Application Note, V 2.1, Sept. 2002

AP32025

TriCore PCP

Smart Interrupt Service via PCP

Microcontrollers

Never stop thinking.

Revision History:	2002-09	**V 2.1**
Previous Version:	2001-10	V 2.0

Page	Subjects (major changes since last revision)
All	Adjustments to Infineon template
9, 14, 16	Corrected bit number in R7 for channel enable to bit 6
All	Adaption from TC10GP to TC1775 (deleted not implemented register, register addresses adapted.)

We Listen to Your Comments

Any information within this document that you feel is wrong, unclear or missing at all? Your feedback will help us to continuously improve the quality of this document. Please send your proposal (including a reference to this document) to:
mcdocu.comments@infineon.com

Edition 2002-09

Published by
Infineon Technologies AG
81726 München, Germany

LEGAL DISCLAIMER

THE INFORMATION GIVEN IN THIS APPLICATION NOTE IS GIVEN AS A HINT FOR THE IMPLEMENTATION OF THE INFINEON TECHNOLOGIES COMPONENT ONLY AND SHALL NOT BE REGARDED AS ANY DESCRIPTION OR WARRANTY OF A CERTAIN FUNCTIONALITY, CONDITION OR QUALITY OF THE INFINEON TECHNOLOGIES COMPONENT. THE RECIPIENT OF THIS APPLICATION NOTE MUST VERIFY ANY FUNCTION DESCRIBED HEREIN IN THE REAL APPLICATION. INFINEON TECHNOLOGIES HEREBY DISCLAIMS ANY AND ALL WARRANTIES AND LIABILITIES OF ANY KIND (INCLUDING WITHOUT LIMITATION WARRANTIES OF NON-INFRINGEMENT OF INTELLECTUAL PROPERTY RIGHTS OF ANY THIRD PARTY) WITH RESPECT TO ANY AND ALL INFORMATION GIVEN IN THIS APPLICATION NOTE.

Information

For further information on technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies Office (**www.infineon.com**).

Warnings

Due to technical requirements components may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies Office.

Infineon Technologies Components may only be used in life-support devices or systems with the express written approval of Infineon Technologies, if a failure of such components can reasonably be expected to cause the failure of that life-support device or system, or to affect the safety or effectiveness of that device or system. Life support devices or systems are intended to be implanted in the human body, or to support and/or maintain and sustain and/or protect human life. If they fail, it is reasonable to assume that the health of the user or other persons may be endangered.

Table of Contents **Page**

1 Introduction

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 Peripheral Control Processor, PCP. These units are called Service Providers.

The Peripheral Control Processor PCP performs most of the tasks that are normally done by a DMA controller and CPU interrupt service routines. It could be easily considered the host processor's first line of defense as an interrupt handler engine. The PCP off-loads the CPU from most of the time critical interrupts, easing the implementation of systems based on operating systems.

The first part of this ApNote will give a basic understanding of the PCP architecture. This contains the two different ways to use the Code Memory, the 8 General Purpose Registers and the three different context settings. Later in the second part, this concept will be explained and shown on different examples based on the TC1775.

Note: This ApNote includes general information about the PCP and self-explanatory examples, but there is further information about the PCP in the "User's Manual" of the TriCore.

2 The PCP Structure

The PCP has a “Channel Program” associated with each interrupt number. This could be thought of as the interrupt routine for a given interrupt source. When an interrupt of number “n” is received by the PCP Core, it restores the context associated with number “n” from the PRAM, and begins executing Channel Program “n” from the Code Memory until it encounters a terminating condition - usually an EXIT instruction. At that point it saves the current context for number “n” back into the PRAM. If there is a new pending interrupt it starts this process again. If there is no new pending interrupt, the PCP stops until there is a new interrupt to process.

2.1 Code Memory Organization

There are two ways the Code Memory may be configured:

One mode is configured by setting the configuration register bit PCPCS.RCB = 1. This forces the PCP Core to begin each Channel Program from a known fixed point in Code Memory that is related to the interrupt number. This is like a traditional interrupt vector table. At the entry point there will typically be a jump to the rest of the Channel’s code, because there is only the limited space for two instructions (2x16-bit) in this table. Another possibility is to have only one instruction and then an EXIT instruction in this table (ex.: COPY then EXIT)

When PCPCS.RCB = 1 (True), the Channel Entry Table address for a specific Channel is:

- From an FPI[1] Master: Code Base Address + 04h x Channel Number (SRPN[2])
- PCP Core PC Address: 02h x Channel Number (SRPN)

The other mode (PCPCS.RCP = 0) allows the PCP Core to begin execution at the PC address restored as part of the Channel Program Context (R7 [31:16]). This mode allows for code to be contiguous and start at any arbitrary address. It also allows the implementation of interrupt driven state machines, and even the sharing of code across multiple programs with different context.

In this mode the Code Memory does not have to be partitioned; only the value of R7 [31:16] must refer to the correct start PC. The location of code for specific channels is arbitrary within the PCODE.

Note: The interrupt system does not support a Channel number 0 (zero), so entry 0 is never used.

[1] Flexible Peripheral Interconnection
[2] Service Request Priority Number

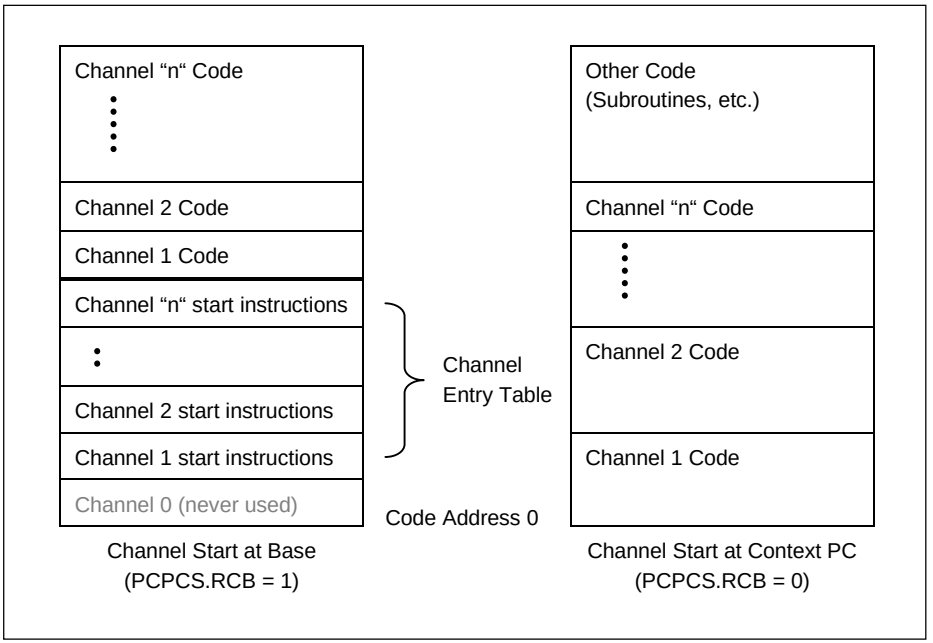

Figure 1 Code Memory Organization

2.2 PCP Parameter RAM (PRAM) Organization

The PRAM is used to store the context for each Channel as well as general data. It is also an area the PCP and the host processor or other FPI bus masters can use to communicate and share data.

In order to accommodate different system needs, there are three possible context sizes. These are static configurations and once the PCP is configured, it always uses that size. The difference between them is equal to the number of the General Registers that are saved/restored, and consequently equal to the number of registers that should be considered “global” or volatile. This is based on a global configuration.

The three methods are Full Context Save, Small Context Save, and Minimum Context Save. A Full Context is defined as the Save/Restore of the registers R0 - R7. Small Context is made up of registers R4 - R7. Registers R0 - R3 are still available for use, but their stability from one interrupt to the next is not guaranteed. This Context size still allows the complete operation of a DMA Channel to continue properly. Minimum Context is the Save/Restore of R6 - R7.

Note: In the case of Small and Minimum Contexts, the unsaved/unrestored registers can be thought of as "globals" that any channel can use, change or leave as constants - for example base address pointers.

While a portion of the PRAM beginning at local address zero (0) is always implicitly used for the context save areas of the channel programs, the remaining area can be used for channel-specific or general data storage. A programmable 8-bit data pointer (DPTR), concatenated with a 6-bit offset, is provided for arbitrary access to the PRAM. Every channel has its own data pointer DPTR, which is very much like a segment register. It points to a 64-word base address and it is possible to make different channels have separate, non-overlapping data areas, or they can share data areas. The locations of general parameters in memory are arbitrary and are determined by the programmer. A Channel Program may also dynamically change DTPR so that it can access large amounts of memory.

As in the Code Memory address, it may be difficult at times to determine exactly where a given word is located in the address space. The PCP Core addresses the PRAM as 32-bit words. There is no concept of half-word or byte accesses to PRAM. However, from the FPI bus a master must use byte addresses in order to access memory. This means the address for the same piece of data is:

- PCP Address: "n" (32 bit word)
- From an FPI Master: PRAM Base Address + (04h x "n")

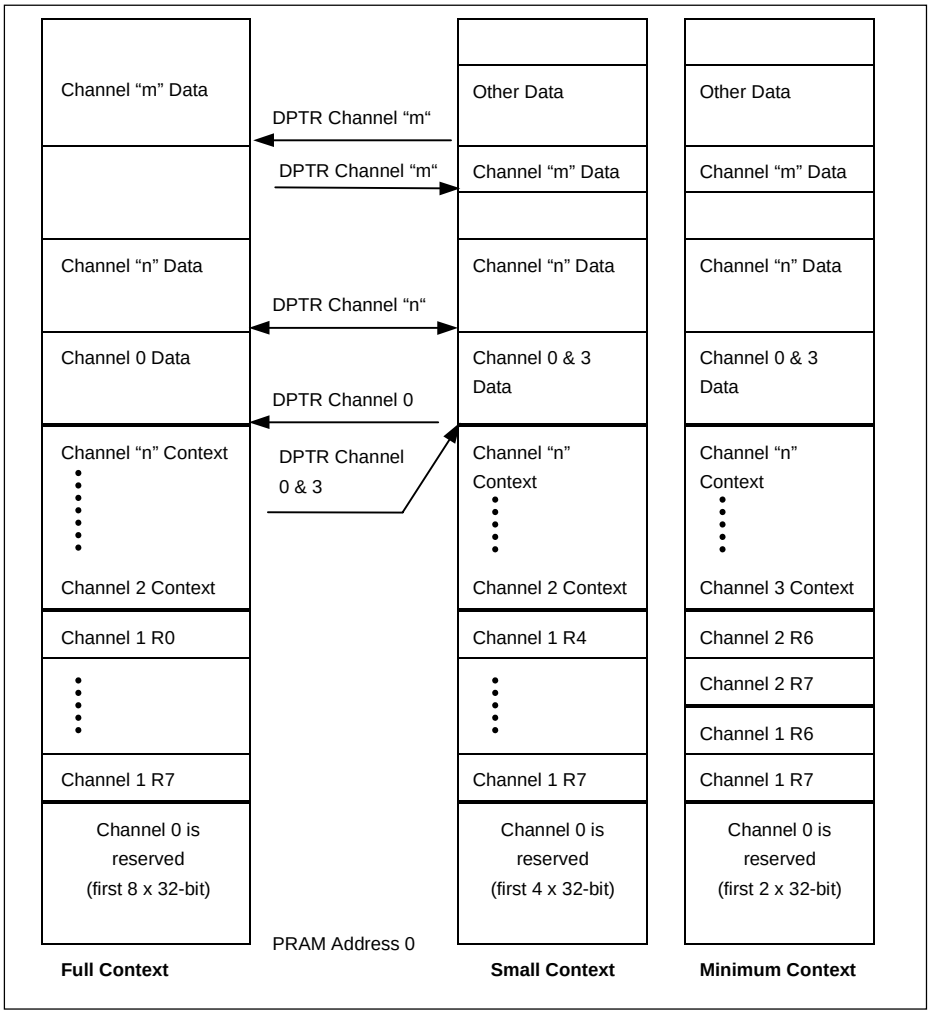

Figure 2 PRAM Organization

Note: The interrupt system does not support a Channel number 0, so entry 0 is never used.

2.3 Context Registers

The program-accessible register file of the PCP is composed of eight General Purpose Registers (GPRs). These registers are all accessible by PCP programs directly as part of the PCP instruction set. Source and destination registers must be specified in most instructions. Of the eight registers, register R7 is the only one that may not be used as a GPR.

Table 1 The General Purpose Registers

Register	Implicit Use	Description
R0	Accumulator	Implicit target for some arithmetic and logical instructions
R1		32 bit general-use register
R2	Return Address	32 bit general-use register
R3		32 bit general-use register
R4	SRC (Source)	Source Pointer for COPY instruction
R5	DST (Destination)	Destination Pointer for COPY instruction

R0 is often used as the destination for instructions that do not have enough fields for complete destination description. These are called out in the individual instruction descriptions.

R1 - R3 are general-purpose registers. By convention, it is suggested to avail R2 as “return address” register when call/return structures are used.

R4 and R5 are general-purpose registers. However, when the COPY instruction is used, it accesses R4 as the source address, which is a full 32-bit address. R5 is used as the destination address. COPY utilizes these registers to maintain the address pointers. As a result, both values may or may not be updated by the COPY instruction depending on the options used in the instruction.

Table 2 The General Purpose Register R6

R6	CPPN/ SRPN/ CNT1	32 bit general-use register / multi purpose register
Name	**Bits**	**Description**
CNT1	R6[11:0]	Outer Loop count for COPY instruction or EXIT instruction
TOS	R6[15:14]	Type Of Service for EXIT interrupt
SRPN	R6[23:16]	SRPN for EXIT interrupt
CPPN	R6[31:24]	Current PCP Priority Number posted to PICU

Register R6 may be availed as a general-purpose register, but there are several times where its use is reserved. If the instruction COPY is used, then the field R6.CNT1 must be properly configured. If an EXIT instruction is used that causes an interrupt, R6.SRPN and R6.TOS must be configured properly immediately before the EXIT. If interrupt priority management is utilized, then R6.CPPN is required immediately before an EXIT is executed.

Table 3 The General Purpose Register R7

R7	**DPTR / Flags**	**PRAM Data Pointer (DPTR) and Status register bits**
Name	**Bits**	**Description**
Z	R7[0]	Zero
N	R7[1]	Negative
C	R7[2]	Carry
V	R7[3]	Overflow
CN1Z	R7[4]	CNT1 = 0
INT	R7[5]	1 = Enable Interruption of Channel. May be used to dynamically alter interrupt enable
CEN	R7[6]	1 = Channel Enable
-	R7[7]	Reserved
DPTR	R7[15:8]	Data Pointer Segment Address for PRAM accesses
PC	R7[31:16]	Reserved – Location of Next PC in context

Register R7 is an exception with respect to the other registers that can not be used as a general-purpose register. It serves for purposes similar to those of a Program Status Word known from traditional processors.

R7 holds the flag bits, a channel enable/disable control bit, and the data pointer (DPTR) into the PRAM. The upper 16 bits of R7 are reserved.

2.4 Example Configurations

After this short overview about the PCP Code- and Data Memory, it is time to show how this can be implemented in your software. The following section will explain how to do this on three different examples.

This examples and the used instructions are commented, but you should refer to the TriCore “User’s Manual“ if you want to have a more detailed information.

Note: The full generic format and description of the registers PCPCS, PCPICR is given in the “User’s Manual” of the TriCore with a detailed description of each bit and bit field.

2.4.1 Normal, Multi-Channel

- Full Context model (all registers are saved and restored).
- Channel Programs always start at their base addresses (0x2 * Channel number = PC of the PCP).
- Interrupt arbitration of 4 cycles at 2 clocks per cycle.

The following table describes the minimum configuration updates required for a fully operational PCP to accommodate the channel programs described below.

Table 4 Normal, Multi-channel Configuration

FPI Address	Register Value	Description
0xF0003F10	0xnn000011	PCPCS. Enable PCP, Restart Channels at Base, Full Context, nn = error interrupt number to CPU.
0xF0003F20	0x00000000	PCPICR. Four arbitration cycles, two clocks per arbitration round.

Note: Each Channel program is contained in two instructions, or it must branch to the continuation point of the program further on in Code memory. In the example below, it branches to the main program. There is also code sharing between channels.

Note: R7 context of each channel must have bit 6 = 1 in order the channel is enabled.

Normal, Multi Channel - Program & Context

```
;This program is designed to provide a series of channels to execute
;that will respond to CPU stimuli (interrupts). It additionally
;causes interrupt traffic on the PCP side.
;This also shows how multiple channel programs can use the same piece
;of code, where the difference in function is implicit in the context.
;For PCPCS.RCB = 1 the channels will always start from the
;Channel Entry Point.

;Channel Entry Point Table
.org 0x2              ;start of Channel 1 (SRPN 1)
CH1: JC.A CH1A, cc_UC ;Channel 1
.org 0x4              ;start of Channel 2 (SRPN 2)
CH2: JC.A CH24A, cc_UC;Channel 2
.org 0x6              ;start of Channel 3 (SRPN 3)
CH3: JC.A CH3A, cc_UC ;Channel 3
.org 0x8              ;start of Channel 4 (SRPN 4)
CH4: JC.A CH24A, cc_UC;Channel 4

CH1A:                 ;Channel 1 Program outputs note
ST.IF [R3], 0x8, SIZE=32      ;outputs note from R0
EXIT EC=0, ST=0, INT=0, EP=0, cc_UC  ;exit, no interrupts, PC at next

CH24A:                ;Channel 2 & 4 Program
                      ;outputs note and interrupts CPU
ST.IF [R3], 0x8, SIZE=32      ;outputs note from R0
EXIT EC=0, ST=0, INT=1, EP=0, cc_UC  ;exit, interrupt CPU from R6

CH3A:                 ;Channel 3 Program
                      ;outputs note and interrupts PCP
ST.IF [R3], 0x8, SIZE=32      ;outputs note from R0
EXIT EC=0, ST=0, INT=1, EP=0, cc_UC  ;exit, interrupt PCP from R6
```

```
;A linear list of PRAM context for this program,
;beginning at address 0 (zero) is
.SDECL "pcp.data", DATA       ;instruction for the locator (TASKING)
.SECT  "pcp.data"             ;to allocate this code in the PCP-PRAM
.space 32                     ;Channel 0 is not used
Ch1:    ;Channel 1 Full-Context
.word 0x00000040       ;R7 channel enable set, DPTR=0
.word 0x01000000       ;R6
.word 0x00000000       ;R5
.word 0x00000000       ;R4
.word 0xD0000000       ;R3 verify base address
.word 0x00000000       ;R2
.word 0x00000000       ;R1
.word 0x40000001       ;R0 note value
Ch2:    ;Channel 2 Full-Context
.word 0x00000040       ;R7 channel enable set, DPTR=0
.word 0x02200000       ;R6 CPU service, interrupt number 0x20
.word 0x00000000       ;R5
.word 0x00000000       ;R4
.word 0xD0000000       ;R3 verify base address
.word 0x00000000       ;R2
.word 0x00000000       ;R1
.word 0x40000002       ;R0 note value
Ch3:    ;Channel 3 Full-Context
.word 0x00000040       ;R7 channel enable set, DPTR=0
.word 0x03044000       ;R6 PCP service, interrupt number 0x04
.word 0x00000000       ;R5
.word 0x00000000       ;R4
.word 0xD0000000       ;R3 verify base address
.word 0x00000000       ;R2
.word 0x00000000       ;R1
.word 0x40000003       ;R0 note value
Ch4:    ;Channel 4 Full-Context
.word 0x00000040       ;R7 channel enable set, DPTR=0
.word 0x04400000       ;R6 CPU service, interrupt number 0x40
.word 0x00000000       ;R5
.word 0x00000000       ;R4
.word 0xD0000000       ;R3 verify base address
.word 0x00000000       ;R2
.word 0x00000000       ;R1
.word 0x40000004       ;R0 note value
```

2.4.2 Restore PC, Multi-Channel

- Full Context model (all registers are saved and restored).
- Programs always start at their restored address that is present in the channel context. This allows more compact programs with better linearity.
- Interrupt arbitration of 4 cycles at 2 clocks per cycle.

The following table describes the minimum configuration updates required for a fully operational PCP to accommodate the channel programs described below.

Table 5 Restore PC, Multi-channel Configuration

FPI Address	Register Value	Description
0xF0003F10	0xnn000001	PCPCS. Enable PCP, Restart from PC in Context, Full Context, nn = error interrupt number to CPU
0xF0003F20	0x00000000	PCPICR. four arbitration cycles, two clocks per arbitration round.

Note: In the code below, each Channel jumps to the instruction immediately before the channel program start. This instruction slot holds the EXIT. After exiting, the NextPC value will be stored as the PC in the context, and the channel will begin operation at its beginning. This relies on the context to be configured correctly at boot time. Also notice that each channel start address is independent of the actual channel number.

Restore PC, Multi-Channel - Program & Context

```
;This program is designed to provide a series of channels to execute
;that will respond to CPU stimuli (interrupts). It additionally
;causes interrupt traffic on the PCP side.
;This also shows how multiple channel programs can use the same pieces
;of code, where the difference in function is implicit in the context.
;It will work with PCPCS.RCB = 0 - that means the channel will
;always starts from the restored context PC.

.org 0x0
CH15:                    ;Channel Program 1 & 5
EXIT EC=0, ST=0, INT=0, EP=1, cc_UC   ;exit, no interrupt, PC @ next
ST.IF [R3], 0x8, SIZE=32 ;output note from R0
```

```
JC CH15, cc_UC            ;loop back before exit

CH24:                     ;Channel 2 & 4 Program
EXIT EC=0, ST=0, INT=1, EP=1, cc_UC   ;exit, interrupt to CPU, PC @ next
                          ;Outputs message and interrupts CPU
ST.IF [R3], 0x8, SIZE=32 ;output note from R0
JC CH24, cc_UC            ;loop back before exit

CH3:                      ;Channel 3 Program
EXIT EC=0, ST=0, INT=1, EP=1, cc_UC   ;exit, interrupt to PCP,  PC @
                                      ;next STATE0:
   COMP.I R5, 0x0         ;compare to state number it should be
   ADD.I R5, 0x1          ;increment state number
EXIT EC=0, ST=0, INT=0, EP=1, cc_UC   ;benign exit, PC @ next
                                      ;STATE1:
   COMP.I R5, 0x1         ;compare to state number it should be
   JC ERROR, cc_NZ        ;jump to error routine if not correct
   ADD.I R5, 0x1          ;increment state number
EXIT EC=0, ST=0, INT=0, EP=1, cc_UC   ;benign exit, PC @ next
                                      ;STATE2:
   COMP.I R5, 0x2         ;compare to state number it should be
   LD.I R5, 0x0           ;reset state number
JC CH3, cc_UC             ;jump back to start of state machine
                          ;ERROR:
   ST.IF [R3], 0x8, SIZE=32   ;output note from R0
JC CH3, cc_UC             ;jump back to start of state machine

;A linear list of PRAM context for this program,
;beginning at address 0 (zero) is

.SDECL "pcp.data", DATA   ;instruction for the locator (TASKING)
.SECT  "pcp.data"         ;to allocate this code in the PCP-PRAM
.space 32                 ;Channel 0 is not used

CH1:                      ;Channel 1 Full-Context
.word 0x00000040          ;R7 channel enable set, DPTR=0, restore PC=1
.word 0x01000000          ;R6
.word 0x00000000          ;R5
.word 0x00000000          ;R4
.word 0xD0000000          ;R3 verify base address
.word 0x00000000          ;R2
.word 0x00000000          ;R1
.word 0x40000001          ;R0 note value
```

```
Ch2:                    ;Channel 2 Full-Context
.word 0x00030040        ;R7 channel enable set, DPTR=0, restore PC=4
.word 0x02200000        ;R6 CPU service, interrupt number 0x20
.word 0x00000000        ;R5
.word 0x00000000        ;R4
.word 0xD0000000        ;R3 verify base address
.word 0x00000000        ;R2
.word 0x00000000        ;R1
.word 0x40000002        ;R0 note value

Ch3:                    ;Channel 3 Full-Context
.word 0x00060040        ;R7 channel enable set, DPTR=0, restore PC=7
.word 0x03044000        ;R6 PCP service, interrupt number 0x04
.word 0x00000000        ;R5 state number
.word 0x00000000        ;R4
.word 0xD0000000        ;R3 verify base address
.word 0x00000000        ;R2
.word 0x00000000        ;R1
.word 0x40000003        ;R0 note value

Ch4:                    ;Channel 4 Full-Context
.word 0x00030040        ;R7 channel enable set, DPTR=0, restore PC=4
.word 0x04400005        ;R6 CPU service, interrupt number 0x40
.word 0x00000000        ;R5
.word 0x00000000        ;R4
.word 0xD0000000        ;R3 verify base address
.word 0x00000000        ;R2
.word 0x00000000        ;R1
.word 0x40000004        ;R0 note value

Ch5:                    ;Channel 5 Full-Context
.word 0x00000040        ;R7 channel enable set, DPTR=0,restore PC=1
.word 0x05000000        ;R6
.word 0x00000000        ;R5
.word 0x00000000        ;R4
.word 0xD0000000        ;R3 verify base address
.word 0x00000000        ;R2
.word 0x00000000        ;R1
.word 0x40000005        ;R0 note value
```

Note: R7 context of each channel must have bit 6 = 1 in order for the channel to operate.

2.4.3 Small Context

- Small Context model (only R4-R7 registers are saved and restored).
- R0 - R3 are used as global registers.
- Channel Programs always start at their restored PC.
- Interrupt arbitration of 4 cycles at 2 clocks per cycle.

The following table describes the minimum configuration updates required for a fully operational PCP to accommodate the channel programs described below.

Table 6 Small Context Configuration

FPI Address	Register Value	Description
0xF0003F10	0xnn000041	PCPCS. Enable PCP, Restart Channels at restored PC, Small Context, nn = error interrupt number to CPU.
0xF0003F20	0x00000000	PCPICR. Four arbitration cycles, two clocks per arbitration round.

Small Context - Program & Context

```
;This program is designed to provide a series of channels to execute
;that will respond to CPU stimuli (interrupts). It additionally
;causes interrupt traffic on the PCP side.
;This also shows how multiple channel programs can use the same pieces
;of code, where the difference in function is implicit in the context.
;It will work with PCPCS.RCB = 0 - that means the channel
;will always starts from the restored context PC.

CH5:                    ;Initialization Channel Program would be run
                        ;by interrupt from
                        ;CPU at time 0 (boot procedure) sets up global
                        ;data in R3
EXIT EC=0, ST=0, INT=0, EP=1, cc_UC ;exit, no interrupts, PC @ next
LD.I R3, 0              ;clear R3
LDL.IU R3, D000         ;build D0000000 as verify base address
JC CH5, cc_UC           ;loop back before exit

CH13:                   ;Channel Program 1 & 3
EXIT EC=0, ST=0, INT=0, EP=1, cc_UC   ;exit, no interrupts, PC @ next
```

```
ST.IF [R3], 0x8, SIZE=32 ;output note from R0
JC CH13, cc_UC           ;loop back before exit

CH24:                    ;Channel 2 & 4 Program
EXIT EC=0, ST=0, INT=1, EP=1, cc_UC   ;exit, interrupt to CPU
ST.IF [R3], 0x8, SIZE=32 ;output note from R0
JC CH24, cc_UC           ;loop back before exit


.SDECL "pcp.data", DATA  ;instruction for the locator (TASKING)
.SECT  "pcp.data"        ;to allocate this code in the PCP-PRAM
.space 16                ;Channel 0 is not used
Ch1:                     ;Channel 1 Small-Context
.word 0x00050040         ;R7 channel enable set, DPTR=0, restore PC=5
.word 0x01000000         ;R6
.word 0x40000001         ;R5
.word 0xD0000000         ;R4
Ch2:                     ;Channel 2 Small-Context
.word 0x00080040         ;R7 channel enable set, DPTR=0, restore PC=8
.word 0x02600000         ;R6 CPU service, interrupt number 0x60
.word 0x00000000         ;R5
.word 0x00000000         ;R4
Ch3:                     ;Channel 3 Small-Context
.word 0x00050040         ;R7 channel enable set, DPTR=0, restore PC=5
.word 0x03000000         ;R6
.word 0x00000000         ;R5
.word 0x00000000         ;R4
Ch4:                     ;Channel 4 Small-Context
.word 0x00080040         ;R7 channel enable set, DPTR=0, restore PC=8
.word 0x04300000         ;R6 CPU service, interrupt number 0x30
.word 0x00000000         ;R5
.word 0x00000000         ;R4
Ch5:                     ;Channel 5 Small-Context
.word 0x00010040         ;R7 channel enable set, DPTR=0, restore PC=1
.word 0x05000000         ;R6
.word 0x00000000         ;R5
.word 0x00000000         ;R4
```

Note: R7 context of each channel must have bit 6 = 1 in order for the channel to operate.

3 A PCP program: The PWM (Pulse With Modulation) in simple steps

Every large program consists of many different small functions. This chapter will show in basic steps how to write a PCP program using an example. The purpose of this chapter is to improve the readers understanding how modular a program should be. For this reason the PC Resume Mode has been chosen, where the re-entry PC for the next Channel Service is defined through the PC stored with the context (R7.PC) at the Channels EXIT (interrupt-driven state machine). Further, this program is using the Full Context Mode.

Note: The example will be upgraded from step to step and it is executable in every state, but the listing shown below is only the PCP part of the program. See the Appendix for a full program listing.

3.1 The PCP-Program Structure

The following is a short description on the principal structure of an interrupt-driven state machine. This is the structure used for the example later on. The scheme at the left shows how the program flow is organized.

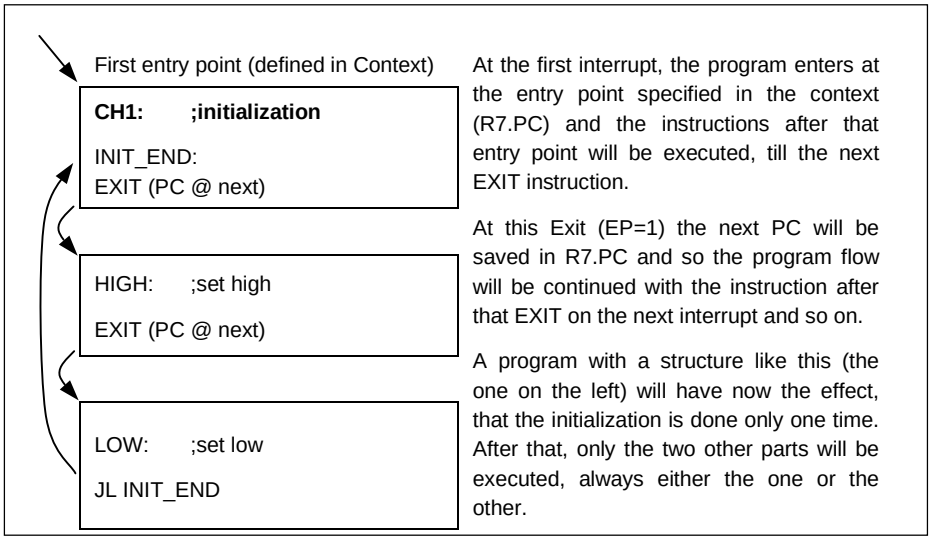

Figure 3 Principle structure of the interrupt states

After this, the principal structure of the program should be clear. The next step is to implement these thoughts and to create an executable program.

The first thing to do is the initialization of the PCP. Since “Restore PC and Full Context“ is chosen, this function must include only the steps described in the chapter ”Restore PC, Multi Channel“ above.

Another important thing is the initialization of one of the timers, because a timer is used to generate periodical Interrupt Service Requests (ISR) to the PCP. Once started, the timer will count until it overflows and this is used to generate an ISR to the PCP. Then the timer will be reloaded with the value stored in the timer reload register and it starts counting again. The functionality of the timer is dependent of many different factors, but this is not the subject of this ApNote. The only interesting thing for us is, that this timer can be reloaded with different reload values and so the time from one to the next Interrupt Service Request can be influenced.

Note: You will find additional information about the timer functionality in the ApNote “Using the Timer Interrupt System“ (AP3224)

The basic configuration is now done and we can start to fill the program structure explained above with code. In this first evolution step we will add only really simple pieces of code. This code is self-explanatory and commented, but for more details on some instructions you should refer to the “User’s Manual“. The following is now a PCP program to visualize how this could look like, but this is only the PCP part of the program. This program has the functionality to toggle the pin P13.5 . This Pin is connected to a Toogle LED on the TriBoard TC1775. So you can check the result with an osciloscope or simply watching the LED.

Example: PCP Program – Pin toggling

```
.org 0x2                          ;start code at address 0x2
CH1:  ;Channel 1
      CLR.F [R4], 0x5             ;set P13.5 to Low
      LDL.IU R1, 0xF000           ;store address of the Port Direction
      LDL.IL R1, 0x3518           ;P13 Direction Control Register in R1
      SET.F [R1], 0x5, SIZE=16    ;set P13.5 to Output

      LDL.IU R0, 0x0000
      LDL.IL R0, 0x000F           ;store 0x0000000F in R0
      LDL.IU R1, 0xF000           ;store address of the Timer Run
      LDL.IL R1, 0x0760           ;Control Register (T012RUN) in R1
      OR.F R0, [R1]               ;create T012RUN |= 0x0000000F;
      ST.F R0, [R1]               ;and store value in T012RUN

INIT_END:                         ;end of the initialization
EXIT EC=0, ST=0, INT=0, EP=1, cc_UC
```

```
                                ;exit, no interrupts, start @next PC

HIGH:                           ;set high
      SET.F [R4], 5             ;set P13.5
EXIT EC=0, ST=0, INT=0, EP=1, cc_UC
                                ;exit, no interrupt, start @ next PC
LOW:                            ; set to low
      CLR.F [R4], 5             ;clear P13.5
EXIT EC=0, ST=0, INT=0, EP=1, cc_UC
                                ; exit, no interrupt, start @ next PC
      JL INIT_END


.SDECL "pcp.context", DATA
.SECT  "pcp.context"
.space 32

Ch1:
.word 0x00020140                ;R7 DPTR 0x1, begin @ 0x2, chan.enable
.word 0x00010000                ;R6 CPU Interrupt Configuration
.word 0x0                       ;R5
.word 0xF0003510                ;R4 P13 Output Register
.word 0x0                       ;R3
.word 0x0                       ;R2
.word 0x0                       ;R1
.word 0x0                       ;R0
```

3.2 Simple PWM

In this second step, the program will be upgraded from a program with an arbitrary reload value to a program with a well defined reload value for the timer. Before, the timer reload register was on any value and so the pin was the same time on high as on low. The idea is now, to define a High-time and a Period duration in the context and to use this values to change the reload value for each state separately.

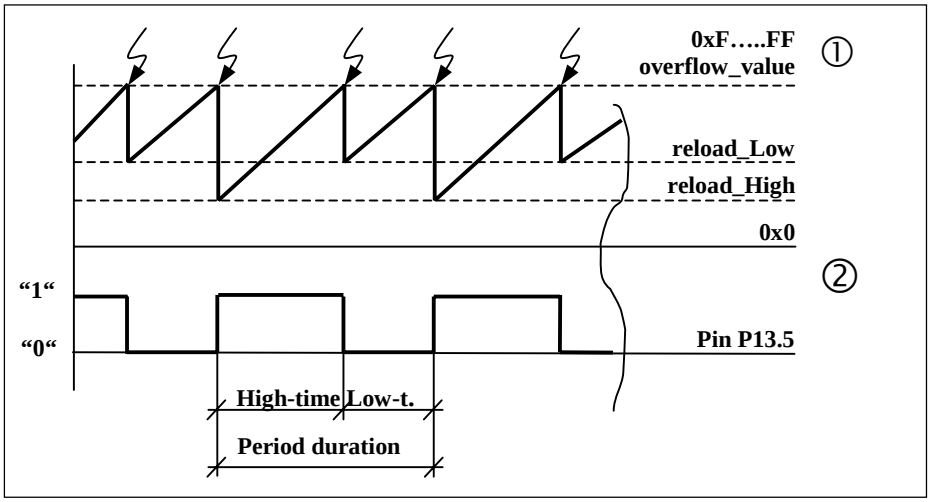

Figure 4 Timer overflows generate a PWM output signal

① That means, if the timer overflows from reload value reload_Low, the timer immediately generates an Interrupt Service Request (ISR) and restarts from reload_High the value specified in the reload register.

② Meanwhile the PCP-interrupt handler (started through the ISR) sets P0.0 to ”1” and changes the reload register to the value reload_ Low and vice versa.

This example shows, that if the code structure is modular, the program can be upgraded quite easily. With only some small changes we could modify the code above from a bit toggling function to the steering of a DC motor below.

Example: PCP Program – Simple PWM

```
.org 0x2                          ;start code at address 0x2
CH1:  ;Channel 1
      CLR.F [R4], 0x5             ;set P13.5 to Low
      LDL.IU R1, 0xF000           ;store address of the Port Direction
      LDL.IL R1, 0x3518           ;P13 Direction Control Register in R1
      SET.F [R1], 0x5, SIZE=16    ;set P13.5 to Output

      LDL.IU R0, 0x0000
      LDL.IL R0, 0x000F           ;store 0x0000000F in R0
      LDL.IU R1, 0xF000           ;store address of the Timer Run
      LDL.IL R1, 0x0760           ;Control Register (T012RUN) in R1
      OR.F R0, [R1]               ;create T012RUN |= 0x0000000F;
```

```
      ST.F R0, [R1]              ;and store value in T012RUN

INIT_END:                        ;end of the initialization
EXIT EC=0, ST=0, INT=0, EP=1, cc_UC
                                 ;exit, no interrupts, start @next PC
HIGH:                            ;set high
      SET.F [R4], 5              ;set P13.5
      LDL.IU R1, 0xFFFF
      LDL.IL R1, 0xFFFF
      SUB R1, R3, cc_UC          ;0xF..FF - Period
      ADD R1, R2, cc_UC          ;+ High time = reload_Low

      ST.F R1 , [R5], Size=32    ;store Reload Value
EXIT EC=0, ST=0, INT=0, EP=1, cc_UC
                                 ;exit, no interrupt, start @ next PC
LOW:                             ; set to low
      CLR.F [R4], 5              ;clear P13.5
      LDL.IU R1, 0xFFFF
      LDL.IL R1, 0xFFFF
      SUB R1, R2, cc_UC          ;0xF..FF – High time = reload_High
      ST.F R1 , [R5], Size=32    ;store Reload Value
EXIT EC=0, ST=0, INT=0, EP=1, cc_UC
                                 ; exit, no interrupt, start @ next PC
      JL INIT_END

.SDECL "pcp.context", DATA       ;declare pcp.data section
.SECT  "pcp.context"
.space 32                        ;Channel 0 is not used
Ch1:
.word 0x00020040                 ;R7 begin address 0x2, channel enable
.word 0x0                        ;R6
.word 0xF000073C                 ;R5 Reload Register (32-bit)
.word 0xF0003510                 ;R4 P13 Output Register
.word 0x00000400                 ;R3 Period length (ex. 0x400 clocks)
.word 0x00000100                 ;R2 High Time (ex. 0x100 clocks)
.word 0x0                        ;R1
.word 0x0                        ;R0
```

3.3 Complex PWM and interrupt to the CPU

The previous code showed, how a store (or similar instruction) through the FPI could look like and also, how to use the context as data storage from one program execution to the next. Many different instructions where used to show them in “action“ and to visualize how they are working.

Now, this last evolution step shows: the usage of the PRAM as general data storage accessed by setting the DPTR, how the counter CNT1 (part of register R6) could be used and how to generate an interrupt to the CPU. To visualize this functionalitys the program will be upgraded to a complex PWM. In the previous step of the example the code generated an output signal on pin P13.5 with a well defined high and low phase. This phases where defined by the values stored in the context.

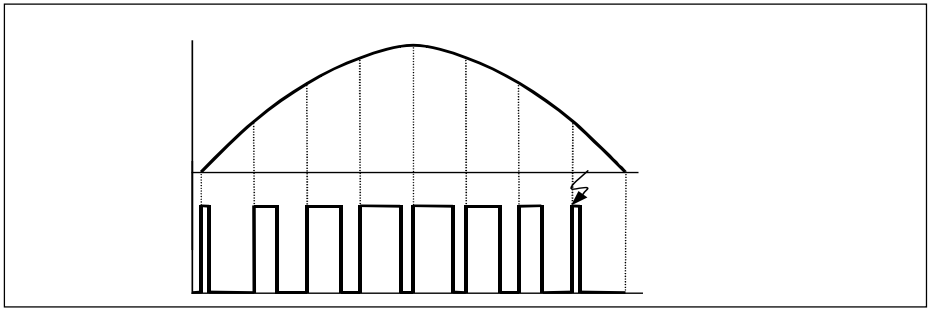

Figure 5 Complex PWM signal

So the next idea is to change this values in the context periodically to approximate a sinus half wave. This could be done by reading values for the High-time from a table stored in the PRAM. We can do this really simple using the code we had in the previous example and add only a small function that reads a value from the PRAM depending of a counter state.

Another implemented thing you will see below is that the code generates an interrupt request to the CPU at the beginning of the last high phase.

This is done in order to increment a counter with the CPU, but the CPU interrupt handler could also contain a routine to recalculate the values for a feedback control system and store them in the PRAM. A system like this could be the automatic control for an one phase a.c motor (1~) and it could easily be upgraded to the regulation of a 3 phase a.c. motor (3~) using three PCP channels and three timers as input for them.

Example: PCP Program – Complex PWM and interrupt to the CPU

```
.org 0x2                          ;start code at address 0x2
CH1:  ;Channel 1
      CLR.F [R4], 0x5, SIZE=16    ;set P13.5 to Low
      LDL.IU R1, 0xF000           ;store address of the Port Direction
      LDL.IL R1, 0x3518           ;P13 Direction Control Register in R1

      SET.F [R1], 0x5             ;set P13.5 to Output
```

A PCP program: The PWM (Pulse With Modulation) in simple steps

```
      LDL.IU R0, 0x0000
      LDL.IL R0, 0x000F          ;store 0x0000000F in R0
      LDL.IU R1, 0xF000          ;store address of the Timer Run
      LDL.IL R1, 0x0760          ;Control Register (T012RUN) in R1
      OR.F R0, [R1]              ;create T012RUN |= 0x0000000F;

      ST.F R0, [R1]              ;and store value in T012RUN
      LD.P R2, [R6], cc_UC       ;load value from table in R2
                                 ;load asdress = (DPTR + offset in R6)
INIT_END:                        ;end of the initialization
EXIT EC=0, ST=0, INT=0, EP=1, cc_UC   ;exit, no interrupts,
                                      ;decrement CNT1, PC @ next

HIGH: ;set high
      SET.F [R4], 5              ;set P13.5
      LDL.IU R1, 0xFFFF
      LDL.IL R1, 0xFFFF
      SUB R1, R3, cc_UC          ;0xF..FF - Period
      ADD R1, R2, cc_UC          ;+ High time = reload_Low

      ST.F R1 , [R5], Size=32    ;store Reload Value
      JC NOZERO, cc_CNN          ; jump if CNT1<>0
      SET R6, 0x3                ;set counter CNT1 to 0x8 (initial.)
NOZERO:
EXIT EC=1, ST=0, INT=0, EP=1, cc_UC  ;exit, interrupt to CPU

LOW:  ; set to low
      CLR.F [R4], 5              ;clear P13.5
      LD.P R2, [R6], cc_UC       ;load value from table in R2
                                 ; load asdress = (DPTR + offset in R6)
      LDL.IU R1, 0xFFFF
      LDL.IL R1, 0xFFFF
      SUB R1, R2, cc_UC          ;0xF..FF - High time = reload_High
      ST.F R1, [R5], Size=32     ;store Reload Value

 EXIT EC=0, ST=0, INT=1, EP=1, cc_CNZ  ;exit, interrupt to CPU

      JL HIGH

.align 2

.SDECL "pcp.context", DATA
.SECT  "pcp.context"
.space 32                        ;channel 0 is not used

Ch1:
      word 0x00020140            ;R7 DPTR 0x1, begin @ 0x2, chan.enable
      word 0x00010007            ;R6 CPU Interrupt Config. and CNT1=7
      word 0xF000073C            ;R5 Reload Register (32-bit)
      word 0xF0003510            ;R4 P13 Output Register
      word 0x00000400            ;R3 Period length (ex. 0x400 clocks)
      word 0x000000C0            ;R2 High Time (ex. 0x100 clocks)
      word 0x0                   ;R1
      word 0x0                   ;R0
```

A PCP program: The PWM (Pulse With Modulation) in simple steps

```
.SDECL "pcp.data", DATA, FPI AT 0xF0010100
.SECT  "pcp.data"

      word 0x000000C0           ;Data Section (DPTR 0x1)
      word 0x000001C0           ;starting at word 64 (PCP @ 0x40)
      word 0x00000280           ;
      word 0x00000300           ;this table contains
      word 0x00000300           ;test values, but it
      word 0x00000280           ;could contain CPU
      word 0x000001C0           ;calculated data
      word 0x0000000f           ;
```

4 Hints on how to program the PCP

In this section, are several basic tricks and techniques outlined to help plan channel programs. There are also hints on configuring a channel's context.

4.1 Initial PC

There are two places where a Channel Program can begin operation. The first is the Channel Entry Table location. If PCPCS.RCB = 1 (True) then the Channel Program is forced to always start at its Channel Entry Table location no matter what the restored context value to the PC is. This is much like an interrupt vector location for that channel. When the Channel is started, the PC is set to 02h* Channel Number (SRPN).

The second option is to have the Channel begin executing at whatever address its restored context holds in R7.PC. If PCPCS.RCB = 0 (False) then the Channel Program will simply begin executing at whatever PC is restored in the context R7.PC.

This is an important distinction that has a large impact on how the code memory is configured by the programmer, and what the initial PC is in the Channel Context that is loaded in the PRAM Context Save Area at boot time.

4.1.1 Channel Entry Table

Channel Entry Table operation (PCPCS.RCB = 1) is straightforward. Each time a channel is invoked, it is forced to start from its Channel Entry Table location regardless of its previous context or PC state.

The Channel Entry Table is also used, if the EXIT instruction is executed with EP=0, then the PC saved in the Context Save will be the location in the table for that channel. That means that the next time the channel is started, it will begin operation at the Channel Entry Table.

Note: If EP=0 or PCPCS.RCB = 1 is used, a Channel Entry Table must be provided at the base of Code Memory.

4.1.2 PC Resume

Using the PC Resume operation can be less obvious. Before exiting, a Channel must configure its context PC so that it restarts at the desired location. In this way a mix of interrupt driven state machines can be created as individual Channel Programs. Channels that always start at their beginning, or channels that have a mix of state machine and restart can also be made. An example of a "restarting" Channel is shown below. Before exiting, the channel branches back to CH16 and then exits. This will

leave NextPC as the channel START. When the channel is originally configured by the programmer, the PC field in the R7 context should also be set to START.

```
CH16:                                   ;Channel Program 16
EXIT EC=0, ST=0, INT=0, EP=1, cc_UC     ;exit no intr, leave PC @ next
START:                                  ;nominal channel start address
ST.IF [R3], 0x8, SIZE=32                ;output note from R0
JC CH16, cc_UC                          ;loop back before exit
```

Similarly, an interrupt-driven state machine can be created by exiting with NextPC pointing to the start of the next state. For example (see below) a program starting at address STATE0 proceeds after the first interrupt till the address before STATE1, where the Channel is left ready for the next state, STATE1 in the state machine. After the next interrupt it executes till the address before STATE2 and the Channel is left ready for the next state, STATE2. After another interrupt, it proceeds through STATE2. The channel jumps back to START, which is the address before STATE0, and then the state machine is complete, and it is ready to restart in STATE0.

```
;This program is intended to test the sequence of exit / operate just as
;if you were implementing an interrupt driven state machine. It requires
;a periodic sequence of interrupts.
START:
      EXIT EC=0, ST=0, INT=0, EP=1, cc_UC ;benign exit
STATE0:
      COMP.I R5, 0x0    ;compare to interrupt number it should be
      JC ERROR, cc_NZ   ;jump to error routine if not correct
      ADD.I R5, 0x1     ;increment state number
      EXIT EC=0, ST=0, INT=0, EP=1, cc_UC ;benign exit
STATE1:
      COMP.I R5, 0x1    ;compare to interrupt number it should be
      JC ERROR, cc_NZ   ;jump to error routine if not correct
      ADD.I R5, 0x1     ;increment state number
      EXIT EC=0, ST=0, INT=0, EP=1, cc_UC ;benign exit
STATE2:
      COMP.I R5, 0x2    ;compare to interrupt number it should be
      JC ERROR, cc_NZ   ;jump to error routine if not correct
      LD.I R5, 0x0      ;reset state number
      JC START, cc_UC   ;jump back to start of state machine
ERROR:
```

4.2 Channel Management for Minimum Context

If Small or Minimum contexts are used, only some of the registers are saved and restored, including when a channel interruption occurs. The other registers are unaffected by save and restore. If channel interrupts are enabled, that means there is a problem insuring the integrity of the Registers that are not included in the context. However, the Channel program may still use all registers reliably while allowing interrupts from higher priority Channels. This can be done in one of two ways:

- Channel-independent Global Registers in R0 - R3 (small), or R0 - R5 (minimum)
- Managed Interrupt Windows

4.2.1 Global Registers

In the first case no extra effort is required. These registers hold global pointers or constants, and no channel is allowed to change them.

Note: There does need to have an initial Channel Program that sets these values at or near boot time. There are two choices to implement this. There is either a boot interrupt Channel Program that is invoked once to perform initialization, or there is a program that routinely loads these values as a matter of course, and it is invoked at boot time, or as the very first interrupt.

4.2.2 Interrupt Windows

For managed interrupt windows, the Channel Program must periodically either save R0 - R3 (or R0 – R5 depending on Context Size), or choose a time when the values in them are not needed. At that predetermined point, the Channel Program can momentarily allow interrupts, allowing any higher priority Channel to take precedence. This is done in the following example:

```
SET R7, 6   ;enable Interrupts by setting R7.INT
            ;potential higher interrupt taken here between instructions
CLR R7, 6   ;disable Interrupts by clearing R7.INT
```

This will momentarily allow any pending interrupt greater than the current PCPICR.CPPN to take control of the PCP. If an interrupt is taken, the code will clear the bit upon re-entering the Channel Program, and will proceed until its next allowed interrupt “window”.

4.3 Dispatch of Low Priority Tasks

A higher-priority Channel Program may wish to start a low-priority background task, or periodically pause and re-start itself later when there is no other action required. This can be accomplished in several ways:

- Post a SRPN to a free SRN on the FPI bus, then EXIT.
- Perform an EXIT with the posted interrupt to the PCP and the Channel Number to be started.
- A Single Channel can act as a list-driven task dispatcher.

The first approach is simple, but uses a system SRN resource. It does allow continuous channel operation without using the interrupt stack or having the potential of blocking other processes use of the PCP.

The second approach allows a looping Channel to continue operation in the background without using any interrupt stack entries. It will also always be superseded by any higher priority tasks.

The third approach uses a “master” channel which can dispatch other non-interrupt driven channel programs in a round-robin fashion. In this way multiple tasks could be continuously operated without over using the PCP service request queue, or the interrupt “stack”. This implies polling interrupts in the peripheral SRN’s rather than allowing the natural interrupt structure to dispatch tasks.

4.4 Code Reuse Across Channels (Call / Return)

A special Jump instruction is included to allow subroutine “calls” from multiple Channels. A routine may be jumped to directly, and then returned from using the JC.IA instruction. JC.IA allows a “calling” Channel to set aside a register for its return address, which will typically be NextPC. The called program can then execute a JC.IA to the address stored in the register specified. The programmer will have to adopt a calling convention on which register holds the return address. Register R2 is recommended for this use.

Example:

The main routine loads the return address "RETURN" into R2. Then it jumps to the known subroutine location "SUB." The subroutine, when done, simply jumps to the address held in R2, which is the return address "RETURN."

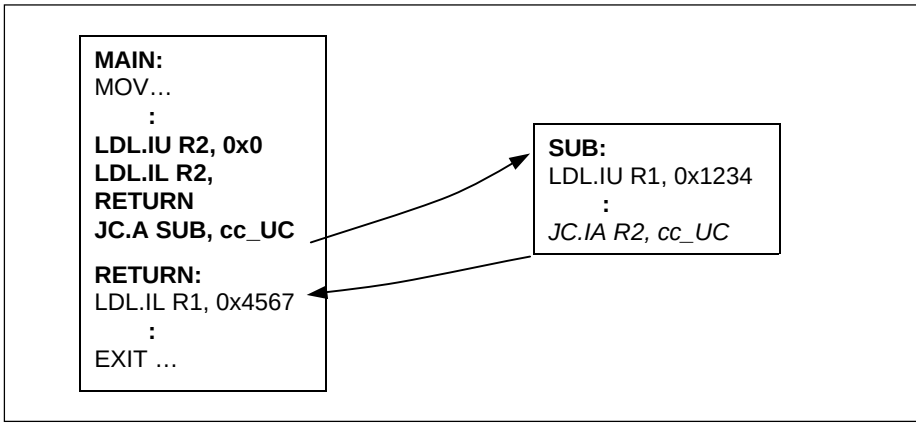

Figure 6 Call/Return functionality

4.5 Case-like Code Switches (Computed Go-To)

In order to implement a multi-way branch for branch on bit or branch on state constructs, a special instruction is included: JC.I. This instruction allows a conditional relative jump based on an index held in a register. If this instruction is combined with a table of jumps, a powerful switch can be implemented. The “default” case (condition code = false) is the next instruction, as is the jump with register index = 0. The index register should be checked for range before the jump into the table is performed.

Example:

```
         COMP.I R3 ,6      ;compare R3 to #6 -
                           ;the number of entries in the table + 1
         JC.I R3, cc_ULT   ;if less than 6, perform jump
DEFAULT: JL case_0         ;destination if R3 = 0 or condition = false
         JL case_1         ;destination if R3 = 1
         JL case_2         ;destination if R3 = 2
         JL case_3         ;destination if R3 = 3
         JL case_4         ;destination if R3 = 4
         JL case_5         ;destination if R3 = 5
```

5 Appendix - PWM program listing

The following is the program listing of the main function used for the PCP program examples in chapter 3 (PCP program - PWM).

“main.c“

```
#define vuint  volatile unsigned int
#define ICR           0xFE2C  //Interrupt Control Register of the CPU
#define GPTU_SRC0     *(vuint*)0xF00007FC //GPTSRC0
#define GPTU_SRSEL    *(vuint*)0xF00007DC //Source Selection Reg.
#define GPTU_CLC      *(vuint*)0xF0000700 //Clock and Power Control
#define GPTU_TOSEL    *(vuint*)0xF000072C //GPT Output Source Sel. Reg.
#define PCP_CS        *(vuint*)0xF0003F10 //PCP Control and Status Reg.
#define PCP_ICR       *(vuint*)0xF0003F20 //PCP Interrupt Control Reg.
#define GPTU_T012RUN  *(vuint*)0xF0000760 //Run Control Register
#define GPTU_T0RDCBA  *(vuint*)0xF000073C //Reload Register (32-bit)
#define GPTU_T0DCBA   *(vuint*)0xF0000734 //Count Register (32-bit)
#define GPTU_T01OTS   *(vuint*)0xF0000714 //Output, Trigger Selection
#define GPTU_T01IRS   *(vuint*)0xF0000710 //Input & Reload Source
#define P13_ALTSEL0   *(vuint*)0xF0003544 //Alternate Port Mode0
#define P13_DIR       *(vuint*)0xF0003518 //Port direction

int isrCounter;

void initPCPfull(void)    //function to initialize the PCP
{
    PCP_CS  = 0x00020801; //Enable PCP, PC from Context, Full context
    PCP_ICR = 0x30000000; //1 arb. cycles, 2 clocks per arbitration
}

void initT0(void)         //function to initialize T0 registers
{
    clear_endinit();
        // setting clock to GPTU (enable)
    GPTU_CLC = 0x100;
        // setting endinit, password access to wdtcon0
    set_endinit();

    GPTU_T01IRS = 0x000700FC;  //concatenate A/B/C/D-blocks(32-bit)
                               //reload on overflow of D-block
                               //select mod_clk as A-block clock input
    GPTU_T01OTS = 0x00000303;  //overflow of block D triggers SR00
    GPTU_T0DCBA = 0xFFFFFC00;  //initial load value for count register
    GPTU_T0RDCBA = 0xFFFFFC00; //reload register (T0 is upcounting)
    GPTU_TOSEL = 0x00000000;   //disable automatic bit toggling by timer
```

```
    P13_ALTSEL0 |= 0x0001;       //select alternative output function
    P13_DIR |= 0x0001;           //set direction register
    GPTU_SRSEL = 0xC0000000;     //assign GTSRC0 the source SR00
    GPTU_SRC0 = 0x00001401;      //enable service request node 0, &
    GPTU_T012RUN |= 0x0000000F;  //stop timer T0
}

void _interrupt(1) ext_int(void) // interrupt routine for priority 1
{
    isrCounter++;                // increment counter
}

void main()
{
    _bisr(0x00);              //enable Interrupt System and set CCPN=0

    initT0();                //function to initialize T0 registers
    initPCPfull();           //function to initialize the PCP (full
                             //context and restart from PC in Context)
   GPTU_SRC0 |= 0x8000;       //set ISR to node GTSRC0

    while(1)
        {
        }
}
```

“pwm.pcp“

See program listings in chapter 3 (PCP program – PWM).

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