define DB4_M6       canreg[107]
#define DB5_M6       canreg[108]
#define DB6_M6       canreg[109]
#define DB7_M6       canreg[110]
```

```
/* CAN Message 7 Registers */
#define MCR0_M7      canreg[112]
#define MCR1_M7      canreg[113]
#define UAR0_M7      canreg[114]
#define UAR1_M7      canreg[115]
#define LAR0_M7      canreg[116]
```

```
#define LAR1_M7      canreg[117]
#define MCFG_M7     canreg[118]
#define DB0_M7      canreg[119]
#define DB1_M7      canreg[120]
#define DB2_M7      canreg[121]
#define DB3_M7      canreg[122]
#define DB4_M7      canreg[123]
#define DB5_M7      canreg[124]
#define DB6_M7      canreg[125]
#define DB7_M7      canreg[126]
```

```
/* CAN Message 8 Registers */
```

```
#define MCR0_M8     canreg[128]
#define MCR1_M8     canreg[129]
#define UAR0_M8     canreg[130]
#define UAR1_M8     canreg[131]
#define LAR0_M8     canreg[132]
#define LAR1_M8     canreg[133]
#define MCFG_M8     canreg[134]
#define DB0_M8      canreg[135]
#define DB1_M8      canreg[136]
#define DB2_M8      canreg[137]
#define DB3_M8      canreg[138]
#define DB4_M8      canreg[139]
#define DB5_M8      canreg[130]
#define DB6_M8      canreg[131]
#define DB7_M8      canreg[132]
```

```
/* CAN Message 9 Registers */
```

```
#define MCR0_M9     canreg[144]
#define MCR1_M9     canreg[145]
#define UAR0_M9     canreg[146]
#define UAR1_M9     canreg[147]
#define LAR0_M9     canreg[148]
#define LAR1_M9     canreg[149]
#define MCFG_M9     canreg[150]
#define DB0_M9      canreg[151]
#define DB1_M9      canreg[152]
#define DB2_M9      canreg[153]
#define DB3_M9      canreg[154]
#define DB4_M9      canreg[155]
#define DB5_M9      canreg[156]
#define DB6_M9      canreg[157]
#define DB7_M9      canreg[158]
```

```
/* CAN Message 10 Registers */
```

```
#define MCR0_M10    canreg[160]
#define MCR1_M10    canreg[161]
#define UAR0_M10    canreg[162]
#define UAR1_M10    canreg[163]
#define LAR0_M10    canreg[164]
#define LAR1_M10    canreg[165]
#define MCFG_M10    canreg[166]
#define DB0_M10     canreg[167]
#define DB1_M10     canreg[168]
#define DB2_M10     canreg[169]
#define DB3_M10     canreg[170]
#define DB4_M10     canreg[171]
#define DB5_M10     canreg[172]
#define DB6_M10     canreg[173]
#define DB7_M10     canreg[174]
```

```
/* CAN Message 11 Registers */
```

```
#define MCR0_M11    canreg[176]
#define MCR1_M11    canreg[177]
#define UAR0_M11    canreg[178]
#define UAR1_M11    canreg[179]
```

```
#define LAR0_M11 canreg[180]
#define LAR1_M11 canreg[181]
#define MCFG_M11 canreg[182]
#define DB0_M11 canreg[183]
#define DB1_M11 canreg[184]
#define DB2_M11 canreg[185]
#define DB3_M11 canreg[186]
#define DB4_M11 canreg[187]
#define DB5_M11 canreg[188]
#define DB6_M11 canreg[189]
#define DB7_M11 canreg[190]
```

```
/* CAN Message 12 Registers */
#define MCR0_M12 canreg[192]
#define MCR1_M12 canreg[193]
#define UAR0_M12 canreg[194]
#define UAR1_M12 canreg[195]
#define LAR0_M12 canreg[196]
#define LAR1_M12 canreg[197]
#define MCFG_M12 canreg[198]
#define DB0_M12 canreg[199]
#define DB1_M12 canreg[200]
#define DB2_M12 canreg[201]
#define DB3_M12 canreg[202]
#define DB4_M12 canreg[203]
#define DB5_M12 canreg[204]
#define DB6_M12 canreg[205]
#define DB7_M12 canreg[206]
```

```
/* CAN Message 13 Registers */
#define MCR0_M13 canreg[208]
#define MCR1_M13 canreg[209]
#define UAR0_M13 canreg[210]
#define UAR1_M13 canreg[211]
#define LAR0_M13 canreg[212]
#define LAR1_M13 canreg[213]
#define MCFG_M13 canreg[214]
#define DB0_M13 canreg[215]
#define DB1_M13 canreg[216]
#define DB2_M13 canreg[217]
#define DB3_M13 canreg[218]
#define DB4_M13 canreg[219]
#define DB5_M13 canreg[220]
#define DB6_M13 canreg[221]
#define DB7_M13 canreg[222]
```

```
/* CAN Message 14 Registers */
#define MCR0_M14 canreg[224]
#define MCR1_M14 canreg[225]
#define UAR0_M14 canreg[226]
#define UAR1_M14 canreg[227]
#define LAR0_M14 canreg[228]
#define LAR1_M14 canreg[229]
#define MCFG_M14 canreg[230]
#define DB0_M14 canreg[231]
#define DB1_M14 canreg[232]
#define DB2_M14 canreg[233]
#define DB3_M14 canreg[234]
#define DB4_M14 canreg[235]
#define DB5_M14 canreg[236]
#define DB6_M14 canreg[237]
#define DB7_M14 canreg[238]
```

```
/* CAN Message 15 Registers */
#define MCR0_M15 canreg[240]
#define MCR1_M15 canreg[241]
#define UAR0_M15 canreg[242]
```

```
#define UAR1_M15 canreg[243]
#define LAR0_M15 canreg[244]
#define LAR1_M15 canreg[245]
#define MCFG_M15 canreg[246]
#define DB0_M15 canreg[247]
#define DB1_M15 canreg[248]
#define DB2_M15 canreg[249]
#define DB3_M15 canreg[250]
#define DB4_M15 canreg[251]
#define DB5_M15 canreg[252]
#define DB6_M15 canreg[253]
#define DB7_M15 canreg[254]
```

B INTC515C.H

```
/* **** */
/*                                     */
/*      „INTC515C.H“                 */
/*                                     */
/*      symbolic interruptnumbers for C151C-interrupt routines */
/*                                     */
/* **** */
#define EXTI0      0    // (03H) external interrupt 0
#define TIMER0     1    // (0BH) Timer 0 Overflow
#define EXTI1      2    // (13H) external interrupt 1
#define TIMER1     3    // (1BH) Timer 1 Overflow
#define SINT       4    // (23H) serial interrupt
#define TIMER2     5    // (2BH) Timer 2 Overflow
#define ADCI       8    // (43H) A/D-Converter interrupt
#define EXTI2      9    // (4BH) external interrupt 2
#define EXTI3     10    // (53H) external interrupt 3
#define EXTI4     11    // (5BH) external interrupt 4
#define EXTI5     12    // (63H) external interrupt 5
#define EXTI6     13    // (6BH) external interrupt 6
#define PWD       15    // (7BH) Power Down interrupt
#define CANI      17    // (8BH) CAN interrupt
#define SSCI      18    // (93H) SSC interrupt
#define EXTI7     20    // (A3H) external interrupt 7
#define EXTI8     21    // (ABH) external interrupt 8
```

C REGC515C.H

```
/* (c) Copyright SIEMENS 1996 , All rights reserved. */
/* Register Declarations for the C515C Processor      */
/* **** */
/* BYTE Register */
sfr P0      = 0x80;
sfr SP      = 0x81;
sfr DPL     = 0x82;
sfr DPH     = 0x83;
sfr WDTREL  = 0x86;
sfr PCON    = 0x87;
sfr TCON    = 0x88;
sfr TMOD    = 0x89;
sfr TL0     = 0x8A;
sfr TL1     = 0x8B;
sfr TH0     = 0x8C;
sfr TH1     = 0x8D;

sfr P1      = 0x90;
sfr XPAGE   = 0x91;
sfr DPSEL   = 0x92;
sfr SSCCON  = 0x93;
sfr STB     = 0x94;
```

```

sfr  SRB      = 0x95;
sfr  SSCMOD   = 0x96;
sfr  SCON     = 0x98;
sfr  SBUF     = 0x99;
sfr  IEN2     = 0x9A;

sfr  P2       = 0xA0;
sfr  IEN0     = 0xA8;
sfr  IP0      = 0xA9;
sfr  SRELL    = 0xAA;
sfr  SCF      = 0xAB;
sfr  SCIEN    = 0xAC;

sfr  P3       = 0xB0;
sfr  SYSCON   = 0xB1;
sfr  IEN1     = 0xB8;
sfr  IP1      = 0xB9;
sfr  SRELH    = 0xBA;

sfr  IRCON    = 0xC0;
sfr  CCEN     = 0xC1;
sfr  CCL1     = 0xC2;
sfr  CCH1     = 0xC3;
sfr  CCL2     = 0xC4;
sfr  CCH2     = 0xC5;
sfr  CCL3     = 0xC6;
sfr  CCH3     = 0xC7;
sfr  T2CON    = 0xC8;
sfr  CRCL     = 0xCA;
sfr  CRCH     = 0xCB;
sfr  TL2      = 0xCC;
sfr  TH2      = 0xCD;

sfr  PSW      = 0xD0;
sfr  ADCON0   = 0xD8;
sfr  ADDATH   = 0xD9;
sfr  ADDATL   = 0xDA;
sfr  P6       = 0xDB;
sfr  ADCON1   = 0xDC;
sfr  P7       = 0xDD;
sfr  CCPL     = 0xDE;
sfr  CCPH     = 0xDF;

sfr  ACC      = 0xE0;
sfr  P4       = 0xE8;

sfr  B        = 0xF0;
sfr  P5       = 0xF8;
sfr  DIR5     = 0xF8;

/***** */
/* BIT Register      */

/* TCON */
sbit TF1      = 0x8F;
sbit TR1      = 0x8E;
sbit TF0      = 0x8D;
sbit TR0      = 0x8C;
sbit IE1      = 0x8B;
sbit IT1      = 0x8A;
sbit IE0      = 0x89;
sbit IT0      = 0x88;
/* SCON */
sbit SM0      = 0x9F;
sbit SM1      = 0x9E;
sbit SM2      = 0x9D;
sbit REN      = 0x9C;

```

```

sbit TB8      = 0x9B;
sbit RB8      = 0x9A;
sbit TI       = 0x99;
sbit RI       = 0x98;
/* IEN0 */
sbit EAL      = 0xAF;
sbit WDT      = 0xAE;
sbit ET2      = 0xAD;
sbit ES       = 0xAC;
sbit ET1      = 0xAB;
sbit EX1      = 0xAA;
sbit ET0      = 0xA9;
sbit EX0      = 0xA8;
/* IEN1 */
sbit EXEN2    = 0xBF;
sbit SWDT     = 0xBE;
sbit EX6M     = 0xBD;
sbit EX5      = 0xBC;
sbit EX4      = 0xBB;
sbit EX3      = 0xBA;
sbit EX2      = 0xB9;
sbit EADC     = 0xB8;
/* P3 */
sbit RD       = 0xB7;
sbit WR       = 0xB6;
sbit T1       = 0xB5;
sbit T0       = 0xB4;
sbit INT1     = 0xB3;
sbit INT0     = 0xB2;
sbit TXD      = 0xB1;
sbit RXD      = 0xB0;
/* T2CON */
sbit T2PS     = 0xCF;
sbit I3FR     = 0xCE;
sbit I2FR     = 0xCD;
sbit T2R1     = 0xCC;
sbit T2R0     = 0xCB;
sbit T2CM     = 0xCA;
sbit T2L1     = 0xC9;
sbit T2I0     = 0xC8;
/* IRCON */
sbit EXF2     = 0xC7;
sbit TF2      = 0xC6;
sbit IEX6     = 0xC5;
sbit IEX5     = 0xC4;
sbit IEX4     = 0xC3;
sbit IEX3     = 0xC2;
sbit IEX2     = 0xC1;
sbit IADC     = 0xC0;
/* ADCON0 */
sbit BD       = 0xDF;
sbit CLK      = 0xDE;
sbit ADEX     = 0xDD;
sbit BSY      = 0xDC;
sbit ADM      = 0xDB;
sbit MX2      = 0xDA;
sbit MX1      = 0xD9;
sbit MX0      = 0xD8;
/* PSW */
sbit CY       = 0xD7;
sbit AC       = 0xD6;
sbit F0       = 0xD5;
sbit RS1      = 0xD4;
sbit RS0      = 0xD3;
sbit OV       = 0xD2;
sbit F1       = 0xD1;
sbit P        = 0xD0;

```