Manual Addendum

Tiger *plus*

Blank Page

Index

Blank Page

Installation

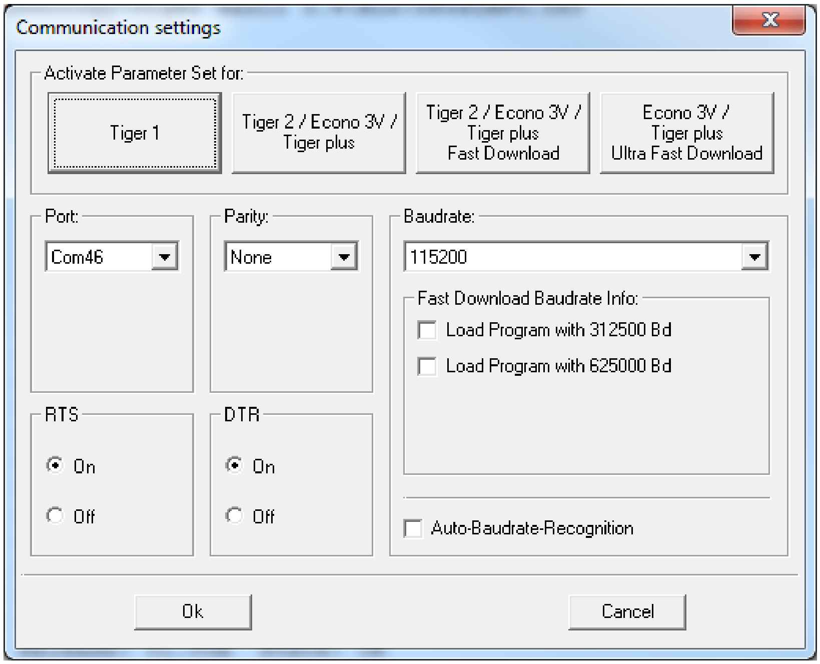
In order to work with Tiger *plus* using an existing compiler-version 5.4, several new files are required, please copy them into particular directories of your existing Tiger-BASIC installation. This concerns the following files:

file name(s):	file type:	copy to:
Tgbas32.exe	new compiler-version	..\Bin
*.TDP	Device drivers for Tiger *plus*	..\Bin
Tac0000.TAP	System file for Tiger *plus*	..\Bin
Tac0000_.TAP	System file for Tiger *plus*	..\Bin
Tac0100.TAP	System file for Tiger *plus*	..\Bin
Tac0100_.TAP	System file for Tiger *plus*	..\Bin
Tac0200.TAP	System file for Tiger *plus*	..\Bin
Tac0200_.TAP	System file for Tiger *plus*	..\Bin
Tac0300.TAP	System file for Tiger *plus*	..\Bin
Tac0300_.TAP	System file for Tiger *plus*	..\Bin
Tac0400.TAP	System file for Tiger *plus*	..\Bin
Tac0400_.TAP	System file for Tiger *plus*	..\Bin
Thinfo0.THP	System file for Tiger *plus*	..\Bin
Define_a.INC	general symbol-definitions	..\Include
Ufunc4.INC	definitions user-function-codes	..\Include

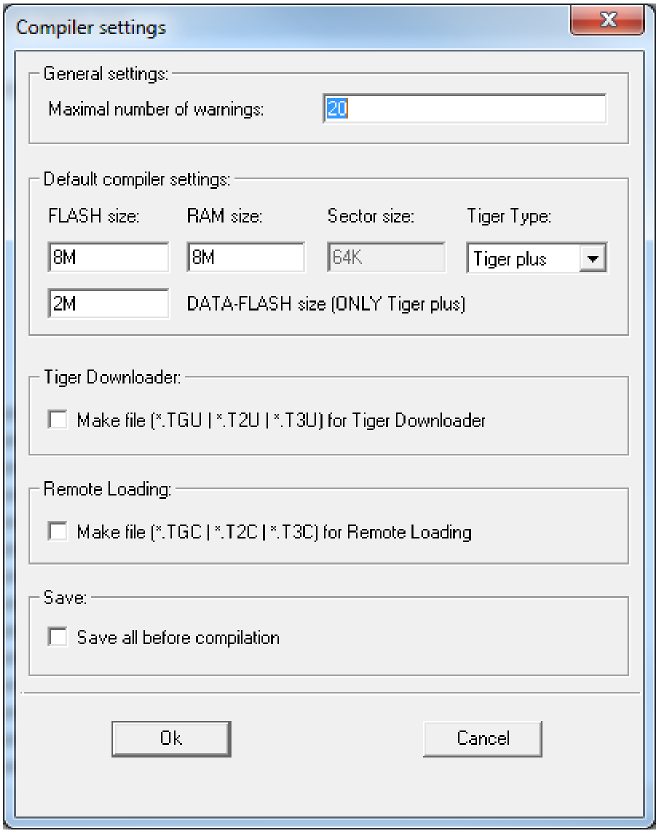
Development environment

Please consider the following in the Tiger-BASIC IDE when employing Tiger *plus*:

- The interface-settings, to be found in the **Options / Communication** menu, are to be adjusted so that the baud rate is 115,200 and parity is set to "none". Just press the "Tiger plus" button to activate this setting. **The Tiger *plus* module also supports the fast download (312500 Bd) and the ultra-fast download (625000 Bd) option.**



- The Tiger *plus* module will be recognized by its development environment automatically. If a program has to be compiled for the Tiger *plus*, without a module being connected, the module type has to be set to “Tiger plus” in the menu **Options / Compiler.**



Tiger *plus* module

Hardware

Aside from the very small basic differences between the classical Tiny-Tiger and the new Tiny Tiger *plus* such as the additional rows of pins, there are differences in certain pins, which have obviously not changed in their function when compared to the Tiny-Tiger. However, the differences are the following:

- In the Tiger *plus*, the pins L33..L37, L60..L67, L70..L73, L80..L87, as well as L90..L95 have a voltage range of 0 to 3.3 V as outputs and an input voltage range of 0 to 5 V. As digital inputs the I/O are 5V tolerant.

Software

A further change in the Tiger *plus* concerns the software, viz. the file type STRING: Theoretically, strings with a length of up to 2 GB can be processed. In practice, therefore, the length of a string is only restricted by the size of the module's RAM.

String length

In the Tiger *plus*, the maximum length of a string is no longer restricted (only by the RAM). Therefore, even more data can be put into a string. This is to be taken with a grain of salt, though, since the duration of the operations increases correspondingly for very large strings. Very large strings can also influence the timing of the multi-tasking system, since one BASIC instruction is always completed before switching to the next task.

This has also influence to the DATA instruction. For further information, please refer to page 10.

Tiger-BASIC Preprocessor Instructions

#define TIGER_PLUS

#define TIGER_PLUS *(or #define TIGER_2 or #define TIGER_1)*

Function: The symbolic constants "TIGER_1", "TIGER_2" and "TIGER_PLUS" are automatically generated by the compiler and can be applied for managing the module-dependent branches of the source code. Creating these defines in your code may result in unwanted effects running your program and should thus be avoided.

Example for installing serial driver with different baud rates using Tiger *plus*:

```
#ifdef TIGER_PLUS
  INSTALL_DEVICE #SER, "SER1B_K1.TDP", &
    BD_115_200, DP_8N, JA, &                        ' settings for SER0
    BD_115_200, DP_8N, JA                           ' settings for SER1
#else
  INSTALL_DEVICE #SER, "SER1B_K1.TDD", &
    BD_38_400, DP_8N, JA, &                         ' settings for SER0
    BD_38_400, DP_8N, JA                            ' settings for SER1
#endif
```

Tiger-BASIC Compiler Instructions

USER_FREQUENCY

USER_FREQUENCY SPEED_100

Function: The Tiger *plus* CPU speed is adjusted with *USER_FREQUENCY*. Without this instruction, the default speed is *SPEED_25*. You are free to increase or decrease the CPU speed and adapt it to your application. Please refer the Tiger *plus* datasheet for typ. power consumption.

Options for *USER_FREQUENCY*:

No	Symbol	Description
1	SPEED_25	25% Speed **(default)**
2	SPEED_50	50% Speed
4	SPEED_100	100% Speed (Full Speed)

DATA

DATA Type Constlist

Function: Initializes a data field in the Flash-memory.

Parameters:

	B	W	L	S	F	
Type	●	●	●	-	-	determines the type of data BYTE, WORD, LONG, REAL, STRING, FILTER, or FILE and determines the values to be saved.
Constlist	●	●	●	-	-	is a list of constants of the type BYTE, WORD, LONG, STRING, or FILE and determines the values to be saved.

There is one difference to the Tiger-1, because of the new string length. Character strings are saved with details of their length, e.g.:
"Hello" -> 05 00 **00 00** 'H' 'e' 'l' 'l' 'o'; a total of 9 bytes.

Updated functions

SYSVARN

RES = SYSVARN (FunctionNo, Parameter2)

Returns the Value of a LONG system variable. Numeric type system variables, other than real, are tested with this function. The test can also trigger system functions.

Parameters:

	B	W	L	S	F	
FunctionNo	●	●	●	-	-	is a variable, constant or expression of the type BYTE, WORD, LONG and is the number of the inquiry.
Parameter2	●	●	●	-	-	can have various meanings or is sometimes a random number (Dummy).
						Function value:
RES	-	-	●	-	-	is of the type LONG. An automatic type conversion takes place during the assignment.

The function numbers are assigned names in the Include file DEFINE_A.INC; these can be found in the table below.

Include the file 'DEFINE_A.INC' to use symbols, as function numbers may change in future developments of Tiger BASIC®. New and updated Functions of SYSVARN:

Symbol	No	2nd parameter	Description
FLASH_SIZE	‹33›	Dummy	Size of Program-Flash in bytes
FLASH_SEC	‹34›	Dummy	Number of sectors in Program-Flash
FLASH_SSIZE	‹35›	Dummy	Flash sector size
FLASH_ASEC	‹36›	Dummy	Number of Flash sectors
FLASH_GSIZE	‹37›	Dummy	Size of Flash memory in bytes
FLASH_DSEC	‹38›	Dummy	Number of Flash sectors for User-Data
FLASH_DSIZE	‹39›	Dummy	Size of Flash memory in bytes for User-Data

Symbol	No	2nd parameter	Description
FLASH_DMODE	‹40›	Dummy	0=system waits during Flash operations 1=system continues to run during Flash operations
PFLASH_DSIZE	‹41›	Dummy	Size of Flash memory in bytes for User-Data inside the Program Flash
DFLASH_SIZE	‹42›	Dummy	Size of Data-Flash in bytes
FLASH_BUSY	‹43›	Dummy	Busy flag for usage with ERASE_FLASH_SECTOR function 1 = busy 0 = not busy
BACKUP_RAM_SIZE	‹53›	Dummy	Size of Backup RAM memory in bytes
TIGER_MODULE	‹69›	Dummy	Tiger module Type: 003H = module family E3V 083H = module family TINY-Tiger 084H = module family TINY-Tiger 2 092H = module family ECONO-Tiger *plus* 093H = module family TINY-Tiger *plus* 094H = module family TINY-Tiger 2 *plus* 09AH = module family BASIC-Tiger *plus* 0AAH = module family A (BASIC-Tiger)

Program examples:

```
'----------------------------------------------------------------------
' Name: SYSVARN_FLASH.TIG
'----------------------------------------------------------------------
USER_VAR_STRICT
#INCLUDE DEFINE_A.INC                        ' include global definitions

TASK MAIN                                    ' begin task MAIN
' install LCD-driver (BASIC-Tiger)
  INSTALL DEVICE #LCD, "LCD1.TDD"
' install LCD-driver (TINY-Tiger)
' INSTALL DEVICE #1, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8
  PRINT #LCD, "<1>Flash size:"; SYSVARN (FLASH_GSIZE, -1)/1024; "K"
  PRINT #LCD, "Data-F.:";       SYSVARN (DFLASH_SIZE, -1)/1024; "K"
  PRINT #LCD, "Prog-F.:";       SYSVARN (FLASH_SIZE, -1)/1024; "K"
  PRINT #LCD, "Prog-F. U-Dat:"; SYSVARN (PFLASH_DSIZE, -1)/1024; "K";
END
```

```
'----------------------------------------------------------------------
' Name: SYSVARN_MODULE_TYPE.TIG
'----------------------------------------------------------------------
TASK MAIN ' begin task MAIN

' install LCD-driver (BASIC-Tiger)
  INSTALL DEVICE #LCD, "LCD1.TDD"

  PRINT #LCD, "<1>Module type:"; SYSVARN (TIGER_MODULE, -1)
  switch SYSVARN (TIGER_MODULE, -1)
  case 003H:
    PRINT #LCD, "E3V"
  case 083H:
    PRINT #LCD, "TINY-Tiger"
  case 084H:
    PRINT #LCD, "TINY-Tiger 2"
  case 092H:
    PRINT #LCD, "ECONO-Tiger plus"
  case 093H:
    PRINT #LCD, "TINY-Tiger plus"
  case 094H:
    PRINT #LCD, "TINY-Tiger 2 plus"
  case 09AH:
    PRINT #LCD, "BASIC-Tiger plus"
  case 0AAH:
    PRINT #LCD, "BASIC-Tiger"
  endswitch

END
```

READ_T_CODE$

RES$ = READ_T_CODE$(Option)

Reads out the unique serial number (T-Code) of a Tiger plus module.

Parameters:

	B	W	L	S	F	
Option	●	●	●	-	-	is a variable, constant or expression of the type BYTE, WORD, LONG and is the number of the inquiry. 0: T-Code / serial number (12 bytes) 1: software version (6 bytes) 2-4: dummy (each 5 bytes) 128: complete String (all 128 bytes) **Function value:**
RES$	-	-	-	●	-	is of the type STRING. Contains the requested information.

The T-Code will be unique for all Tiger plus modules but might have overlaps with the Tiny Tiger 2 T-Codes. An additional check of the module type is recommended.

While Tiny Tiger 2 had 5 bytes for the software version, Tiger plus modules use 6 bytes and the Tiger plus will return only zeros for parameters 2 to 4. Therefore, reading all 128 bytes will result in: [12 bytes T-Code] [6 bytes software version] [110 zeros]

Program example:

```
'----------------------------------------------------------------------------
'    Name:  Read_T_Code.tig
'----------------------------------------------------------------------------
task main
  string read$(128)

  #ifdef TIGER_PLUS
  read$ = read_t_code$(0)        ' T-Code / serial number
  read$ = read_t_code$(1)        ' Software version
  read$ = read_t_code$(128)      ' complete string (all 128 Bytes)
  #endif
end
```

New functions

READ_BACKUP_RAM

String:	RES	=	READ_BACKUP_RAM$ (Address, Number, Success_code)
Byte/Word/Long:	RES	=	READN_BACKUP_RAM (Address, Number, Success_code)
Real:	RES	=	READR_BACKUP_RAM (Address, Number, Success_code)

This function reads a group of bytes from the backup RAM memory location given by **Address** into **RES**. The number of bytes within the group read from backup RAM is given by the value **Number**.

Parameters:

	B	W	L	S	F	
Address	●	●	●	-	-	Is a variable, constant or expression of the type BYTE, WORD or LONG and specifies the starting address in the backup RAM from where the bytes are to be read.
Number	●	●	●	-	-	Is a variable, constant or expression of the type BYTE, WORD or LONG and specifies the number of bytes to be read. The number of bytes that the variable can accept is also the maximum number of bytes that can be read.
Success_code	-	-	●	-	-	Output: is a variable of the type LONG and returns the result of the function as follows: 0 = OK. Bytes were read as intended. -1 = Warning: **Number** limited to maximum size of **RES**. -2 = Warning: reached end of backup RAM memory. -4 = Warning: address out of backup RAM area. -17 = Warning: READ_BACKUP_RAM functions are not supported in this Tiger module. **Function value:**
RES	●	●	●	●	●	Is of type BYTE, WORD, LONG, REAL or STRING and contains the bytes read from backup RAM.

! READ_BACKUP_RAM functions are supported as of module version 3.10m! The module version can be inquired at runtime with the function SYSVARN or via the command *View->Tiger-Status* in the TIGER-Basic IDE.

The backup RAM is powered from Batt. Input voltage, when the main Vcc supply is powered off. To retain the content of the backup RAM when Vcc is turned off, Batt. input pin needs to be connected to an optional standby voltage supplied by a battery or by another source. It can be considered as an internal EEPROM with unlimited erase cycles when Batt. input is always present.

When the Tiger is supplied by Vcc, the backup RAM is powered from Vcc which replaces the Batt. input power supply to save battery life.

Typically, the size of the backup RAM is 2 Kbyte. To read out the real size of the backup RAM of your module, please use the SYSVARN function:

```
RAM_SIZE = SYSVARN(BACKUP_RAM_SIZE, 0) 'get the size of Backup RAM
```

Program example:

```
'-----------------------------------------------------------------------------
'Name:  READ_BACKUP_RAM$1.TIG
'-----------------------------------------------------------------------------
user_var_strict
#include define_a.inc                     ' include global definitions
#include ufunc4.inc                       ' include global definitions

task main                                 ' begin task MAIN
  long llResult                           ' error/sucess code
  string slBackupRam$                     ' result of READ_BACKUP_RAM

  install_device #LCD, "lcd1.tdd"         ' install LCD-driver

  ' write "Hello World!" to backup RAM
  llResult = WRITE_BACKUP_RAM(0, "Hello World!", 0, 12)

  ' read from backup RAM
  slBackupRam$ = READ_BACKUP_RAM$(0, 12, llResult)

  print #LCD, "<1>BACKUP_RAM:"            ' print result
  print #LCD, slBackupRam$                ' to LCD
end
```

Please ensure there was no power down before reading out the backup RAM contents, in the case of power down, these contents are lost. The easiest way is to use the RTC device driver. The RTC uses the same Batt. Input as the backup RAM. There is a User-function-code to read out the voltage low detection. It is recommended to use an additional magic number to validate the backup RAM content.

```
'-----------------------------------------------------------------------------
'Name:  READ_BACKUP_RAM$2.TIG
'-----------------------------------------------------------------------------
user_var_strict
#include define_a.inc                  ' include global definitions
#include ufunc4.inc                    ' include global definitions

#define MAGIC_NUMBER    0DEADBEEFH     ' Magic number (validate backup RAM)

task main                              ' begin task MAIN
  long llResult                        ' error/sucess code
  string slBackupRam$                  ' result of READ_BACKUP_RAM
  long llVoltage                       ' voltage down flag from RTC
  long llRTCstat                       ' status of RTC
  long llMagicNumber                   ' Magic number

  install_device #LCD, "lcd1.tdd"      ' install LCD-driver
  install_device #RTC, "rtc1.tdd"      ' install RTC-driver

  print #1,"<1>installing RTC";        '
  llRTCstat = RTC_INITIAL              '
  while llRTCstat < RTC_NO_RTC         ' while searching for RTC
    get #RTC, #0, #UFCI_RTC_STAT0, 1, llRTCstat ' get status of RTC
    wait_duration 200                           '
  endwhile

  if llRTCstat = RTC_PRESENT then                        ' if RTC found
    ' read out magic number from backup RAM
    llMagicNumber = READN_BACKUP_RAM(0, 4, llResult)
    get #RTC, #0, #UFCI_RTC_VOLTAGE, 0, llVoltage        ' get Voltage Low
    if llVoltage = RTC_VOLTAGE_LOW OR &                  ' was power down?
       llMagicNumber <> MAGIC_NUMBER then                ' wrong magicnumber?
      put #RTC, 0                                        ' start RTC
      print #LCD, "<1>Save String to"                    '
      print #LCD, "backup RAM..."                        '

      ' write "Hello World!" to backup RAM
      llResult = WRITE_BACKUP_RAM(4, "Hello World!", 0, 12)
      if llResult = 0 then                               ' check success code
        ' write magic number to validate content of backup RAM
        llResult = WRITE_BACKUP_RAM(0, MAGIC_NUMBER, 0, 4)
        if llResult = 0 then                             ' check success code
          wait_duration 1000                             ' wait 1 second
          restart_prog()                                 ' reset Tiger
        else
          print #LCD, "<1>Error:"; llResult              ' print error number
        endif
      else                                               '
        print #LCD, "<1>Error:"; llResult                ' print error number
      endif
    else                                                 '
      slBackupRam$ = READ_BACKUP_RAM$(4, 12, llResult)   ' read backup RAM
      if llResult = 0 then                               ' check success code
        print #LCD, "<1>BACKUP_RAM:"                     ' print result
        print #LCD, slBackupRam$                         ' to LCD
      else                                               '
        print #LCD, "<1>Error:"; llResult                ' print error number
```

```
      endif                                                   '
    endif                                                     '
  endif                                                       '
end                                                           ' end task MAIN
```

See also: WRITE_BACKUP_RAM

WRITE_BACKUP_RAM

RES = WRITE_BACKUP_RAM (Dst_Address, Source, Src_Offset, Number)

This function writes **Number** bytes from **Source** with **Src_Offset** to **Dst_Address** in the backup RAM.

Parameters:

	B	W	L	S	F	
Dst_Address	●	●	●	-	-	Is a variable, constant or expression of the type BYTE, WORD or LONG and specifies the starting address in the backup RAM from where the bytes are to be written.
Source	●	●	●	●	●	Is a variable, constant or expression of the type BYTE, WORD, LONG, REAL or STRING and specifies the data to write to the backup RAM.
Src_Offset	●	●	●	-	-	Is a variable, constant or expression of the type BYTE, WORD or LONG and specifies the start position in **Source** to write from. With numeric values, **Src_Offset** 0 means the lowest byte. With a data type STRING **Src_Offset** 0 is the first byte in the string.
Number	●	●	●	-	-	Is a variable, constant or expression of the type BYTE, WORD or LONG and specifies the number of bytes to be written. The number of bytes is limited through the length of **Source** and the length of the backup RAM.
						Function value:
RES	-	-	●	-	-	Is a variable of the type LONG and returns the result of the function as follows: 0 = OK. Bytes were written as intended. -1 = Warning: not enough bytes in **Source** variable. **Number** limited to length of **Source**. -2 = Warning: reached end of backup RAM memory. -3 = Warning: no **Source** bytes. -4 = Warning: address out of backup RAM area. -17 = Warning: WRITE_BACKUP_RAM not supported in this Tiger module

!

WRITE_BACKUP_RAM is supported as of module version 3.10m! The module version can be inquired at runtime with the function SYSVARN or via the command *View->Tiger-Status* in the TIGER-Basic IDE.

For a detailed description of the backup RAM please refer to READ_BACKUP_RAM.

Program example:

```
'------------------------------------------------------------------------------
'Name:  WRITE_BACKUP_RAM.TIG
'------------------------------------------------------------------------------
user_var_strict
#include define_a.inc                        ' include global definitions
#include ufunc4.inc                          ' include global definitions

task main                                    ' begin task MAIN
  long llResult                              ' error/sucess code
  string slBackupRam$                        ' result of WRITE_BACKUP_RAM(String)
  long llBackupRam                           ' result of WRITE_BACKUP_RAM(Long)
  real rlBackupRam                           ' result of WRITE_BACKUP_RAM(Real)

  install_device #LCD, "lcd1.tdd"            ' install LCD-driver

  llResult = WRITE_BACKUP_RAM(0, "Hello World!", 0, 12) ' write String
  llResult = WRITE_BACKUP_RAM(12, 123, 0, 4)     ' write 123 to backup RAM
  llResult = WRITE_BACKUP_RAM(16, 1.23, 0, 8)    ' write 1.23 to backup RAM

  slBackupRam$ = READ_BACKUP_RAM$(0, 12, llResult)      ' read String
  llBackupRam  = READN_BACKUP_RAM(12, 4, llResult)      ' read Long
  rlBackupRam  = READR_BACKUP_RAM(16, 8, llResult)      ' read Real

  print #LCD, "<1>BACKUP_RAM:"                   ' print result to LCD
  print #LCD, slBackupRam$                       ' String
  print #LCD, llBackupRam                        ' Long
  print #LCD, rlBackupRam                        ' Real
end                                              ' end task MAIN
```

See also: READ_BACKUP_RAM

OTYPE_PIN

OTYPE_PIN Log_Portadr., Bitposition , Output_Type

Configures the output type of an individual pin within a bit-oriented internal I/O port.

Parameters:

	B	W	L	S	F	
Log_Portadr	●	●	●	-	-	is a variable, constant or expression of the type BYTE, WORD or LONG and specifies the logical port address.
Bitposition	●	●	●	-	-	is a variable, constant or expression of the type BYTE, WORD or LONG and specifies the position of the bit. For Bitposition > 7 the complete port is set.
Output_Type	●	●	●	-	-	is a variable, constant or expression of the type BYTE, WORD or LONG and specifies the output type of the I/O line.

Output_Type	I/O-Pin
0	Push pull (reset state)
1	Open-drain

Program example:

```
'----------------------------------------------------------------------
' Name:  OTYPE_PIN.TIG
'----------------------------------------------------------------------
TASK MAIN                                'begin task MAIN
  OTYPE_PIN 8, 7, 1                      'Set bit 7 as open-drain
  DIR_PIN   8, 7, 0                      'port 8, bit 7 is output
  LOOP 9999999                           'many loops
    OUT 8,10000000b, 128                 'set port 8, bit 7 open-drain high
    WAIT_DURATION 500                    'wait 500 ms
    OUT 8,10000000b, 0                   'set port 8, bit 7 open-drain low
    WAIT_DURATION 500                    'wait 500 ms
  ENDLOOP
END                                      'end task MAIN
```

See also: OTYPE_PORT

OTYPE_PORT

OTYPE_PORT Log_Portadr., Output_Type

Configures the output type of all pins of an internal I/O port.

Parameters:

	B	W	L	S	F	
Log_Portadr	●	●	●	-	-	is a variable, constant or expression of the type BYTE, WORD or LONG and specifies the logical port address.
Output_Type	●	●	●	-	-	is a variable, constant or expression of the type BYTE, WORD or LONG and specifies the output type of the I/O lines bitwise.

Output_Type	I/O-Portbit
0	Push pull (reset state)
1	Open-drain

The instruction

OTYPE_PORT 8, 1

sets pin 0 to open drain and all other pins to push pull:

OTYPE_PORT 8, 1							
L87	L86	L85	L84	L83	L82	L81	L80
push pull	push pull	push pull	push pull	push pull	push pull	push pull	Open drain

Program example:

```
'------------------------------------------------------------------
' Name: OTYPE_PORT.TIG
'------------------------------------------------------------------
TASK MAIN                                   'begin task MAIN
  OTYPE_PORT 8, 255                         'Port 8 is open-drain
  DIR_PORT 8, 0                             'Set Port 8 as output
  OUT 8, 255, 01010101b                     'set all even bits open drain high
END                                         'end task MAIN
```

See also: OTYPE_PIN

PU_PD_PIN

PU_PD_PIN Log_Portadr., Bitposition , PullUp_PullDown

Configures the pull-up or pull-down of an individual pin within a bit-oriented internal I/O port.

Parameters:

	B	W	L	S	F	
Log_Portadr	●	●	●	-	-	is a variable, constant or expression of the type BYTE, WORD or LONG and specifies the logical port address.
Bitposition	●	●	●	-	-	is a variable, constant or expression of the type BYTE, WORD or LONG and specifies the position of the bit. For Bitposition › 7 the complete port is set.
PullUp_PullDown	●	●	●	-	-	is a variable, constant or expression of the type BYTE, WORD or LONG and specifies the pull-up or pull-down of the I/O line.

PullUp_PullDown	I/O-Pin
0	No pull-up, pull-down
1	Pull-up
2	Pull-down

Program example:

```
'---------------------------------------------------------------------
' Name: PU_PD_PIN.TIG
'---------------------------------------------------------------------
TASK MAIN                                'begin task MAIN
  PU_PD_PIN 8, 0, 0                      'No pull-up or pull-down on L80
END                                      'end task MAIN
```

See also: PU_PD_PORT

PU_PD_PORT

OTYPE_PORT Log_Portadr., PullUp_PullDown

Configures the pull-up or pull-down of all pins of an internal I/O port.

Parameters:

	B	W	L	S	F	
Log_Portadr	●	●	●	-	-	is a variable, constant or expression of the type BYTE, WORD or LONG and specifies the logical port address.
PullUp_PullDown	●	●	●	-	-	is a variable, constant or expression of the type BYTE, WORD or LONG and specifies the pull-up or pull-down of the I/O lines bitwise as 16-bit value.

PullUp_PullDown	I/O-Portbit
0 / 00b	No pull-up, pull-down
1 / 01b	Pull-up
2 / 10b	Pull-down

Pull-up pull-down 16-bit value																
Pin	7		6		5		4		3		2		1		0	
Bit-No.	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

The instruction

PU_PD_PORT 8, 024H

sets pin1to pull-up, pin 2 to pull-down and all other pins to no pull-up, pull down

PU_PD_PORT 8, 024H							
L87	L86	L85	L84	L83	L82	L81	L80
No pull-up, pull-down	No pull-up, pull-down	No pull-up, pull-down	No pull-up, pull-down	No pull-up, pull-down	Pull-down	Pull-up	No pull-up, pull-down

Program example:

```
'----------------------------------------------------------------------
' Name: PU_PD_PORT.TIG
'----------------------------------------------------------------------
TASK MAIN                                'begin task MAIN
  PU_PD_PORT 8, 0000000000000000b        'No pull-up or pull-down on port 8
END                                      'end task MAIN
```

See also: PU_PD_PIN

ERASE_FLASH_SECTOR

ERASE_FLASH_SECTOR Start address, Length [, Error handling]

Deletes one FLASH-sector without blocking BASIC code execution.

Parameters:

	B	W	L	S	F	
Start address	●	●	●	-	-	is the FLASH address where the erase process is to start. This must be exactly a sector's start address.
Length	●	●	●	-	-	is the number of bytes, which are to be erased. The length must always be exactly the length of one sector.

ERASE_FLASH_SECTOR instruction is supported as of Tiger IDE version **6.0.23** with Tiger plus Firmware **3.12a** or newer!

Tiger BASIC® programs can use the Data FLASH to store data. The first FLASH address that can be used for data storage is 0, the last address which can be used depends on the length of the Data FLASH. Precise values can be obtained by inquiring the system variables with the function SYSVARN.

ERASE_FLASH_SECTOR can be used to erase a single sector. The exact start address of the sector must be known and the erase length must be the sector length. Otherwise, this instruction will not be carried out during the runtime (generates runtime error). If the ERASE_FLASH_SECTOR command is successfully initiated, the FLASH is busy for a short while and cannot be addressed.

This instruction can use its own Error handling in the form of a subroutine or branch.
Notation:

ERASE_FLASH_SECTOR ***Start address, Length*****, ON_ERROR_CALL** ***Subroutine***
ERASE_FLASH_SECTOR ***Start address, Length*****, ON_ERROR_GOTO** ***Label***

Unlike ERASE_FLASH, ERASE_FLASH_SECTOR does not wait for the erase process to finish before returning. ERASE_FLASH_SECTOR is executed in the background, parallel to BASIC code execution.

Attention: If any flash operation is executed while an erase is in progress, the flash operation will wait for the end of the erase!

To better control flash operation and waiting time you can use the SYSVARN with FLASH_BUSY to check if the erase has finished.

Program example:

```
'----------------------------------------------------------------------
'    Name:  Erase_flash_sector.tig
'----------------------------------------------------------------------
user_var_strict
#include define_a.inc

task main
  long user_flash_size
  long i

  install device #LCD, "LCD1.TDP"
  run_task erase_complete_flash

  for i = 0 to 99999
    print #1, "<1BH>A<0><0><0F0H>running";i;" sec"
    wait_duration 1000
  next
end

task erase_complete_flash
    long flash_sectors
    long sector_size
    long i
    long busy

    flash_sectors   = sysvarn ( FLASH_DSEC,  0 )
    sector_size     = sysvarn ( FLASH_SSIZE, 0 )

    for i = 0 to flash_sectors - 1
        print #1, "<1BH>A<0><1><0F0H>sectors erased:"; i
        erase_flash_sector i * sector_size, sector_size
        busy = 1
        while busy > 0
            busy = sysvarn(FLASH_BUSY, 0)
        endwhile
    next

    print #1, "<1BH>A<0><2><0F0H>erase flash finished"
end
```

See also: SYSVARN

Device drivers

On principle, all device-drivers that can be found for the Tiger 1 (BASIC-Tiger, TINY-Tiger, Econo-Tiger) are also available for the Tiger *plus*. A distinction is made in the naming, however:

*.TDD:	Device driver for Tiger 1
*.TD2:	Device driver for Tiger 2
*.TDP:	Device driver for Tiger *plus*

There might be some differences for some drivers due to special specifications of the Tiger *plus*. These will be talked about in more detail later on.

There is no need to rename existing Tiger-1 device drivers in your source code. The BASIC compiler choose the correct device driver for the connected module.

Example for installing the serial driver for Tiger *plus* (and Tiger 1 and Tiger 2):

```
INSTALL_DEVICE #SER, "SER1B_K1.TDD", &
  BD_115_200, DP_8N, JA, &                    ' settings for SER0
  BD_115_200, DP_8N, JA                       ' settings for SER1
```

SER1B – serial interfaces

The SER1B serial interfaces only differ within the possible options for baudrates.

Baudrates:

Nr.	Symbol	Meaning	BASIC-Tiger TINY-Tiger Econo-Tiger	TINY-Tiger 2	Tiger *plus*
0	BD_50	50 Bd			
1	BD_75	75 Bd			
2	BD_110	110 Bd			
3	BD_150	150 Bd			
4	BD_200	200 Bd			
5	BD_300	300 Bd	available	available	
6	BD_600	600 Bd	available	available	
7	BD_900	900 Bd		available	available
8	BD_1_200	1,200 Bd	available	available	available
9	BD_1_800	1,800 Bd		available	available
10	BD_2_400	2,400 Bd	available	available	available
11	BD_3_600	3,600 Bd		available	available
12	BD_4_800	4,800 Bd	available	available	available
13	BD_7_200	7,200 Bd		available	available
14	BD_9_600	9,600 Bd	available	available	available
15	BD_14_400	14,400 Bd		available	available
16	BD_19_200	19,200 Bd	available	available	available
17	BD_28_800	28,800 Bd		available	available
18	BD_38_400	38,400 Bd	available	available	available
19	BD_57_600	57,600 Bd		available	available
20	BD_76_800	76,800 Bd	available	available	available
21	BD_115_200	115,200 Bd		available	available
22	BD_153_600	153,600 Bd	available	available	available
23	BD_230_400	230,400 Bd			
24	BD_307_200	307,200 Bd		available	available

Nr.	Symbol	Meaning	BASIC-Tiger TINY-Tiger Econo-Tiger	TINY-Tiger 2	Tiger *plus*
25	BD_460_800	460,800 Bd			
26	BD_614_400	614,400 Bd		available	available
32	BD_31_250	31,250 Bd	available	available	available
33	BD_62_500	62,500 Bd	available	available	available
34	BD_EXT	external Oscillator / 16 Connect to CTS pin		available	
35	BD_10_400	10,400 Bd		available	available
36	BD_41_600	41,600 Bd		available	available
37	BD_100_000	100,000 Bd		available	available
38	BD_26_000	26,000 Bd		available	available

!

There is no more UFCI_SER_TX_LOCK support in Tiger plus.

In Tiger plus by UFCI_SER_9ADR it is possible to get only the address that is set by UFCO_SER_9ADR. If address was not set yet, then default address 0 will be returned.

RTC1.TDP

The device-driver 'RTC1' supports the internal real time clock.

File name: RTC1.TDP

INSTALL DEVICE #*D*, "RTC1.TDP" [, P1]

D	is a constant, variable or an expression of data type WORD, LONG, BYTE in the range 0...63 and stands for the device number of the driver.
P1	is a flag and determines whether the driver uses real hardware RTC or software RTC. YES: the driver uses real hardware RTC (default value). NO: the driver uses software RTC.

Attention: In contrast to Tiger 1, the alarm time for Tiger *plus* can be set to a maximum of 1 month in advance.

User-function-codes of the RTC1.TDP

RTC1-user-function-codes and the corresponding answers of the driver:

No.	Symbol	Description
160	UFCI_RTC_STAT0	Status of the RTC chip
		Answer of the driver:
0	RTC_INITIAL	State immediately after power-on
1	RTC_INSTALLING	Installing still continues
2	RTC_NO_RTC	No RTC hardware available
3	RTC_PRESENT	OK, RTC hardware present
4	RTC_RETRY	Repeated attempt to find RTC
161	UFCI_RTC_STAT1	Status of the RTC device driver
		Answer of the driver:
0	RTC_READY	Ready
1	RTC_BUSY	Busy
162	UFCI_RTC_VOLTAGE	Status voltage drop
		Answer of the driver:
0	RTC_READY	There was no voltage drop, clock still running as initialized
1	RTC_VOLTAGE_LOW	Voltage of clock had been gone; it was initialized again at the install device.

Program sample:

```
'----------------------------------------------------------------------
' Name: RTC1_Tiger_plus.TIG
'----------------------------------------------------------------------
#include define_a.inc
#include ufunc4.inc                                     'User Function Codes

task Main                                               'begin task main
  long seconds, prev_sec, voltage                       'declare variables of
                                                        'type long
  install_device #LCD, "LCD1.TDD"                       'install LCD-driver
  install_device #RTC, "RTC1.TDD"                       'install RTC-driver

  RTCSTAT = RTC_INITIAL
  while RTCSTAT < RTC_NO_RTC                            'while searching for RTC
    get #RTC, #0, #UFCI_RTC_STAT0, 1, RTCSTAT           'get status of RTC
    print #LCD,"<1>installing";
    wait_duration 200
  endwhile
  if RTCSTAT = RTC_PRESENT then                         'if RTC found
    seconds = 12345678                                  'preset value
    get #RTC, #0, #UFCI_RTC_VOLTAGE, 0, voltage         'get Voltage Low Bit
    if voltage = RTC_VOLTAGE_LOW then
      print #LCD, "<01>";                               'cursor to top left
      print #LCD, "Voltage Low"                         'print to LCD
      print #LCD, "setting time";                       'print to LCD
      wait_duration 2000                                'give some time to
                                                        'notice text on LCD
      put #RTC, seconds                                 'set RTC in absolute
                                                        'seconds
    else
      print #LCD, "<01>";                               'cursor to top left
      print #LCD, "NO Voltage Low"                      'print to LCD
      print #LCD, "not setting time";                   'print to LCD
      wait_duration 2000                                'give some time to
                                                        'notice text on LCD
    endif

    while 1 = 1                                         'endless loop
      prev_sec = seconds                                'store old time
      while seconds = prev_sec                          'while current = old
                                                        'time
        get #RTC, 0, seconds                            'read RTC
      endwhile
      print #LCD,"<1>RTC-Time =<0>";seconds;            'if new time, show it
    endwhile
  else                                                  'if no RTC
    print #LCD, "<1>No RTC found"
  endif
end                                                     'end task main
```

ANALOG1.TDP

The device driver 'ANALOG1' reads the instantaneous value of the analog inputs.

INSTALL DEVICE #D, "ANALOG1.TDP"

D is a constant, variable or an expression of data type WORD, LONG, BYTE in the range 0...63 and stands for the device number of the driver.

The device driver ANALOG1.TDD reads the internal analog inputs. The instantaneous values are read. The resolution is 8 bit if BYTEs are read (e.g.: GET #n,#sa,1,CHAR) or 10 bit if WORD or LONG values are read. For secondary addresses from 100 12-bit values are read.

The resolution can be improved and the noise "calculated out" with the aid of the FIFO buffer and the command INTEGRAL_FIFO.

Secondary addresses for TINY, BASIC and ECONO Tiger

Sec. address	Function	Instruction
0	Reads from A/D channel 0 (8 bit or 10 bit)	GET
1	Reads from A/D channel 1 (8 bit or 10 bit)	GET
2	Reads from A/D channel 2 (8 bit or 10 bit)	GET
3	Reads from A/D channel 3 (8 bit or 10 bit)	GET
4	Reads all 4 A/D channels (8 bit)	GET
5	Reads all 4 A/D channels (10 bit)	GET
100	Reads from A/D channel 0 (12 bit)	GET
101	Reads from A/D channel 1 (12 bit)	GET
102	Reads from A/D channel 2 (12 bit)	GET
103	Reads from A/D channel 3 (12 bit)	GET
112	Reads all 4 A/D channels (12 bit)	GET

Secondary addresses for Tiger 2

Sec. address	Function	Instruction
0	Reads from A/D channel 0 (8 bit or 10 bit)	GET
1	Reads from A/D channel 1 (8 bit or 10 bit)	GET
2	Reads from A/D channel 2 (8 bit or 10 bit)	GET
3	Reads from A/D channel 3 (8 bit or 10 bit)	GET
4	Reads from A/D channel 4 (8 bit or 10 bit)	GET
5	Reads from A/D channel 5 (8 bit or 10 bit)	GET
6	Reads from A/D channel 6 (8 bit or 10 bit)	GET
7	Reads from A/D channel 7 (8 bit or 10 bit)	GET
8	Reads from A/D channel 8 (8 bit or 10 bit)	GET
9	Reads from A/D channel 9 (8 bit or 10 bit)	GET
10	Reads from A/D channel 10 (8 bit or 10 bit)	GET
11	Reads from A/D channel 11 (8 bit or 10 bit)	GET
12	Reads all 12 A/D channels (8 bit)	GET
13	Reads all 12 A/D channels (10 bit)	GET
100	Reads from A/D channel 0 (12 bit)	GET
101	Reads from A/D channel 1 (12 bit)	GET
102	Reads from A/D channel 2 (12 bit)	GET
103	Reads from A/D channel 3 (12 bit)	GET
104	Reads from A/D channel 4 (12 bit)	GET
105	Reads from A/D channel 5 (12 bit)	GET
106	Reads from A/D channel 6 (12 bit)	GET
107	Reads from A/D channel 7 (12 bit)	GET
108	Reads from A/D channel 8 (12 bit)	GET
109	Reads from A/D channel 9 (12 bit)	GET
110	Reads from A/D channel 10 (12 bit)	GET
111	Reads from A/D channel 11 (12 bit)	GET
112	Reads all 12 A/D channels (12 bit)	GET

Examples:

GET #AD1, #0, 1, value	reads from the Analog1 driver from A/D-channel 0 exactly 1 byte into variable 'value' (8 bit resolution). Value is of type BYTE, WORD or LONG.
GET #AD1, #1, 2, value	reads from the Analog1 driver from A/D-channel 1 exactly 2 bytes into variable 'value' (10 bit resolution). Value is of type WORD or LONG.
GET #AD1, #102, 2, value	reads from the Analog1 driver from A/D-channel 2 exactly 2 bytes into variable 'value' (12 bit resolution). Value is of type WORD or LONG.
GET #AD1, #112, 0, V$	reads from the Analog1 driver from all A/D-channels exactly 2 byte per channel into V$ (12 bit resolution). V$ is of type STRING and must be large enough to accommodate 8 Bytes for TINY, BASIC and ECONO Tiger (4 A/D channels) or 24 Bytes for Tiger-2 (12 A/D channels). The low value byte from channel 0 is the first byte. The value of a channel can, e.g., be read from the string like this (CH = channel number): **Value = NFROMS (V$, CH*2, 2)**

Program sample:

```
'---------------------------------------------------------------------
'    Name:  ANALOG1_T2plus.tig
'---------------------------------------------------------------------
user_var_strict                          '
#include define_a.inc

TASK Main                                ' begin Task MAIN
  ARRAY Value(12) OF LONG                ' LONG-Array declaration
  String result$                         ' String declaration
  LONG K                                 ' LONG variable declaration
  byte pos                               ' BYTE variable declaration

  INSTALL_DEVICE #LCD, "LCD1.TD2"        ' install LCD-Driver (Tiger 2)
  INSTALL_DEVICE #AD1, "ANALOG1.TDP"     ' Analog-Inputs Device Driver

  ' 1. example: Read out ONLY 1 channel and with 8-Bit resolution
  FOR K = 0 TO 11                        ' 12 channels (0 - 11)
    GET   #AD1, #K, 1, Value(K)          ' read out value from ADC from
                                         ' channel K 8-Bit resolution(1 Byte)
    PRINT #LCD, "<1>";                   ' delete LCD
    PRINT #LCD, "Single Ch. 8-Bit:"      ' show info on LCD
    PRINT #LCD, "AD"; K; ":";            ' show channel number
    PRINT #LCD, Value(K)                 ' show value on LCD
    WAIT_DURATION 500                    ' wait 500ms
  NEXT                                   ' next channel
  WAIT_DURATION 1000                     ' wait 1 second

  ' 2. example: Read out ONLY 1 channel and with 10-Bit resolution
  FOR K = 0 TO 11                        ' 12 channels (0 - 11)
    GET   #AD1, #K, 2, Value(K)          ' read out value from ADC from
                                         ' ch. K 10-Bit resolution(2 Byte)
    PRINT #LCD, "<1>";                   ' delete LCD
    PRINT #LCD, "Single Ch. 10-Bit:"     ' show info on LCD
    PRINT #LCD, "AD"; K; ":";            ' show channel number
    PRINT #LCD, Value(K)                 ' show value on LCD
    WAIT_DURATION 500                    ' wait 500ms
  NEXT                                   ' next channel
  WAIT_DURATION 1000                     ' wait 1 second

  ' 3. example: Read out ONLY 1 channel with 12-Bit resolution
  FOR K = 0 TO 11                        ' 12 channels (0 - 11)
    GET   #AD1, #K+100, 2, Value(K)      ' read out value from ADC from
                                         ' ch. K 12-Bit resolution (2 Byte)
    PRINT #LCD, "<1>";                   ' delete LCD
    PRINT #LCD, "Single Ch. 12-Bit:"     ' show info on LCD
    PRINT #LCD, "AD"; K; ":";            ' show channel number
    PRINT #LCD, Value(K)                 ' show value on LCD
    WAIT_DURATION 500                    ' wait 500ms
  NEXT                                   ' next channel
  WAIT_DURATION 1000                     ' wait 1 second

  ' 4. example: Read out ALL Channels with 8-Bit resolution
  GET #AD1, #12, 12, result$             ' read ALL channels with 8-Bit
                                         ' resolution in String (12 Byte)
  FOR pos=0 TO 11 STEP 1                 ' 12 channels (0 - 11)
    PRINT #LCD, "<1>";                   ' delete LCD
    PRINT #LCD, "All Ch. 8-Bit:"         ' show info on LCD
```

```
    PRINT #LCD, "AD"; pos; ":";          ' show channel number
    PRINT #LCD, NFROMS(result$,pos,1)    ' show value of channel
    WAIT_DURATION 500                    ' wait 500ms
  NEXT                                                 '
  WAIT_DURATION 1000                     ' wait 1 second

  ' 5. example:         Read out ALL Channels with 10-Bit resolution
  GET #AD1, #13, 24, result$             ' read ALL channels with 10-Bit
                                         ' resolution in String (24 Byte)
  FOR pos=0 TO 11 STEP 1                 ' 12 channels (0 - 11)
    PRINT #LCD, "<1>";                   ' delete LCD
    PRINT #LCD, "All Ch. 10-Bit:"        ' show info on LCD
    PRINT #LCD, "AD"; pos; ":";          ' show channel number
    PRINT #LCD, NFROMS(result$,pos*2,2)  ' show result to LCD
    WAIT_DURATION 500                    ' wait 500ms
  NEXT                                                 '

  ' 6. example: Read out ALL Channels with 12-Bit resolution
  GET #AD1, #112, 24, result$            ' read ALL channels with 12-Bit
                                         ' resolution in String (24 Byte)
  FOR pos=0 TO 11 STEP 1                 ' 12 channels (0 - 11)
    PRINT #LCD, "<1>";                   ' delete LCD
    PRINT #LCD, "All Ch. 12-Bit:"        ' show info on LCD
    PRINT #LCD, "AD"; pos; ":";          ' show channel number
    PRINT #LCD, NFROMS(result$,pos*2,2)  ' show result to LCD
    WAIT_DURATION 500                    ' wait 500ms
  NEXT                                                 '

END
```

ANALOG2.TDP

The device-driver ANALOG2 reads in analog values controlled by the time basis device driver 'TIMERA' and stores them in a FIFO-buffer (FIFO=First-In-First-Out) or a string.

Further information about ANALOG2.TDP:

- User-function-codes
- Measuring with trigger

File name: ANALOG2.TDP

INSTALL DEVICE #D, "ANALOG2.TDP"

D is a constant, variable or an expression of data type WORD, LONG, BYTE in the range 0...63 and stands for the device number of the driver.

The device driver ANALOG2.TD2 reads in analog values from the internal analog channels into a FIFO buffer or a string. The measurements are synchronized with the help of the time basis driver 'TIMERA.TD2' so that they are taken independent of the BASIC program and up to high speeds. The time basis driver provides a basic frequency that is divided down through the prescaler of the driver ANALOG2 to the actual measuring rate. The setting of the prescaler can be changed through commands (user-function-code) to the driver.

Please note: TIMERA.TD2 must be integrated before ANALOG2.TD2.

The driver supports the resolutions 8-bit, 10-bit and 12-bit. The 12-bit resolution is extrapolated from a 10-bit reading using numerical integration. The analog values can be read in either into a string or a FIFO buffer. The following reading modes are supported:

- from a single channel (0, 1, 2, 3)
- from channel 0 and 1
- from channel 0, 1 and 2
- from channel 0, 1, 2 and 3

There are therefore many different settings, from which channel in what resolution to where the analog values are read in. For this purpose, the speed (measure or sample rate) can be adjusted in many ways. In addition, options can be selected that relate to the behavior of the reading as far as strings or FIFO-buffer is concerned. Therefore, following is some information concerning the differences between 'measurement in string' and 'measurement in FIFO' and what has to be paid attention to with the different settings.

For setting up the analog measuring system, there are several user-function codes, which are defined as symbolical names in UFUNCn.INC. Settings that have been carried out once are maintained and must not be done again before each measurement. If options are given explicitly at the start of the measurement (offset in the string, number of measurements), then these are valid only for this one measurement. The settings that have been made beforehand with the help of the user-function-codes will be maintained.

The following table shows an overview of the function-codes of this driver. The file UFUNCx.INC must be integrated, so that the compiler knows the command symbols.

User-function-codes of the ANALOG2.TD2

User-function-codes of the ANALOG2.TD2 for setting of parameters (PUT):

No.	Symbol	Description
46	UFCO_AD2_RESET	Set all parameters to default values
128	UFCO_AD2_CHAN	Set single channel mode (FIFO, STRING): 0, 1, 2, 3 (default: 1) This channel is also the measured channel in the mode multi-channel measurement, if only one channel is set.
129	UFCO_AD2_RESO	Set resolution (FIFO, STRING): 8 = 8-bit (default) 10 = 10-bit 12 = 12-bit
130	UFCO_AD2_INTEG	Integration-width at 12-bit (FIFO, STRING): 16, 32, 64, or 128 (default: 16)
131	UFCO_AD2_STOVL	Flag: "Stop-on-FIFO-overflow" (FIFO) 0 = YES n = no = wrap-around for FIFO It is always stopped with strings.
132	UFCO_AD2_CNT	Number of measures (per channel) (FIFO) 0 = endless (only for FIFO, default) n = number (LONG)
133	UFCO_AD2_PSCAL	Pre-scaler, divides the basic frequency of the driver "TIMERA.TDD" down (FIFO, STRING): 0,1 = without pre-scaler n = divider (WORD)
134	UFCO_AD2_STOP	Stop AD-sampling (FIFO, STRING): only DUMMY-parameter

No.	Symbol	Description
136	UFCO_AD2_SCAN	Set multi-channel mode and number of channels (FIFO, STRING): n = 1: the last channel to be set with UFCO_AD2_CHAN n = 2: 2-channel: Ch-0, Ch-1 n = 3: 3-channel: Ch-0, Ch-1, Ch-2 n = 4: 4-channel: Ch-0, Ch-1, Ch-2, Ch-3
137	UFCO_AD2_ISAMP	Integral-samples (FIFO, STRING): tells which measurement is to be written into the target buffer (e.g. every 2nd, every 10th, ...). Is only valid when INTEGRATION is done (only for 12-bit) values: 1...65535 (WORD)
138	UFCO_AD2_TRIG_SAMPLE	Sets the number of samples that are measured after the trigger event occurs and at the same time activates the trigger mode. To deactivate, set to 0FFFFH.
139	UFCO_AD2_TRIG_HLEV	Sets the high trigger level. When measurement is **exceeding** this value, the trigger event sets in. **Exactly 4, 8 or 12 WORDs are expected** (one WORD for each channel)
140	UFCO_AD2_TRIG_LLEV	Sets the low trigger level. When measurement is **falling below** this value, the trigger event sets in. **Exactly 4, 8 or 12 WORDs are expected** (one WORD for each channel)
143	UFCO_AD2_PSCIMM	Sets the pre-scaler during the running measurement.

User-function-codes of the ANALOG2.TD2 for reading in parameters (GET):

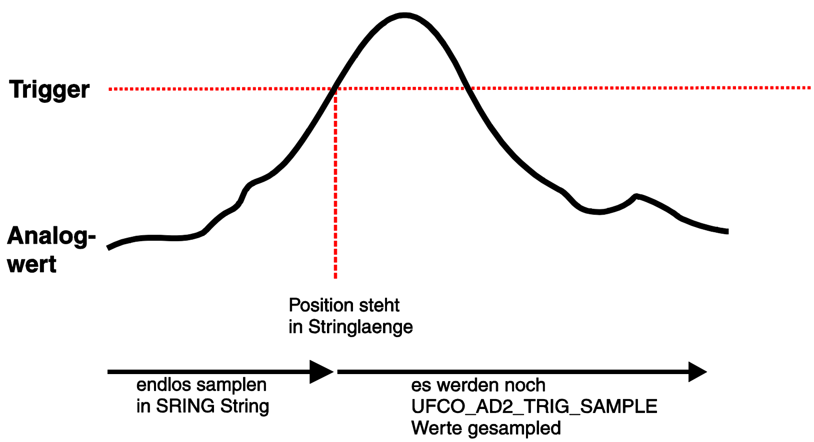
No.	Symbol	Description
68	UFCI_CPU_LOAD	Read the CPU-performance that is consumed by this driver (100%=10.000)
99	UFCI_DEV_VERS	Version of the driver
145	AD2_MEAS_ACT	Reads out if driver is currently measuring. 0 = not running 1 = running
146	AD2_RELOAD_FLAG	Reads out if a reload string is available for continuous sampling 0 = no reload string available 1 = reload string is available
147	AD2_MEAS_REST	Number of remaining measurements that fit into used FIFO or STRING + reload STRING
148	AD2_TRIG_POS	Reads out the trigger position, when the trigger event has occurred
149	AD2_STRI_WRITE	Reads out the current writing position in the string
150	AD2_STRI_OVL	Reads out, whether the string has already overrun once in trigger mode. 0: string overrun at least one time 0FFH: String has not overrun yet

Measuring with trigger

Measuring with trigger is activated with the User-Function-Code UFCO_AD2_TRIG_SAMPLE. When a value is set here, a trigger is used for sampling, to work without trigger again, this value simply has to be set to 0FFFFH.

When measuring with trigger, first, there is endless sampling. When the end of the string is reached, writing continues at the beginning, in this case the string is a ring buffer, who continuously keeps the most recent values. The length of the string at this time is 0FFFFFFFFH for Tiny Tiger 2 and Tiger plus series and 0FFFFH for first Tiger generation. This does not correspond to the real length, but is a flag for the situation that the trigger event has not occurred yet. As soon as the string overflows for the first time, you will read out a 0 with the User-Function-Code UFCI_AD2_STRI_OVL. The most recent writing position can continually be queried with the User-Function-Code UFCI_AD2_STRI_WRITE.

As soon as the measurement value in a channel exceeds the set trigger limit(s), the trigger event sets in. The length of the string now has the value 0FFFFFFFEH for Tiny Tiger 2 and Tiger plus series and 0FFFEH for first Tiger generation, so that it becomes clear that the trigger has already occurred. Now, exactly as many samples are done as were set in the User-Function-Code UFCO_AD_TRIG_SAMPLE, then the measurement is stopped. The length of the string is set to the position at which the trigger event occurred; the length thus is a marking. After that, the length of the string should be set back to the maximum length in the BASIC program. Now the string can be evaluated. A new measurement can be started normally at any time.



Measuring with trigger is restricted to strings and not possible with FIFO !!!

Program sample:

```
user_var_strict
#INCLUDE DEFINE_A.INC                         ' common defines
#INCLUDE UFUNC4.INC                           ' User Function Codes
#define MLEN    200
#define TLEVEL  700
STRING M$ (MLEN)                               ' meassurement-string (global!)

TASK MAIN                                     ' begin Task MAIN
' TIMER-A driver installation (Zeitbasis Timer: 1001Hz)
  INSTALL_DEVICE #TA, "TIMERA.TD2", 3, 156
' ANALOG-2 driver installation
  INSTALL_DEVICE #AD2, "ANALOG2.TD2"

  word  t0,t1,t2,t3                           ' trigger level
  long K

  t0 = TLEVEL                                 ' set trigger level for channel 0
  t1 = TLEVEL                                 ' set trigger level for channel 1
  t2 = TLEVEL                                 ' set trigger level for channel 2
  t3 = TLEVEL                                 ' set trigger level for channel 3

  M$=""                                       ' meassurement-string empty
  PUT #AD2,#0,#UFCO_AD2_PSCAL, 0              ' no pre-scaler
  PUT #AD2,#0,#UFCO_AD2_RESO, 10              ' resolution
  PUT #AD2,#0,#UFCO_AD2_CHAN, 0               ' channel
  PUT #AD2,#0,#UFCO_AD2_SCAN, 4               ' no. of channels
  PUT #AD2,#0,#UFCO_AD2_TRIG_SAMPLE, 10       ' samples after trigger
  PUT #AD2,#0,#UFCO_AD2_TRIG_HLEV, t0, t1, t2, t3 ' set trigger for channels
  PUT #AD2,M$                                 '

  #ifdef TIGER_1          ' codeblock for 1st generation Tiger
  K = 0FFFFH              ' init k
  while K >= 0FFFEH       ' wait for trigger and
                          ' end of meassurement
    K = len(M$)           ' read flag
  endwhile
  #endif

  #ifdef TIGER_2          ' codeblock for 2st generation Tiger
  K = 0FFFFFFFFH          ' init k
  while K < 0             ' wait for trigger and
                          ' end of meassurement
    K = len(M$)           ' read flag
  endwhile
  #endif

  #ifdef TIGER_PLUS       ' codeblock for Tiger plus
  K = 0FFFFFFFFH          ' init k
  while K < 0             ' wait for trigger and
                          ' end of meassurement
    K = len(M$)           ' read flag
  endwhile
  #endif

  set_len$(M$,MLEN)       ' meassurement finished, set real length
END                                           ' Ende Task MAIN
```

The low level trigger works analog to this. When the measured value falls below the trigger level, the trigger event occurs. High level and low level triggers can be combined in any way; both can be used for one channel at the same time, as well.

If a trigger is to be turned off for a channel, it is set to a limit value which can never be exceeded. For the low level trigger 0 is selected, for the high level trigger 0FFFFH is selected, e.g.

When the trigger measurement is activated, but all triggers are deactivated, the string is simply sampled into, which can of course be read out at any time, until the measurement is stopped manually.

Please note:

At the 8-bit trigger measurement only the lower 8 bit of the trigger level are taken into account. The value 100H thus corresponds to an 8-bit trigger value of 0!

CAN-Bus

The device driver 'CAN1_xx.TDP' supports the internal CAN interface of the TINY-Tiger 2 plus.

This section contains:

Differences to TCAN & Tiny Tiger 2

The CAN1 device driver of the Tiny Tiger 2 plus has minor deviations from the versions used with Basic-Tiger-CAN (TCAN) and Tiny Tiger 2.

Dual-Filter configuration

Like the Tiny Tiger 2, the Tiny Tiger 2 plus does not support the dual filter mode present in the TCAN version. Only single 32bit filters are usable.
To set more than one CODE and MASK combination, please refer to the section **Setting of more access codes in standard format** or **Setting of more access codes in extended format**

User-Function-Codes that are no longer present

UFCI_CAN_ALC
UFCI_CAN_ECC
UFCI_CAN_EWL
UFCI_CAN_RMC
UFCO_ERRC_RESET
UFCO_CAN_CMD
UFCO_CAN_EWL

Setting multiple access codes with global acceptance mask

Using the global mask for additional access codes on Tiny Tiger 2 had the effect, that the IDE bit was ignored, even when it was set in the global acceptance mask.
Tiny Tiger 2 plus will now use the IDE bit correctly.
If your program needs to ignore the IDE bit, set it to "do not care" in the global acceptance mask.
See section **Set Access-Code and Access-Mask** for details on mask bits and **Setting of more access codes in standard format** or **Setting of more access codes in extended format** for details on the usage of global and local acceptance mask

Bus-Off recovery

The CAN chip will recover from Bus-Off (become error active again) automatically. It will start the recovering sequence (128 occurrences of 11 consecutive recessive bits monitored on CANRX) automatically after it has entered Bus-Off state.

Description of the device driver CAN1_xx.TDP

This device driver enables input and output on the CAN-bus in connection with the TINY-Tiger 2 plus. The parameters of the CAN interface can be specified during installation of the driver. Some parameters can also be changed during the running time by commands to the driver.

File names: CAN1_K8.TDP (with 8K buffers)
CAN1_K1.TDP (with 1K buffers)
CAN1_R1.TDP (with 256 byte buffers)

INSTALL DEVICE #*D*, "CAN1_xx.TDP" [, *Code, Mask, Bt0, Bt1, Mod, Outctrl*]

D is a constant, variable or expression of the data type BYTE, WORD, LONG in the range 0...63 and stands for the device number of the driver.

Code is a parameter to determine the Access-Code. 'Code' is always 4 bytes long. The range of values for the Access code with standard frames is 0...7FFh and with extended frames 0...1FFF FFFF.
Default value: 0

Mask is a parameter to determine the acceptance filter. 'Mask' is always 4 bytes long.
Default value: 0FFFFFFFFh

Bt0 (Bustiming-Register-0) is a parameter to determine the baud rate-prescalers and the synchronisation step (1 byte). This determines the transfer rate together with Bt1.
Default value: 0

Bt1 (Bustiming-Register-1) is a parameter to determine the Bus-Timing and the number of samples during receipt (1 byte). This also determines the transfer rate together with Bt0.
Default value: 2Fh (Tseg1=15, Tseg2=2)

Mod is a parameter to determine the mode (1 byte) .
Default value: 0

Bit	Symbol	if bit set ('1')
1	CAN_LISTEN	Listen-Only-Mode
2	CAN_SELFTEST	Selftest-Mode
3		reserved
4	CAN_SLEEP	Sleep-Mode
0,5,6		reserved

If the Listen-Only mode is installed the driver tries to automatically recognize the bit rate on the bus on the basis of a table with predefined bit rates.

Outctrl is a dummy parameter. Default value is 1Ah.

Example for an installation for 500 kBit:

```
install_device #CAN, "CAN1_K1.TD2", &
0,0,0,0, &                          ' access code
0ffh,0ffh,0ffh,0ffh, &              ' access mask
0,2Fh, &                            ' bustim1, bustim2
0,1Ah                               ' mode, outctrl
```

CAN messages in the I/O-buffer of the driver

The I/O buffers of the Tiger-BASIC-CAN device driver always contains complete CAN messages and no further bytes. A CAN message starts with the Frame-Info-byte, which determines whether this is a message with an 11 or 29-Bit-Identifier and how many data bytes are contained therein. The Frame-Info-Byte also contains the RTR-bit. This is followed by 3 Identifier-bytes (standard frame) or 5 Identifier-bytes (extended frame) and then the data bytes depending on the frame type. A CAN message can transfer 0...8 bytes as useful data.

The Frame-Info-Byte also contains information on

- the frame type (11 or 29 ID-Bits)
- the number of data bytes (0...8)
- whether this is a Remote-Transmit-Request

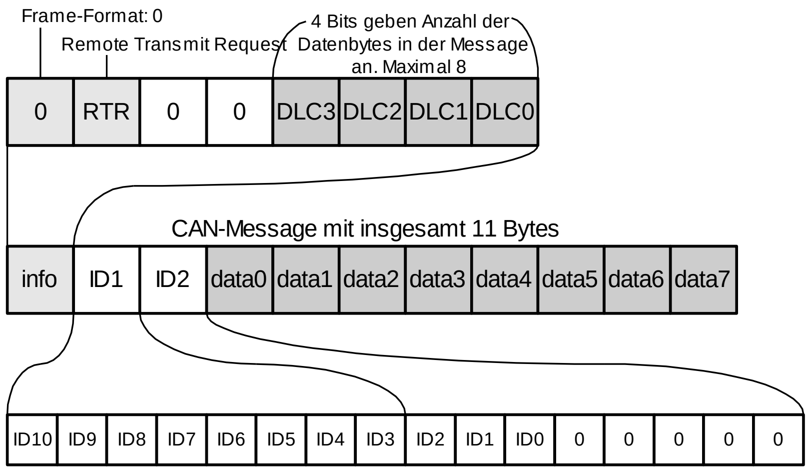
The Identifier can

- be 29 bits long and the occupies 4 bytes in the buffer
- be 11 bits long and then occupies 2 bytes in the buffer

A standard frame occupies a maximum of 11 bytes, an extended frame a maximum of 13 bytes in the buffer. If the device driver does not have at least 13 bytes free in the buffer free during receipt the message will be rejected and an error registered 'Buffer overflow'. Between 341 messages (only standard frames without data) and 78 message (only extended frames, all with 8 data bytes) fit in a 1kByte buffer depending on the length of the individually received CAN message.

Standard frame

The illustration shows the structure of the standard frame with enlarged Frame-Info-Byte (top) and the ID-byte (enlarged bottom). The length of the message is set automatically by the device driver. The 11 ID-bits must first be flush left with the highest-order bit in the two bytes, as shown in the illustration.



Structure of the 'Standard Frame'

Standard Frame, Info-bits:

FF	Frame-Format bit, here FF=0. 0: Standard Frame 1: extended Frame
RTR	Remote Transmit Request, send request. Messages with a set RTR-bit will be responded directly by the driver, if a reply is specified.
DLC	4 bits specify the number of data bytes in the message (0...8). This bit sets the device driver.

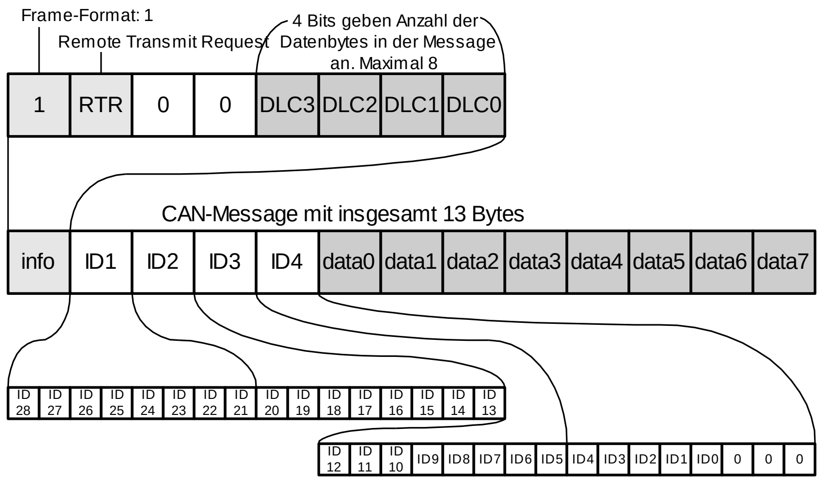
The 11-Bit-Identifier of the CAN message can be found in both ID-bytes, offset by 5 bits to the left. The format here is 'high-byte first', unlike the WORD variables in Tiger-BASIC which are 'low-byte first'.

The ID-bytes are followed by as many data bytes as specified by DLC.

Example for the generation of standard frames in Tiger-BASIC:

```
t_id = 7FFh shl 5                   ' Transmit-ID, left-aligned in WORD
' Standard frame with frame info byte, 2 empty ID bytes, data
msg$ = "<0><0><0>" + data$
msg$ = ntos$ ( msg$, 1, -2, t_id ) ' fit in ID with high-byte first
                                    ' length is set by driver
print #CAN, msg$;                   ' PRINT, with semicolon!!
' or
put #CAN, msg$
```

Extended Frame



Structure of the 'extended Frame'

Extended Frame, Info-Bits:

FF — Frame-Format-Bit, here FF=1.
0: Standard Frame
1: extended Frame

RTR — Remote Transmit Request, send request. Messages with a set RTR-bit will be responded directly by the driver, if a reply is specified.

DLC — 4 bits specify the number of data bytes in the message (0...8).

The 29-Bit-Identifier of the CAN message can be found in the 4 ID-bytes, offset by 3 bits to the left. The format here is 'high-byte first', unlike the LONG-variables which are 'low-byte first'.

The ID-bytes are followed by as many data bytes as specified by DLC.

Example for the generation of extended frames in Tiger-BASIC®:

```
t_id = 1FFFFFFFh shl 3              ' Transmit-ID, left-aligned in LONG
' extended frame with frame info byte, 4 empty ID bytes, data
msg$ = "<80h><0><0><0><0>" + data$
msg$ = ntos$ ( msg$, 1, -4, t_id ) ' fit in ID with high-byte first
                                    ' length is set by driver
print #CAN, msg$;                   ' PRINT with semicolon!!
' or
put #CAN, msg$
```

CAN User-Function-Codes

User-Function-Codes for inquiries (Instruction GET):

No	Symbol Prefix UFCI_	Description
1	UFCI_IBU_FILL	No. of bytes in input buffer (Byte)
2	UFCI_IBU_FREE	Free space in input buffer (Byte)
3	UFCI_IBU_VOL	Size of input buffer (Byte)
33	UFCI_OBU_FILL	Number of bytes in output buffer (Byte)
34	UFCI_OBU_FREE	Free space in output buffer (Byte)
35	UFCI_OBU_VOL	Size of output buffer (Byte)
65	UFCI_LAST_ERRC	Last error code
99	UFCI_DEV_VERS	Driver version
144	UFCI_CAN_EERR	Byte 1+2: Buffer overflow count counter is reset after reading
152	UFCI_CAN_MODE	reads CAN register MODE
153	UFCI_CAN_STAT	reads CAN register STAT
154	UFCI_CAN_CODE	get CAN register CODE0
155	UFCI_CAN_MASK	get CAN register MASK0
158	UFCI_CAN_RXERR	reads copy from ‘rx error counter register’
159	UFCI_CAN_TXERR	reads copy from ‘tx error counter register’
161	UFCI_CAN_BUSY	get CAN busy state

User-Function-Codes for output (Instruction PUT):

No	Symbol Prefix: UFCO_	Description
1	UFCO_IBU_ERASE	Delete input buffer
33	UFCO_OBU_ERASE	Delete output buffer
136	UFCO_CAN_MODE	sets CAN register MODE
138	UFCO_CAN_CODE	sets CAN register CODE
139	UFCO_CAN_MASK	sets CAN register MASK
140	UFCO_CAN_BUSTIM0	sets CAN register BUSTIM0
141	UFCO_CAN_BUSTIM1	sets CAN register BUSTIM1
162	UFCO_CAN_LAM	sets local acceptance mask (only channel-16)
176	UFCO_CAN_RESET	Resets and reinstalls the CAN bus
193	UFCO_CAN_RESRM	Resets and reinitializes the CAN bus

Reinstall CAN driver

PUT #D, #0, #UFCO_CAN_RESET, *Code,* [*Mask, Bt0, Bt1, Mod, Outctrl*]

D	is a constant, variable or expression of the data type BYTE, WORD, LONG in the range from 0→63 and stands for the device number of the drivers.
Code	is a parameter to determine the Access-Code. 'Code' is always 4 bytes long. The range of values for the Access code with standard frames is 0...7FFh and with extended frames 0...1FFF FFFF. Default value: 0
Mask	is a parameter to determine the acceptance filter. 'Mask' is always 4 bytes long. Default value: 0FFFFFFFFh
Bt0	(Bustiming-Register-0) is a parameter to determine the baud rate-prescalers and the synchronisation step (1 byte). This determines the transfer rate together with Bt1. Default value: 0
Bt1	(Bustiming-Register-1) is a parameter to determine the Bus-Timing and the number of samples during receipt (1 byte). This also determines the transfer rate together with Bt0. Default value: 2Fh (Tseg1=15, Tseg2=2)
Mod	is a parameter to determine the mode (1 byte) . Default value: 0

Bit	Symbol	if bit set (‘1’)
1	CAN_LISTEN	Listen-Only-Mode
2	CAN_SELFTEST	Selftest-Mode
3		reserved
4	CAN_SLEEP	Sleep-Mode
0,5,6		reserved

If the Listen-Only mode is installed the driver tries to automatically recognize the bit rate on the bus on the basis of a table with predefined bit rates.

Outctrl is a dummy parameter. Default value is 1Ah.

This command forces a master reset and reinstalls the driver. Everything is reinitialized, including the buffers. All previously made settings are lost. The parameters are the same as those for the install device.

Example:

```
put #CAN, #0, #UFCO_CAN_RESET, &
0,0,0,0, &                          ' access code
0ffh,0ffh,0ffh,0ffh, &              ' access mask
0,2Fh, &                            ' bustim1, bustim2
0,1Ah                               ' mode, outctrl
```

Master reset

PUT #D, #0, #UFCO_CAN_RESRM, dummy

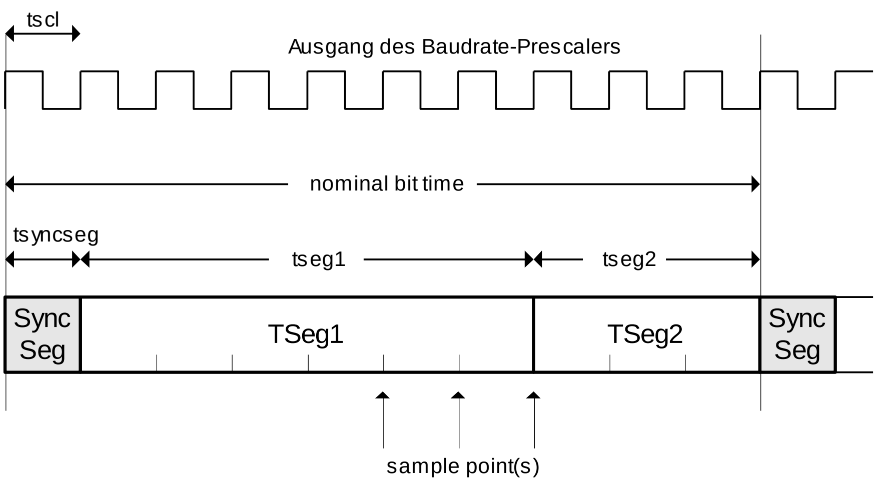
D	is a constant, variable or expression of the data type BYTE, WORD, LONG in the range from 0→63 and stands for the device number of the drivers.
dummy	is a constant, variable or expression of the data type BYTE, WORD, LONG in the range from 0→63 and stands for the device number of the drivers.

This command forces a master reset and a re-initialization of the CAN bus. The previously used settings are kept. The buffers are not affected by this.

Bus-Timing and transfer rate

The transfer rate is determined by the length of a bit. A bit is made up of three sections which in turn consist of individual time segments:

- Sync-Segment, always one time segment long.
- TSEG1 is between 5 and 15 time segments long. The bit is sampled during receipt within Tseg1.
- TSEG2 is between 2 and 7 time segments long.



Structure of a bit:

The unit of a time segment is determined in the Bustiming-Register 0, the number of time segments which make up TSEG1 and TSEG2 in the Bustiming-Register 1.

Bustiming-Register 0

The length of a time segment 'tscl' is determined in the **Bustiming-Register 0**, by the baud rate-prescaler **BRP**. The 6-bit prescaler can assume values between 0 and 31.

1 Time segment: $t_{scl} = 0{,}1 * (BRP+1)$ µsec

1 Bit time = Tsync + Tseg1 + Tseg2

The upper bits in this register determine the synchronization step. The value **SJW** determines the maximum number of clock cycles by which a bit may be shortened or extended to compensate phase differences between different bus controllers through resynchronization.

Bustiming-Register 0							
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
SJW1	SJW0	BRP5	BRP4	BRP3	BRP2	BRP1	BRP0

Bustiming-Register 1

Bustiming-Register-1 determines the number of time segments in **Tseg1** and **Tseg2** and how often the received bit is sampled (once or three times).

Bustiming-Register 1							
Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
SAM	TSEG2.2	TSEG2.1	TSEG2.0	TSEG1.3	TSEG1.2	TSEG1.1	TSEG1.0

SAM=1: The bus is sampled three times. Recommend for slow and medium-speed buses if filtration of spikes on the bus brings advantages.

SAM=0: The bus is sampled once. Recommend for fast buses.

Which values of Tseg1 and Tseg2 guarantee a safe receipt depends on the physical characteristics of the transmission medium, including driver components, optical coupling device. These characteristics finally determine the achievable baud rate and line length.

Some common settings can be found in the following table (achievable bus lengths are only references):

Bit rate	Bustim0	Bustim1	Bt1 Tseg1	Bt1 Tseg2	Bus length
1 Mbit	0	45h	5	4	25m
500 kBit	0	5Ch	12	5	100m
250 kBit	1	5Ch	12	5	250m
125 kBit	3	5Ch	12	5	500m
100 kBit	4	5Ch	12	5	650m

The bit rate can be specified during installation of the driver by parameters.

During the running time the Bustiming settings can be changed using User-Function-Codes.
Note: the output buffer should be empty whilst setting Bustim0 or Bustim1 since the internal CAN chip is temporarily in the rest mode. It is also temporarily not ready to receive.

Example: set 100kBit acc. to above table during the running time:

```
PUT #CAN, #0, #UFCO_CAN_BUSTIM0, 4
PUT #CAN, #0, #UFCO_CAN_BUSTIM1, 5CH
```

Error Register

Both the correct receipt of a CAN message and faulty statuses on the CAN bus trigger a Receiver-Interrupt. During the Interrupt-processing the device driver determines whether a fault-free package has been received or whether errors have occurred. In any case the values associated with error statuses will be refreshed and be given a User-Function code for the next error inquiry. If further errors occur before the error inquiry the later error code will be saved in each case.

The following error inquiries are possible:

User-Function-Code	Bit(s)	Meaning
UFCI_CAN_STAT	0	Receive Buffer Status: 0: empty 1: full
	1	Receive Overrun: 0: no 1: yes Data-Overrun. Occurs if a new CAN-Message is received although there is not enough space in the receive area of the CAN-Chip. This does not relate to the buffer of the device driver.
	2	Transmit Buffer: 0: blocked 1: free
	3	Send: 0: active 1: done
	4	Receive: 0: free 1: active
	5	Send: 0: free 1: active
	6	Error: 0: ok 1: one or both error counters (RXERR, TXERR) have exceeded the value set for Error-Warning-Limit.
	7	Bus-Status: 0: ON 1: OFF If OFF the CAN-Hardware no longer takes part in activities on the bus.
UFCI_CAN_RXERR	0...7	Rx-error counter. counts up with receive errors and back down again to 0 with a correct receipt.
UFCI_CAN_TXERR	0...7	Tx-error counter. counts up with send errors and back down again to 0 if sent correctly.

Arbitration-Lost error

The inquiry of the ALC-Register can provide more information about that bit position at which the bus access was lost. At first the highest-order Identifier bit appears on the CAN bus after the start bit. 10 further Identifier bits follow in the case of a standard frame. Since the 'Extended Frames' must be compatible with the standard frames these 10 Identifier bits are always followed by an RTR-bit. The next bit now decides whether this is a Standard-Frame or an 'Extended Frame'. It is called the IDE bit, **Id**entifier **E**xtension. The remaining 18 Identifier bits follow a reserved bit in the case of the 'Extended Frame'. The Arbitration-Lost-Register can follow arbitration up to the 31st bit, i.e. up to the RTR-bit of an 'Extended Frame'.

Since all participants access the bus simultaneously, the first recessive bit which is overwritten by a dominant bit shows the lost bus access. The bit position is hereby a measure of the priority of the participant which prevents bus access.

Remember: The buffered value is refreshed in the DEVICE at every Interrupt. Since the ALC register of the CAN hardware is reset when it is read, an Arbitration-Lost error which has occurred and been registered once will be overwritten at the next correct receipt. Single Arbitration-Lost statuses can therefore only be recorded if there is sufficient time to read out the value from the driver. Repetitive Arbitration-Lost statuses are recorded statistically.

RXERR receive error counter

The receive error counter is read out at every CAN-Interrupt in the DEVICE driver. The last value can be inquired with a User-Function code. The inquiry doesn't change the meter reading.

```
...
get #CAN, #0, #UFCI_CAN_RXERR, 1, rx_err
...
```

If the meter reading exceeds the set Error-Warning limit (standard: 96) bit 6 will be set in the status register.

If the meter reading exceeds 127, the internal CAN chip switches to the 'Bus-Error-Passive' mode. In this mode the CAN-hardware sends no further error telegrams but continues to send and receive its telegrams. Error-free data telegrams on the bus reduce the error counter again.

TXERR send error counter

The send error counter in the device driver will be read out in the event of Error-Interrupts. The last value can be inquired with a User-Function code. The inquiry doesn't change the meter reading.

```
...
get #CAN, #0, #UFCI_CAN_TXERR, 1, tx_err
...
```

If the meter reading exceeds the set Error-Warning limit (standard: 96) bit 6 will be set in the status register.

If the meter reading exceeds 127, the internal CAN chip switches to the 'Bus-Error-Passive' mode. In this mode the CAN-hardware sends no further error telegrams but continues to send and receive its telegrams. Error-free data telegrams on the bus reduce the error counter again.

If the meter reading exceeds 255, the CAN chip switches to the 'Bus-Off status'. The CAN chip will recover from Bus-Off (become error active again) automatically. It will start the recovering sequence (128 occurrences of 11 consecutive recessive bits monitored on CANRX) automatically after it has entered Bus-Off state.

Receive filter with Code and Mask

The set Access-Code together with the Access-Filter determines which CAN-messages are received. The Access-Mask sets bits to 'don't care' if necessary. The bits of the received Identifiers which are not 'don't care' must correspond with the code so that the message can be received.

There now follow instructions for:

- Set Access-Code and Access-Mask
- Standard-Frame with Single filter configuration
- Extended Frame with Single filter configuration
- Standard-Frame with Dual filter configuration
- Extended Frame with Dual filter configuration

The received CAN-message can be present as a Standard-Frame or as an Extended-Frame.

Set Access-Code and Access-Mask

Access-Code and Access-Mask are registers and part of the CAN hardware and are set during installation of the device driver. If no parameters are specified Access-Code is set to 0 and Access-Mask to 0FFFFFFFFh so that all messages pass through the filter.

The code and the mask can be seen as simple bit patterns or as numbers. For example, a LONG number is suitable to store the bits of the Access-Code or the Access-Mask . One problem here is that the CAN number starts with the highest-order byte, the Tiger-BASIC LONG number however with the lowest-order:

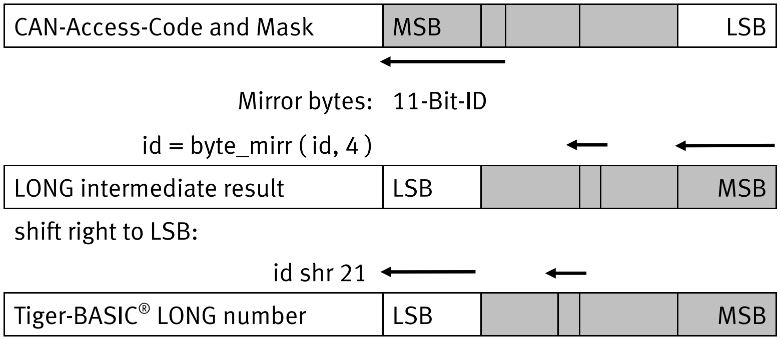
CAN-Access-Code and Mask	MSB			LSB

Tiger-BASIC® LONG number	LSB			MSB

In addition the 11 bits and/or 29 bits are flush left in the 32 bit for the Identifier depending on the frame type. Numbers start, however, on the right with the lowest bit and have no 'don't care' bit to the right of this. There can be a zero to the left of a number, but this is not important.

If you therefore wish to see the Identifier from the Access-Code as a number the bytes first have to be mirrored and

- the value of the Access-Code shifted 21 bits (5+16) to the right with an 11-Bit Identifier
- the value of the Access-Code shifted 3 bits to the right with a 29-Bit Identifier.



Conversely: if you hav110e a number and want to store it in a CAN register Access-Code or Access-Mask then

- the bits in the number first have to be moved to the left
- then the bytes in the number mirrored

Remember that the Function NTOS$ can mirror the bytes by specifying a negative value as an argument for the number of bytes:

- **msg$ = ntos$ (msg$, 1, -2, t_id)** inserts an 11-bit Identifier present as a WORD number with the ID-bits in the correct position into a string and hereby mirrors the bytes.
- **msg$ = ntos$ (msg$, 1, -4, t_id)** does the same for a 29-bit Identifier, which is present as a LONG number with the ID-bits at the correct position.

The sequence does not change in a string:

id$ = “‹1Fh›‹AAh›‹BBh›‹33h›“

or

id$ = “1F AA BB 33“%

Step the following example program to understand these conditions in the 'Monitored expressions'.

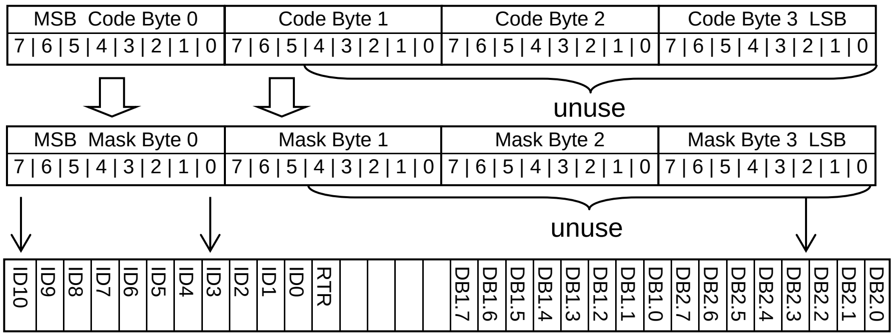
Program example:

```
'-----------------------------------------------------------------
'Name: CAN_SET_FILTER.TIG
'sets filter configuration
'demostrates how to set accress code and access mask
'in different variations
'only one CAN-Tiger is necessary as nothing is sent or received
'Please use the command 'Watches' from the menu 'View'

'-----------------------------------------------------------------
user var strict                     'check var declarations
#INCLUDE UFUNC3.INC                 'User Function Codes
#INCLUDE DEFINE_A.INC               'general symbol definitions
#INCLUDE CAN.INC                    'CAN definitions

LONG ac_code, ac_mask
STRING id$

'-----------------------------------------------------------------
TASK MAIN
  install_device #LCD, "LCD1.TDD" 'install LCD-driver
  install_device #CAN, "CAN1_K1.TDD", & 'install CAN-driver
    "12 34 56 78 &                  'access code
     EF FF FE FF &                  'access mask
     10 45 &                        'bustim1, bustim2
     08 1A"%                        'single filter mode, outctrl

  using "UH<8><8>   0 0 0 4 4"      'to display ID in whole program

'show access code und access mask after installation
  get #CAN, #0, #UFCI_CAN_CODE, 0, ac_code
  ac_code = byte_mirr ( ac_code, 4 ) 'byte order mirrored for LONG
  print_using #LCD, "<1>ac_code:";ac_code
  get #CAN, #0, #UFCI_CAN_MASK, 0, ac_mask 'and read
  ac_mask = byte_mirr ( ac_mask, 4 ) 'byte order mirrored for LONG
  print_using #LCD, "ac_mask:";ac_mask
'the same lines are in show_codemask
  wait_duration 1000

'see byte order ('watches' id$ and ac_code)
  get #CAN, #0, #UFCI_CAN_CODE, 4, id$ 'test: read access code
  get #CAN, #0, #UFCI_CAN_CODE, 0, ac_code 'and read into a LONG
  wait_duration 1000


  ac_code = byte_mirr ( (1FFFFFFFFh shl 3), 4 )'biggest access code
  put #CAN, #0, #UFCO_CAN_CODE, ac_code 'and set
  call show_codemask                'and display
  wait_duration 1000

'this is the same:
  id$ = "FF FF FF F8"%              '1FFFFFFF left bound
  put #CAN, #0, #UFCO_CAN_CODE, id$ 'and set
  call show_codemask                'and display
  wait_duration 1000

'set new code for the following read test
```

```
  ac_code = byte_mirr ( (12345678h shl 3), 4 ) 'becomes 0C0B3A291h
  put #CAN, #0, #UFCO_CAN_CODE, ac_code 'and set
  call show_codemask                    'and display
  wait_duration 1000
'step from here
  get #CAN, #0, #UFCI_CAN_CODE, 0, ac_code 'see byte order
  ac_code = byte_mirr ( ac_code, 4 )       'after each step
  ac_code = ac_code shr 3
  print_using #LCD, "<1>ac_code:";ac_code

END


'-----------------------------------------------------------------
'displays access code and access mask an
'-----------------------------------------------------------------
SUB show_codemask
  get #CAN, #0, #UFCI_CAN_CODE, 0, ac_code
  ac_code = byte_mirr ( ac_code, 4 ) 'byte order mirrored for LONG
  print_using #LCD, "<1>ac_code:";ac_code
  get #CAN, #0, #UFCI_CAN_MASK, 0, ac_mask 'and read
  ac_mask = byte_mirr ( ac_mask, 4 ) 'byte order mirrored for LONG
  print_using #LCD, "ac_mask:";ac_mask
END
```

Standard-Frame with Single-Filter configuration

In the 'single filter' mode with a **Standard-Frame**, all ID-bits are passed through the Access filter and compared with the set code. Only the ID Bits are compared, but NOT the RTR Bit or the data Bytes.



In the example program CAN_FILTER_SS.TIG the Access-Code is set to 4EE0 0000 after installation. The mask determine which bits of the set code are relevant. The value F11F FFFF has a total of 6 '0'-bits within the area of the Identifier (the 11 bit left-adjusted) which indicate that these bits in the message on the bus must correspond with the Access-Code so that the message will be received. The test shows that those values with an 'E' or 'F' in the second position and an 'E' in the third position come through. Thus, exactly those messages whose bits match the relevant bits of the Access-Code will be received

The illustration shows the Access-Code, Access-Mask and an Identifier as an example. Only the ID-bits are shown. The other bits in the example are 'don't care' any way:

	ID10	ID9	ID8	ID7	ID6	ID5	ID4	ID3	ID2	ID1	ID0
Code: 4EEh	0	1	0	0	1	1	1	0	1	1	1
Mask: F11h	1	1	1	1	0	0	0	1	0	0	0
x=not relevant	x	x	x	x	1	1	1	x	1	1	1
ID: 0Eeh	0	0	0	0	1	1	1	0	1	1	1
ID: 7Feh	0	1	1	1	1	1	1	1	1	1	1

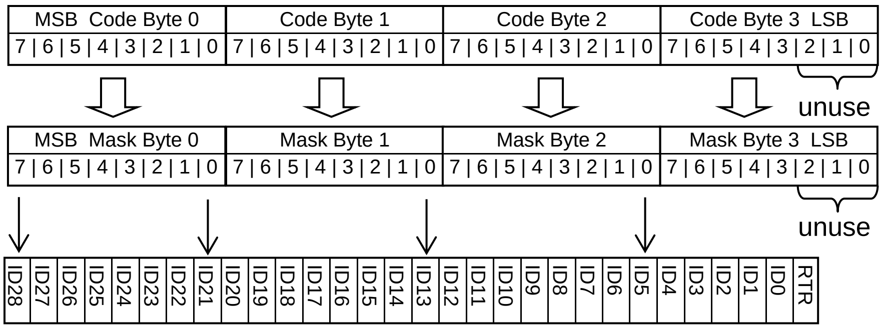
Program example:

```
'----------------------------------------------------------------
'Name: CAN_Filter_SS.TIG
'single filter configuration
'sends standard frames with different IDs for filter test
'receives filtered CAN messages and displays on LCD
'knows standard and extended frame
'connect a second CAN-Tiger with the same program
'----------------------------------------------------------------
user var strict                    'check var declarations
#INCLUDE UFUNC3.INC                'User Function Codes
#INCLUDE DEFINE_A.INC              'general symbol definitions
#INCLUDE CAN.INC                   'CAN definitions

BYTE frameformat, msg_len, can_stat
LONG ac_code, ac_mask
LONG r_id                          'Rx ID
STRING id$(4), msg$(13), data$(8)

'----------------------------------------------------------------
TASK MAIN
  BYTE ever                        'for endless loop
  WORD ibu_fill                    'input buffer fill level

  install_device #LCD, "LCD1.TDD" 'install LCD-driver
  install_device #CAN, "CAN1_K1.TDD", & 'install CAN-driver
    "4E E0 00 00 &                 'access code
     F1 1F FF FF &                 'access mask
     10 45 &                       'bustim1, bustim2
     08 1A"%                       'single filter mode, outctrl

'code and mask are set like this now:
'01001110111 RTR --data-- --data-- code (relevant 11 bits)
'11110001000  1  11111111 11111111 mask (bits 0 count, 1=don't care)
'thus messages with the following bit pattern will pass:
'01001110111 RTR --data-- --data-- code (relevant 11 bits)
'xxxx111x111  x  xxxxxxxx xxxxxxxx
'received frames are 0EEh, 0FEh, 1EEh, 1FEh, etc

  using "UH<8><8>   0 0 0 4 4"
  get #CAN, #0, #UFCI_CAN_CODE, 0, ac_code
  ac_code = byte_mirr ( ac_code, 4 ) 'byte order mirrored for LONG
  print_using #LCD, "<1>ac_code:";ac_code

  get #CAN, #0, #UFCI_CAN_MASK, 0, ac_mask 'and read
  ac_mask = byte_mirr ( ac_mask, 4 ) 'byte order mirrored for LONG
  print_using #LCD, "ac_mask:";ac_mask

  run_task generate_frames              'generates incrementing IDs

'display now IDs of received frames
  for ever = 0 to 0 step 0               'endless loop
    get #CAN, #0, #UFCI_IBU_FILL, 0, ibu_fill

    if ibu_fill > 2 then           'if at least one message
      get #CAN, #0, 1, frameformat 'get frame info byte
      msg_len = frameformat bitand 1111b 'length
```

```
      if frameformat bitand 80h = 0 then 'if standard frame
        get #CAN, #0, CAN_ID11_LEN, r_id 'get ID bytes
        r_id = byte_mirr ( r_id, 2 )
        disable_tsw
        using "UH<4><4>   0 0 0 0 4"
      else                              'else it is extended frame
        get #CAN, #0, CAN_ID29_LEN, r_id'and no SLIO message
        r_id = byte_mirr ( r_id, 4 )
        disable_tsw
        using "UH<8><8>   0 0 0 4 4"
      endif
      print_using #LCD, "<1Bh>A<0><2><0F0h>ID rcvd:";r_id;
      enable_tsw

      if msg_len > 0 then               'if contains data
        get #CAN, #0, msg_len, data$    'get them out of the buffer
      endif
    endif

' HEX format for one byte

  next
END


'-----------------------------------------------------------------
'generates standard frames with incrementing ID
'-----------------------------------------------------------------
TASK generate_frames
  BYTE ever                       'for endless loop
  WORD obu_free                   'output buffer free space
  LONG t_id                       'Tx ID
  STRING msg$(13)

  t_id  = 0                       'standard identifier
  for ever = 0 to 0 step 0        'endless loop
    get #CAN, #0, #UFCI_OBU_FREE, 0, obu_free
    if obu_free > 13 then
'frame info 0 = standard, 2 ID bytes, no data
      msg$ = "<0><0><0>"
      msg$ = ntos$ ( msg$, 1, -2, t_id ) 'insert ID high byte 1st
      put #CAN, #0, msg$          'send a standard frame message
      disable_tsw
      using "UH<4><4>   0 0 0 0 4" 'to display ID
      print_using #LCD, "<1Bh>A<0><3><0F0h>ID sent:";t_id;
      enable_tsw
                                  'this counts up t_id by 1
                                  'when considering the shift by 5
                                  'of the extended ID
      t_id = t_id + 100000b       'next ID
      t_id = t_id bitand 0FFFFh   'remain with standard fraem ID
    endif
    wait_duration 30
  next
END
```

Extended Frame with Single-Filter configuration

With an **Extended-Frame** all ID-bits are passed through the filter. The 3 lowest bits should be masked 'don't care' for reasons of compatibility.



Program example:

```
'-----------------------------------------------------------------
'Name: CAN_Filter_ES.TIG
'single filter configuration
'sends extended frames with different IDs for filter test
'receives filtered CAN messages and displays on LCD
'knows standard and extended frame
'connect a second CAN-Tiger with the same program
'-----------------------------------------------------------------
user var strict                   'check var declarations
#INCLUDE UFUNC3.INC               'User Function Codes
#INCLUDE DEFINE_A.INC             'general symbol definitions
#INCLUDE CAN.INC                  'CAN definitions

BYTE frameformat, msg_len, can_stat
LONG ac_code, ac_mask
LONG r_id
STRING id$(4), msg$(13), data$(8)

'-----------------------------------------------------------------
TASK MAIN
  BYTE ever                       'for endless loop
  WORD ibu_fill                   'input buffer fill level

  install_device #LCD, "LCD1.TDD" 'install LCD-driver
  install_device #CAN, "CAN1_K1.TDD", & 'install CAN-driver
    "6D 55 D9 98 &                'access code
     EF FF FE FF &                'access mask
     10 45 &                      'bustim1, bustim2
     08 1A"%                      'single filter mode, outctrl

  using "UH<8><8>   0 0 0 4 4"    'to display ID in whole program

  get #CAN, #0, #UFCI_CAN_CODE, 4, id$ 'test: read access code
  'check byte order with View - Watches
  get #CAN, #0, #UFCI_CAN_CODE, 0, ac_code
  ac_code = byte_mirr ( ac_code, 4 ) 'byte order mirrored for LONG
  print_using #LCD, "<1>ac_code:";ac_code
  wait_duration 2000

'code and mask will be set for extended frames like this now:

'87654321 09876543 21098765 43210Rxx RTR, 2x don't care
'01101101 01010101 11011001 10011000 code (29 relevant bits+RTR)
'11101111 11111111 11111110 11111111 mask (0-bits are relevant)
'RTR and not used bits don't care
'thus messages with the following bit pattern will pass:
'xxx0xxxx xxxxxxxx xxxxxxx1 xxxxxxxx
'bit 5 must be set and bit 25 must be 0


  ac_code = byte_mirr ( (0DAABB33h shl 3), 4 ) '         new access code
  put #CAN, #0, #UFCO_CAN_CODE, ac_code 'and set
'this is the same:
' id$ = "FD 55 D9 98"%                  ' new access code
' put #CAN, #0, #UFCO_CAN_CODE, id$     ' and set
```

```
'check again byte order with View - Watches
  get #CAN, #0, #UFCI_CAN_CODE, 4, id$ 'read access code into string
'or read like this, but must mirror for LONG
  get #CAN, #0, #UFCI_CAN_CODE, 0, ac_code 'and read into a LONG
  ac_code = byte_mirr ( ac_code, 4 )
  print_using #LCD, "<1>ac_code:";ac_code
  wait_duration 1000

  ac_mask = byte_mirr ( 0EFFFFEFFh, 4 ) 'access mask
  put #CAN, #0, #UFCO_CAN_MASK, ac_mask 'set
  get #CAN, #0, #UFCI_CAN_MASK, 0, ac_mask 'and read
  ac_mask = byte_mirr ( ac_mask, 4 ) 'byte order mirrored for LONG
  print_using #LCD, "ac_mask:";ac_mask

  run_task generate_frames              'generates incrementing IDs

'display now IDs of received frames
  for ever = 0 to 0 step 0               'endless loop
    get #CAN, #0, #UFCI_IBU_FILL, 0, ibu_fill

    if ibu_fill > 2 then          'if at least one message
      get #CAN, #0, 1, frameformat 'get frame info byte
      msg_len = frameformat bitand 1111b 'length
      if frameformat bitand 80h = 0 then 'if standard frame
        get #CAN, #0, CAN_ID11_LEN, r_id 'get ID bytes
        r_id = byte_mirr ( r_id, 2 )
        r_id = r_id shr 5
      else                              'else it is extended frame
        get #CAN, #0, CAN_ID29_LEN, r_id'and no SLIO message
        r_id = byte_mirr ( r_id, 4 )
        r_id = r_id shr 3
        if msg_len > 0 then             'if contains data
          get #CAN, #0, msg_len, data$  'get them and free the buffer
        endif
      endif
      disable_tsw
      using "UH<8><8>   0 0 0 4 4"    ' display ID
      print_using #LCD, "<1Bh>A<0><2><0F0h>ID rcvd:";r_id;
      enable_tsw

      if msg_len > 0 then               'if contains data
        get #CAN, #0, msg_len, data$    'get them out of the buffer
      endif
    endif

' HEX format for one byte

  next
END


'-----------------------------------------------------------------
'generates extended frames with incrementing ID
'-----------------------------------------------------------------
TASK generate_frames
  BYTE ever
  WORD obu_free
  LONG t_id
  STRING msg$(13)

  using "UH<8><8>   0 0 0 4 4"    'to display ID in whole program
```

```
  t_id  = 0AABB00h shl 3           'extended identifier
  for ever = 0 to 0 step 0         'endless loop
    get #CAN, #0, #UFCI_OBU_FREE, 0, obu_free
    if obu_free > 13 then
'frame info 80h = extended, 4 ID bytes, no data
      msg$ = "<80h><0><0><0><0>"
      msg$ = ntos$ ( msg$, 1, -4, t_id ) 'insert ID high byte 1st
      put #CAN, #0, msg$           'send a standard frame message
      print_using #LCD, "<1Bh>A<0><3><0F0h>ID sent:";t_id shr 3;
                                  'this counts by 1 in bytes 0 and 3
                                  'when considering the shift by 3
                                  'of the extended ID
      t_id = t_id + 08000008h     'next ID
    endif
    wait_duration 50
  next
END
```

Setting of more access codes in standard format

Secondary addresses 3…15 can be used for additional access codes. If the AME Bit is set, the global acceptance filter is used for filtering, otherwise no filter is used.

Secondary address 16 can be used for one more additional access code. If The AME Bit is set, the local acceptance filter is used for filtering, otherwise no filter is used.

Sec.-Adr.	Function
3	Sets one more access code (global mask)
4	Sets one more access code (global mask)
5	Sets one more access code (global mask)
6	Sets one more access code (global mask)
7	Sets one more access code (global mask)
8	Sets one more access code (global mask)
9	Sets one more access code (global mask)
10	Sets one more access code (global mask)
11	Sets one more access code (global mask)
12	Sets one more access code (global mask)
13	Sets one more access code (global mask)
14	Sets one more access code (global mask)
15	Sets one more access code (global mask)
16	Sets one more access code (local mask)

PUT #CAN, #CH, *"‹ID0›‹ID1›‹ID2›‹ID3›"*

‹CH› contains the channel number 3…16.

‹ID0› contains the identifiers 3…10.

‹ID1› contains the identifiers 0…2.

‹ID2› is zero.

‹ID3› contains acceptance mask enable bit and identifier extension bit.

```
slCode$ = "10 00 00 00"%             ' only ID = 80H
PUT #CAN, #3, slCode$                ' set code (without any mask)
```

	Code Byte 0							
Bit No.	7	6	5	4	3	2	1	0
Function	ID10	ID9	ID8	ID7	ID6	ID5	ID4	ID3

Identifiers <ID3> to <ID10> are stored

	Code Byte 1							
Bit No.	7	6	5	4	3	2	1	0
Function	ID2	ID1	ID0					

Identifiers <ID0> to <ID2> are stored

	Code Byte 2							
Bit No.	7	6	5	4	3	2	1	0
Function								

	Code Byte 3							
Bit No.	7	6	5	4	3	2	1	0
Function						IDE	AME	

	Acceptance mask enable
0	Acceptance mask is not used for acceptance filtering
1	Acceptance mask is used for acceptance filtering

	Identifier extension Bit (for use without mask)
0	Standard format (11-bit identifier)
1	Extended format (29-bit identifier)

Setting of the local acceptance mask in standard format

The local acceptance mask is used **only** for access code 16. Channel-16 is a special access code with its own local acceptance mask. If no other code matches, the incoming CAN message is compared with channel 16 Code and the local acceptance mask (NOT the global acceptance mask)!

PUT #CAN, #0, #UFCO_CAN_LAM, *"‹M0›‹M1›‹M2›‹M3›"*

‹M0›	contains the mask bits for identifiers 3...10.
‹M1›	contains the mask bits for identifiers 0...2.
‹M2›	dummy data (zero).
‹M3›	dummy data (zero).

```
slCode$ = "FF FF C0 00"%                     ' set mask
PUT #CAN, #0, #UFCO_CAN_LAM, slCode$         ' set local acceptance mask

slCode$ = "00 00 3F FE"%                     ' all IDs = xxxx7FFH
PUT #CAN, #16, slCode$                       ' set code (with local mask)
```

	Mask Byte 0							
Bit No.	7	6	5	4	3	2	1	0
Function	ID10	ID9	ID8	ID7	ID6	ID5	ID4	ID3

→ Mask Bytes for identifiers <ID3> to <ID10> are stored

	Mask Byte 1							
Bit No.	7	6	5	4	3	2	1	0
Function	ID2	ID1	ID0					

→ Mask Bytes for identifiers <ID0> to <ID2> are stored

	Mask Byte 2							
Bit No.	7	6	5	4	3	2	1	0
Function								

	Mask Byte 3							
Bit No.	7	6	5	4	3	2	1	0
Function								

Setting of more access codes in extended format

Secondary addresses 3...15 can be used for additional access codes. If the AME Bit is set, the global acceptance filter is used for filtering, otherwise no filter is used.

Secondary address 16 can be used for one more additional access code. If The AME Bit is set, the local acceptance filter is used for filtering, otherwise no filter is used.

Sec.-Adr.	Function
3	Sets one more access code (global mask)
4	Sets one more access code (global mask)
5	Sets one more access code (global mask)
6	Sets one more access code (global mask)
7	Sets one more access code (global mask)
8	Sets one more access code (global mask)
9	Sets one more access code (global mask)
10	Sets one more access code (global mask)
11	Sets one more access code (global mask)
12	Sets one more access code (global mask)
13	Sets one more access code (global mask)
14	Sets one more access code (global mask)
15	Sets one more access code (global mask)
16	Sets one more access code (local mask)

PUT #CAN, #CH, *“‹ID0›‹ID1›‹ID2›‹ID3›”*

‹CH›	contains the channel number 3...16.
‹ID0›	contains the identifiers 21...28.
‹ID1›	contains the identifiers 13...20.
‹ID2›	contains the identifiers 5...12.
‹ID3›	contains the identifiers 0...4, the acceptance mask enable bit and identifier extension bit.

```
slCode$ = "00 00 00 0C"%       ' only ID = 1H (extended format)
PUT #CAN, #3, slCode$          ' set code (without any mask)

slCode$ = "00 00 3F FE"%       ' all IDs = xxxx7FFH (extended format)
PUT #CAN, #4, slCode$          ' set code (with global mask)
```

	Code Byte 0							
Bit No.	7	6	5	4	3	2	1	0
Function	ID28	ID27	ID26	ID25	ID24	ID23	ID22	ID21

Identifiers <ID21> to <ID28> are stored

	Code Byte 1							
Bit No.	7	6	5	4	3	2	1	0
Function	ID20	ID19	ID18	ID17	ID16	ID15	ID14	ID13

Identifiers <ID13> to <ID20> are stored

	Code Byte 2							
Bit No.	7	6	5	4	3	2	1	0
Function	ID12	ID11	ID10	ID9	ID8	ID7	ID6	ID5

Identifiers <ID5> to <ID12> are stored

	Code Byte 3							
Bit No.	7	6	5	4	3	2	1	0
Function	ID4	ID3	ID2	ID1	ID0	IDE	AME	

	Acceptance mask enable
0	Acceptance mask is not used for acceptance filtering
1	Acceptance mask is used for acceptance filtering

	Identifier extension Bit (for use without mask)
0	Standard format (11-bit identifier)
1	Extended format (29-bit identifier)

Identifiers <ID0> to <ID4> are stored

Setting of the local acceptance mask in extended format

The local acceptance mask is used **only** for access code 16. Channel-16 is a special access code with its own local acceptance mask. If no other code matches, the incoming CAN message is compared with channel 16 Code and the local acceptance mask (NOT the global acceptance mask)!

PUT #CAN, #0, #UFCO_CAN_LAM, ***"‹M0›‹M1›‹M2›‹M3›"***

‹M0›	contains the mask bits for identifiers 21...28.
‹M1›	contains the mask bits for identifiers 13...20.
‹M2›	contains the mask bits for identifiers 5...12.
‹M3›	contains the mask bits for identifiers 0...4.

	Mask Byte 0							
Bit No.	7	6	5	4	3	2	1	0
Function	ID28	ID27	ID26	ID25	ID24	ID23	ID22	ID21

Mask Bytes for identifiers <ID21> to <ID28> are stored

	Mask Byte 1							
Bit No.	7	6	5	4	3	2	1	0
Function	ID20	ID19	ID18	ID17	ID16	ID15	ID14	ID13

Mask Bytes for identifiers <ID13> to <ID20> are stored

	Mask Byte 2							
Bit No.	7	6	5	4	3	2	1	0
Function	ID12	ID11	ID10	ID9	ID8	ID7	ID6	ID5

Mask Bytes for identifiers <ID5> to <ID12> are stored

	Mask Byte 3							
Bit No.	7	6	5	4	3	2	1	0
Function	ID4	ID3	ID2	ID1	ID0			

Mask Bytes for identifiers <ID0> to <ID4> are stored

Sending CAN messages

The CAN device driver supports the following methods of dispatch:

Send single messages which contain 0...8 characters and whose Identifiers can be specified individually as required. Every CAN message is output with a PUT or Print instruction. With the Print instruction you must remember that the version will be formatted and any additional bytes (CR, LF) appended.

Send data, which may also contain more the 8 characters. The device driver creates as many CAN data packets from this are needed to dispatch the complete amount and uses the Identifier specified at the start of the string. The data are transferred to the buffer with a single PUT or PRINT instruction.

Reply to a 'Remote Transmission Request' by providing a message especially for this purpose in the device driver. The message provided will be automatically sent by the driver if an RTR-Message is received.

The CAN device driver expect a CAN message in the predefined format as an argument. The first byte will be interpreted as a Frame-Format byte . The next 2 or 4 bytes are the message's Identifier depending on the Frame-format. A typical CAN output as a Standard Frame looks as follows:

PUT #CAN, #0, *“‹Frame-Format›‹ID1›‹ID2›data”*

‹Frame-Format›	contains information that this is a Standard-Frame.
‹ID1›	contains the upper bits 3...10 of the Identifier.
‹ID2›	contains the lower bits 0...2 of the Identifier at the bit positions 5, 6 and 7. The remaining bits in this byte are insignificant.
data	are data bytes which are transferred in the message. 0...8 data bytes are possible.

With 0...8 data bytes this generates a CAN message. If more than 8 data bytes are contained the device driver packs the data into several CAN messages and uses the same Identifier.

PUT #CAN, #0, *“‹Frame-Format›‹ID1›‹ID2›abcdefghijklmnopqrs”*

becomes the following CAN messages:

“‹Frame-Format›‹ID1›‹ID2›abcdefgh”

“‹Frame-Format›‹ID1›‹ID2›ijklmnop”

“‹Frame-Format›‹ID1›‹ID2›qrs”

If the data are sent via the secondary address 1 the RTR-bit will be set in the message and thus a 'Remote Transmission Request' produced.

A single message with a maximum of 8 data bytes at the secondary address 2 leaves a response which will be sent when the device driver itself receives a 'Remote transmission Request'.

Sec.-Adr.	Function
0	Normal data dispatch
1	Data dispatch with 'Remote transmission Request'
2	Deposit a response message which will be sent when the device driver itself receives a 'Remote Transmission Request'.

The following program shows a simple send example for **standard frame** CAN-messages.

Program example:

```
'----------------------------------------------------------------
'Name: CAN_TX_STANDARD.TIG
'sends 'the quick brown fox' via CAN in standard frames
'connect a receiving CAN device, e.g. a Tiger with CAN_RX.TIG
'----------------------------------------------------------------
user var strict                    'check var declarations
#INCLUDE UFUNC3.INC                'User Function Codes
#INCLUDE DEFINE_A.INC              'general symbol definitions
#INCLUDE CAN.INC                   'CAN definitions

'----------------------------------------------------------------
TASK MAIN
  BYTE ever, i_msg, can_stat
  WORD obu_free                    'output buffer space
  WORD t_id                        'transmit ID
  STRING data$, msg$(11)

  install_device #LCD, "LCD1.TDD" 'install LCD-driver
  install_device #CAN, "CAN1_K1.TDD", & 'install CAN-driver
    "50 A0 00 00 &                 'access code
     FF FF FF FF &                 'access mask
     10 45 &                       'bustim1, bustim2
     08 1A"%                       'single filter mode, outctrl

  data$ = "the quick brown fox jumps over the lazy dog"
  i_msg = 0                        'index for running text
  t_id  = 155h shl 5               'standard identifier

  for ever = 0 to 0 step 0         'endless loop
    get #CAN, #0, #UFCI_OBU_FREE, 0, obu_free
    print #LCD, "<1Bh>A<0><1><0F0h>OBU_FREE:";obu_free;"    ";
    if obu_free > 11 then
      msg$ = &  'frame info 0 = standard, 2 ID bytes, data
      "<0><0><0>" + mid$ ( data$, i_msg, 8 )'nfo, ID
      msg$ = ntos$ ( msg$, 1, -2, t_id ) 'insert ID high byte 1st
      print #CAN, #0, msg$;        'send a standard frame message
      i_msg = i_msg + 1            'advance string index
      if i_msg > len(data$)-8 then 'check limit
        i_msg = 0
      endif
    endif                          'check CAN state
    get #CAN, #0, #UFCI_CAN_STAT, 0, can_stat
    using "UH<2><2>   0 0 0 0 2"   'HEX format for a byte
    print_using #LCD, "<1Bh>A<0><0><0F0h>CAN-State:";can_stat;
    wait_duration 200
  next
END
```

The following program shows a simple send example for **extended frame** CAN-messages.

Program example:

```
'------------------------------------------------------------------
'Name: CAN_TXEXTENDED.TIG
'sends 'the quick brown fox' via CAN in extended frames
'connect a receiving CAN device, e.g. a CAN-Tiger
'------------------------------------------------------------------
user var strict                   'check var declarations
#INCLUDE UFUNC3.INC               'User Function Codes
#INCLUDE DEFINE_A.INC             'general symbol definitions
#INCLUDE CAN.INC                  'CAN definitions

'------------------------------------------------------------------
TASK MAIN
  BYTE ever, i_msg, can_stat
  WORD obu_free                   'output buffer space
  LONG t_id                       'extended ID 4 bytes
  STRING data$, msg$(13)

  install_device #LCD, "LCD1.TDD" 'install LCD-driver
  install_device #CAN, "CAN1_K1.TDD", & 'install CAN-driver
    "50 A0 00 00 &                'access code
     FF FF FF FF &                'access mask
     10 45 &                      'bustim1, bustim2
     08 1A"%                      'single filter mode, outctrl

  data$ = "the quick brown fox jumps over the lazy dog"
  i_msg = 0                       'index for running text
  t_id = 01733F055h shl 3         'extended identifier

  for ever = 0 to 0 step 0        'endless loop
    get #CAN, #0, #UFCI_OBU_FREE, 0, obu_free
    print #LCD, "<1Bh>A<0><1><0F0h>OBU_FREE:";obu_free;"    ";
    if obu_free > 13 then
      msg$ = &  'frame info 80h = exetended, 4 ID bytes, data
      "<80h><0><0><0><0>" + mid$ ( data$, i_msg, 8 )
      msg$ = ntos$ ( msg$, 1, -4, t_id ) 'insert ID high byte 1st
      print #CAN, #0, msg$;       'send an extended frame message
      i_msg = i_msg + 1           'advance string index
      if i_msg > len(data$)-8 then ' check limit
        i_msg = 0
      endif
    endif                         'check CAN state
    get #CAN, #0, #UFCI_CAN_STAT, 0, can_stat
    using "UH<2><2>   0 0 0 0 2"  'HEX format for a byte
    print_using #LCD, "<1Bh>A<0><0><0F0h>CAN-State:";can_stat;
    wait_duration 200
  next
END
```

Receive CAN messages

The CAN device driver receives CAN messages and put these in the receive buffer. Reading out the receive buffer with the CAN device driver is a special process and differs from reading out other buffers (e.g. of the serial or parallel driver), since here the messages in the buffer can contain further information in addition to the data. The messages will always be read completely and processed according to the message type:

Two read modes read differently from the secondary addresses 0 and 1:

Sec.Adr.	
0	The bytes in the CAN message will be read as they are in the buffer, including Frame-Format and ID-bytes.
1	Only data bytes will be read. Frame-Format and ID-bytes will be ignored. The length information of partially read CAN messages will be automatically corrected in the buffer .

Caution: the CAN-message must be read completely from the secondary address 0 since otherwise the next read operation will not start with the Frame-Info byte of the next CAN message.

Single messages containing 0...8 characters and whose frame format ID and Identifier precede the data bytes are read out via the secondary address 0. The Frame-Info byte will at first be read to determine whether this is a 'Standard-Frame' or an 'extended Frame' and how many data bytes are contained therein. The ID-bytes which indicate the application-specific type of message will then be read. The data bytes will then be read in.

The example program CAN_RX1.TIG reads the received messages from the buffer, distinguishes thereby between standard frames and extended frames and shows these in a hexadecimal form.

Program example:

```
user_var_strict

#INCLUDE UFUNC3.INC                  ' User Function Codes
#INCLUDE DEFINE_A.INC                ' allg. Symbol-Definitionen
#INCLUDE CAN.INC                     ' CAN-Definitionen

task main
  BYTE frameformat, msg_len
  WORD ibu_fill
  LONG ac_code, ac_mask, r_id
  string slCode$(4), data$(8)

  INSTALL DEVICE #SER, "SER1B_K4.TD2", &
  BD_38_400,DP_8N,NEIN,BD_38_400,DP_8N,NEIN

  install_device #CAN, "CAN1_K8.TD2", & ' install CAN-driver
    "00 00 00 00 &                   ' access code
     FF FF FF FF &                   ' access mask
     01 5C &                         ' bustim1, bustim2
     00 1A"%                         ' dual filter mode, outctrl

  Print #SER,#0, "Can Receive All!"

  while 1 = 1
    get #CAN, #0, #UFCI_IBU_FILL, 0, ibu_fill
    if ibu_fill > 2 then                            ' if there is a message
      get #CAN, #0, 1, frameformat                  ' get Frame-Info-Byte
      msg_len = frameformat bitand 1111b            ' length
      if frameformat bitand 80h = 0 then            ' if Standard-Frame
        get #CAN, #0, CAN_ID11_LEN, r_id            ' get ID-Bytes
        r_id = byte_mirr ( r_id, 2 )                '
        r_id = r_id SHR 5                           '
        using "UH<8><3>   0 0 0 0 3"      ' fuer ID Anzeige
      else                                          ' it is extended frame
        get #CAN, #0, CAN_ID29_LEN, r_id            '
        r_id = byte_mirr ( r_id, 4 )                '
        r_id = r_id SHR 3                           '
        using "UH<8><8>   0 0 0 4 4"      ' fuer ID Anzeige
      endif
      print_using #SER, #0, "ID:"; r_id; ", ";   ' show ID
      using "UH<1><1>   0 0 0 0 1"        ' zeige Laenge an
      print_using #SER, #0, "DLC:";msg_len ; ", ";

      if msg_len > 0 then                           ' if there are data bytes
        get #CAN, #0, msg_len, data$                ' read out data
      endif
      if bit(frameformat, 6) = 1 then               ' RTR Message?
         data$ = ""
         print #SER, #0, "RTR Message";
      endif
      print #SER, #0, data$
    endif
  endwhile
end
```

Data is read out via the secondary address 1 irrespective of the Frame-Format and Identifier bytes. The device driver only reads the data bytes and ignores the Identifier. Incompletely read CAN messages keep their frame format and ID byte, the length is corrected accordingly by the driver so that the next read operation again finds an intact CAN-message in the buffer.

Program example:

```
'-----------------------------------------------------------------
'Name: CAN_RX2.TIG
'receives CAN data and displays them, ignores IDs
'displays data as text (send ASCII only)
'displays also status
'connect a sending CAN device, e.g. a Tiger with CAN_TXS.TIG
'-----------------------------------------------------------------
user var strict                  'check var declarations
#INCLUDE UFUNC3.INC              'User Function Codes
#INCLUDE DEFINE_A.INC            'general symbol definitions
#INCLUDE CAN.INC                 'CAN definitions

'-----------------------------------------------------------------
TASK MAIN
  BYTE ever, frameformat, msg_len, can_stat
  WORD ibu_fill                  'output buffer fill level
  LONG r_id
  STRING id$(4), data$, line$

  install_device #LCD, "LCD1.TDD" 'install LCD-driver
  install_device #CAN, "CAN1_K1.TDD", & 'install CAN-driver
    "50 A0 00 00 &               'access code
     FF FF FF FF &               'access mask
     10 45 &                     'bustim1, bustim2
     08 1A"%                     'single filter mode, outctrl

  print #LCD, "<1Bh>A<0><0><0F0h>STAT LEN ID";

  line$ = ""
  for ever = 0 to 0 step 0       'endless loop
    get #CAN, #0, #UFCI_IBU_FILL, 0, ibu_fill
    print #LCD, "<1Bh>A<0><3><0F0h>IBU_FILL:";ibu_fill;"    ";
    get #CAN, #1, 0, data$
    if data$ <> "" then
      line$ = line$ + data$
      if len(line$) > 20 then     'if longer than LCD line
        line$ = right$ ( line$, 20 )
      endif
      print #LCD, "<1Bh>A<0><2><0F0h>";line$;
    endif
    get #CAN, #0, #UFCI_CAN_STAT, 0, can_stat
    using "UH<2><2>   0 0 0 0 2"  'HEX format for a byte
    print_using #LCD, "<1Bh>A<1><1><0F0h>";can_stat;
  next
END
```

Receipt of a 'Remote Transmission Request' leads to a message which has been especially provided for this purpose in the device driver being sent. The received CAN message would otherwise be treated as a CAN message without Remote Transmission Request'.

Program example:

```
'-----------------------------------------------------------------
'Name: CAN_RTR.TIG
'prepares a RTR-message and sends then 2 different messages
'in a loop.
'RTR message and loop message have different IDs
'connect a CAN device which uses a RTR message to get the
'response, e.g. a CAN Tiger with CAN_RTRS.TIG
'-----------------------------------------------------------------
user var strict                  'check var declarations
#INCLUDE UFUNC3.INC              'User Function Codes
#INCLUDE DEFINE_A.INC            'general symbol definitions
#INCLUDE CAN.INC                 'CAN definitions

'-----------------------------------------------------------------
TASK MAIN
  BYTE ever                      'endless loop
  STRING rtr_msg$(13)

  install_device #LCD, "LCD1.TDD" 'install LCD-driver
  install_device #CAN, "CAN1_K1.TDD", & 'install CAN-driver
    "50 A0 00 00 &               'access code
     FF FF FF FF &               'access mask
     10 45 &                     'bustim1, bustim2
     08 1A"%                     'single filter mode, outctrl

  rtr_msg$ = "<0><0FFh><0E0h>RTR-resp"'RTR response string as standard frame
  put #CAN, #2, rtr_msg$         'prepare device driver
  print #LCD, "RTR-message prepared"
                                 'now do something else
  for ever = 0 to 0 step 0       'endless loop
    wait_duration 3000
    put #CAN, #0, "<0><0FFh><0C0h>abcdefgh"
    wait_duration 3000
    put #CAN, #0, "<0><0FFh><080h>ijklmnop"
  next
END
```

CAN RTR messages

'Remote Transmission Request' messages are sent with secondary address 1. A RTR message never contains data bytes. In some cases the data length (DLC) contains the number of bytes that are required from the data frame. In this case you have to add dummy data to your message. The length of the dummy data specifies the data length (DLC) bits. Every CAN message is output with a PUT or Print instruction. With the Print instruction you must remember that the version will be formatted and any additional bytes (CR, LF) appended.

Receiving a 'Remote Transmission Request' messages is the same as receiving all other CAN messages. If the RTR bit is set and DLC is greater than 0, you have to get the data from the CAN Buffer. These data bytes are dummies, ignore them. After getting the dummy bytes, you can continue getting the next CAN message.

The CAN device driver expect a CAN message in the predefined format as an argument. The first byte will be interpreted as a Frame-Format byte . The next 2 or 4 bytes are the message's Identifier depending on the Frame-format. A typical CAN output as a Standard Frame looks as follows:

PUT #CAN, #1, ***"‹Frame-Format›‹ID1›‹ID2›data"***

‹Frame-Format›	contains information that this is a Standard-Frame.
‹ID1›	contains the upper bits 3...10 of the Identifier.
‹ID2›	contains the lower bits 0...2 of the Identifier at the bit positions 5, 6 and 7. The remaining bits in this byte are insignificant.
data	are dummy data bytes which specifies the DLC length of the RTR message. 0...8 data bytes are possible.

Sending a RTR message with DLC=0 (standard format):

```
msg$ = "<0><0><0>"
msg$ = ntos$ ( msg$, 1, -2, t_id )
put #CAN, #1, msg$
```

Sending a RTR message with DLC=8 (standard format):

```
msg$ = "<0><0><0>"+"12345678"
msg$ = ntos$ ( msg$, 1, -2, t_id )
put #CAN, #1, msg$
```

Program example receiving:

```
user_var_strict

#INCLUDE UFUNC3.INC                  ' User Function Codes
#INCLUDE DEFINE_A.INC                ' allg. Symbol-Definitionen
#INCLUDE CAN.INC                     ' CAN-Definitionen

task main
  BYTE frameformat, msg_len
  WORD ibu_fill
  LONG ac_code, ac_mask, r_id
  string slCode$(4), data$(8)

  INSTALL DEVICE #SER, "SER1B_K4.TD2",&
  BD_38_400,DP_8N,NEIN,BD_38_400,DP_8N,NEIN

  install_device #CAN, "CAN1_K8.TD2", & ' install CAN-driver
    "00 00 00 00 &                   ' access code
     FF FF FF FF &                   ' access mask
     01 5C &                         ' bustim1, bustim2
     00 1A"%                         ' dual filter mode, outctrl

  Print #SER,#0, "Can Receive All!"

  while 1 = 1
    get #CAN, #0, #UFCI_IBU_FILL, 0, ibu_fill
    if ibu_fill > 2 then                            ' if there is a message
      get #CAN, #0, 1, frameformat                  ' get Frame-Info-Byte
      msg_len = frameformat bitand 1111b            ' length
      if frameformat bitand 80h = 0 then            ' if Standard-Frame
        get #CAN, #0, CAN_ID11_LEN, r_id            ' get ID-Bytes
        r_id = byte_mirr ( r_id, 2 )                '
        r_id = r_id SHR 5                           '
        using "UH<8><3>   0 0 0 0 3"      ' fuer ID Anzeige
      else                                          ' it is extended frame
        get #CAN, #0, CAN_ID29_LEN, r_id            '
        r_id = byte_mirr ( r_id, 4 )                '
        r_id = r_id SHR 3                           '
        using "UH<8><8>   0 0 0 4 4"      ' fuer ID Anzeige
      endif
      print_using #SER, #0, "ID:"; r_id; ", ";   ' show ID
      using "UH<1><1>   0 0 0 0 1"        ' zeige Laenge an
      print_using #SER, #0, "DLC:";msg_len ; ", ";

      if msg_len > 0 then                           ' if there are data bytes
        get #CAN, #0, msg_len, data$                ' read out data
      endif
      if bit(frameformat, 6) = 1 then               ' RTR Message?
         data$ = ""
         print #SER, #0, "RTR Message";
      endif
      print #SER, #0, data$
    endif
  endwhile
end
```

I/O buffer

CAN messages consist of a Frame-Format byte, an Identifier and a maximum of 8 data bytes. The Identifier occupies 2 bytes in the case of a 'Standard frame'. With an 'extended Frame' the Identifier is 4 bytes long. Every message is stored in the buffer together with the Frame-Format byte and the Identifier. If a message no longer fits into the buffer the PUT instruction waits during sending until space is again available in the buffer. During receipt the message will be rejected and an Overflow error registered.

Number of data bytes	occupied in the buffer	
	Standard Frame	extended Frame
0	3	5
8	11	13

Note: if a string containing more than 8 data bytes is transferred to the buffer with only one single PUT instruction, space will be needed for additional Identifiers since the date is split between several CAN messages.

Both incoming and sent data will be buffered in a buffer. Size, level or remaining space of the input and output buffer as well as the driver version can be inquired with the User-Function codes.

During both output and receipt, a buffer will be regarded as being as full as soon as less than 13 bytes are free. A CAN message in Extended-Frame format is 13 bytes long. This limit applies since half CAN messages cannot be stored.

User-Function-Codes for inquiries (instruction GET):

If there is not enough space in the output buffer and you nevertheless wish to output the instruction PUT or Print (and thus the complete task) waits until space once again becomes free in the buffer. This waiting can be avoided by inquiring the free space in the buffer before output.

Example: only output if still sufficient free space in the output buffer:

```
GET #CAN, #0, #UFCI_OBU_FREE, 0, wVarFree
IF wVarFree > (LEN(A$)) THEN
  PUT #CAN, #0, A$
ENDIF
```

Example: check whether there is a message in the input buffer (the shortest possible message is 3 bytes long):

```
GET #CAN, #0, #UFCI_IBU_FILL, 0, wVarFill
IF wVarFill > 2 THEN
  ` lies die CAN-Nachricht
ENDIF
```

Automatic bit rate detection

If the driver is installed in the 'Listen-Only' mode it tries to automatically recognize the bit rate. In the 'listen-only' mode the CAN chip itself cannot send anything so that the otherwise familiar error telegrams will not be produced as long as the bit rate has not been recognized. Which bit rates are actually recognized can be set in a table. If no table is transferred during installation an internal table will be used.

The following prerequisites must be met to detect the bit rate:

- An operative bus with data traffic is assumed, i.e. there must be at least two active participants who send something.
- The table must contain the correct bit rate.

The bit rate detection starts with the first setting from the table, as a rule the highest possible bit rate. No receive error occurs with the next data packet on the CAN bus if the bit rate is already correct. If a receive error does however occur, then the driver switches to the next bit rate in the table and waits for a new CAN telegram. The driver waits in every case until sufficient CAN telegrams have either enabled a recognition of the bit rate or the table of possible values has been processed three times. If the bit rate wasn't recognized, the CAN device driver will not be installed. If CAN telegrams are only sent very rarely over the bus and the correct bit rate is only at the end of the table, the detection takes accordingly longer. If the bit rate wasn't recognized, the device driver quits the 'listen-only' mode.

The table contains the settings for the registers 'bustim0' and 'bustim1' in the CAN chip. 2 bytes will therefore be needed for every setting. The table must contain at least 4 bytes otherwise the internal table which contains the following values will be used

Program example:

```
'----------------------------------------------------------------------
'Name: CAN_ABR.TIG
'auto bitrate selection from pre-defined table
'rest similar to CAN_RX1.TIG
'connect with a CAN bus with sending devices
'----------------------------------------------------------------------
user var strict                   'check var declarations
#INCLUDE UFUNC3.INC               'User Function Codes
#INCLUDE DEFINE_A.INC             'general symbol definitions
#INCLUDE CAN.INC                  'CAN definitions

'----------------------------------------------------------------------
TASK MAIN
  BYTE ever, frameformat, msg_len, can_stat
  WORD ibu_fill                   'input buffer fill level
  LONG r_id                       'received ID
  STRING msg$(8), data$(8)

  install_device #LCD, "LCD1.TDD" 'install LCD-driver
  print #LCD, "trying to find <10><13>CAN bitrate.<10><13>Please wait..."
  install_device #CAN, "CAN1_K1.TDD", & 'install CAN-driver
    "50 A0 00 00 & 'access code
     FF FF FF FF & 'access mask
     00 00 &       'bustim1, bustim2
     0A 1A &       'single filter + listen only, outctrl
     00 43 &       '1 Mbit      here on table with bytes
     00 5C &       '500 kbit     for bustim0 and bustim1
     01 5C &       '250 kbit            for auto bitrate
     03 5C &       '125 kbit                   detection
     04 5C &       '100 kbit
     09 5C &       ' 50 kbit
     10 45 &       ' 49 kbit for SLIO: TSYNC + TSEG1 + TSEG2 = 10
     0F 7F &       ' 25 kbit
     1F 7F"%       ' 12.5 kbit

  print #LCD, "<1>STAT LEN ID";

  for ever = 0 to 0 step 0        'endless loop
    get #CAN, #0, #UFCI_IBU_FILL, 0, ibu_fill
    print #LCD, "<1Bh>A<0><3><0F0h>IBU_FILL:";ibu_fill;"    ";
    if ibu_fill > 3 then          'if at least one message
      get #CAN, #0, 1, frameformat 'which frame format?
      msg_len = frameformat bitand 1111b
      if frameformat bitand 80h = 0 then  'if standard frame
        get #CAN, #0, CAN_ID11_LEN, r_id 'get ID bytes
        r_id = byte_mirr ( r_id, 2 )     'byte order for Tiger WORD
        r_id = r_id shr 5                'shift right bound
        using "UH<8><3>   0 0 0 0 3"     'to display ID
      else                               'else it is extended frame
        get #CAN, #0, CAN_ID29_LEN, r_id 'get ID bytes
        r_id = byte_mirr ( r_id, 4 )     'low byte 1st in LONG
        r_id = r_id shr 3                'shift right bound
        using "UH<8><8>   0 0 0 4 4"     'to display ID
      endif
      print_using #LCD, "<1Bh>A<9><1><0F0h>";r_id;
```

```
      using "UH<1><1>   0 0 0 0 1"       'display length
      print_using #LCD, "<1Bh>A<6><1><0F0h>";msg_len;
      if msg_len > 0 then                'if contains data
        get #CAN, #0, msg_len, data$     'get them and display
        msg$ = "        "                '8 spaces
        msg$ = stos$ ( msg$, 0, data$, msg_len )'prepare for LCD field
        print #LCD, "<1Bh>A<0><2><0F0h>data:";msg$;
      else
        print #LCD, ;" RTR      ";
      endif
    endif

    get #CAN, #0, #UFCI_CAN_STAT, 0, can_stat 'CAN status
    using "UH<2><2>   0 0 0 0 2"  'HEX format for one byte
    print_using #LCD, "<1Bh>A<1><1><0F0h>";can_stat;
  next
END
```

A short introduction to CAN

CAN is an abbreviation for Controllers Area Network. Originally, CAN was developed as a communications protocol to exchange information in motor vehicles. CAN is now just as common in automation engineering and domestic engineering. The basis for the CAN bus is a hardware which makes the connection to the CAN bus and takes care of the actual message dispatch and message receipt, similar to a UART at the RS 232 interface, though checksums, error control and repetition of the messages in the event of errors as well as bus arbitration and bus prioritization. There are a number of manufacturers who have implemented the CAN-interface on their processor and there are external CAN chips which can be connected to processors which do not have a CAN-interface 'on-board'.

Compact data packets are sent on the CAN bus, referred to in the following as CAN messages. A message consists of an Identifier and between 0 and 8 data bytes from a user point of view. There are two variants of the bit protocol on the bus, **with 11-Bit-Identifiers** in accordance with CAN 2.0A and with **29-Bit-Identifiers** in accordance with CAN 2.0B. Both variants exist next to each other, and both have their advantages and disadvantages. Modern chips support either CAN2.0B or at least accept the existence of 29 bit Identifiers on the (CAN2.0B passive).

Bus accesses and access priorities are defined by the CAN specification and are handled completely by the CAN hardware. The application software places the CAN message with a 'label' in the CAN send mail box. The label, or Identifier, is not however an address label but an identification of the contents of the CAN message, e.g. the temperature information from sensor 'A', or the adjustment information for pressure controller 'X'. Any bus user for whose application the message is important will be programmed to accept this message . The sender cannot find out whether any other node has accepted the message.

A **receiving filter** in the CAN hardware pre-filters the messages according to certain criteria so that all messages reach the application. The biggest differences between the different implementations of CAN hardware are in the receiving part. Both the manner of the filtration and the number of the messages which are saved in the receive mail box are very different. An attempt is made to only allow those messages through the filter, which are important for the application.

So-called **'Remote Transmission Requests'** can be sent out on the CAN bus. The corresponding bus users are requested to respond with a specific message. Thus, for example, the request to report the 'Temperature Boiler 2' can appear on the bus. The applications in the single CAN nodes determine whether a response will be made to such send requests and the contents of the response.

The **bus accesses** take place in a fixed time grid. All bus users synchronize themselves with every bus access. The accesses take place at the same time. The idle level on the bus is the '1'. This level is not the dominant one. A '1' can be overwritten by a '0', thus the term 'dominant' for the '0'. A bus access starts with a **dominant** '0'. This is followed by the '1' and '0' levels of the Identifier, starting with the highest-order bit. The lower priority bus users have '1'-bits in the higher-order bit positions and can therefore be overwritten by the prioritized bus users with a '0'. As soon as a user is unable to place his '1' during a bus access he aborts the bus access to try again later. This renewed trial is carried out automatically by the CAN hardware and need not be programmed in the application, which knows nothing at all of this. Only if a bus access proves impossible after a number of attempts, and the bus therefore apparently permanently occupied by dominant users, will the application be able to recognize this status by an inquiry to the error registers of the CAN hardware.

The most concise differences to the majority of other networks and bus systems are compared here:

Most other industrial bus systems	CAN bus
Every user receives an address and messages are given a destination address, sometimes together with an origin address.	There aren't any addresses. The messages are provided with a content declaration instead of the address. The users have programmable input filters which allow certain messages to pass through.
An acknowledgement of receipt is often scheduled. The receiver then confirms the correct receipt of the transmission.	At the end of a message package the CAN hardware confirms that this has been received correctly on the bus (Acknowledge). Whether any user has in fact accepted the message is unknown.
Rules exist for the bus access so that two users never use the bus simultaneously.	Several users can access the bus with CAN simultaneously. Prioritized users replace the others, who automatically access the bus later, during the access. The bus access is handled completely by the CAN hardware.

Error situations

In the following, some error situations are listed and it will be shown how these can be recognized .

Error	Possible cause
What is seen on the Scope: a user permanently and continually sends on the bus although the application only wanted to send a single message.	The sending user, or better: their hardware, receives no Acknowledge from another bus user. The CAN hardware thus sends the message again and again. Possible reasons: Only one active user is on the bus. The others are either unavailable, switched off or have not been initialized. The bit rate of this participant doesn't correspond with the bit rate of the other bus users.
Messages which are safely sent don't arrive.	Receive errors occur. Have the error register shown to be able to draw conclusions on the error. If the error registers are all right, it could be that the filters don not let the Identifier pass.
When sending, the error register is set immediately.	The bus is possibly permanently occupied by a higher prioritized user (overload) or the bit rate is wrong. Is there another active user? At least one bus user must set the ACK bit.

Documentation History

Version of Documentation	Description / Changes
001	- First release
002	- OTYPE_PIN, OTYPE_PORT, PU_PD_PIN, PU_PD_PORT
003	- SER1B baudrates - SYSVARN: BACKUP_RAM_SIZE - READ_BACKUP_RAM / WRITE_BACKUP_RAM - RTC1 User-function-codes
004	- ANALOG2
005	- RTC1 example
006	- Font bug fixed
007	- CAN-Bus
008	- SYSVARN: FLASH_BUSY - ERASE_FLASH_SECTOR
009	- ANALOG1
010	- ERASE_FLASH_SECTOR Tiger plus Firmware notice added
011	- READ_T_CODE$