Application Note, V 2.1, Sept. 2002

AP32022

TriCore

First steps through the TriCore Interrupt System

Microcontrollers

Never stop thinking.

Revision History:	**2002-09**	V 2.1
Previous Version:	V 2.0	2000-01

Page	Subjects (major changes since last revision)
all	Layout adjustments
all	Layout changed to new template
all	Logo changed to Infineon.
10	Added Section 2.3 Enabling the GPTU by setting the Clock Control Register CLC.
16	Added Section 3.4 CLC - Clock Control Register.

1 Overview

Interrupts are used to respond to asynchronous requests from a certain part of the microcontroller that needs to be serviced. Each peripheral in the TriCore as well as the Bus Control Unit, the Debug Unit, the Peripheral Control Processor (PCP) and the CPU itself can generate an interrupt request.

The interrupt system of the TriCore offers two options to service an interrupt request. An interrupt request can either be serviced by the CPU or by the PCP. These units are called Service Providers. The term service request instead of interrupt request is often used to indicate that an interrupt source can be serviced by different Service Providers.

The TriCore contains two interrupt control units, one for the CPU (ICU), and one for the PCP (PICU). The Interrupt Control Unit manages the interrupt system and performs all the actions necessary to arbitrate incoming interrupt requests, to find the one with the highest priority, and to determine whether to interrupt the service provider or not. These control units, ICU and PICU, handle the interrupt arbitration and the proper communication with the CPU and PCP, respectively.

This ApNote will show how to setup the TriCore registers to perform an interrupt service, and it will focus on the CPU as the basic service provider. The PCP is more sophisticated and needs to be covered separately.

For a CPU interrupt, the entry code for the Interrupt Service Routine (ISR) is a block within a vector of code blocks. Each code block provides an entry for one interrupt source with an assigned priority number, which is programmable. The service routine uses the priority number to determine the location of the entry code block. The prioritization of service routines enables nested interrupts and the use of interrupt priority groups.

Note: You will find additional information about the Interrupt System of the TriCore in the User's Manual.

2 The basic steps to an Interrupt

This chapter contains a guideline on how to setup the registers of the TriCore to perform an interrupt service. This is only one easy solution and it is possible to find better solutions for specific tasks. More information about some sophisticated settings is included in the chapters "Core Registers"and "Interrupt System" of the User’s Manual.

2.1 The general settings

To secure that the Interrupt System is able to work correctly, it is important that the following steps are part of the program. By using a tool from Tasking, Greenhills or High-Tech a startup file like CSTART.ASM or CRT0.TRI will be inluded "automatically" and the following steps should be done. Then you have to warrant only that this startup file is correct and the following initializations are done.

Note: Currently the syntax used by Tasking and Greenhills compilers is a little bit different and for this you will find examples for both tools below.

What’s to do?

- **BIV - set the Base Address of Interrupt Vector Table**

The BIV contains the Base Address of the Interrupt Vector Table. This register gives you the possibility to allocate the interrupt vector table anywhere in the available code memory. Control must be taken regarding the alignment of the address contained in the BIV register.

```
Example Tasking (CSTART.ASM):
.extern   (CODE) _lc_u_int_tab
    :
BIV  .equ   0x2ffffe20             ; BIV register
    :
CONST.D    d0,_lc_u_int_tab        ; const.d is a macro
mtcr #(BIV & 0xffff),d0            ; move value in register BIV
isync                          ; prevent unexpected pipeline side effects


Example Greenhills (CRT0.TRI):
BIV  .equ   0x2ffffe20             ; BIV register
    :
movh d2, %hi(__ghsbegin_interrupts)
addi d2, d2, %lo(__ghsbegin_interrupts)
mtcr (BIV & 0xffff), d2            ; move value in register BIV
isync                          ; prevent unexpected pipeline side effects
```

- **ISP - set the Interrupt Stack Pointer**

The Interrupt Stack Pointer (ISP) helps to prevent Interrupt Service Routines (ISRs) from accessing the private stack areas and possibly interfering with the software managed task's context. An automatic switch to the use of the Interrupt Stack Pointer instead of the private Stack Pointer is implemented in the TriCore architecture.

```
Example Tasking (CSTART.ASM):
.extern    _lc_ue_istack               ; interrupt stack end
    :
ISP  .equ  0x2ffffe28                  ; Interrupt Stack Pointer
    :
CONST.D    d0,_lc_ue_istack            ; const.d is a macro
mtcr  #(ISP & 0xffff), d0              ; initialize interrupt stack pointer
isync
```

```
Example Greenhills (CRT0.TRI):
ISP  .equ  0x2ffffe28
    :
movh d2, %hi(__ghsbegin_intstack)    ; Set Interrupt Stack Pointer
addi d2, d2, %lo(__ghsbegin_intstack)
mtcr (ISP & 0xffff), d2
isync
```

- **Context - you have to do the initialization of the Context list**

The context is subdivided into the upper context and the lower context. The upper context consists of the upper address registers, A10 - A15, and the upper data registers, D8 - D15. These registers are designated to be non-volatile, for purposes of function calling. The upper context also includes the PCXI and PSW registers. The lower context consists of the lower address registers, A2 through A7, the lower data registers, D0 through D7, and the PC.

Both upper and lower contexts include a Link Word. Contexts are saved in fixed-size areas and they are linked together via the link word.

The upper context is saved automatically on interrupts and is restored on returns. The lower context is saved and restored explicitly by the Interrupt Service Routine (ISR) if the ISR needs to use more registers than provided by the upper context.

```
Example Tasking (CSTART.ASM):
.extern _lc_ub_csa                ; context save area begin
.extern _lc_ue_csa                ; context save area end
    :
PCXI  .equ  0x2ffffe00            ; Previous Context Information
```

```
FCX  .equ  0x2ffffe38         ; Free CSA List Head Pointer
LCX  .equ  0x2ffffe3c         ; Free CSA List Limit Pointer
    :
;--- clear PCX field (incl. PCXS) in PCXI for CrossView’s stack trace:
mfcr  d0,#PCXI
movh  d1,#0xfff0
and16    d0,d1
mtcr #(PCXI & 0xffff),d0
isync

;--- setup context lists:
     CONST.A a3,_lc_ub_csa    ; const.a is a macro
     mov16.d d0,a3
     extr.u  d0,d0,#28,#4     ; extract segment number
     sh   d0,d0,#16           ; D0 = shifted segment number
     mov16   d1,#0
     mov16.d d2,a3            ; D2 = previous CSA
     st.w [a3+]64,d1          ; store last null pointer
     mov16.d d1,a3
     extr.u  d1,d1,#6,#16     ; get CSA index
     or16 d1,d0               ; add segment number
     mtcr #(FCX & 0xffff),d1  ; initialize LCX
     isync
     CONST.A a5,(CSA-2)       ; A5 = loop counter
loop:
     extr.u d1,d2,#6,#16      ; get CSA index
     or16 d1,d0               ; add segment number
     mov16.d d2,a3
     st.w [a3+]64,d1          ; store "next" pointer
     loop16   a5,loop
     extr.u   d1,d2,#6,#16    ; get CSA index
     or16 d1,d0               ; add segment number
     mtcr #(FCX & 0xffff),d1  ; initialize FCX
     isync

     mov16   d4,#0            ; argc = 0
     movz16.a a4              ; argv = 0

;*********************************
```

```
Example Greenhills (CRT0.TRI):
CTXCOUNT  .equ  64            ; Number of contexts
PCXI   .equ  0x2ffffe00
FCX    .equ  0x2ffffe38
LCX    .equ  0x2ffffe3c
    :
```

```
; Set PCXI
    mov16 d2, 0
    mtcr  (PCXI & 0xffff), d2
    isync

; Initialize context areas
    movh   d2, %hi(__ghsbegin_contexts)
    addi   d2, d2, %lo(__ghsbegin_contexts)
    extr.u d3, d2, 6, 16
    sh     d4, d2, -12
    movh   d5, 0x000f
    and16  d4, d5
    or16   d3, d4
    add       d4, d3, 1
    mov.a  a3, d2
    lea       a4, CTXCOUNT-1
initcsa:
    st.w   [a3+]64, d4
    add16  d4, 1
    loop16 a4, initcsa
    add16  d4, -1

    mtcr  (FCX & 0xffff), d3
    isync
    mtcr  (LCX & 0xffff), d4
    isync
; Protection registers
; Priority Numbers in SRNodes
```

2.2 Service request specific Individual settings

The next step is the initialization of some service request specific registers. This step is not a part of the startup file and so you have to add this in any case.

- **xxSRC - set the Service Request Control Register**

An interrupt or service request from a module connects to a Service Request Node (SRN). Each Service Request Node contains a Service Request Control Register (xxSRC) and the necessary logic for the communication with the requesting source and the two interrupt arbitration busses. To do the individual configuration of an interrupt you have to setup the Service Request Control Register.

All Service Request Control Registers in the TriCore, have the same format and they hold the individual control bits to enable/disable the request, to assign a priority number and to direct the request to one of the two service providers. A request status bit shows whether the request is active or not.

Besides being activated by the associated module through hardware, each request can also be set or reset through software via two additional control bits.

Note: The full generic format and description of a service request control register, xxSRC ('xx' refers to the requesting source), is given in the section "Appendix" further below with a detailed description of each bit and bit field.

- **enable/disable the request**

The bit xxSRE (= xxSRC[12]) controls if a service request is allowed or not. Set this bit to **0 to disable** or set it to **1 to enable the service request**.

- **direct the request to one of the two service providers**

With these two bits xxTOS (= xxSRC[11:10]) you can set either the CPU or the PCP as the Service Provider. Set these two bits to **0b00 for CPU** or set them to **0b01 for PCP**. Depending on the value in xxTOS, now you have to setup the register ICR for the CPU or the register PCPICR for the PCP. These registers contain the same bits. For this reason we will explain only the modification of ICR in detail.

- **assign a priority number**

The 8-bit Service Request Priority Number xxSRPN (= xxSRC[7:0]) of each Interrupt Request Control Register must be either equal to 0x00, which means that this service request is never serviced, or equal to a value **between 0x01 (lowest priority) and 0xFF (highest priority)**.

The Interrupt with the **highest priority will be served first**. That means that even a running interrupt can be intermitted if an interrupt with a higher priority appears (ICR.IE must be enabled again). In this case the interrupt with the lower priority will be discontinued until the interrupt with the higher priority has been served.

A clever and reasonable setup of the priority can increase the speed of an Interrupt Service. By using minor priorities for the interrupt routines, it is possible to save cycles for the arbitration. If an interrupt handler is very short, it may fit entirely within the 8 words available in the vector code segment. Otherwise, the code stored at the entry location can either span several vector entries or should contain some initial instructions, followed by a jump to the rest of the handler.

Note: See the User's Manual chapter "The Interrupt Vector Table" for hints on how to set this priority number.

```
Example Tasking and Greenhills:
                                    // GPTU1 Service Request Control Register 0
GPTU1_GTSRC0 = 0x00001001;            // enable service request,
                                     // set CPU as service provider, priority 1
```

- **ICR - set the Interrupt Control Register (CPU)**

The Interrupt Control Unit contains an Interrupt Control Register (ICR), which holds the current CPU priority number (CCPN), the global interrupt enable/disable bit (IE), the pending interrupt priority number PIPN, as well as two bits to control the required number of interrupt arbitration cycles. See the chapter "Appendix" for a detailed description of each bit and bit field.

- **CPU priority number (CCPN)**

The CPU priority **should normally be at 0x00**, because it makes no sense to set this to a higher priority (in a normal program). But it is possible to set the priority to another value for test purpose.

- **the global interrupt enable/disable bit (IE)**

The bit IRC.IE indicates the enable/disable state of the Interrupt System. For **IRC.IE=1(0)** the global Interrupt System is **enabled(disabled)**.

- **specific bits to control the interrupt arbitration cycles (CONECYC, CARBCYC)**

The value for the arbitration cycles **CARBCYC** (=ICR[25:24]) depends on the setting of the interrupt priorities. This field controls the number of arbitration cycles used to determine the request with the highest priority. The following Table gives an overview on the options for the arbitration cycle control.

Number of Arbitration Cycles	4	3	2	1
Coding for bit field CARBCYC	**0b00**	**0b01**	**0b10**	**0b11**
Relevant bits on the SRPNs	7:0	5:0	3:0	1:0
Range of priority numbers	**1..255**	**1..63**	**1..15**	**1..3**

Another way to increase the speed of an interrupt service is the correct setting of the bit **CONECYC** (=IRC[26]). This bit determines the number of clocks per arbitration cycle. Setting this bit to one can only be used with lower system frequencies. Please refer to the TC10GP Data Sheet for the exact limit frequency.

*Note: Below you can see an example on setup of the register ICR. Note that you need to use **mtcr to write and mfcr to read from a core register**. After a **mtcr** instruction it is recommended to use an **isync** instruction (synchronize instruction stream) in order to avoid unexpected pipeline side effects.*

```
Example Tasking and Greenhills:
// set ICR to ->  0x0 7 000 1 00
// CONECYC to 0b01 (1 clock per arbitration cycle), set CPU priority to 0
// CARBCYC to 0b11 for priority 1…3, enable the interrupt system
#define  ICR  0x2FFFFE2C     // Interrupt Control Register of the CPU
// For Greenhills: enable the options: allow #pragma asm,
// allow // style comments, Japanese Automotive C, -Xintvectbyentry
_mtcr((ICR & 0xFFFF),((_mfcr(ICR & 0xFFFF) & 0xF0FFF000) | 0x07000100));
_isync();  // to synchronize instruction stream
// Note: This syntax is Tasking specific,
// but you can change it very easily in to the Greenhills syntax
// using two underlines "__" instead of one "_" before intrinsics
// like ISYNC !
```

2.3 Enabling the GPTU by setting the Clock Register CLC

A number of on-chip modules of the TriCore, including the peripherals, the EBU and the PCP each have clock-control registers, CLC. The CLC register provides overall clock control for the GPTU and needs to be configured at the beginning. At startup, the default state of the peripherals, including the Timer, is disabled. Write operations to the registers of disabled modules are not allowed. However, the clock-control register CLC of a disabled module unit can be written. An attempt to write to any of the other writable registers of a disabled module besides CLC will cause the Bus Control Unit (BCU) to generate a bus error interrupt request signal.

The GPTU has an 8-bit control field in the CLC register for Run Mode clock control (CLC_RMC). The clock divider circuit in the timer consists of an 8-bit down-counter with an 8-bit reload value. When the timer is not disabled, the reload value is taken from the contents of CLC_RMC. A value of 0 in CLC_RMC disables the clock signals to the unit.

If RMC is not equal to 0, the Run Mode clock for a unit is

$$f_{GPTU} = f_{SYSTEM} / <RMC_{GPTU}>$$

Where <RMC> is the contents of the CLC_RMC field with a range of 1..255.

In this case, set the RMC bit field (GPTUCLC[15:8]) to '0x01', making the timer frequency the same as the system frequency.

Note: This CLC register is protected by the ENDINIT bit in the WDTCON0 register, so special code needs to be used before the CLC register can be written to. Refer to ApNotes AP3221 "What is the ENDINIT bit and how to handle it" and AP3219 "How to use the Watchdog Timer of the TriCore" for more information. Also refer to the software example in section 2.5 for the unlock_wdtcon() and lock_wdtcon() functions.

```
unlock_wdtcon();                // password access, clearing ENDINIT bit
GPTU1_CLC = 0x00000100;          // enable GPTUx module
lock_wdtcon();                   // password access, setting ENDINIT bit
```

2.4 Software Interrupt

Now, after the previous steps the interrupt system should be configured correctly. Combine the parts and add to them an easy interrupt function where a counter will be incremented and a complete interrupt program is done . To trigger an interrupt in this program we are setting the interrupt request bit (xxSETR) by software.

We have now this small example code and it will work correctly if you followed the previous instructions. The example below includes a lightly different part then described above. It uses the BISR instruction to enable the Interrupt System and to set the CPU priority, because it is not always necessary to change the ICR by the MTCR instructions.

```
Example Tasking and Greenhills (common part):
//please see for Greenhills the notes in the chapter before

#define  ICR    0x2FFFFE2C        // Interrupt Control Register of the CPU
#define  GPTU1_GTSRC0  *((volatile unsigned int*) 0xF00007FC)    //GPTSRC0
#define GPTU1_CLC    *((volatile uint*)0xF0000700)  // Clock Control Reg.
#define WDTCON0 0xF0000020           // Watchdog Control Register
#define WDTCON1 0xF0000024           // Watchdog Control Register 1

int isrCounter;                      // variable for counting interrupts

void unlock_wdtcon(void)             // unlock function watchdog
{
 uint wcon0, wcon1;

   wcon0 = *(uint *)(WDTCON0);
   wcon1 = *(uint *)(WDTCON1);
   wcon0 &= 0xFFFFFF03;
   wcon0 |= 0xF0;
   wcon0 |= (wcon1 & 0xC);
   wcon0 ^= 0x2;
   *(uint *)(WDTCON0) = wcon0;       // modify access to WDTCON0
   wcon0 &= 0xFFFFFFF0;
   wcon0 |= 2;
   *(uint *)(WDTCON0) = wcon0;
}
```

```
void lock_wdtcon(void)                    // lock function watchdog
{
   uint wcon0, wcon1;

   wcon0 = *(uint *)(WDTCON0);
   wcon1 = *(uint *)(WDTCON1);
   wcon0 &= 0xFFFFFF03;
   wcon0 |= 0xF0;
   wcon0 |= (wcon1 & 0xC);
   wcon0 ^= 0x2;
   *(uint *)(WDTCON0) = wcon0;            // modify access to WDTCON0
   wcon0 &= 0xFFFFFFF0;
   wcon0 |= 3;
   *(uint *)(WDTCON0) = wcon0;
}

void main()
{
unlock_wdtcon();            // password access, clearing ENDINIT bit
GPTU1_CLC = 0x00000100;     // enable GPTUx module
lock_wdtcon();              // password access, setting ENDINIT bit
GPTU1_GTSRC0 = 0x00001001; // GTSRC0: priority 1, enabled, CPU service

_bisr(0x00);        // enable Interrupt System and set CCPN to 0x00
_isync();           // to synchronize instruction stream

while(1)
   {
    GPTU1_GTSRC0 |= 0x00008000;       // interrupt by software (xxSETR)
   }
}

Tasking specific:
void _interrupt(1) isr_count(void)    // interrupt routine for priority 1
{
   isrCounter++;                      // increment counter
}
```

```
Greenhills specific:
//please see for Greenhills the notes in the chapter before

#pragma intvect isr_counter #0x01    //init interrupt vector table

__interrupt void isr_counter(void)   // Example interrupt routine
{
   isrCounter++;                     // increment counter
}
// Note: This syntax is Tasking specific,
// but you can change it very easily in to the Greenhills syntax
// using two underlines "__" instead of one "_" before intrinsics
// like ISYNC !
```

3 Appendix

3.1 xxSRC - Source xx Service Request Control Register

Reset value: 0000’0000$_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
xx SETR	**xx CLRR**	**xx SRR**	**xx SRE**	**xxTOS**		**0**	**0**	**xxSRPN**							
w	w	rh	rw	rw				rw							

Field	Bit	Typ	Description
xxSETR	xxSRC[15]	w	**Source xx Request Set Bit** 0 No action. 1 Set xxSRR (no action if xxCLRR = 1). Written value is not stored. Read returns 0.
xxCLRR	xxSRC[14]	w	Source xx Request Clear Bit: 0 No action. 1 Clear xxSRR (no action if xxSETR = 1). Written value is not stored. Read returns 0.
xxSRR	xxSRC[13]	rh	Source xx Service Request Flag: 0 No service request pending. 1 A service request is pending.
xxSRE	xxSRC[12]	rw	Source xx Service Request Enable Control. 0 Service request is disabled. 1 Service request is enabled.
xxTOS	xxSRC[11:10]	rw	Source xx Type-of-Service Control: 00 Request CPU service (Service Provider 0). 01 Request PCP service (Service Provider 1). TOS[1] is read-only. Read returns 0.
xxSRPN	xxSRC[7:0]	rw	Source xx Service Request Priority Number. 0x00 A service request on this priority is never serviced. 0x01 Lowest priority number. 0xFF Highest priority number.
0		r	These bit positions are read-only, returning 0 when read. Writing to these bit positions has no effect. These positions are reserved for future extensions, and it is advised to always write a 0 to these bit positions when writing to the register in order to preserve compatibility with future derivatives.

3.2 ICR - Interrupt Control Register

Reset value: 0000’0000$_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	**0**	**0**	**0**	**0**	**C ONE CYC**	**CARBCYC**		**PIPN**							
					rw	rw		rh							

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	**0**	**0**	**0**	**0**	**0**	**0**	**IE**	**CCPN**							
							rwh	rwh							

Field	Bit	Typ	Description
CONECYC	ICR[26]	rw	Clocks per Arbitration Cycle Control: 0 Two clocks per arbitration cycle (default). 1 One clock per arbitration cycle.
CARBCYC	ICR[25:24]	rw	Number of Arbitration Cycles Control: 00 Four arbitration cycles (default). 01 Three arbitration cycles. 10 Two arbitration cycles. 11 One arbitration cycle.
PIPN	ICR[23:16]	rh	Pending Interrupt Priority Number: 0x00 No valid pending request. 0xYY A request with priority YY is pending.
IE	ICR[8]	rwh	Global Interrupt Enable Bit: 0 Interrupt system is globally disabled. 1 Interrupt system is globally enabled.
CCPN	ICR[7:0]	rwh	Current CPU Priority Number
0		r	These bit positions are read-only, returning 0 when read. Writing to these bit positions has no effect. These positions are reserved for future extensions, and it is advised to always write a 0 to these bit positions when writing to the register in order to preserve compatibility with future derivatives.

3.3 Service Request Sources

Module	Description	Control Register
GPTUx	GPTUx Service Request Nodes 0	GTSRC0
GPTUx	GPTUx Service Request Nodes 1	GTSRC1
GPTUx	GPTUx Service Request Nodes 2	GTSRC2
GPTUx	GPTUx Service Request Nodes 3	GTSRC3
GPTUx	GPTUx Service Request Nodes 4	GTSRC4
GPTUx	GPTUx Service Request Nodes 5	GTSRC5
GPTUx	GPTUx Service Request Nodes 6	GTSRC6
GPTUx	GPTUx Service Request Nodes 7	GTSRC7
ASC0	ASC0 Transmit Buffer Service Request Node	S0TBSRC
ASC0	ASC0 Receive Service Request Node	S0RSRC
ASC0	ASC0 Transmit Service Request Node	S0TSRC
ASC0	ASC0 Error Service Request Node	S0ESRC
ASC1	ASC1 Transmit Service Request Node	S1TSRC
ASC1	ASC1 Transmit Buffer Service Request Node	S1TBSRC
ASC1	ASC1 Receive Service Request Node	S1RSRC
ASC1	ASC1 Error Service Request Node	S1ESRC
SSC0	SSC0 Transmit Service Request Node	SC0TSRC
SSC0	SSC0 Receive Service Request Node	SC0RSRC
SSC0	SSC0 Error Service Request Node	SC0ESRC
BCU	BCU Error Service Request Node	BCUSRC
Debug	Software Breakpoint Service Request Node	SBSRC0
CPU	CPU Software Service Request Node 0	CPUSRC0

3.4 CLC - Clock Control Register

Reset value: 0000’0002$_H$

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RMC								**0**	**0**	**FS OE**	**SB WE**	**EDIS**	**SP EN**	**DISS**	**DISR**
rw										rw	w	rw	rw	rh	rw

Field	Bit	Typ	Description
RMC	[15:8]	rw	GPTU Clock Divider for Normal Mode
FSOE	[5]	rw	GPTU Fast Shut-Off Enable
SBWE	[4]	w	GPTU Suspend Bit Write Enable
EDIS	[3]	rw	GPTU External Request Disable
SPEN	[2]	rw	GPTU Suspend Enable Bit
DISS	[1]	rh	GPTU Disable Status Bit
DISR	[0]	rw	GPTU Disable Request Bit
0		r	Unimplemented, reserved.

Infineon goes for Business Excellence

“Business excellence means intelligent approaches and clearly defined processes, which are both constantly under review and ultimately lead to good operating results.
Better operating results and business excellence mean less idleness and wastefulness for all of us, more professional success, more accurate information, a better overview and, thereby, less frustration and more satisfaction.”

Dr. Ulrich Schumacher

http://www.infineon.com

Published by Infineon Technologies AG