Manual Addendum

TINY-Tiger 2

Blank Page

Index

Blank Page

Installation

In order to work with Tiger2 using an existing compiler-version 5.2, several new files are required, which have to be copied into particular directories of your existing Tiger-BASIC installation. This concerns the following files:

file name(s):	file tipe:	copy to:
Tgbas32.exe	new compiler-version	..\Bin
*.TD2	Device drivers for Tiger 2	..\Bin
Tac0100.TA2	System file for Tiger 2	..\Bin
Tac0100_.TA2	System file for Tiger 2	..\Bin
Thinfo0.TH2	System file for Tiger 2	..\Bin
Define_a.INC	general symbol-definitions	..\Include
Ufunc4.INC	definitions user-function-codes	..\Include

Development environment

The following needs to be considered in the Tiger-BASIC IDE when employing TINY-Tiger 2:

- 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”.

- The TINY-Tiger 2 module will be recognized by its development environment automatically. If a program has to be compiled for the TINY-Tiger 2, without a module being connected, the module type has to be set to “Tiger 2” in the menu **Options / Compiler.**

TINY-Tiger 2 module

Hardware

Aside from the basic differences between the classical TINY-Tiger and the new TINY-Tiger 2 such as the number of pins and therefore the number of I/O's due to 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 TINY-Tiger 2, the pins L33...L37, L40...L42, L60...L67, L70...L73 as well as L80...L87 have a voltage range of 0 to 3.3 V as outputs. In the TINY-Tiger, these pins have a voltage range of 0 to 5 V.

- Pin 43 (battery input) of the TINY-Tiger serves as an input for the battery buffering of the SRAM as well as the RTC (real time clock). In the TINY-Tiger 2, there are two separate pins: pin 43a (BATT-RAM) for the buffering of the SRAM as well as pin 43b (BATT-RTC) for the buffering of the real time clock.

Software

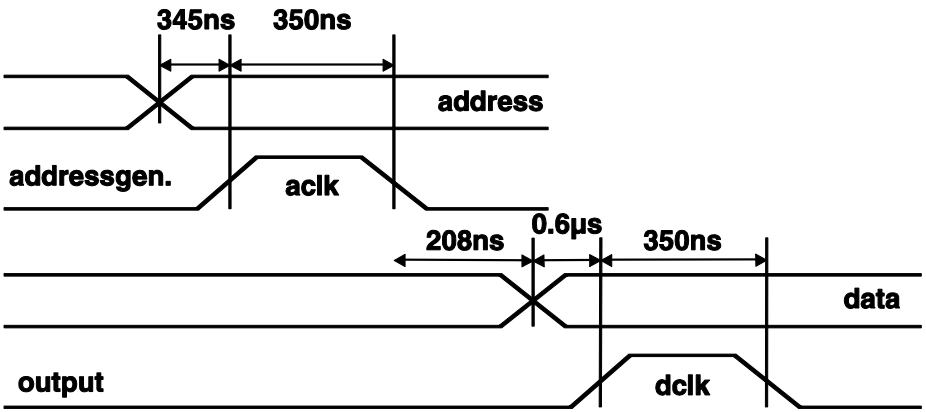
A further change in the TINY-Tiger 2 concerns the software, viz. the file type STRING: Theoretically, strings with a length of up to 2 GB can be processed. In practice, therefore, the length of a string is only restricted by the size of the module's SRAM.

String length

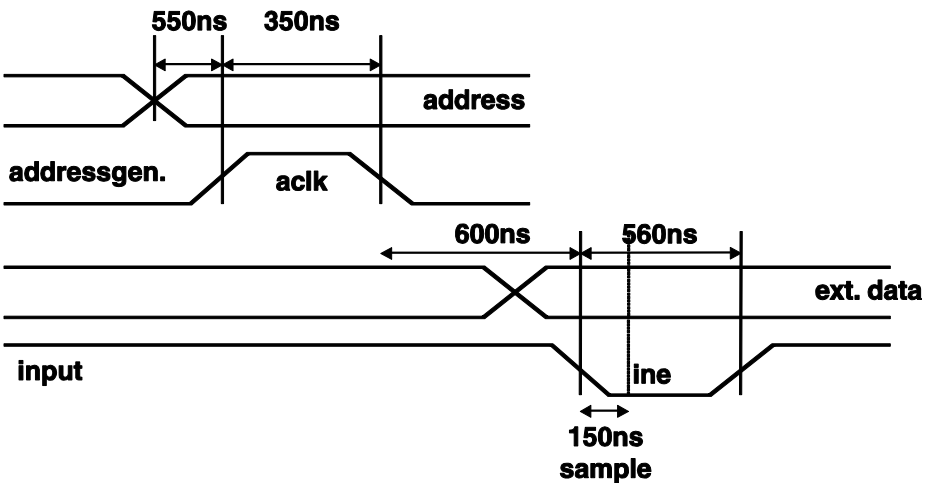
In the Tiger 2, the maximum length of a string is no longer restricted (only by the RAM). Therefore, even more data can be put into a string. This is to be taken with a grain of salt, though, since the duration of the operations increases correspondingly for very large strings. Very large strings can also influence the timing of the multi-tasking system, since one BASIC instruction is always completed before a change in tasks can take place.

XBUS Timing

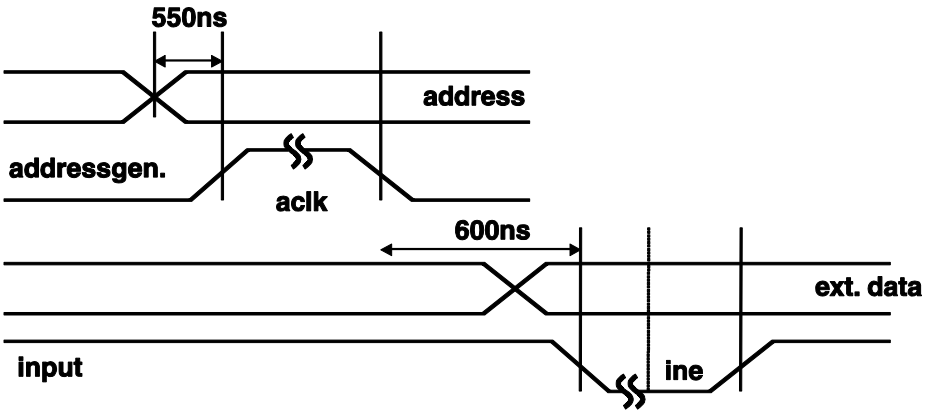
Output timing of extended I/O system (TINY-Tiger 2):



Input timing of extended I/O system (TINY-Tiger 2):



Input timing of extended I/O system with Delay (TINY-Tiger 2). For details about slowing down the bus speed, please look at documentation of *XIN* with delay in *New_and_updated_Functions_since_5_2.* This function is typically used with Tiger-2, if devices running with Tiger-1 are too slow for the use with Tiger-2.



New functions: Interrupts

In the Tiger 2, one can operate interrupts as well as multi-tasking. This happens very easily through the new interrupt tasks. Instead of polling a pin or determining certain time distances with wait_duration or diff_ticks, you can now configure interrupts. In total there are now 4 different interrupt sources with 4 different priorities. It is important to first configure the interrupts correctly and then to start them.

The interrupts are handled with in a separate task, which is provided **only** for the handling of the interrupts. Normal tasks cannot be assigned to an interrupt and interrupt tasks cannot be started normally like any other task. The INTTASK starts automatically, if the corresponding interrupt source is set off.

If several interrupts occur simultaneously, the interrupt with the highest priority will be carried out first. An interrupt in process can **only** be interrupted by an interrupt of a **higher priority**.

Priority	Interrupt source
0	INTM1
1	Timer
2	INTM3
3	INTM4

When an interrupt occurs, the normal multitasking system is stopped. The current instruction is carried through and after that the whole program comes to a halt. Only the INTTASK is being carried out. After quitting the INTTASK, the program returns to where it has been interrupted. The INTTASK can only be interrupted by higher interrupts, which will also be carried through first, and only after this does the program return to the interrupt.

Nota bene: Please note that the execution of an interrupt should occur fast. It is therefore better to refrain from very complex instructions and various instructions such as wait_duration or loops since the interrupt handler has to be operated as fast as possible, just in case a new interrupt occurs during the process. This interrupt can be buffered optionally, however, it is more correct when no other interrupt occurs during the

interrupt handling. This goes especially critical with timer interrupts, as they occur in equal time sequences.

The following tasks and instructions can be used for interrupts:

INTTASK	=> defines a task for the handling of interrupts
SET_INT	=> assigns an INTTASK to an interrupt
CONFIG_TIMER_INT	=> writes byte above HDQ
ENABLE_INT	=> activates one or all interrupts
DISABLE_INT	=> deactivates one or all interrupts

INTM4

The INTM4 interrupt is a very special one, and has to be dealt with carefully. It can be used on BASIC level and also implements a counter (Long), which can be controlled by functions.

The BASIC interrupts occur no more frequently than every ms. Which means that they are constrained to some extent in their timing capabilities.

The counter will, however, be counted up on every recognized interrupt, so no interrupt will be missed. This interrupt therefore also suits perfectly for counting interrupts. This happens automatically and the handling does not have to be implemented into an INTTASK. The counting is of course very much faster than counting the values in an INTTASK!

For the interrupt INTM4 you can use the following functions:

COUNTS	=> reading the counter
DIFF_COUNTS	=> difference
SET_COUNTS	=> setting the counter

PC mode

After the download of the basic program from the IDE, **all** interrupts are by default **turned off**. Since the time response is different in the PC mode, you are therefore are secured from an overflow of interrupts. Of course, the interrupts can easily be turned on again. When canceling the lock-out of the interrupts, they will occur exactly as set out in the program.

Following the same principle, the buffering of the interrupts can be controlled. The buffering can therefore be stopped or one can let it work as described in the program.

To modify the settings, please go to:

Debug -> Debug interrupts...

There, the interrupts can be turned off globally or individually, and the buffering can be set for each interrupt.

INTTASK

```
INTTASK Name
  :
  :
END
```

Function: INTTASK begins a task that is made especially for the handling of interrupts. After the task has been assigned to an interrupt, the task will automatically be called when the interrupt is triggered.

Please note: The INTTASK can be written like any other task, it has local variables, as well as access to all global variables.

Important: Please note that the execution of an interrupt should occur fast. It is therefore better to refrain from very complex instructions and various instructions such as wait_duration or loops since the interrupt handler has to be operated as fast as possible, just in case a new interrupt occurs during the process. This interrupt can be buffered optionally, however, it is more correct when no other interrupt occurs during the interrupt handling. This goes especially critical with timer interrupts, as they occur in equal time sequences.

SET_INT

SET_INT task name, interrupt number

Function: Assigns an INTTASK to an interrupt.

Parameters:

	B	W	L	S	F	
Task name	-	-	-	-	-	The name of an existing INTTASK will be quoted. This task will be started when the interrupt is triggered.
Interrupt number	●	●	●	-	-	Number of interrupt signal that starts the task: 1: INTM1 2: Timer interrupt 3: INTM3 4: INTM4

Please note: Before the interrupts are activated, an INTTASK has to be assigned to the active interrupts!!!

CONFIG_TIMER_INT

CONFIG_TIMER_INT prescaler, interval, postscaler

Function: The interval for the timer interrupt is being set.

Parameters:

	B	W	L	S	F	
Prescaler	●	●	●	-	-	Determines the frequency/resolution for a timer tick. 0: 0.4µs 1: 1.6µs 2: 6.4µs
Interval	●	●	●	-	-	Value between 0....65535 Determines the interval length. More details further down in the text.
Postscaler	●	●	●	-	-	Sets the number of the already determined time intervals, until a timer interrupt is triggered.

If you require an interval of 0.4s at a resolution of 6.4 µs you simply divide:

0.4s / 6.4µs = 62,500

Therefore 62,500 is given as 2nd parameter, if you require an interval of 0.4s with a resolution of 6.4µs.

➔ desired time / prescaler = interval

If you want to increase the time span further, you can increase the postscaler. The postscaler determines how often the time span that has been set before has to occur, until an interrupt is triggered. If the time span is 0.4 seconds, as above, and the postscaler is set to 10, the interrupt will be triggered every 4 seconds. It is therefore possible to create very large time spans (up to › 7 hours). The postscaler must not exceed the value 65535 (0FFFFH). If you do not want a postscaler, put in the value 1 or 0.

Please note:

The timer interrupt must not be chosen too narrowly, since, in general, the processing is to be faster than the time span for the interrupt. When the INTTASK is not ready yet, and another interrupt does still occur, there will be timing problems and the rest of the program will not be carried out.

ENABLE_INT

ENABLE_INT Int-No, buffering

Function: One interrupt, or optionally all interrupts are activated.

Parameters:

	B	W	L	S	F	
Int-No	●	●	●	-	-	Number of interrupt signal to be enabled: 1: INTM1 2: Timer interrupt 3: INTM3 4: INTM4 255: *all* interrupts
buffering	●	●	●	-	-	0= interrupt is buffered 1= interrupt is not buffered

Please note: Before the interrupts are activated, an INTTASK with SET_INT has to be assigned to them. In the case of the timer interrupt, it has to be configured additionally with CONFIG_TIMER_INT.

Please note: When the interrupt is **not buffered**, the next interrupt can only occur when the corresponding INTTASK is brought to completion. All identical interrupts before this moment will be ignored. When the interrupts are **buffered**, the interrupts that occur during the corresponding INTTASK, will be carried out immediately after the INTTASK, so they will be attached to the interrupt. *The maximum number of interrupts per interrupt source that can be buffered is 255!*

DISABLE_INT

DISABLE_INT Int-No

Function: Deactivates one or all interrupts.

Parameters:

	B	W	L	S	F	
Int-No	●	●	●	-	-	Number of interrupt signal to be disabled: 1: INTM1 2: Timer interrupt 3: INTM3 4: INTM4 255: *all* interrupts

Please note: After calling up the instruction, the interrupt will no longer be triggered. The buffered interrupts up to now will still be executed. They will **not be dismissed**.

Sample program:

```
#include define_a.inc

' Global variables
long INT1
long INT2
long INT3
long main_cnt


task main

  install_device #0, "LCD1.TD2"  ' install LCD driver

  ' init Vars
  INT1 = 0
  INT2 = 0
  INT3 = 0
  main_cnt = 0


  ' Show start status of the (interrupt) values
  print #0, "<1BH>A"; CHR$(0); CHR$(1); "<0F0H>"; INT1
  print #0, "<1BH>A"; CHR$(0); CHR$(2); "<0F0H>"; INT2
  print #0, "<1BH>A"; CHR$(0); CHR$(3); "<0F0H>"; INT3


  ' set interrupt vectors
  set_int one, 1
  set_int two, 2
  set_int three, 3


  config_timer_int 2, 62500, 10 ' configure timer interrupt ( 4 sec )

  enable_int 255, 0             ' start ALL interrupts


  while 1=1
    print #0, "<1BH>A"; CHR$(0); CHR$(0); "<0F0H>"; main_cnt
    wait_duration 500
    main_cnt = main_cnt + 1
  endwhile
end



inttask eins
  INT1 = INT1 + 1
  print #0, "<1BH>A"; CHR$(0); CHR$(1); "<0F0H>"; INT1
end

inttask zwei
  INT2 = INT2 + 1
  print #0, "<1BH>A"; CHR$(0); CHR$(2); "<0F0H>"; INT2
end
```

```
inttask drei
  INT3 = INT3 + 1
  print #0, "<1BH>A"; CHR$(0); CHR$(3); "<0F0H>"; INT3
end
```

COUNTS

long = COUNTS ()

Function: Get the current counter reading of the INTM4 interrupt.

COUNTS delivers the number of the counted INTM4 interrupts. The COUNTS counter overruns after 2,147,483,647 counts. The counter remains in the positive values margin of the LONG numbers. The difference between two counter readings is determined by the function DIFF_COUNTS and takes into account that the counter might just have overrun. With SET_COUNTS, the counter is set to a new value.

DIFF_COUNTS

diff = DIFF_COUNTS (x)

Function: Determines the number of occurred interrupts from a given moment.

Parameters:

	B	W	L	S	F	
x	●	●	●	-	-	Value of the saved counter reading
diff	●	●	●	-	-	Difference between current counter reading and saved counter reading x.

The COUNTS counter overruns after 2,147,483,647 counts. The counter remains in the positive values margin of the LONG numbers. The function DIFF_COUNTS delivers the difference between the given value and the current value of the counter correctly even in the moment of the overrun.

SET_COUNTS

SET_COUNTS (cnt)

Function: Set the INTM4 counter to a certain value.

Parameters:

	B	W	L	S	F	
cnt	●	●	●	-	-	Counter will be set to this value.

The counter counts the INTM4 interrupts. The COUNTS counter overruns after 2,147,483,647 interrupts. The counter remains in the positive values margin of the LONG numbers. The current counter reading is read with the function COUNTS. The difference between two counter readings is determined by the function DIFF_COUNTS and takes into account that the counter might just have overrun.

SLEEP

SLEEP ()

Function: Tiger goes into the energy saving modus and the basic program stops at this point, until an INTM4 occurs. The Tiger 2 now consumes only about 30 mA.

Please note: After executing this instruction, the Tiger 2 stops immediately. To continue the program, an interrupt has to be generated on INTM4. Then the program will be continued behind the SLEEP function.

Further functions:

SHIFT_OUT

SHIFT_OUT *Log_iPortaddr, data pin, clock pin, variable, number*

Function: Clocked, serial output to external chips. SHIFT_OUT transfers the indicated **number** of bits of the **variable** through the **data pin**. For each bit of data, a clock impulse is produced by double inversion of the **clock pin**. Data pin and clock pin are on one internal port with the address **Log_iPortadr**.

Parameters:

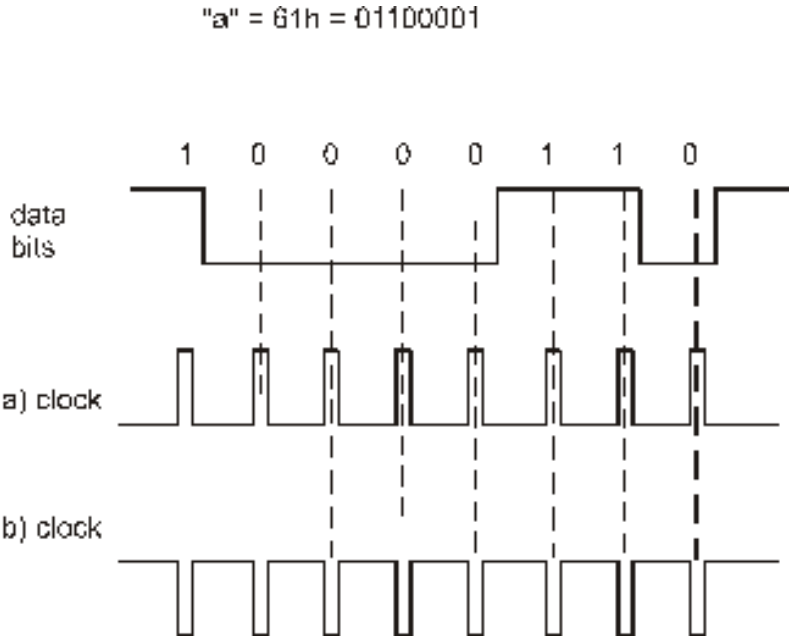
	B	W	L	S	F	
Log_iPortaddr	●	●	●	-	-	Logical, internal port address
Data pin	●	●	●	-	-	Number of pin at which data bits are put out.
Clock pin	●	●	●	-	-	Number of pin that is used as clock pin.
Variable	●	●	●	●	-	Contains the data to be written at data pin.
Number	●	●	●	-	-	With numerical variables 'Number' determines how many bits are written. With a positive number, the least significant bit (LSB) will be sent first, with a negative number, the most significant bit (MSB) will be sent first. The number of bits to send is restricted to 32. Spare bits with BYTE or WORD are written as 0-bits. With variables of the type STRING, the given number is valid for every single byte and is restricted to 8. The whole string is always written. A REAL number (8 bit * 8 byte) can be shifted, if it is first converted to a LONG-number with the help of the functions RTL or LREAL and HREAL.

SHIFT_OUT writes a serial data stream clocked to an output-pin of an internal port. A second pin of the port is used as clock pin. The clock is set by the TINY-

Tiger® 2 module. A clock impulse is generated by inverting the clock pin twice. The idle state can therefore be preset. A SHIFT_OUT instruction writes a maximum of 32 bits with whole numbered numerical variables, always 64 bits with REAL numbers and with strings always the whole string. For strings, the quantity tells how many bits of each byte are being written.

Applications of the SHIFT_OUT instruction are e.g. shift registers or connections of several modules, when the serial ports are already occupied for other purposes.

The picture shows the transfer of the character ‘a’ with high active (a) and with low active (b) level on the clock line.



On the Tiger 2® modules, the cycle time is 1.4 microseconds as a standard. With SET_SYSVARN, a system variable can be set that slows down the clock, up to a cycle time of 5.8 microseconds.

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 TINY-Tiger 2. A distinction is made in the naming, however:

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

There might be some differences for some drivers due to special specifications of the TINY-Tiger 2. These will be talked about in more detail later on.

ANALOG1.TD2

Since the TINY-Tiger 2 now offers 12 analog inputs, the secondary addresses chance slightly when reading in. Here are some examples:

GET #4, #0, 1, value reads from the Analog1 driver (here: address 4) from A/D-channel 0 exactly 1 byte into variable 'value' (8 bit resolution). Value is of type BYTE, WORD or LONG.

GET #4, #4, 2, value reads from the Analog1 driver from A/D-channel 4 exactly 2 bytes into variable 'value' (10 bit resolution). Value is of type WORD or LONG.

GET #4, #5, 2, value reads from the Analog1 driver from A/D-channel 5 exactly 2 bytes into variable 'value' (10 bit resolution). Value is of type WORD or LONG.

GET #4, #11, 2, value reads from the Analog1 driver from A/D-channels 11 exactly 2 bytes into variable 'value' (10 bit resolution). Value is of type WORD or LONG.

**GET #4, #12, 12, V$** reads from the Analog1 driver from the A/D-channels 0...11 exactly 1 byte per channel into V$ (8 bit resolution). V$ is of type STRING and must be large enough to accommodate 12 bytes. The 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, 1)

**GET #4, #13, 24, V$** reads from the Analog1 driver from the A/D-channels 0...11 exactly 2 bytes per channel into V$ (10 bit resolution). V$ is of type STRING and must be large enough to accommodate 24 bytes. 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)

A/D inputs with ANALOG2.TD2

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.TD2:

- User-function-codes
- Measuring in FIFO
- Measuring in string
- Measuring with 12 bit
- Setting of the sample-rate
- Measuring with trigger

File name: ANALOG2.TDD

INSTALL DEVICE #*D*, "ANALOG2.TDD"

D is a constant, variable or an expression of the data type BYTE, WORD or LONG in the range of 0...63 and represents 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, 4, 5, 6, 7, 8, 9, 10, or 11)
- from channel 0 and 1
- from channel 0, 1 and 2
- from channel 0, 1, 2 and 3
- from channel 0, 1, 2, 3, 4, 5, 6, and 7 (only string)
- from channel 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, and 11 (only string)

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 different 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 particular 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 prefix: UFCO_	Description
46	UFCO_AD2_RESET	Set all parameters to standard values
128	UFCO_AD2_CHAN	Set single channel mode (FIFO, STRING): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 (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
135	UFCO_AD2_GROF	Set growth-flag (STRING) 0 = spontaneous assignment of size at the end of a row of measurements. else = continual increase of string-length with every measurement (when string size increases!)
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

No.	Symbol prefix: UFCO_	Description
		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 n = 8: 8-channel: Ch-0 ... Ch-7 n = 12: 12 channel: Ch-0 ... Ch-11
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):

Nr	Symbol prefix: UFCI_	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
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 in FIFO

First determine the desired resolution, the desired maximum measuring rate as well as the number of channels.

If you want to produce a stream of measurement data with no interruptions, you have to measure into a FIFO buffer. How the measured data is removed from the FIFO buffer and processed determines the maximum possible measuring rate that there is without the FIFO buffer running over and this causing measured data to be lost. The larger the FIFO buffer, the larger the fluctuations in the speed of the processing of the data are allowed to be. The measuring in a FIFO buffer as well as the collecting of the data, however, is slightly slower than measuring in a string. When the FIFO buffer is full, the device driver stops the measurement. For further measurements, it has to be started again, or the “wrap-around” needs to be turned on beforehand.

With 12-bit resolution (and only there!) the integration depth can be set (size of the internal integration buffer). The number of measurements can be reduced, by not transferring every internally read measurement into the string or the FIFO buffer. In that way, measurements are made that are further apart in the time scale, but which are reduced in noise.

After all settings have been made with the help of the User-Function-Codes, the measurement in a FIFO buffer is started this way:

PUT #D, *FIFO_Name*

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

FIFO_Name is the FIFO buffer, into which the measurements are written. The buffer is FIFO of byte with 8-bit measurements and FIFO of word with 10-bit or 12-bit measurements. The FIFO buffer is set automatically to EMPTY at the beginning.

Please note, that with integration (12-bit), the measured values are valid only when the internal integration buffer is filled once.

The measurement can be aborted with the User-Function-Code UFCO_AD2_STOP.

Sample program:

```
'---------------------------------------------------------------------
' Name: ANALOG2F.TIG
'---------------------------------------------------------------------
#INCLUDE DEFINE_A.INC                    ' general definitions
#INCLUDE UFUNC3.INC                      ' User Function Codes

TASK MAIN                                ' Beginning Task MAIN
  FIFO SAMPLE (256) OF WORD              ' Sample-Buffer
  WORD A, B, C, D                        ' Var. for analog values
' install LCD-driver (BASIC-Tiger)
  INSTALL DEVICE #LCD, "LCD1.TD2"
' install LCD-driver (TINY-Tiger)
'  INSTALL DEVICE #LCD, "LCD1.TD2", 0, 0, 0, 0, 0, 0, 80h, 8
' install TIMER-A driver(time basis clock: 1001Hz)
  INSTALL_DEVICE #TA, "TIMERA.TD2", 3, 156
' install ANALOG-2 driver
  INSTALL_DEVICE #AD2, "ANALOG2.TD2"

  PUT #AD2,#0,#UFCO_AD2_RESO, 10         ' resolution
  PUT #AD2,#0,#UFCO_AD2_SCAN, 4          ' number of channels
  PUT #AD2,#0,#UFCO_AD2_STOVL, 0         ' stop on overflow
  PUT #AD2,#0,#UFCO_AD2_PSCAL, 5         ' Pre-Scaler: 1001/5=200S/sec
  PUT #AD2,SAMPLE                        ' Start measurements

  K = 0
  WHILE K < 127                          ' End when a FIFO is full
    K = LEN_FIFO(SAMPLE)                 '
    PRINT #LCD, "<1>Length=";K
  ENDWHILE
                                         ' ----------------------------
  WHILE LEN_FIFO(SAMPLE) > 4             ' show FIFO
    GET_FIFO SAMPLE, A
    PRINT #LCD, "<1bh>A<12><0><0f0h>";A;
    GET_FIFO SAMPLE, B
    PRINT #LCD, "<1bh>A<12><1><0f0h>";B;
    GET_FIFO SAMPLE, C
    PRINT #LCD, "<1bh>A<12><2><0f0h>";C;
    GET_FIFO SAMPLE, D
    PRINT #LCD, "<1bh>A<12><3><0f0h>";D;
  ENDWHILE
  PRINT #LCD, "<1Bh>A<0><3><0F0h>ready";
END                                      ' End Task MAIN
```

Measuring in String

First, determine the desired resolution, the desired (maximum) measuring rate, as well as the maximum number of channels.

If you want to produce successive measuring sections, it is advisable to measure in a string. Advantages: the measurement requires less CPU performance and the processing is faster than with measurements in FIFO buffer, due to the string processing functions. When the string is to be passed on serially, e.g., it can be sent directly in pieces of 240 bytes each (this is the restriction of the instructions PRINT and PUT). The measurement is stopped, when the string is full.

To read in analog values into a string, first, a string is declared in the fitting length. The time basis driver TIMERA.TD2 is integrated and set to the highest basic frequency that is required in the application. Further settings such as pre-scaler, resolution, number of channels as well as number of measurements are set.

! There always has to be a measurement-string! Not permitted are thus variables that live only temporarily, like local strings (in subroutines) or temporary strings (expressions). **Correct: global or task-local strings.**

The measurement does not have to be written in the string from position 0. Also, the string does not necessarily have to be empty. An offset can be named, from which position in the string data is written. Values before the writing position are maintained, when they have been defined beforehand. If the string was shorter than the offset, there are undefined values in front of the writing position.

The measurement is terminated either when the set number of measurements is reached or when the string is full. However, the string does not necessarily reach its maximum length: when there are still 2 free bytes in the string at a 4-channel measurement with 8-bit resolution, the measurement does not take place, because at each measurement 4 bytes are created. The string length thus remains 2 bytes under the maximum length.

When all settings have been made with the help of the User-Function-Codes the measurement in a string is started as follows:

PUT #D, *String* [, *Offset*, *number*, *Growth_Flag*]

D	is a variable, a constant, or an expression of the data type BYTE, WORD, LONG in the range between 0...63 and stands for the device number of the driver.
String	is the string, into which the measured values are written. **The string needs to be static, i.e. global or task-local.** Note: the string is set to EMPTY at the beginning.
Offset	is a variable, a constant or an expression of the type BYTE, WORD or LONG and determines the offset, when the measured values are to be written into the string from a position unequal to 0. Default = 0 (beginning of string).
Number	is a variable, a constant or an expression of the type BYTE, WORD or LONG and determines the number of measurements. For multi-channel measurements, more bytes per measurement are produced accordingly. 10-bit or 12-bit measurements produce 2 bytes per measurement and per channel. When quantity is 0, the measurement continues to the end of the string.
Growth_Flag	is a variable, a constant or an expression of the type BYTE, WORD, or LONG and determines whether the size of the string grows steadily with the measurements or it is set after finishing the measurement. 0: string grows steadily Unequal 0: string size set after measurements.

When the specifications Offset, quantity or Growth_Flag are missing, the settings that have been made with the User-Function-Code before are valid. When the specifications Offset, Quantity or Growth_Flag are present, they are only valid for this measurement and do not influence the general settings.

When the set quantity of measurements leads to exceeding the string size, the measurement is stopped when there is no more space in the string for a further measurement. This can be the case before reaching the maximum string length, when less than 8 bytes are available in a string for a 4-channel 10-bit measurement (8 bytes per measurement), e.g.

The measurement can be aborted with the User-Function-Code UFCO_AD2_STOP.

Program sample:

```
'-------------------------------------------------------------------------
' Name: ANALOG2S.TIG
'-------------------------------------------------------------------------
#INCLUDE DEFINE_A.INC                    ' general definitions
#INCLUDE UFUNC3.INC                      ' User Function Codes
STRING M$ (150)                          ' measurement value string (global!)

TASK MAIN                                ' start task MAIN
' install LCD-driver (BASIC-Tiger)
  INSTALL DEVICE #LCD, "LCD1.TD2"
' install LCD-driver (TINY-Tiger)
'  INSTALL DEVICE #LCD, "LCD1.TD2", 0, 0, 0, 0, 0, 0, 80h, 8
' install TIMER-A driver (time basis timer: 1001Hz)
  INSTALL_DEVICE #TA, "TIMERA.TD2", 3, 156
' install ANALOG-2 driver
  INSTALL_DEVICE #AD2, "ANALOG2.TD2"

  M$=""                                  ' measurement-string empty
  PUT #AD2,#0,#UFCO_AD2_PSCAL, 5         ' pre-scaler: 1001/5=200S/sec
  PUT #AD2,#0,#UFCO_AD2_RESO, 8          ' resolution
  PUT #AD2,#0,#UFCO_AD2_SCAN, 4          ' number of channels
  PUT #AD2,M$,0,300,1                    ' starting pos., no of measurements
                                         ' bigger than string!
  K = 0                                  ' end when string is full
  WHILE K < 148                          ' string does not reach length 150
    K = LEN(M$)                          ' but 4Ch x 37 = 148
    PRINT #LCD, "<1>Length=";K
  ENDWHILE
                                         ' ----------------------------
  FOR I = 0 TO LEN(M$)-4 STEP 4          ' show STRING
    PRINT #LCD, "<1Bh>A<12><0><0F0h>0:";NFROMS(M$,I,1);
    PRINT #LCD, "<1Bh>A<12><1><0F0h>1:";NFROMS(M$,I+1,1);
    PRINT #LCD, "<1Bh>A<12><2><0F0h>2:";NFROMS(M$,I+2,1);
    PRINT #LCD, "<1Bh>A<12><3><0F0h>3:";NFROMS(M$,I+3,1);
    WAIT_DURATION 1000                   ' wait 1 sec.
  NEXT
  PRINT #LCD, "<1Bh>A<0><3><0F0h>ready";
END                                      ' end of task MAIN
```

Measurements with 12-bit

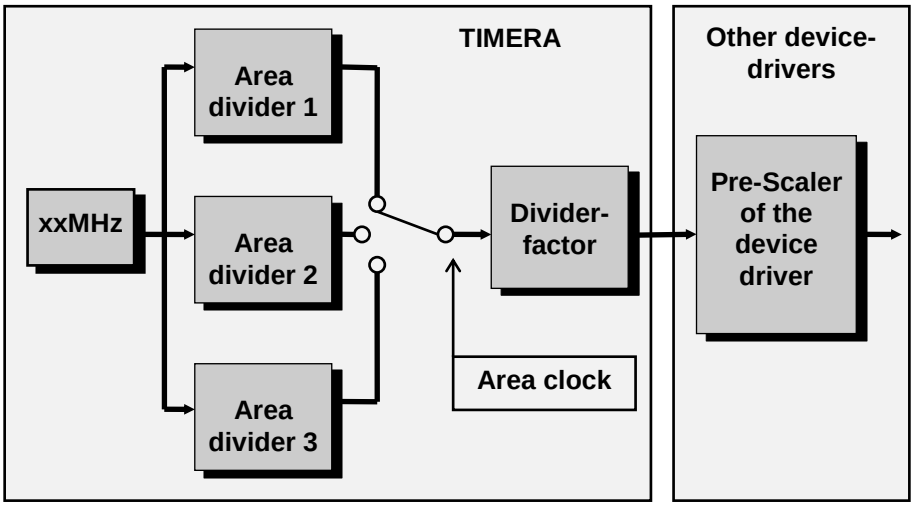
For 12-bit resolution, the integration depth can be set (size of the internal integration buffer). The number of measurements can be reduced, when not every internally produced measurement value is transferred into the string or FIFO buffer. In that way, temporally more separated measurements are made, which are, however, adjusted through a certain integration depth.

```
PUT #7, #0, #UFCO_AD2_RESO, 12   ' set 12-bit resolution
PUT #7, #0, #UFCO_AD2_INTEG, 32 ' 32 bit integration buffer
PUT #7, #0, #UFCO_AD2_ISAMP, 9   ' only every 9th measurement is selected
                                 ' the sample rate is divided by 9
```

The larger the integration buffer, the more accurate the measurement. However, the low pass filter effect becomes larger, i.e. fast signal changes are filtered out.

Setting the sample-rate

The measurement rate or sample-rate is deduced from the basic frequency of the device driver TIMERA. The pre-scaler of the device driver ANALOG2 divides the basic frequency for this purpose:



Further information about setting the device-driver TIMERA can be found in the description of the time base driver. The pre-scaler is set with the User-Function-Code UFCO_AD2_PSCAL. Examples can be found further up under “Measuring in FIFO” and “Measuring in string”.

!

This device driver together with the driver TIMERA can, with a ‘fast’ setting, use up so much of the CPU-performance, that other tasks are hindered very much. With the User-Function-Code UFCI_CPU_LOAD, the CPU load, caused only by this driver, can be queried.

The driver cannot accept certain settings, which would lead to an overloading of the system.

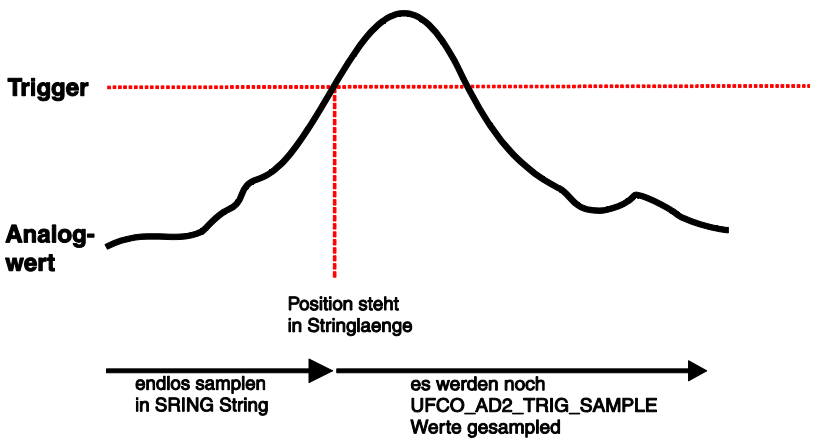
Please note: TIMERA must be installed **before** ANALOG2 can be installed.

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. 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, 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:

```
#INCLUDE DEFINE_A.INC                    ' general definitions
#INCLUDE UFUNC3.INC                      ' user function codes

#define UFCO_AD2_TRIG_SAMPLE    08AH
#define UFCO_AD2_TRIG_HLEV      08BH

#define MLEN    200
#define TLEVEL  700

STRING M$ (MLEN)                          ' measurement value string(global!)

TASK MAIN                                ' start task MAIN
' install LCD-driver (TINY-Tiger)
'  INSTALL DEVICE #LCD, "LCD1.TD2", 0, 0, 0, 0, 0, 0, 80h, 8
' install TIMER-A driver (time basis timer: 1001Hz)
  INSTALL_DEVICE #TA, "TIMERA.TD2", 3, 156
' install ANALOG-2 driver
  INSTALL_DEVICE #AD2, "ANALOG2.TD2"

  word  t1,t2,t3,t4                      ' trigger level

  t1 = TLEVEL                            ' set trigger level
  t2 = TLEVEL                            ' set trigger level
  t3 = TLEVEL                            ' set trigger level
  t4 = TLEVEL                            ' set trigger level

  M$=""                                  ' measurement-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          ' number of channels

  PUT #AD2,#0,#UFCO_AD2_TRIG_SAMPLE, 10           ' samples after trigger
  PUT #AD2,#0,#UFCO_AD2_TRIG_HLEV, t1, t2, t3, t4 ' set trigger for channels

  PUT #AD2,M$                            '

  K = 0FFFFH                             ' init k
  while k =   0FFFFH                     ' wait for trigger
    k = len(M$)                          ' read out trigger flag
  endwhile

  set_len$(M$,MLEN)                      ' measurement completed

END                                      ' end of 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!

RTC1.TD2

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

File name: RTC1.TD2

INSTALL DEVICE #*D*, "RTC1.TD2"

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.

Particularly the calendar function is new, and is much more comfortable than a “pure seconds” RTC, because no conversion has to be made anymore. The seconds are maintained for reasons of compatibility, however. At run-time, the calendar function is faster than the seconds function.

The clock is build into the module and can be buffered through the battery input. This clock keeps running as long as it is buffered.

The file ‘TIMECVT.TIG’ contains sample subroutines that convert the seconds counter into minutes, hours and dates. There are similar subroutines for setting the seconds counter. All sample subroutines concerning the clock in the file ‘TIMECVT.TIG’ assume that the counter has started at 0.00 o’clock of January 1st, 1980, with 0 seconds. You can set any starting point in your system; however, the available subroutines cannot be used anymore.

The alarm function is supported only by the real-time-clock. Setting of the alarm time is done with secondary address 1 or 4. Setting of the alarm time causes the clock to switch the alarm pin to ‘high’ after a short delay. When the alarm time is reached, the real time clock switches the alarm pin of the BASIC-Tiger® module back to ‘low’. In contrast to Tiger 1, where to alarm time can be set in steps of 1 second, the alarm time for Tiger 2 is set in steps of 1 minute to a maximum of 1 month in advance. If the alarm time is to be set with secondary address 1 (in seconds), it is internally rounded down to the last full minute.

Secondary address	Function
0	Setting and reading the time (seconds)
1	Setting the alarm time (seconds)
3	Setting and reading the time (numerical string)
4	Setting the alarm time (numerical string)
5	Reading the time (clear text string)

User-function-codes of the RTC1.TD2

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.

! When comparing the read time with a reference time, always use the phrasing ‘larger’, ‘smaller’, ‘larger or equal’ or ‘smaller or equal’, never use ‘equal’. The clock can occasionally skip a second due to internal corrective factors.

Setting and reading the time in numerical strings

Byte-No.	Description
0	Hours
1	Minutes
2	Seconds
3	Day (of month)
4	Month
5	Year (low byte)
6	Year (high byte)
7	Day of week (Sunday=0, Wednesday=3)

PUT #D, #3, date_*string*

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.

date_string is a constant, variable or an expression of data type STRING and contains the date with time, formatted as in the table above. Byte 7 is a dummy byte; the day of week will be calculated automatically by the device driver since version 1.01d.

! Since RTC1.TD2 1.01d the day of week will be calculated automatically by the device driver. You need to pass 7 Bytes to set the time. Every further byte will be ignored.

GET #D, #3, 0, date_*string*

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.

date_string is a constant, variable or an expression of data type STRING and contains the date with time, formatted as in the table above.

Reading the time as a clear text string

For some applications it is advisable to read out the clock in clear text. Here date and time are already formatted for e.g. the output on an LCD. The formatting is as follows:

"hh:mm:ss:dd:mon:yyyy:dow"

For example:

"17:17:00:13:Jun:2006:Tue"

GET #D, #5, 0, date_*string*

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.
date_string	is a constant, variable or an expression of data type STRING and contains the date with time, formatted in clear text. This string can also be output on the LCD. The string has to be able to hold at least 24 bytes!

Setting the alarm time with numerical string

If one wants to set the alarm time calendrical, it can be set one month in advance. The day of the alarm (might be in this month or in the next) has to be given together with the time. Then the alarm is set off. Minutes are the smallest units of resolution for the alarm in Tiger 2.

Byte-No.	Description
0	Hours
1	Minutes
2	Day (of month)

PUT #D, #4, alarm_*string*

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.

alarm_string is a constant, variable or an expression of data type STRING and contains the time of the alarm formatted according to the table above.

Setting and reading the time in seconds

PUT #D, #0, seconds

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.

seconds is a constant, variable or an expression of data type LONG and contains the number of seconds to which the RTC is to be set.

GET #D, #0, 0, seconds

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.

seconds is a variable of data type LONG and contains the current number of seconds in the RTC.

Setting the alarm time in seconds

If the alarm is to be set in seconds, it can be set one month in advance. The number of seconds is specified. Then the alarm is set off. Minutes are the smallest unit of resolution in the Tiger 2. **Should the seconds not amount to a full minute, it is rounded down to the last full minute!!**

PUT #D, #1, seconds

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.
seconds	is a constant, variable or an expression of data type LONG and contains the time of the alarm in seconds.

Sample program:

```
#define UFCI_RTC_VOLTAGE 162
#define RTC_VOLTAGE_LOW 1


task main
  install_device #0, "LCD1.TD2"
  install_device #1, "RTC1.TD2"
  string DATE$
  long secs
  long voltage




  GET #1, #0, #UFCI_RTC_VOLTAGE,0, voltage      ' get voltage low bit
  IF voltage = RTC_VOLTAGE_LOW THEN
    PRINT #0,"<01>";                            ' cursor to top left
    PRINT #0, "Voltage low"                     ' print to LCD
    PRINT #0, "Set time"                        ' print to LCD

  '               00:00:00   01.Jan.     1980   Wednesday
    PUT #1, #3, "<0h><0h><0h><1h><01H><0BCH><7H><3H>"
  ' set alarm:      00:02   01.(Jan.)
    PUT #1, #4, "<00H><02H><01H>"
  ELSE
    PRINT #0,"<01>";                            ' cursor to top left
    PRINT #0, "No voltage low"                  ' print to LCD
    PRINT #0, "Don't set time"                  ' print to LCD
  ENDIF

  wait_duration 2000

  while 1=1
    GET #1,#5, 0, DATE$                 ' Get date as printable string
    GET #1,#0, 0, secs                  ' Get date in seconds
    PRINT #0,"<01>";                    ' cursor to top left
    PRINT #0, DATE$                     ' print to LCD
    PRINT #0, secs                      ' print to LCD
    wait_duration 500
  endwhile

end
```

MF2_xxxx.TD2 – MF-II PC keyboard

The device-driver ‘MF2_xxxx’ enables the connection of a PC keyboard of type MF-II. External components are, apart from the MF-II keyboard connector, only two resistors.

File name: MF2_84pp.TD2

INSTALL DEVICE #*D*, "MF2_84Pp.TD2"

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.

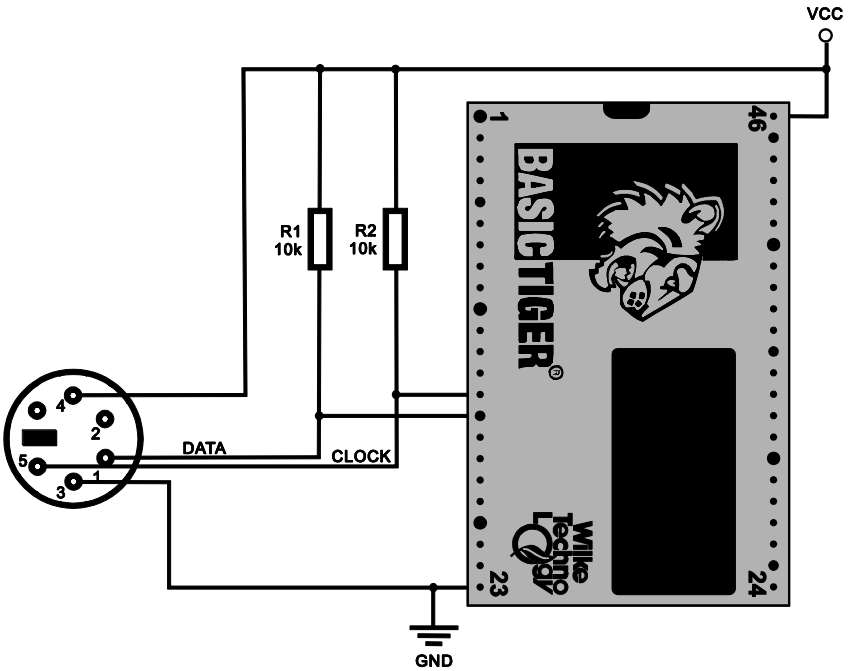
Pp in the file name stands for:
P: internal port
p: pin for data line of the keyboard.

For Tiger 2, the clock line is always on L84 !

The lines for clock and data are provided with pull-up resistors against VCC. The power supply of the keyboard is provided at the keyboard connector. The power requirements are described in the data sheets of the keyboard.

Size and filling level of the input buffer as well as the version of the driver can be queried with the help of the User-Function-Codes.

Example of a connection of a PS/2 MF-II keyboard (view on PS/2 plug):



Since an MF-II keyboard does not send ASCII-codes, but requires further steps of code conversion, quite complex measures are necessary to get the desired keyboard function. As a basis, and in addition to the sample program, several include-files are provided as well, that can be adjusted to individual needs. **MF2_TR.INC** is the only include file to be integrated into the application. **MF2_TR.INC** itself integrates all further include files.

The application calls up subroutines that can be found in the files **MF2_TR.INC, MF2_TR_D.INC**. Here the conversion to ASCII takes place. In the next layer **MF2_PH.INC**, **MF2_PH_D.INC** the physical connection to the driver and thus to the keyboard is made.

The initialization **'InitKeybTables'** is called up once before the usage of the keyboard. The argument is the number of the language (1=English, 2=German, 3=English and German).

The subroutine **'InitKeybDev'** with the device number as argument (WORD) is also called up once. When the driver is embedded several times, then 'InitKeybDev' with the according device number is also called up several times.

The subroutine **'GetAsciiKey'** provides in a WORD:

- when no character: 0000h
- when ASCII character: character in low-byte, scan code in high-byte
- when key with extended code: 0 in low-byte, scan code in high-byte

The subroutine **'CheckKeybFlags'** provides information about the present state of the special keys like STRG, ALT, SHIFT, etc.

Byte 0

Bit 0: right Shift-key pressed
Bit 1: left Shift-key pressed
Bit 2: Strg-key pressed
Bit 3: ALT-key pressed
Bit 4: Scroll-Lock is activated
Bit 5: Num-Lock is activated
Bit 6: Caps-Lock is activated
Bit 7: Insert is activated

Byte 1

Bit 0: left Strg-key pressed
Bit 1: left ALT-key pressed
Bit 2: System-Request is pressed
Bit 3: Pause-key is toggled
Bit 4: Scroll-Lock-key pressed
Bit 5: Num-Lock-key pressed
Bit 6: Caps-Lock-key pressed
Bit 7: Insert-key pressed

Byte 2 (LED-display)

Bit 0: Scroll-Lock-display
Bit 1: Num-Lock-display
Bit 2: Caps-Lock-display
further bits are not used

Byte 3

Bit 0: last code was the 'E1 hidden code'
Bit 1: last code was the 'E0 hidden code'
Bit 2: right Strg-key pressed
Bit 3: right ALT-key pressed
further bits are not used

There are several useful subroutines in **'MF2_PH.INC'**:

Subroutine (arguments)	Function
ResetKbd (WORD wDevId)	RESET Keyboard
SetKbdTypematicRate (WORD wDevId; BYTE bTpRate)	Sets typematic rate of the MF-II keyboard
SetKbdIndicators (WORD wDevId; BYTE bLEDsMask)	Sets the LED's of the MF-II keyboard (bLEDsMask, 0=off, 1=on): Bit 0: Scroll-Lock Bit 1: Num-Lock Bit 2 Caps-Lock
ClearKbdBuffer (WORD wDevId)	Deletes the MF-II keyboard buffer
GetKbdScanCode (WORD wDevId; VAR BYTE bCode)	Takes a character from the keyboard buffer. When the buffer is empty, 'bCode' remains unchanged.
SetKbdScanCodeTable (WORD wDevId; BYTE bTableId)	Sets in 'bTableId' the scan-code table for the keyboard
GetKbdBufferFillSize (WORD wDevId; VAR LONG lBufSize)	Reads the filling level of the keyboard buffer

All subroutines for the MF-II keyboard are re-entrant, i.e. several tasks can use them at the same time.

Remark: The MF-II subroutines are written for the scan code set 1.

The following sample program shows, that the usage of the keyboard has become easy with the provided include files from the user's point of view.

Sample program:

```
'---------------------------------------------------------------------
' Name: MF2_1.TIG
' Shows the usage of an MF-II keyboard at the BASIC-Tiger
'---------------------------------------------------------------------
' connect 4 lines of the keyboard
'   MF-II          Tiger
'     GND  <---->  GND
'     +5V  <---->  Vcc
'   CLOCK  <---->  L80 plus 10...22KOhm --> Vcc
'    DATA  <---->  L81 plus 10...22KOhm --> Vcc
'
'---------------------------------------------------------------------
user_var_strict                  ' unconditional variable .declaration
#include UFUNC3.INC              ' user function codes
#include DEFINE_A.INC            ' general symbol definitions
#include MF2_TR.INC              ' subroutines of the transport layer

WORD wKeybDevId1                 ' keyboard device number
LONG lKeybExtFlags1              ' keyboard flags
BYTE bKeybActLang1               ' keyboard layout (language)

'---------------------------------------------------------------------
TASK Main
  WORD wKey                      ' key (WORD)
  BYTE bIsActive                 '
  LONG lComplexMask              '

' install LCD-driver (BASIC-Tiger)
  INSTALL DEVICE #LCD, "LCD1.TDD"
' install LCD-driver (TINY-Tiger)
'  INSTALL DEVICE #LCD, "LCD1.TDD", 0, 0, 0, 0, 0, 0, 80h, 8

  INSTALL DEVICE #KEYB1, "MF2_8081.TDD" ' L80=clock, L81=data

  wKeybDevId1 = KEYB1            ' initialize keyboard variable
  lKeybExtFlags1 = 0
  bKeybActLang1 = LANG_GERMAN
'  bKeybActLang1 = LANG_ENGLISH

  CALL InitKeybTables( bKeybActLang1 ) ' Init step 1
  CALL InitKeybDev( wKeybDevId1 )      ' Init step 2

  LOOP 9999999                   ' many loops
    ' Read a character from the buffer, translate to ASCII
    CALL GetAsciiKey(wKeybDevId1, lKeybExtFlags1, bKeybActLang1, wKey)
    IF wKey <> 0 THEN            ' when valid character
      PRINT #LCD, CHR$(wKey);    ' display on LCD
    ENDIF
  ENDLOOP
END
```

CAN-Bus

The device driver ‘CAN1_xx.TD2’ supports the internal CAN interface of the TINY-Tiger 2 module.

This section contains:

- Description of the device driver CAN1_xx.TD2
- CAN messages in the I/O-buffer of the driver
- CAN User-Function codes
- Bus timing and transfer rate
- Error register
- Receive filter with Code and Mask
- Sending CAN messages
- Receiving CAN messages
- I/O buffer
- Automatic bit rate detection
- CAN-bus hardware connection example
- A short introduction to CAN
- Error situations
- References to CAN

Description of the device driver CAN1_xx.TD2

This device driver enables input and output on the CAN-bus in connection with the TINY-Tiger 2-module. 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.TD2 (with 8K buffers)
CAN1_K1.TD2 (with 1K buffers)
CAN1_R1.TD2 (with 256 byte buffers)

INSTALL DEVICE #*D*, "CAN1_xx.TD2", "*Code, Mask, Bt0, Bt1, Mod, Oc*"

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.
Standard value: 0

Mask is a parameter to determine the acceptance filter. 'Mask' is always 4 bytes long.
Standard 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.
Standard 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.
Standard value: 2Fh (Tseg1=15, Tseg2=2)

Mod is a parameter to determine the mode (1 byte) .
Standard 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. Standard 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

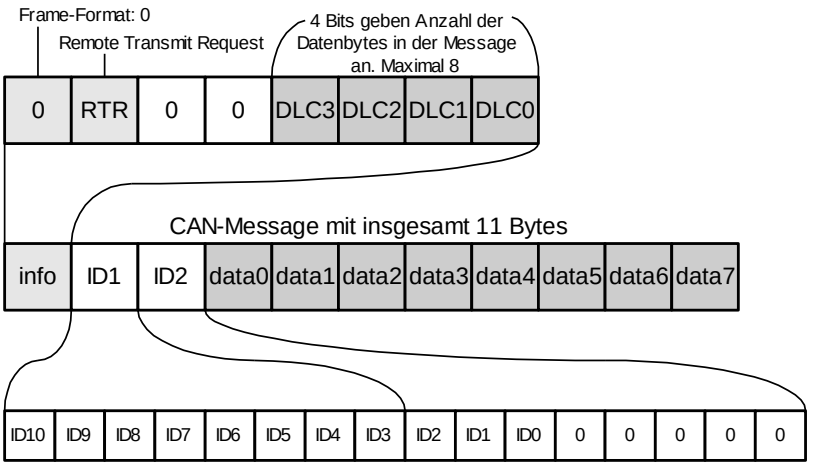
The Identifier can

- 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.

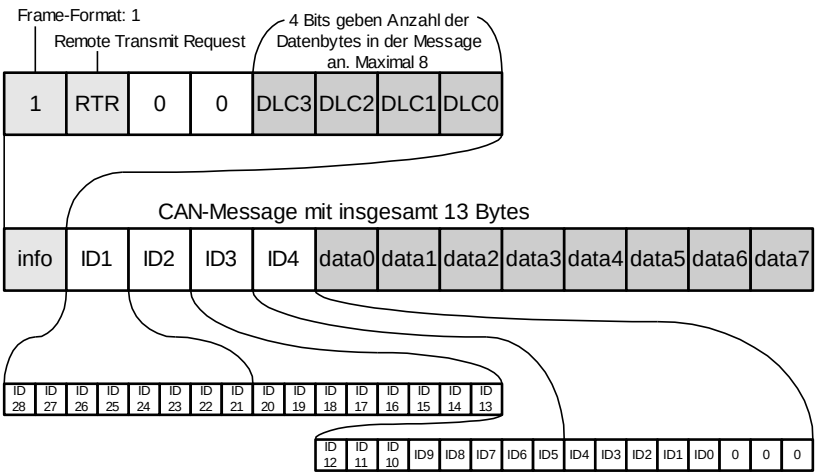
The 11-Bit-Identifier of the CAN message can be found in both ID-bytes, offset by 5 bits to the left. The format here is 'high-byte first', unlike the WORD variables in Tiger-BASIC which are 'low-byte first'.

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 0+1: No. of receive errors Byte 1+2: Buffer overflow count both counters are 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
99	UFCI_DEV_VERS	Driver version

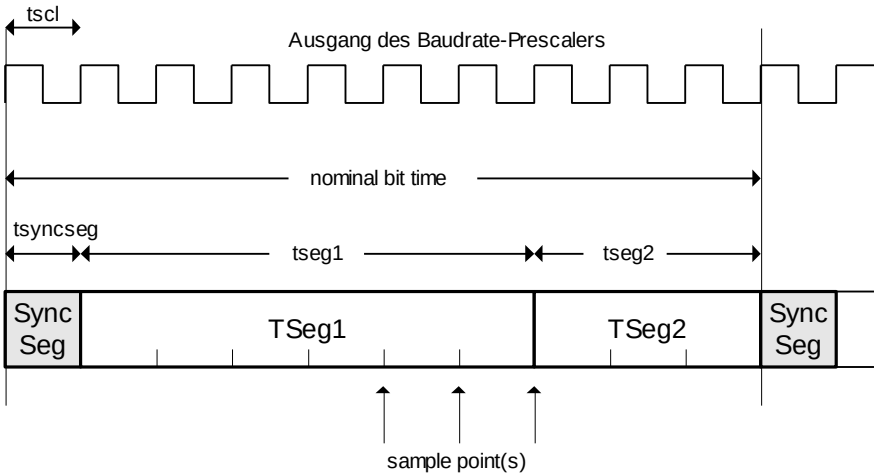
User-Function-Codes for output (Instruction PUT):

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
142	UFCO_CAN_CMD	set CAN register CMD
143	UFCO_CAN_EWL	set CAN error warning limit register
162	UFCO_CAN_LAM	sets local acceptance mask (only channel-16)
176	UFCO_CAN_RESET	CAN soft reset
193	UFCO_CAN_RESRM	set CAN-Chip in RUN mode

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: $t_{..} = 0{,}1 * (BRP+1)$ 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. See also Error-Warning-Limit		
UFCI_CAN_TXERR	0...7	Tx-error counter. counts up with send errors and back down again to 0 if sent correctly. See also Error-Warning-Limit		

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'. This status can only be quit by a hardware reset or software reset.

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:

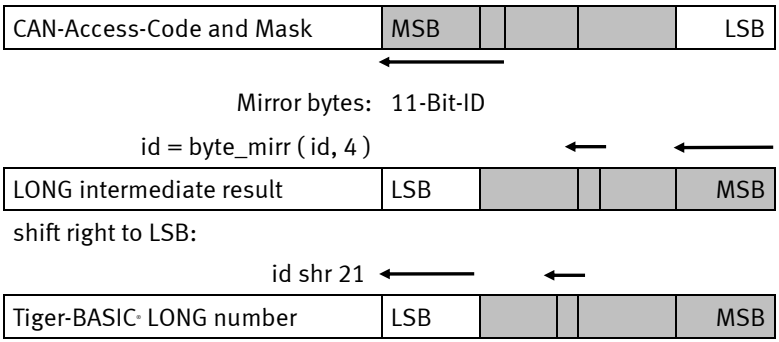
CAN-Access-Code and Mask	MSB			LSB

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'.

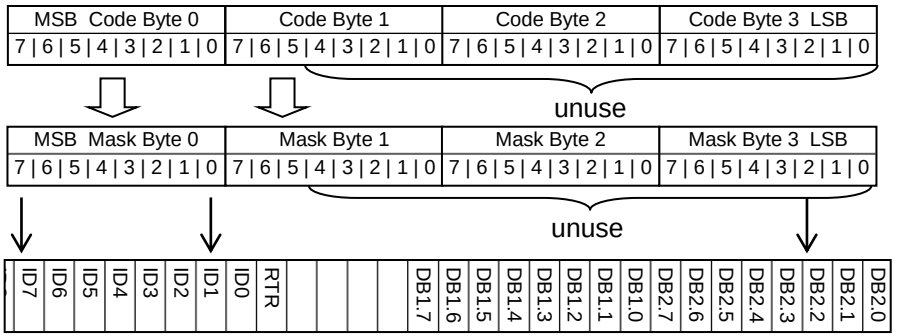
Program example:

```
'-------------------------------------------------------------------
'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 ( (1FFFFFFFh 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

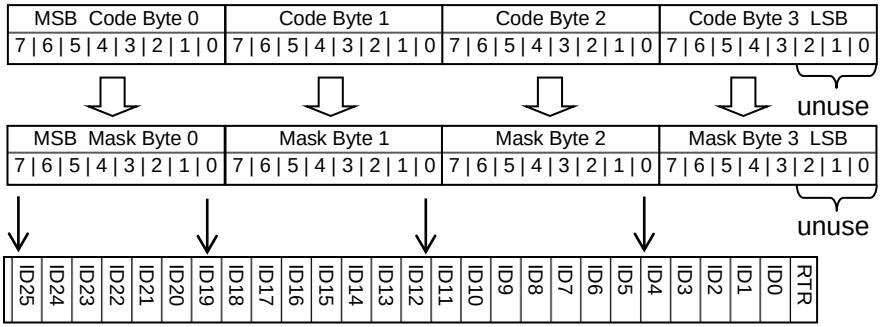
Program example:

```
'-------------------------------------------------------------------
'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	

→ AME

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

→ IDE

	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.

References to CAN

[1] Wolfgang Lawrenz: CAN Controller Area Network, Grundlagen und Praxis. Hüthig Verlag, 1994, ISBN 3-7785-2263-9

[2] Konrad Etschberger: CAN Controller Area Network, Grundlagen, Protokolle, Bausteine, Anwendungen. Verlag Hanser, 1994, ISBN 3-446-17596-2

[3] Bosch CAN Spezifikation Version 2.0 1991

[4] CiA: CAN in Automation e.V. Users Group, Am Weichselgarten 25, D-91058 Erlangen, Germany; Tel: +49 9131 601091, Fax: +49 9131 601092

[5] SJA1000 data book as PDF-file on the Internet: http://www-eu3.semiconductors.com/pip/SJA1000

[6] P82C150 CAN-SLIO data book as PDF-file on the Internet: http://www-eu3.semiconductors.com/pip/P82C150

Extensive additional bibliographical references can be found in the books.

Output pulse with high resolution

The device driver 'PLSOUT1' is able to generate pulses with a resolution of up to 0.4 µsec. The area is determined during installation of the driver. However, the area can also be altered at a later time through commands to the driver.

File name: PLSOUT1.TDD

INSTALL DEVICE #*D*, "PLSOUT1.TDD", *Area*

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 driver.

Area is a parameter to determine the area.

Area	Timers	Resolution	Time area
1	2.500.000 kHz	0.400 µsec	0.0004...26.214 sec
2	625.000 kHz	1.600 µsec	0.0016...104.856 sec
3	156.250 kHz	6.400 µsec	0.0064...419.424 sec

Secondary address 0 selects the channel 0 of the pulse-out-device driver. The possible number of channels depends on the module In BASIC-Tiger® or Tiny-Tiger® modules version 1.0xx only this channel is available. The input pin is always **Pin L86**.

The device driver PLSOUT1 enables a very fast pulse output up to 1.25 MHz hereby uses hardware resources of the BASIC-Tiger® or Tiny-Tiger® module. Since other fast drivers may also need these hardware resources, the simultaneous use of a number of drivers is excluded.

Possible uses of the driver PLSOUT1.TDD together with PLSIN1.TDD in BASIC-Tiger® and Tiny-Tiger® modules

PLSOUT1	PLSIN1
Module Version 1.0xx	
1 channel	—
—	1 channel

Pulse output:

PUT #*D*, #0, *cnt*, *duty*, *cycle*

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 driver.
cnt	is a constant, variable or expression of the data type LONG and specifies the number of pulses to be output.
duty	is a constant, variable or expression of the data type WORD in the range from 0...65535 and specifies the time in units of the set area for which the pulse should be 'low'.
cycle	is a constant, variable or expression of the data type WORD in the range from 0...65535 and specifies the total time of a pulse in units of the set area.

Attention: this device driver needs variables of the above given type: LONG for cnt, and WORD for cycle and duty.

When counting pulses then the current Tiger modules are restricted to the following values of CYCLE and DUTY:
CYCLE, range-1 min. 32, range-2 min. 8, range-3 min. 3
DUTY, range-1 min. 31, range-2 min. 7, range-3 min. 2
Smaller values with COUNT ‹› 0 will cause a runtime error.

User-Function-Codes of PLSOUT1.TDD

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

No	Symbol Prefix: UFCI_	Description
176	UFCI_OPL_STAT	read current Count-Down value 0: last pulse just finishing n: n complete pulses follow -1: already at a standstill 7FFFFFFF: endless output

User-Function-Codes for the device drivers PLSOUT1 for output are defined in the Include-File 'UFUNCn.INC' (instruction PUT):

No	Symbol	Description
144	UFCO_OPL_RNG	set area
145	UFCO_OPL_CNT_ADD	add new number of pulses
146	UFCO_OPL_CNT_SET	set new number of pulses n: generates n pulses 0: soft stop -1: hard stop

Example: (area: 3) output 10 pulses with the cycle time 32µsec (5*6.4µsec) and the Low-Time of 12.8µsec (2* 6.4µsec):

```
PUT #10, #0, 10, 2, 5
```

Example: Set area 2 during the runtime:

```
PUT #10,#0, #UFCO_OPL_RNG, 2
```

Example: read the number of pulses following the pulse which is currently running:

```
GET #10,#0, #UFCI_OPL_STAT, 4, CNT
```

Example: with running, possibly infinite pulse output, set the new number of pulses to 1. The output can thus be aborted, whereby the scanning rate and the frequency are retained, the last pulse is still completely output:

```
PUT #10,#0, #UFCO_OPL_CNT_SET, 1
```

Example: with running, possibly infinite pulse output, set the new number of pulses to 0. This is a soft stop. The current pulse will be finished and then the output is stopped:

```
PUT #10,#0, #UFCO_OPL_CNT_SET, 0
```

Example: with running, possibly infinite pulse output, set the new number of pulses to -1. This is a hard stop. The pulse output is directly stopped and the current pulse will not be finished:

```
PUT #10,#0, #UFCO_OPL_CNT_SET, -1
```

Program example endless:

```
#INCLUDE UFUNC3.INC
#INCLUDE DEFINE_A.INC
LONG no_of_pulses
WORD cycle, duty

TASK MAIN
  install_device #OPL1, "PLSOUT1.TDD", 1        ' Range-1

  no_of_pulses = 0                              ' endless
  cycle = 3                                     ' 1,2 ys
  duty = 1                                      ' 400 ns
  put #OPL1, no_of_pulses, duty, cycle          ' start

END
```

Program example soft stop:

```
user_var_strict
#INCLUDE UFUNC3.INC                              ' User Function Codes
#INCLUDE DEFINE_A.INC                            ' Symbol-Definitions

#define UFCO_OPL_CNT_SET        146
#define UFCO_OPL_CNT_ADD        UFCO_OPL_CNT

LONG no_of_pulses                                ' number of pulses
WORD cycle, duty                                 ' PLSO1-Parameter

TASK MAIN
  BYTE ever                                      ' endlessloop

  install_device #LCD, "LCD1.TD2"                ' LCD driver
  install_device #OPL1, "PLSOUT1.TD2", 3         ' Range 3

  no_of_pulses = 0                               ' endless pulses
  cycle = 200
'                   start, min, max, step, ID
  duty = modulo_updo ( 10,  10, 199,    1,  0 )

  run_task do_po1                                '

  for ever = 0 to 0 step 0                       ' endless loop
    duty = modulo_updo ( duty, 0 )               ' change duty
    print #1, "<1BH>A<0><0><F0H>duty:";duty;"    ";     ' show
    wait_duration 10                             ' wait a bit
  next
END

TASK do_po1
  BYTE ever                                      ' endlessloop
  WORD old_duty, old_cycle                       ' PLSO1-Parameter
  LONG rest

  old_duty = duty                                '
  old_cycle = cycle                              '
  put #OPL1, no_of_pulses, duty, cycle           ' start puls output

  for ever = 0 to 0 step 0                       ' endless loop
    if (duty <> old_duty) or (cycle <> old_cycle) then  '
      old_duty = duty                            ' if changed
      old_cycle = cycle                          ' store current values
      rest = 1                                   ' init
      wait_duration 100                          ' pulse 100ms
      put #OPL1, #0, #UFCO_OPL_CNT_SET, 0        ' soft stop
      while rest > 0                             ' wait for end of pules
        get #OPL1, #0, #UFCI_OPL_STAT, 4, rest   ' rest pulses
      endwhile                                   '
      wait_duration 100                          ' 100ms pause for scope
      put #OPL1, no_of_pulses, duty, cycle       ' set new values
    endif
  next
END
```

Documentation History

Version of Documentation	Description / Changes
010	- CAN device driver
011	- CAN RTR messages
012	- PLSOUT1 - CAN RTR correction
013	- XBUS timing
014	- XBUS timing sample point and delay
015	- Interrupts corrected
016	- RTC: Day of week, Sunday added
017	- LCD-S1D13700 stop ORing alternative layer - RTC: Set Day of Week Tip
018	- LCD-S1D13700 deleted => LCD-S1D13700_xxx.pdf - RTC1: Day of week will be set automatically in numerical string.
019	- CAN: Bitrate example 1MBit Bustiming register changed from 43H to 45H - CAN: UFCI_CAN_EERR - SER1B removed (see SER1B_Updates_xxx.pdf) - CAN: UFCI_CAN_ALC, UFCI_CAN_RMC, UFCI_CAN_ECC removed (not available with Tiger 2)
020	- CAN: UFCI_CAN_ALC, UFCI_CAN_ECC removed from status register table and ECC chapter removed completely (not available in Tiger-2) - CAN TXERR and RXERR: In Bus-Error-Passive mode Bit-7 of the status register will NOT be set.
021	- CAN: OUTCTRL and UFCO_ERRC_RESET removed
022	- Analog-2: Setting of Trigger Level - Analog-2: Number single channels changed
023	- Telephone number changed