Application Note, V3.0, Jul. 2008

AP32009

TC17x6/TC17x7

Safe Cancellation of Service Requests

Microcontrollers

Never stop thinking

Edition 2008-07-15

Published by
Infineon Technologies AG
81726 München, Germany

AP32009		
Revision History:	2008-07	V3.0
Previous Version:	2004-09	V2.0

Page	
	Application note has been extended to TC17x6 and TC17x7.
	Title and template were changed.

Table of Contents **Page**

1 Scope

The scope of this application note is to provide the user with practical hints on how to clear safely service requests for both the PCP2 and the CPU. Assembly code examples are also provided (for the PCP2 only).

The interrupt system of the TriCore Architecture relies basically on hardware control (on the contrary to other architectures like ARM). Management tasks like saving context, clearing the service request bit when the interrupt is acknowledged, etc. are performed automatically by hardware. This reduces the latency time of the interrupt processing and minimizes software overhead.

In given cases, there might be a need to clear service requests “manually”. A practical example would be when two interrupt sources share the same Service Request Node. The first source sends a service request that is acknowledged and the corresponding service routine starts. While it is being processed, the second source sends a request as well. Without any intervention, the Service Provider (either CPU or PCP2) would finish processing the service routine initiated by the first source, and then process again the same routine, initiated this time by the second source. It means that the same routine would be processed twice consecutively, which might be unwanted from a system point view[1].

The user may deal with such a case in two conceptually different ways. The first way takes advantage of the flexibility of the TriCore architecture, which allows the user to set and clear service requests directly. This is performed straight-forwardly by writing in the Service Request Control register (bit “source request clear”) of the related interrupt node. Such a methodology will be referred to as “software cancellation”. However, mixing Software and Hardware control might not be without risk and some general rules need to be fulfilled in order to guarantee the efficiency of the system.

The second way is for the application software to take advantage of the hardware controlled interrupt system. The idea is, instead of directly canceling the request, to actually service a request with a dummy interrupt routine, which does nothing from an application point of view. The hardware takes then care of canceling the request, once it is acknowledged. From the system point of view however, the dummy service does not have any effect. Such a methodology will be referred to as “hardware cancellation” in the rest of this document.

In section 2, the software cancellation method is analyzed. In Section 3, two different methods using hardware cancellation are described. Pros and Cons are discussed in section 4.

This document deals with PCP2 canceling a request addressed to the PCP2 or CPU to the CPU. Cross-cases (i.e. when the PCP2 cancels a request addressed to CPU, or vice-versa) are not treated in this application note.

2 Software Cancellation

2.1 Canceling service requests to the PCP2 by software.

2.1.1 Purpose

The goal is to provide a safe way to cancel a service request addressed to the PCP2 during the execution of a channel. “Safe” means here that it is made sure that no request from a given node occurring during the execution of the “normal” channel code will be serviced.

The subsections below deal with the case when the cancelled request is of lower priority or of the same priority (e.g. 2 sources sharing the same node) than the executing channel priority.

For the sake of completeness, the considerations below can also be applied when the cancelled request is of higher priority than the executing channel. However the very concept of priority makes this case unlikely to

[1] This case should be of course the execption. Normally, peripherals shall only raise interrupts if they require service.

occur in real applications. In this case, it is assumed that the executing channel have nested interrupts disabled during the whole execution of the channel.

2.1.2 ICU and Arbitration

This section intends to give only the necessary information on the PCP2 arbitration protocol to understand the subsequent sections. For exhaustive information on the interrupt system, please refer to the appropriate Product Manual.

Service requests are serviced by the PCP2 depending on their priority ranking. The Interrupt Control Unit (ICU) determines which of the requested service has higher priority. This process is called arbitration.

An arbitration is performed on all the nodes that are active and that have requesting a service (bit SRR = 1 in the corresponding SRC register) at the start of the arbitration. Nodes whose SRR bit is set while an arbitration is already executing do not take part to this on-going arbitration, and have to wait for the next arbitration to take part.

The duration of the arbitration depends on the system configuration

It depends first on how many arbitration rounds are selected (from 1 to 4). This number is related to the number of interrupt nodes used in the system.

- Up to 3 nodes, only 1 round is needed.
- Up to 15 nodes, 2 rounds are needed.
- Up to 63 nodes, 3 rounds are needed.
- Otherwise 4 rounds are needed.

The duration of an arbitration also depends on how long each arbitration round lasts (1 or 2 FPI clock cycles). When the FPI operates at higher frequencies, 2 cycles per round are needed in order to meet frequency time constraints.

In addition, 3 FPI clock cycles are needed by the ICU to start and to terminate the arbitration.

All in all, an arbitration may take up to 11 FPI clock cycles maximum.

Note: This is the time from peripheral having a request, until the PCP2 sees the winning request. This is not wasted PCP2 time, since the PCP2 carries on processing instructions regardless of the interrupt tree until a valid priority is discovered.

At the end of the arbitration, the ICU has determined which node has the highest priority. The ICU requests it to be serviced by the PCP2. The PCP2 chooses then to service it or not, depending on its own state (IDLE or already servicing another service request), if nested interrupts are enabled, etc. Please refer to the user’s manual for more details.

2.1.3 The issue with software cancellation

Let SRNx be a node requesting a service (SRCx.SRR=1) and channel Y the channel being executed by the PCP2. The goal is to cancel safely the request of SRNx before the PCP2 exits channel Y.

The node request can be cancelled by setting bit SRCx.CLRR. Bit SRCx.SRR will be immediately cleared. In addition, a new arbitration will be initiated. If an arbitration is already on-going, then the arbitration initiated by setting SRCx.CLRR will be delayed till the end of the on-going arbitration.

When bit SRR is effectively cleared, either one of the three following cases may occur:

- No arbitration is being executed by the ICU
- An arbitration is running but SRNx does not participate.
- An arbitration is running and SRNx does participate.

In case one and two, channel Y could exit right after SRCx.SRR is effectively cleared. Any request on SRCx would be safely cancelled. Thus no special care must be taken in the channel software for these cases.

In the third case, on the contrary, a problem may potentially occur. The possibility exists that SRNx still participates in the arbitration even after SRR is cleared. It might thus happen that it eventually wins this arbitration. Now, if channel Y exits before the arbitration has finished, then at the end of it, the PCP2 would service channel X (see Fig. 1).

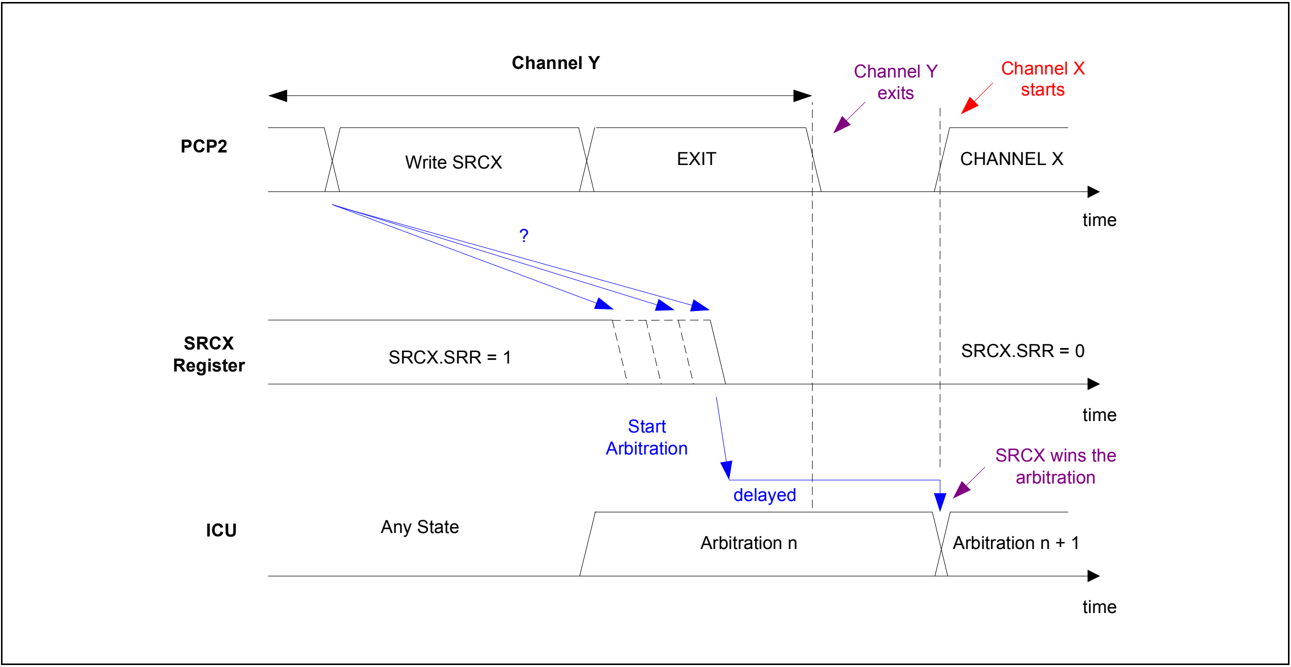


Figure 1 Unsafe cancellation of the service request

The conclusion is that the software has to make sure that the channel exits after any running arbitration is completed.

2.1.4 How long has the PCP2 to wait before exiting?

In the worst case, an arbitration would start the very same clock cycle when bit SRCx.SRR is effectively cleared.

It means that in the worst case, the time between the moment when SRR is effectively cleared and the moment when the PCP2 completes exiting the channel has to be at least 11 FPI clock cycles (since the maximum arbitration duration is 11 cycles).

The minimum execution time for an EXIT instruction is 5 PCP2 clock cycles (including context save).

Depending on the ratio PCP2 clocks to FPI clocks (depends on product options and system configuration), this might result:

- for a system with 1:1 ratio that a minimum of 6 PCP2 cycles (11 – 5 = 6) must take place between the moment the bit SRR is effectively cleared and the PCP2 starts executing the EXIT instruction.
- for a system with 2:1 ratio that a minimum of 17 PCP2 cycles (22 – 5 = 17) must take place between the moment the bit SRR is effectively cleared and the PCP2 starts executing the EXIT instruction.
- even more PCP2 cycles if the ratio of PCP2 frequency to FPI frequency is larger.

2.1.5 Synchronization

In the sections above, it is always mentioned that SRCx.SRR has to be “effectively” cleared.

The reason is, there is an undefined time lag between the moment the channel starts executing the instruction setting CLRR to 1 and the moment when the bit is physically set on hardware level. This time is undefined because of the implementation of the write instruction in the PCP2.

When a FPI Write is issued, the PCP2 posts it and does not wait for an acknowledgement from the FPI target. If the FPI bus is already busy with another transaction, then the PCP2 may execute other instructions before the Write is actually performed.

Thus, the time difference between the Write instruction being issued and it being effectively executed is practically not predictable, because it depends on the whole system implementation.

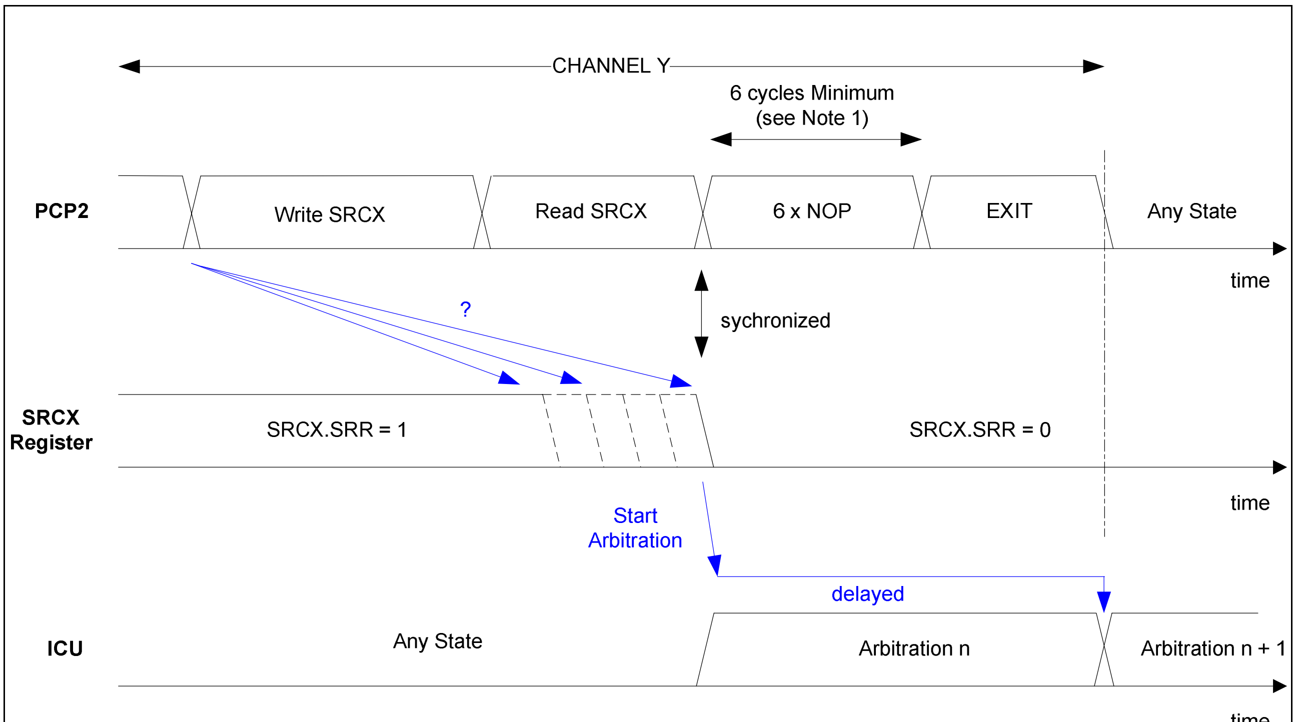
With a Read instruction on the contrary, the PCP2 waits for an acknowledgement of the FPI target before processing the next instruction.

For this reason, the Write instruction that clears bit SRCx.SRR needs to be followed by a Read of register SRCx, in order to synchronize Hardware and Software.

Basically, at the end of the Read SRCx instruction, it is ensured that bit SRCx.SRR has been effectively cleared.

2.1.6 Practical implementation

Let Instruction N be the last instruction (of the normal program flow) of Channel Y. Let SRCx the source to be cancelled. A safe way to cancel interrupt X would be to use following instruction flow (see Fig 2)



Note 1: 6 is the minimum number and might be 17 or higher depending on system capabilities and configuration.

Figure 2 Safe cancellation of the PCP2 service request

Recommended channel program flow

1. Instruction N-1.

Second last instruction of the channel.

2. Disable Nested interrupt (if enabled):R7.IEN=0.

This step is optional, but might be necessary. If nested interrupts are enabled, higher priority interrupts may be acknowledged and processed by the PCP2 at the end of instruction N. It means that the service requested by of SRNx would be cancelled “a long time” after the last instruction of the channel is executed. The cancelled service could thus be a relevant service for the system.

3. Instruction N.

4. Write SRCx.CLRR = 1.

This clears SRCx.SRR. All requests happening before this instruction, i.e. happening before the last “normal” instruction (instruction N) of the channel, will be safely cleared.

5. Read SRCx.

This step is used for synchronization purpose. It is ensured that at the end of this instruction (the latest), bit SRR is cleared.

As seen previously, a minimum of 11 FPI clock cycles need to be inserted from the moment SRR is effectively cleared and the moment the channel exit is completed.

It means equivalently inserting 11 FPI clock cycles from the end of the Read instruction. Since the execution time of the EXIT is at least 5 clock cycles, a minimum of 6 clock cycles instructions need to be inserted between this point and the EXIT instruction.

6. Insert a sufficient number of Nops

Or any code longer than a sufficient number of cycles[1].

7. Enable Nested interrupt R7.IEN=1 (if needed) or insert 1 NOP

8. Exit channel Y

2.1.7 Coding in assembly language

In assembly language, the program flow described in the previous section could be implemented the following way:

Implementation 1

```
START_CHANNEL_Y:
       ; Normal channel Y code…
CANCELLATION:
       LDL.IU R0, SRC_FPI_ADDR_UP   ;load address of SRCx register
       LDL.IL R0, SRC_FPI_ADDR_low  ;load address of SRCx register
       SET.F [R0], 14, SIZE=32      ;set bit CLRR
       LD.F R1, [R0]                ;read register SRCx
       .DUP 6
       NOP                          ;insert 6 NOP or more according
                                    ;to product and configuration.
       .ENDM
       EXIT                         ;normal exit channel Y
```

[1] Instead of NOPs, the user may want to insert here “useful” instructions.

In order to reduce the overhead when no interrupt is requested by node X, one may want first to check the value of SRCx.SRR before going into the cancellation process.

In assembly language, the program flow may be thus implemented the following way:

Implementation 2

```
START_CHANNEL_Y:
       ; Normal channel Y code…
CHECK_SRR:
       LDL.IU R0, SRC_FPI_ADDR_UP   ;load address of SRCx register
       LDL.IL R0, SRC_FPI_ADDR_low  ;load address of SRCx register
       LD.F R1, [R0]                ;Read register SRCx
       CHKB R1, 13                  ;if SRR bit is set, set Carry flag
       JC.A CLEAR_SRR, cc_C         ;if Carry is set, jump.
       EXIT                         ;normal exit for channel Y

CLEAR_SRR:
       SET.F [R0], 14, SIZE=32      ;set bit CLRR
       LD.F R1, [R0]                ;read register SRCx
       .DUP 6
       NOP                          ;insert 6 NOP or more according
                                    ;to product and configuration.
       .ENDM
       EXIT                         ;exit channel Y
```

2.2 Canceling service requests to the CPU by software.

2.2.1 Purpose

1. This section discusses a safe way of canceling a service request to the CPU during the execution of a service routine. “Safe” means here that it is made sure that no request from the source occurring during the execution of the routine will be serviced.
2. The mechanisms involved for the CPU are similar to those discussed in section 2.1. Therefore, only the points specific to the CPU are discussed in this section.
3. Only the case when the cancelled request is of lower priority or of the same priority than the executing service is discussed in this section.

2.2.2 ICU and Arbitration

4. The arbitration protocol which determines which interrupt source should be serviced first is similar to the PCP2 case. The duration of an arbitration also depends on the system configuration (number of arbitration rounds, number of clock cycles per round).The maximum arbitration duration is 11 FPI clock cycles, i.e. 22 CPU clock cycles, assuming a 1:2 frequency ratio.

Note: The 1:2 ratio (worst case for number of inserted NOP) between the CPU and the FPI bus will be assumed in the rest of the document. For a 1:1 ratio, the calculations can be easily derived (by replacing 22 by 11).

2.2.3 Practical implementation.

The same principles as discussed in section 2.1 for the PCP2 also apply for the CPU.

Let Service Y be the running service and SRNx the node whose service request is to cancel. Setting bit SRCx.CLRR immediately initiates an arbitration if the ICU is IDLE. But it may be delayed if another arbitration is already on-going. Thus, once bit SRR has been effectively cleared, the service has to wait until the end of this on-going arbitration before exiting. Therefore, the worst case occurs when bit SRR is effectively cleared the same clock cycle as the beginning of an arbitration.

For the same reason as discussed in section 2.1, a Read instruction has to follow the Write instruction setting CLRR, so that it is made sure that at the completion of the Read instruction, bit CLRR has been set and therefore bit SRR has been cleared.

Since the minimum execution time of the RFE instruction is 2 CPU clock cycles (including context restore), it means eventually that a minimum of 20 CPU clock cycles (22 – 2 = 20) need to be inserted between the Read SRCx and the RFE.

Let N be the final instruction (normal program flow) of service Y. A safe way to cancel the service request of SRNx would be the following (see Fig. 3):

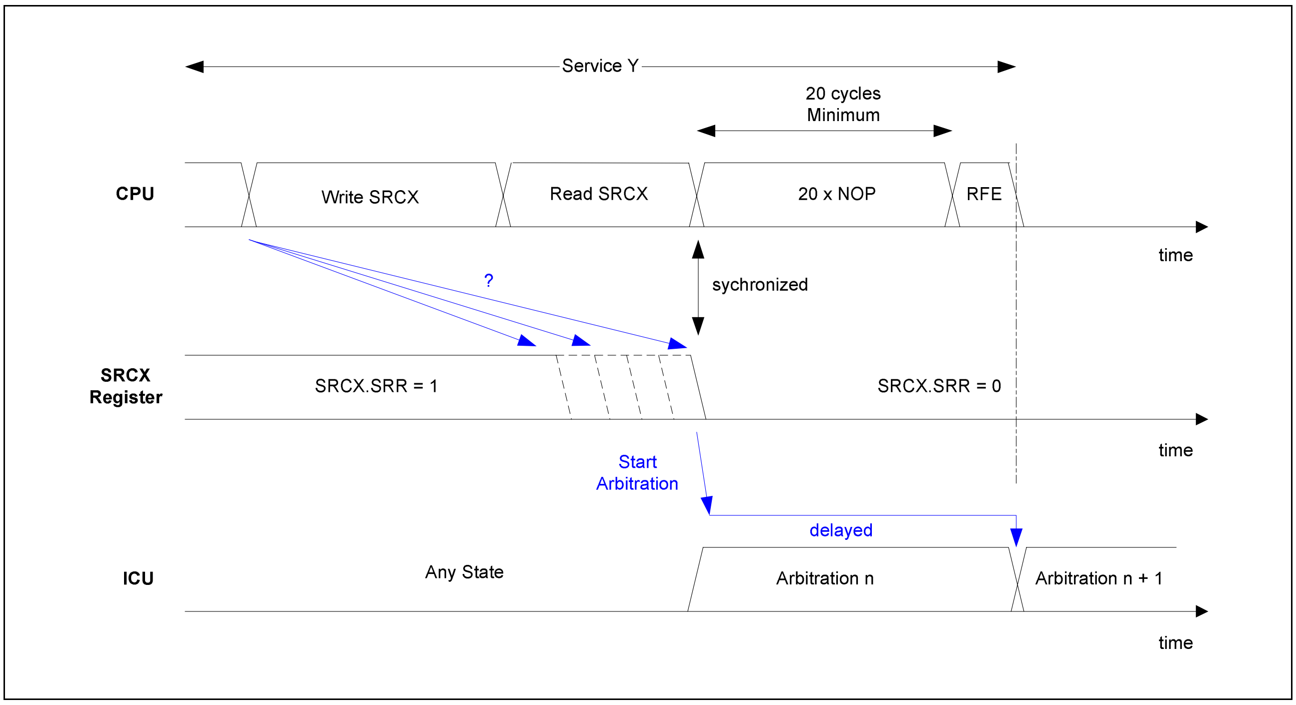


Figure 3 Safe cancellation of the CPU service request

Recommended routine program flow

1. **Instruction N-1**

2. **Disable interrupt (if enabled):ICR.IEN=0.**
Optional, same justification as for the PCP2 case.

3. **Instruction N**

4. **Write SRCx.CLRR = 1**
All service requests happening before this instruction will be safely cancelled.

5. **Read SRCx**
Step used for synchronization.

6. **Insert 19 NOP or any code longer than 19-cycles.**
As Tricore is SuperScalar, this mean up to 38 instructions if instructions selected can be dual-issued.

7. **Insert 1 NOP or enable interrupts (if needed): ICR.IEN=1.**

8. **RFE**

3 Hardware Cancellation

In this section, two different methods for hardware cancellation are presented.

3.1 Hardware cancellation using a global variable.

The explanations below refer to the PCP2. However, a similar reasoning may be performed with the CPU.

3.1.1 Description

Let us consider the same assumption as previously, i.e. the user wants to cancel a request on node X while channel Y is being executed (with priority of X less or equal to the priority of Y).

It can be performed by using a local variable (for example located in the PRAM), and by using some overhead for channels X and Y.

The global variable is first initialized to 0.

At the end of channel Y, it is checked if bit SRCx.SRR is set or not. If not, nothing happens and channel Y exits normally. If yes, then channel Y sets the global variable to a non-zero value and then exits.

At the beginning of channel X, the value of the global variable is tested. If it is zero, standard processing continues. If it is a non-zero, then the channel exits immediately.

As a result, the service request on Node X is actually serviced, but it is transparent for the application.

3.1.2 Practical implementation for the PCP2.

The following program flows may be used for channel X and channel Y.

Channel X:

```
START_CHANNEL_X:
       LD.PI  R0, [offset_glob_var]       ;load PRAM global variable in R0
       JC.A  DUMMY, cc_NZ                 ;jump to DUMMY if variable is not
                                          ;equal to zero
       ;Normal channel X code…
       EXIT                               ;normal exit for channel X
DUMMY:
       LDL.IU R0, 0                       ;sets R0 upper bits to 0
       LDL.IL R0, 0                       ;sets R0 lower bits to 0
       ST.PI  R0, [offset_glob_var]       ;store 0 to PRAM global variable
       EXIT                               ;exit channel
```

Channel Y:

```
START_CHANNEL_Y:
       ; Normal channel Y code…
CHECK_SRR:
       LDL.IU R0, SRC_FPI_ADDR_UP   ;load address of SRCx register
       LDL.IL R0, SRC_FPI_ADDR_low  ;load address of SRCx register
       LD.F R1, [R0]                ;load the value of the SRCx
                                    ;register
       CHKB R1, 13                  ;if SRR bit is set, set Carry flag
       JC.A SET_VARIABLE, cc_C      ;if Carry is set, jump.
       EXIT
SET_VARIABLE:
       LDL.IU R0, 0                 ;sets R0 upper bits to 0
       LDL.IL R0, 1                 ;sets R0 lower bits to 0x1
       ST.PI  R0, [offset_glob_var];store 0x1 to PRAM global variable
       EXIT
```

3.2 Hardware cancellation using the channel start mode (PCP2 only)

The method described here is valid for the PCP2 only. Besides it is not as generic as the methods described so far. It only applies if a channel intends to cancel a request invoking itself. For example, channel X canceling a request on node X. It is besides assumed here that bit PCP_CS.RCB is equal to 0 (start PC as left by last invocation).

This method is based on the fact that when a channel exits, it has the option, at the next invocation, to start either at the channel base address or at the last PC address. This is an option of the EXIT instruction.

Thus, at the end of channel X normal program flow, the PCP2 could check the value of SRCx.SRR bit. If it is not set, the channel exits normally and will restart at the channel base address.

On the contrary, if it is set, it also exits but will restart at the previous PC value. At the next invocation, the channel only performs a single EXIT instruction (restart at channel base).

In assembly language, the following program flow may be used:

Channel X:

```
START_CHANNEL_X:
       ; Normal channel X code…
CHECK_SRR:
       LDL.IU R0, SRC_FPI_ADDR_UP   ;load address of SRCx register
       LDL.IL R0, SRC_FPI_ADDR_low  ;load address of SRCx register
       LD.F R1, [R0]                ;Read register SRCx
       CHKB R1, 13                  ;if SRR bit is set, set Carry flag
       JC.A SRR_SET, cc_C           ;if Carry is set, jump.
       EXIT EP=0                    ;normal exit for channel Y, restart
```

```
                                    ;at channel base address
SRR_SET:
       EXIT EP=1                    ;exits, but will restart at DUMMY_RUN
DUMMY_RUN:
       EXIT EP=0                    ;exits & restart at channel base address
```

4 Software vs. Hardware Cancellation.

In the previous sections, several ways of canceling service requests addressed to the CPU and the PCP2 have been described.

The following criteria are used for analyzing Pros and Cons:

- Portable: is the implementation device specific, or can it be reused for other devices?
- Generic: can any lower priority request be cancelled?
- Deterministic: how much predictable is the overall system behavior? The less the implementation accesses to the internal busses (context load/save, read/write variables etc) or depends on the whole system behavior (amount of interrupts processed, etc), the more "deterministic" it is.
- Valid for CPU.
- Valid for PCP2.
- Cycles consumed if request: what is the minimum amount of FPI cycles used if there is a request to cancel? The values are valid for the PCP2 only.
- Cycles consumed if no request: what is the minimum amount of of FPI cycles used if there is no request to cancel? The values are valid for the PCP2 only.

The Pros and Cons are summarized in Table 1.

Table 1 Software vs. Hardware cancellation: Pros & Cons.

	Software: Implemt. 1	**Software: Implemt. 2**	**Hardware: Glob. Var.**	**Hardware: Chan. Mode.**
Fully Portable	No	No	Yes	Yes
Generic	Yes	Yes	Yes	No
Deterministic	+	+	-	-
Valid for CPU	Yes	Yes	Yes	No
Valid for PCP2	Yes	Yes	Yes	Yes
Res. used if request[1)]	24	37	39-42 [2)]	25-28 [2)]
Res. used if no request[1)]	24	13	17	13

1) The values represent the minimum number of FPI cycles generated by the cancellation program flow.

2) Depending on the size of the context save areas.

From a general point of view, Hardware cancellation methods use more resource. Besides, they go through an additional arbitration phase and the servicing of dummy interrupt routine, which generates extra "random" system behavior (interrupt may be delayed by higher priority services, etc.).

On the other hand, hardware cancellation methods are fully portable from members of the Audo1 through Audo-Future and Audo-S family members. This may not be the case with the Software Cancellation, where code which is optimized for a system configuration (number of interrupt nodes, or ratio of arbitration or CPU cycles per FPI cycle) will need to change if any of those parameters varies).