Please note that Cypress is an Infineon Technologies Company.
The document following this cover page is marked as “Cypress” document as this is the company that originally developed the product. Please note that Infineon will continue to offer the product to new and existing customers as part of the Infineon product portfolio.

Continuity of document content
The fact that Infineon offers the following product as part of the Infineon product portfolio does not lead to any changes to this document. Future revisions will occur when appropriate, and any changes will be set out on the document history page.

Continuity of ordering part numbers
Infineon continues to support existing part numbers. Please continue to use the ordering part numbers listed in the datasheet for ordering.



THIS SPEC IS OBSOLETE

Spec No: 002-05047

Spec Title: AN205047 - Using EI2OS on MB90500 Series Devices

Replaced by: None

AN205047

Using EI²OS on MB90500 Series Devices

This application note describes about the Notes on using EI²OS on MB90500 Series Devices and explains the different behaviour under different conditions.

Contents

1 Introduction....1
1.1 Affected Cores....1
1.2 Interrupts....2
1.3 Extended Intelligent IO Service....3
2 Common Behaviour of L- and LX-Family....4
2.1 Behaviour of L/LX-Family with nested Interrupts....4
2.2 Behaviour of L/LX-family with EI²OS Transfer....4
3 Different Behaviour of L- and LX-Family....5
3.1 Suspended Interrupt Request without EI²OS....5
3.2 Suspended Interrupt Request with EI²OS-Count incomplete....6
3.3 Suspended Interrupt Request with EI²OS-Count complete....8
4 Scheduler and Operating Systems....10
4.1 Stack Analysis....10
4.2 Workaround Programs....11
5 Conclusion....14
6 Document History....15

OBSOLETE

1 Introduction

Some core changes have been done, when developing the LX-family from the L-family. One is the optimised interrupt behaviour, which reduces stack operations. However, there is one specific case, which has to be taken into consideration, if schedulers or operating systems have to be implemented for **MB90500** series MCU. In case of usage of the Extended Intelligent IO Service (EI²OS) the way of storing data on stack can change. The application note explains the different behaviour under different conditions.

The optimised stack operation is described in "3.1.2 Behaviour of LX-Family (EI²OS not used)".

The critical case is described in "3.3.2 Behaviour of MB90500-Series (EI²OS Count complete)". It explains the deviating usage of stack for interrupts.

1.1 Affected Cores

The critical behaviour explained in "3.3.2 Behaviour of MB90500-Series (EI²OS Count complete)" applies to MB90500-series only. Additional stack usage might occur if described conditions are met.

The F²MC-16L family and the MB90400 series of F²MC-16LX family do not show the extra behaviour. Figure 1 lists the series described in this note.

Table 1. Series described by this report

Series	Family	Optimised stack usage with suspended interrupts	Extra stack usage with EI²OS completion interrupt
MB90500	F²MC-16LX	Yes	Yes
MB90400			No
MB90600	F²MC-16L	No	

1.2 Interrupts

Special hardware functions of an MCU can issue an interrupt request. This request will lead to the execution of an interrupt service routine (ISR). This routine interrupts the currently executed program and handles the request of the hardware. Before the execution of the ISR, the MCU core stores a set of registers to the stack (Table 2) in order to save the status of the interrupted program. Because of these stack data the ISR is not finished by normal RET- or RETP-instruction but by RETI-instruction. RETI restores the stack data to the registers and allows continuing the interrupted program.

Table 2. Interrupt stack

Offset	Register		Function
+0	PS		Processor Status (CCR, ILM, RP)
+2	PC		Program Counter (lower 16 bits of instruction pointer IP)
+4	DTB	PCB	Data Bank, Program Counter Bank (higher 8 bits of IP)
+6	DPR	ADB	Direct Page Register, Additional Data Bank
+8	AL		Accumulator lower 16 bits
+10	AH		Accumulator higher 16 bits

If an interrupt service routine is currently executed and another interrupt is requested, the new request is handled either immediately (nested interrupt) or later after the current ISR has been finished. Whether an ISR is executed immediately or delayed for some time depends on a set of registers (Table 3, Table 4).

Table 3. Processor Status register of F²MC-16L/LX

PS (16)								
ILM (3)	RP (5)	CCR (8)						
		-	I	S	T	N	Y	V	C

Table 4. Important registers for interrupt handling

Register		Function
Processor Status	I-flag	The I-flag is part of the Code Condition Register CCR of the processor status PS. It globally enables or disables interrupt request. If it set to 0, interrupts are not handled regardless of other register settings.
	ILM	The Interrupt Level Mask (ILM) is part of the processor status PS. It identifies the level/priority of the current program. Only interrupt request with higher priority (see ICR) can interrupt the current program. Note, higher priority means lower value of the interrupt level value. If a request has a lower level (higher priority) than indicated in ILM, the request can be handled. Before executing the ISR, the core will automatically store the level of the new request in ILM. This locks lower priority requests.
Resource Status	IE-flag	Each resource has its own Interrupt Enable flag (IE or other name). Similar to the global I-flag, this flag enables interrupt handling but for a single resource only.
	C-flag	The Complete flag C (or other name) indicates that the resource has finished processing. The flag could also be used for polling.
	ICR	Each resource of the MCU, which can issue a request, is assigned to a dedicated Interrupt Control Register (ICR). There are 16 different ICR available; one ICR is shared between two resources. The ICR indicates the level/priority of the interrupt request. On request it is compared against a global mask (see ILM). If the ICR-level is same or higher (same or lower priority), the request is not handled until the global mask changes to a higher level than the request. ICR also contains configuration and status data of the EI²OS (see later).

OBSOLETE

Only if all following conditions are met, the ISR is executed:

I-flag = 1 globally interrupt enabled
ILM > ICR request priority is high enough
IE-flag = 1 resource interrupt enable
C = 1 resource operation complete

These conditions apply all the time while interrupts are requested or pending.

LX-family vs. L-family

One difference of L-family and LX-family devices is the stack handling, if an interrupt request is suspended due to an currently running interrupt of higher or same priority.

After the current interrupt is finished the waiting request can be handled and its ISR is started. Before the ISR the interrupt stack of the old ISR is restored to registers. After this the same register values are stored to the new interrupt stack again.

The LX-family devices skip this unnecessary operation and use the same interrupt stack for the next interrupt

1.3 Extended Intelligent IO Service

This DMA-like function can be used in order to handle an interrupt request by an automatic service instead by an ISR. The service allows for a given number requests to copy one byte or word between given addresses on every request. The service can increment the address pointers. This can save a lot of cycles. Instead of executing an ISR (including storing and restoring the interrupt stack) the single data byte/word is copied automatically without executing code. This can be used for writing/reading data to/from Serial IO data register or reading results from ADC.

After transferring the last byte/word an ISR is executed. This EI²OS completion interrupt finalises the operation.

If the resource supports EI²OS, the appropriate ICR is configured by setting the EI²OS-enable flag and by setting a descriptor index. The index points to one out of sixteen possible descriptors, which are located in internal RAM starting at H'0100. It describes the operation of the service (see Table 5 and Table 6).

Table 5. EI²OS descriptor

Offset	Register		Function
+0	BAP		Lower 16 bits of Buffer Address Pointer (usually in RAM)
+2	ISCS	BAPH	Interrupt Service Control Status and upper 8 bits of BAP
+4	IOA		IO register address pointer
+6	DCT		Data Count

Table 6. ISC of EI²OS descriptor

Bit	Flag	Function
0	SE	Termination request enable: allows resources to abort the service
1	DIR	Direction: specifies to copy from or to IO-address
2	BF	Buffer fixed: specifies not to increment the buffer address pointer
3	BW	Byte/Word: specifies to copy bytes or words
4	IF	IO fixed: specifies not to increment the IO address pointer

MB90400-series vs. other series

In the case that an interrupt request for the final transfer of EI²OS is suspended due to currently running interrupt of higher or same priority, a critical stack condition occurs, when the completion interrupt is executed.

At the return form current interrupt, the old interrupt stack is kept and an additional one is stored after EI²OS has been performed (see "3.3.2 Behaviour of MB90500-Series (EI²OS Count complete)").

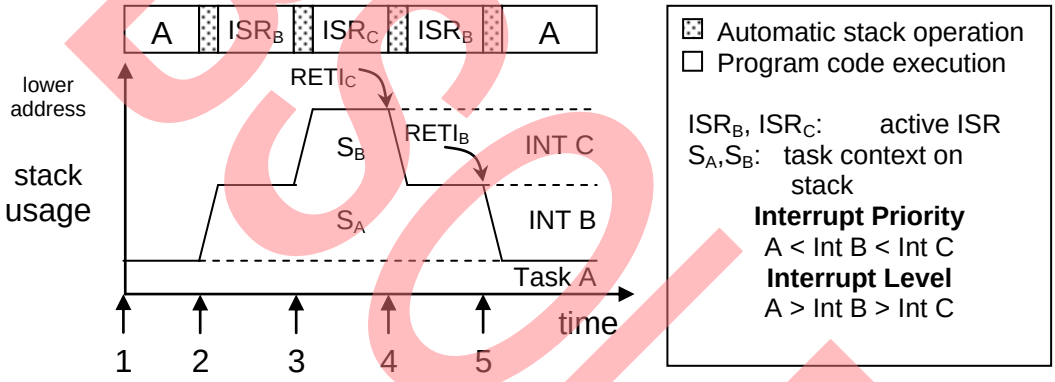
OBSOLETE

2 Common Behaviour of L- and LX-Family

2.1 Behaviour of L/LX-Family with nested Interrupts

1. The case of nested interrupts is identical for L- and LX-family cores. A nested interrupt happens, if the priority of the recent interrupt request is higher than the priority of the task/interrupt, which was running during the request.
2. Any task/program (A) at any interrupt level is executed. Interrupts are globally enabled.
3. Task A is interrupted by Int B. Therefore, an interrupt stack S_A, which describes task A is saved to stack and ISR_B is executed.
4. Another interrupt request INT C occurs with higher priority than Int B. Therefore, the current interrupt is interrupted itself. The task descriptor S_B (containing current state of interrupt Int B) is saved on stack and ISR_C is activated.
5. ISR_D is finished by RETI. The previous task Int B is restored.
6. Finally also ISR_B is finished and task A is restored by RETI.

Figure 1. Stack usage of nested interrupt of L- and LX-family

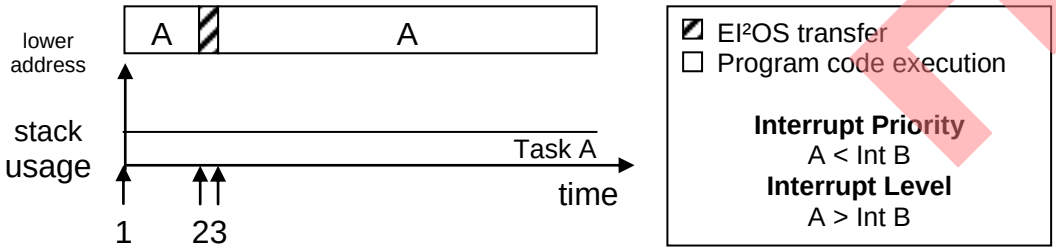


2.2 Behaviour of L/LX-family with EI²OS Transfer

The case of EI²OS transfer the current task/ISR interrupted for a few cycles in order to transfer thebyte/word over the bus.

1. Any task/program (A) at any interrupt level is executed. Interrupts are globally enabled.
2. Task A is interrupted by Int B. Because the EI²OS is enabled by the ICR of Int B, a transfer is performed. The EI²OS counter and pointer are updated.
3. Task A is continued.

Figure 2. Stack usage of nested interrupt of L- and LX-family



OBSOLETE

3 Different Behaviour of L- and LX-Family

3.1 Suspended Interrupt Request without EI²OS

Suspended Interrupt request means that an interrupt is not performed because its priority is not high enough or because interrupts are not enabled globally or for the resource. As soon as the core status allows lower priority interrupts or if interrupts are enabled, the request is handled.

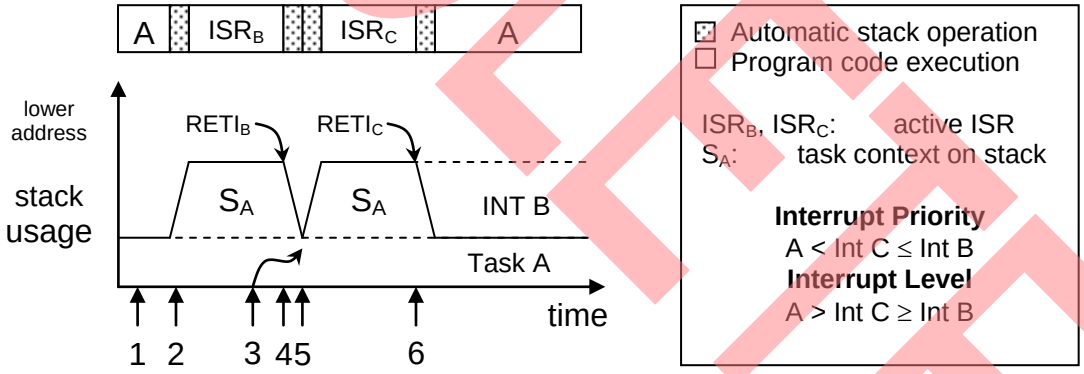
3.1.1 Behaviour of L-Family (EI²OS not used)

For the case of a suspended interrupt the L-family and LX-family behave different.

When executing the RETI instruction, the saved CPU registers will be restored. If another interrupt is already waiting to be executed, just the same register values will be save on stack again.

1. Any task/program (A) at any interrupt level is executed Interrupts are globally enabled.
2. An interrupt (Int B) of higher priority than current task is activated. The current task is saved as S_A on stack and ISR_B is started.
3. Interrupt (Int C) of lower or same priority as Int B but higher priority than task A is requested. Since its priority is not higher than Int B, it is not activated yet. Note, if ISR_B globally disables interrupts (I-flag := 0), INT C is suspended regardless of its priority.
4. ISR_B is finished by RETI. The description S_A of task A is restored on stack by microcode (no program code). If ISR_B had disabled interrupts globally, they will be enabled again because old status was stored on stack.
5. Since Int C is still pending, task A is not continued but ISR_C activated. Not only one instruction of task A has been executed between '4' and '5'. Therefore, the same task descriptor S_A is saved again.
6. ISR_C is finished by RETI. Task A will be restored by writing CPU register values S_A from stack to CPU. After restoring, task A it is continued.

Figure 3. Stack usage of suspended interrupt of L-family



In this situation the LX-family devices behave different from L-family devices. However, LX-family devices behaves also different depending on whether EI²OS (Extended Intelligent IO Service) is calling an interrupt.

3.1.2 Behaviour of LX-Family (EI²OS not used)

The LX-family devices behave different from L-family devices between '4' and '5'. Remember that at '3' an interrupt request occurs that is suspended (small priority or I-flag off).

The L-family will have same contents of the task descriptor on stack after '5' as before '4'. (Between '4' and '5' only micro code and no application code are executed.) Therefore, internal stack operations for restoring and saving are not necessary.

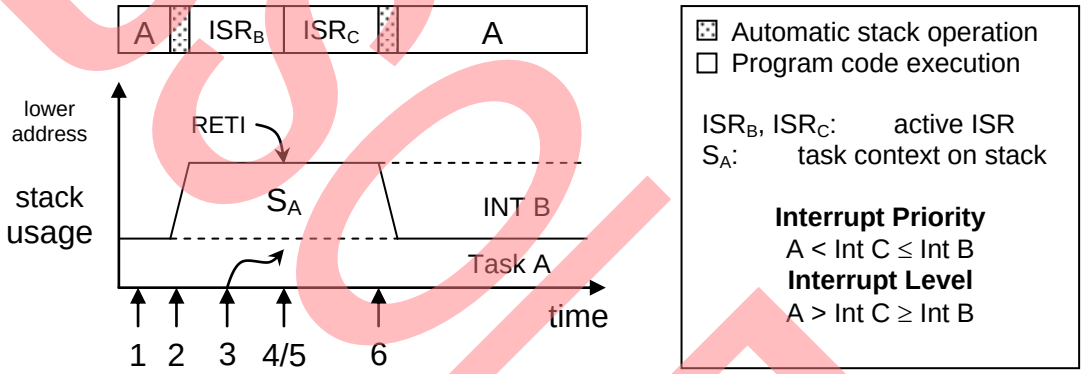
LX-family devices avoid these unnecessary operations.

OBSOLETE

1. Any task/program (A) at any interrupt level is executed Interrupts are globally enabled.
2. An interrupt (Int B) of higher priority than current task is activated. The current task is saved as S_A on stack and ISR_B is started.
3. Interrupt (Int C) of lower or same priority as Int B but higher priority than task A is requested. Since its priority is not higher than Int B, it is not activated yet. Note, if ISR_B globally disables interrupts (I-flag := 0), INT C is suspended regardless of its priority.
4. ISR_B is finished by RETI. . The first word (PS) of the context S_A is written back to PS. Its interrupt level mask is compared against the appropriate interrupt control register ICR of the waiting request. The interrupt level mask ILM is part of the processor state PS[2] that is stored in S_A. If the interrupt level IL of ICR is lower[1] than interrupt level mask ILM of task A, the description S_A is **NOT** restored. The waiting interrupt will be processed immediately.
5. If the ISE bit of ICR is not set (EI²OS disabled), ISR_C will be activated. However, the task context is **NOT** saved. INT C uses the task description that has already been saved by Int B before.
6. ISR_C is finished by RETI. The stack frame S_A is restored and task A is continued.

[1] Lower level means higher priority.
[2] Before RETI-instruction the stack pointer points to PS, which is stored as last word onto stack when context is saved due to interrupt request.

Figure 4. Stack usage of suspended interrupt of LX-family



3.2 Suspended Interrupt Request with EI²OS-Count incomplete

"EI²OS-count incomplete" means that after the recent transfer, there are still other bytes/words pending to transfer. The given number of bytes/words in the descriptor is still unequal zero.

3.2.1 Behaviour of L-Family and MB90400-Series of LX-Family (EI²OS not finished)

If the request '3' is caused by a resource with enabled EI²OS (ISE bit in ICR set), the EI²OS will be activated and the Byte or Word transfer will be performed **after** S_A has been restored. Then DCT (data counter) of the EI²OS descriptor will be decreased. The following processing depends on the value of DCT. In case DCT is not 0 (service not finished), task A is continued (at '5').

1. Any task/program (A) at any interrupt level is executed Interrupts are globally enabled.
2. An interrupt (Int B) of higher priority (lower level) than current task is activated. The current task is saved as S_A on stack and ISR_B is started.
3. Interrupt (Int C) of lower or same priority as Int B but higher priority than task A is requested. Since its priority is not higher than Int B, it is not activated yet. Note, if ISR_B globally disables interrupts (I-flag := 0), INT C is suspended regardless of its priority.
4. ISR_B is finished by RETI. The description S_A of task A is restored in from stack to registers by microcode (no program code). If ISR_B had disabled interrupts globally, they will be enabled again because old status was stored on stack.

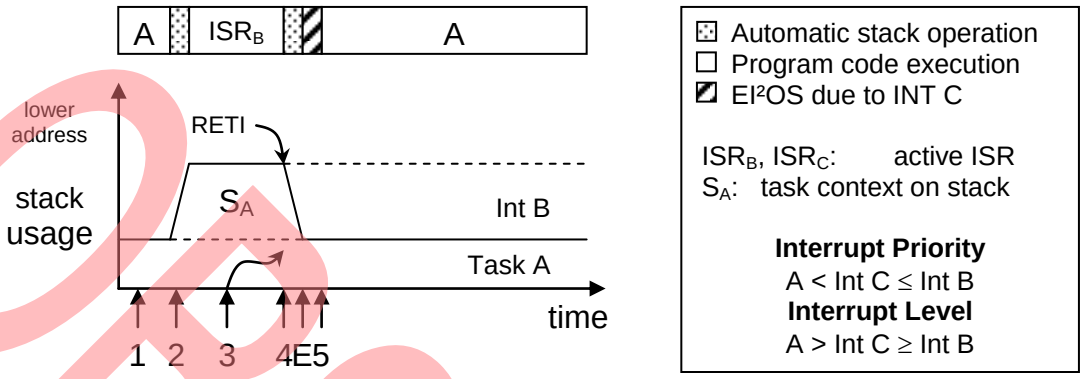
E. Int C is still pending. Since ISE bit in ICR of Int C is set (EI²OS enabled), task A is not continued yet but the EI²OS is started. One data byte or word is transferred from/to IO area and data pointers are updated. The data counter (DCT) of the EI²OS-descriptor is decreased.

OBSOLETE

5. If the new data counter value is NOT zero (service not finished), task A is continued.

Note: If L-family device were used, the context S_A would be restored after before EI²OS is performed.

Figure 5. Stack usage of suspended EI²OS of L-family



3.2.2 Behaviour of MB90500-Series of LX-Family (EI²OS not finished)

If the request '3' is caused by a resource with enabled EI²OS (ISE bit in ICR set), the EI²OS will be activated and the byte/word transfer will be performed **before** S_A is restored. Then DCT (data counter) of the EI²OS descriptor will be decreased. The later processing depends on the value of DCT. In case DCT is not 0 (service not finished), task A is restored (at '4').

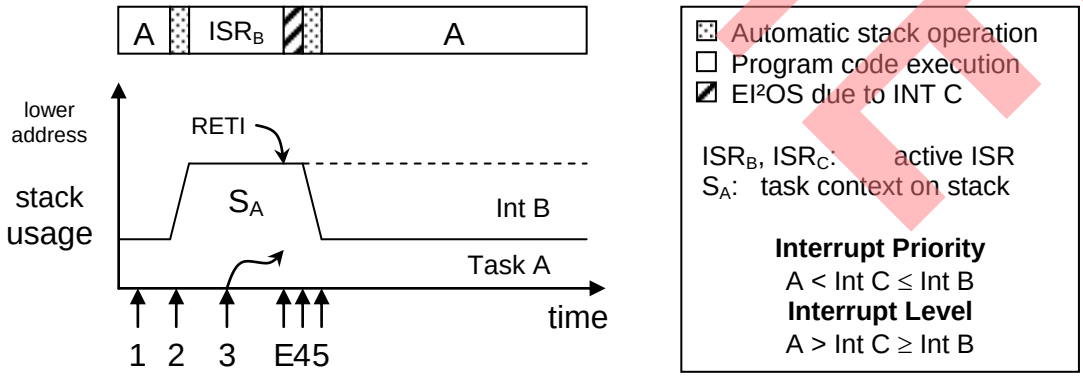
1. Any task/program (A) at any interrupt level is executed Interrupts are globally enabled.
2. An interrupt (Int B) of higher priority (lower level) than current task is activated. The current task is saved as S_A on stack and ISR_B is started.
3. Interrupt (Int C) of lower or same priority as Int B but higher priority than task A is requested. Since its priority is not higher than Int B, it is not activated yet. Note, if ISR_B globally disables interrupts (I-flag := 0), INT C is suspended regardless of its priority.

E. ISR_B is finished by RETI. S_A is not restored yet, because there is a waiting interrupt request. Since ISE bit in ICR of waiting interrupt Int C is set (EI²OS enabled), S_A is not restored yet but the EI²OS is started. One data byte or word is transferred from/to IO area and data pointers are updated. The data counter (DCT) of the EI²OS-descriptor is decreased.

4. If the new counter value is NOT zero (service not finished), operation continues by restoring context S_A of task A.
5. Task A is continued.

Note: If L-family device were used, the context S_A would be restored before EI²OS is performed.

Figure 6. Stack usage of suspended EI²OS of LX-family



OBSOLETE

3.3 Suspended Interrupt Request with EI²OS-Count complete

EI²OS-count complete means that the last byte/word had been transferred. Therefore, the EI²OS will execute an ISR of this resource as "completion interrupt". This allows the software to configure a new EI²OS for the next expected block of data.

In this case not only L- and LX-family behave differently but also MB90400- and MB90500 series (both LX-family) behave differently.

Note: Depending on resource (e.g. SIO) the ISR of the completion interrupt might be executed a second time. This is not subject of this application note. This happens, if the last byte transferred to the resource caused the completion interrupt, and the processing (e.g. transmission) of the last byte is finished later (transmit interrupt). This behaviour is correct and can be avoided by disabling resource IE-flag within completion interrupt.

3.3.1 Behaviour of L-family and MB90400-Series of LX-family (EI²OS Count complete)

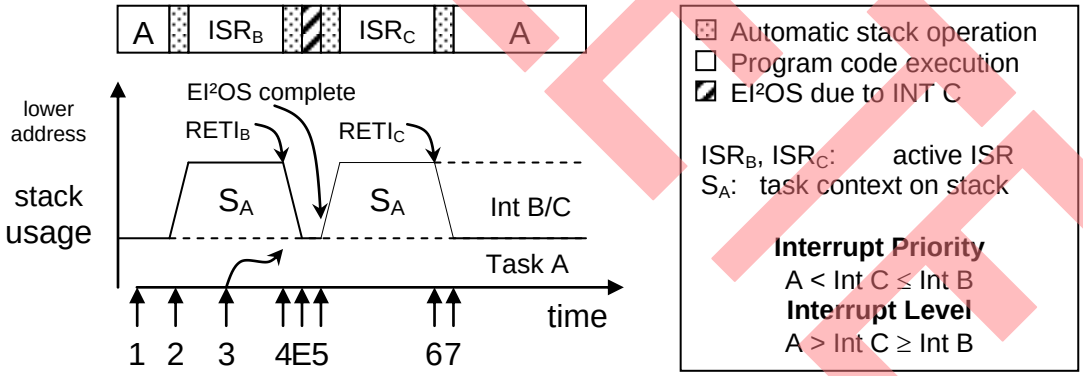
The L-family devices restore the stack before the EI²OS-transfer. If a completion interrupt is executed after the transfer, the same stack data will be saved to stack again.

1. Any task/program (A) at any interrupt level is executed Interrupts are globally enabled.
2. An interrupt (Int B) of higher priority than current task is activated. The current task is saved as S_A on stack and ISR_B is started.
3. Interrupt (Int C) of lower or same priority as Int B but higher priority than task A is requested. Since its priority is not higher than Int B, it is not activated yet. Note, if ISR_B globally disables interrupts (I-flag := 0), INT C is suspended regardless of its priority.
4. Interrupt ISR_B is finished by RETI. MB90600 devices immediately restore S_A. MB90400 devices first check whether

E. Interrupt Int_C is still pending. Since ISE bit in ICR of Int C is set, EI²OS service is started. One byte/word is copied from/to IO area and data pointers are updated. EI²OS counter and pointers are updated.

5. If the new counter is zero, the EI²OS completion interrupt is to be executed. Therefore, task A is save to stack S_A again. Both stacks S_A in Figure 7 are identical. There was no program code executed in-between '4' and '5'.
6. At the end of the ISR_C the instruction RETI is executed. The interrupt stack S_A is restored again.
7. Task A is continued

Figure 7. Stack usage of suspended EI²OS and completion ISR of LX-family



PROFET

3.3.2 Behaviour of MB90500-Series (EI²OS Count complete)

In case of a suspended EI²OS transfer with completion interrupt, MB90500-series devices partly behave as an L-family device. The stack is not restored before the EI²OS-transfer. However, if a completion interrupt is executed after the transfer, an additional interrupt stack is stored, but it should rather use the same interrupt stack or restore and store the same again.

Side Effects

- ISR_C is using a stack level, which uses 12 bytes more than expected by to its priority.
- The saved instruction pointer in S_B points to an instruction (RETI of ISR_B), which is expected to be completely finished already.
- RETI of ISR_B is executed a second time after the completion interrupt ISR_C has been finished.
- The saved processor status in S_B is the same as in S_A (see 4.1.2 Inconsistent Stack in Case of suspended EI²OS Completion Interrupt).

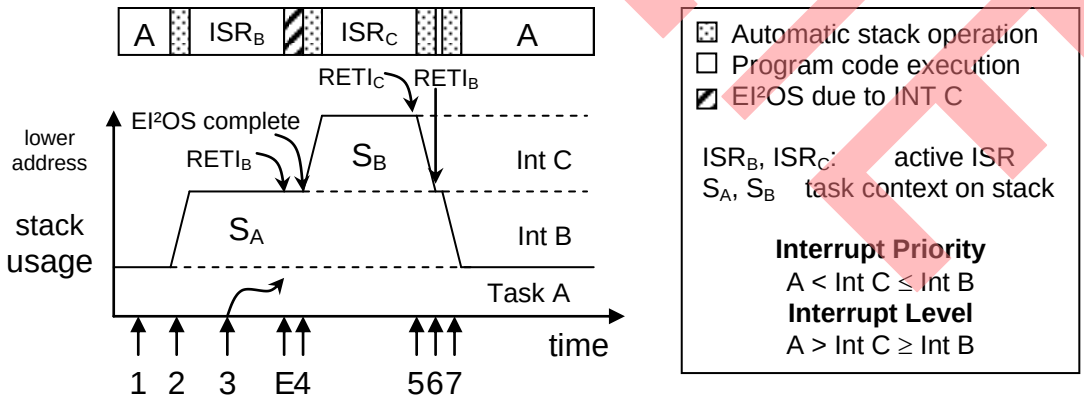
1. Any task/program (A) at any interrupt level is executed Interrupts are globally enabled.
2. An interrupt (Int B) of higher priority than current task is activated. The current task is saved as S_A on stack and ISR_B is started.
3. Interrupt (Int C) of lower or same priority as Int B but higher priority than task A is requested. Since its priority is not higher than Int B, it is not activated yet. Note, if ISR_B globally disables interrupts (I-flag := 0), INT C is suspended regardless of its priority.

E. Interrupt Int B is finished by RETI. The first word (PS) of the context S_A is written back to PS. Its I-flag and ILM is compared against ICR of the waiting interrupt request Int C. Since the interrupt level IL of ICR is lower* than ILM of S_A, the remaining context S_A of task A is not restored yet and the stack pointer remains unchanged. Instruction pointer still points to RETI of ISR_B.

4. Since, ISE bit in ICR of Int C is set, EI²OS service is started. One byte/word is copied from/to IO area and data pointers are updated. The EI²OS counter and pointer is updated.
5. If the new EI²OS-counter is zero, the current context is saved to S_B. The saved instruction pointer of the interrupt stack S_B points to the RETI instruction of ISR_B.
6. At the end of ISR_C the instruction $RETI_C$ is executed. S_B is restored and the instruction pointer points to $RETI_B$.
7. $RETI_B$ of ISR_B is fetched (as at 'E') and executed again. Now the interrupt stack S_A is restored to registers.
8. Task A is continued.

* lower level means higher priority

Figure 8. Stack usage of suspended EI²OS and completion ISR of MB90500-LX-family



OBSOLETE

4 Scheduler and Operating Systems

There are different ways that Scheduler and operating systems can work. They might have system calls, which modify the current stack of an interrupt. E.g. data areas could be copied. However, if these systems use the stack values of S_B in 0 or modify it, operation could fail. Following refers to the behaviour described under 0.

If the interrupt handler ISR_C is calling OS service routines, which clears the stack on its own and uses a context jump with data saved at modified return address, the address calculation may fail. If the OS relies on stack depth and that S_B would not be on top of S_A due to its priority, the OS might not restore S_A. Stack overflow may occur.

However, if an ISR is always executes RETI instruction to exit the interrupt level and if the saved instruction pointer (PCB:PC) on stack is not modified, the stack will be cleared and restored correctly.

4.1 Stack Analysis

4.1.1 Correct Stack in nested Interrupt

Normally, if there are two interrupt stacks on top of each other, the last stored one contains a smaller Interrupt Level Mask ILM.

In case "2.1 Behaviour of L/LX-Family with nested Interrupts" the first stack S_A holds the saved ILM (stored with processor status PS) of task A. The later S_B holds the ILM of Int B and Int C has its ILM in the actual PS register. The priority rises with stack depth. The ILM value decreases with stack depth.

$$ILM(S_A) > ILM(S_B) > ILM(PS\text{-register}) \quad (1)$$

Example 1:Program status after entering the ISR of a nested interrupt

```
>Show register ILM
ILM = 2

>Dump /w %SP
address   +0   +2   +4   +6   +8   +A   +C   +E    --- a s c i i --
000490    60E4 019E 00FF 0100 8002 0100 E0E0 012B  .`......*..@..+.
0004A0    00FF 0100 002A 4001 0000 02C6 00E6 00FF  ....*..@........

>disassemble FF019E
main.c$143     while (!USR0_TDRE);          // wait until the first byte is
sent and
FF019E 6C8421FC           BBC     I:21:4,FF019E
```

Example 1 shows a sample stack (see Table 2, Table 3, Table 4). The Task A is saved on stack as S_A from H'49C to H'4A6 due to Int B. The Int B is saved as S_B on stack from H'490 to H'49A due to Int C.

The current ISR_C is running at level 2 (ILM = 2). The previous interrupt was running at level 3, because the upper 3 bits of saved PS_B at address H'490 are B'011. The level of task A is 7, because the upper bits of saved PS_A at address H'49C are B'111.

The saved instruction pointer in S_B at H'492 points to code of the ISR_B at H'FF019E.

4.1.2 Inconsistent Stack in Case of suspended EI²OS Completion Interrupt

If the conditions of "0 " are met, the stack shows inconsistent data. The stack S_B holds mixture of status of Task A (saved PS) and status of ISR_B at the last instruction (other saved registers). However, the saved S_B is expected to be fully identical to S_A. The reason for inconsistency is the fact that at RETI (see event 'E') of ISR_B only PS of S_A is restored and ISR_C is executed before the other registers are restored.

In case "0 " the first stack S_A holds the saved ILM (stored with processor status PS) of task A. The later S_B also holds the ILM of Task A. Int C has its ILM in the actual PS register. The priority does not rise with stack depth. The ILM value does not decrease with stack depth.

$$ILM(S_A) = ILM(S_B) > ILM(PS\text{-register}) \quad (2)$$

OBSOLETE

Example 2:Program status after entering the ISR of a nested interrupt

```
>Show register ILM
ILM = 2

>Dump /w %SP
address   +0   +2   +4   +6   +8   +A   +C   +E    --- a s c i i --
000490    E0E0 0199 00FF 0100 7FFF 0100 E0E0 012B  ........*..@..+.
0004A0    00FF 0100 002A 4001 0000 02C2 00E6 00FF  ....*..@........

>Disassemble FF0199
main.c$138 }
FF0199 6B                RETI
```

Example 2 shows a sample stack (see Table 2, Table 3, Table 4). The Task A is saved on stack as S_A from H'49C to H'4A6 due to Int B. The Int B is saved as S_B on stack from H'490 to H'49A due to Int C + EI²OS. Normally, Int_C should cause the Task A to be saved again from H'49C to H'4A6.

The current ISR_C is running at level 2 (ILM = 2). The previous interrupt was running at same or lower level than ISR_B (or at any level but with interrupts globally disabled). However, saved PS of S_B is the same as of S_A. The level of task A is 7, because the upper bits of saved PS_A at address H'49C are B'111.

The saved instruction pointer in S_B at H'492 points to RETI of the ISR_B at H'FF0199.

Note: *Interrupt stack inconsistency is not a problem at all, if no* modifications *are done to the stack by the program. At the end of the EI²OS completion interrupt the stack area SB and the stack area SA are correctly restored.*

4.2 Workaround Programs

In order to achieve correct operation, even if modifications are done to the data on stack, the use of workaround code is possible.

The code will check the interrupt stack for the code it is pointing to by instruction pointer (PCB:PC in Table 2). If it is RETI (return from interrupt) the stack is corrected by increasing the stack pointer.

To use the code it is necessary to have the stack pointer pointing to the PS of the interrupt stack (PCB:PC in Table 2). Therefore, PUSHW or LINK must not be executed after entering the interrupt. However, there is only limited control of this, if C-language is used.

To avoid register saving in C with FCC907-compiler the __nosavereg specifier must be applied in addition to the __interrupt specifier of the function. In that case the #pragma register(x) is necessary to assign a separated register bank x to the function.

There is no save way to avoid LINK instruction. LINK sets a function frame (e.g. if local variables are used). Very small function without local variables and no parameters might not use the LINK. However, this is out of control, if the code is changed.

Therefore, an Assembly function should be used, which corrects the stack and then jumps to the C-language function (see Example 3). The Assembly function must be assigned to the interrupt vector.

The combination of Example 3 and Example 5 should best fit for Medium model C-language programs.

OBSOLETE

Example 3:C-language frame for the workaround code

```
__interrupt void AssemblyISR(void);       // prototype of AssemblyISR
#pragma intvect AssemblyISR 37            // assign function to interrupt
                                             (e.g. #37)

#pragma asm
  .GLOBAL _AssemblyISR
  .IMPORT _C_ISR
_AssemblyISR:                             // Assembly routine
    Stack Correction code                 // work around code
  JMPP _C_ISR                             // jump to C-code
#pragma endasm

__interrupt
void C_ISR(void)                          // normal interrupt handler
{
  AnyResource.IntReq = 0;                 // clear interrupt request
}
```

4.2.1 Stack Correction Program for Code Size less than 64KB

For programs with less than 64 KB code it is sufficient to read only the 16Bit-offset of the instruction pointer from stack. Since PCB (upper 8 address bits) remains unchanged, PCB can be used for reading the instruction at the return address (line 5 in Example 4).

Example 4:Workaround code for programs with less than 64 KB code

```
1. AND                CCR, #0xBF          // diasabele interrupts
2. MOVW               A, SP               // load stackpointer
3. ADDW               A, #2               // point to instruction pointer
4. MOVW               A, SPB:@A           // read program counter from stack
5. MOV                A, PCB:@A           // read instruction at return
                                             address
6. CBNE               A, #0x6B,normal1    // is it RETI?
7. ADDSP              #0x0C               // if yes, forward stack
8. normal1:
9. OR                 CCR, #0x40          // enabel interrupts again
```

4.2.2 Stack Correction Program for Code Size less than 64KB

For programs with more than 64 KB code it is necessary to change program bank as well. Therefore, the saved program bank on stack is loaded to ADB register. This might modify the value of ADB (see Example 5). However, ADB is properly restored when returning from interrupt.

Note: If the interrupt handler relies on ADB being the same as it was in the interrupted program, this workaround cannot be used.

OBSOLETE

Example 5:Workaround code for programs with more than 64 KB code and broken ADB

```
  AND  CCR, #0xBF          // diasabele interrupts
  MOVW A, SP               // load stackpointer
  ADDW A, #4               // point to program counter bank
  MOV  A, SPB:@A           // move AL->AH, load PCB to AL
  MOV  ADB, A              // set bank to ADB
  MOVW A, SP               // load stackpointer again
  ADDW A, #2               // point to program counter
  MOVW A, SPB:@A           // read program counter from stack
  MOV  A, ADB:@A           // read instruction from program
  CBNE A, #0x6B,normal2    // is it RETI?
  ADDSP#0x0C               // if yes, point to next int stack
normal2:
  OR CCR, #0x40            // enabel interrupts again
```

4.2.3 Stack Correction Program for Code Size bigger than 64KB and ADB must not be broken

For programs with more than 64 KB code it is necessary to change program bank. Therefore, the saved program bank on stack is loaded to ADB register (see Example 6). This might modify the value of ADB. Therefore, ADB is restored before jumping to the real ISR.

Note: If the interrupt handler relies on ADB being the same as it was in the interrupted program, this workaround should be used.

Example 6:Workaround code for programs with more than 64 KB code and unbroken ADB

```
  AND  CCR, #0xBF          // diasabele interrupts
  MOVW A, SP               // load stackpointer
  ADDW A, #4               // point to program counter bank
  MOV  A, SPB:@A           // load PCB
  MOV  ADB, A              // set bank to ADB
  MOVW A, SP               // load stackpointer again
  ADDW A, #2               // point to program counter
  MOVW A, SPB:@A           // read program counter from stack
  MOV  A, ADB:@A           // read instruction at return address
  MOVW A, SP               // load stackpointer to AL and previous AL to AH
  ADDW A, #6               // point to ADB
  MOV  A, SPB:@A           // read prevous ADB value
  MOV  ADB, A              // restore ADB
  SWAPW                    // reload ADB pointer from AH to AL
  CBNE A, #0x6B,normal3    // is it RETI?
  ADDSP#0x0C               // if yes, point to next int stack
normal3:
  OR CCR, #0x40            // enabel interrupts again
```

OBSOLETE

5 Conclusion

There is no problem with MB90600 series (L-family) and MB90400-series (LX-family). However, with MB90500-series (LX-family) following items needs to be considered in order to avoid misoperation (e.g. stack overflow):

- The data automatically saved by hardware interrupts should not be modified.
- Each interrupt should be finished by RETI instruction.

Otherwise, described workaround should be added to the EI²OS completion interrupt.

This applies only, if followings conditions are met (AND):

1. A interrupt request of lower or same priority occurs, while a program is executed.
2. This waiting request causes EI²OS to be executed.
3. EI²OS counter becomes zero. (Means that the last iteration has been done.) Therefore, it is calling the interrupt handler.

If one of the following conditions are met, normal behaviour can be expected (OR):

1. EI²OS is not used.
2. EI²OS had been executed but the counter did NOT become zero.

OBSOLETE

6 Document History

Document Title: AN205047 - Using EI²OS on MB90500 Series Devices

Document Number: 002-05047

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	-	NOFL	12/17/1997	Initial release
			08/25/2014	Completely revised, graphics improved Condition of necessity of INTP instruction for critical behaviour is not correct, has been removed Behaviour of MB90400 series added Application note tile now refers to MB90500 only (not to all LXfamily) Workaround code included in application note
*A	5129684	NOFL	02/25/2016	Migrated Spansion Application note from MCU-AN-390043-E-V20 **to** Cypress format This product is old and this AN to be Obsoleted

OBSOLETE

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

ARM® Cortex® Microcontrollers	cypress.com/arm
Automotive	cypress.com/automotive
Clocks & Buffers	cypress.com/clocks
Interface	cypress.com/interface
Lighting & Power Control	cypress.com/powerpsoc
Memory	cypress.com/memory
PSoC	cypress.com/psoc
Touch Sensing	cypress.com/touch
USB Controllers	cypress.com/usb
Wireless/RF	cypress.com/wireless

PSoC® Solutions

psoc.cypress.com/solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP

Cypress Developer Community

Community | Forums | Blogs | Video | Training

Technical Support

cypress.com/go/support

OBSOLETE

PSoC is a registered trademark and PSoC Creator is a trademark of Cypress Semiconductor Corp. All other trademarks or registered trademarks referenced herein are the property of their respective owners.

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709

Phone : 408-943-2600
Fax : 408-943-4730
Website : www.cypress.com

© Cypress Semiconductor Corporation, 1997-2016. This document is the property of Cypress Semiconductor Corporation and its subsidiaries, including Spansion LLC ("Cypress"). This document, including any software or firmware included or referenced in this document ("Software"), is owned by Cypress under the intellectual property laws and treaties of the United States and other countries worldwide. Cypress reserves all rights under such laws and treaties and does not, except as specifically stated in this paragraph, grant any license under its patents, copyrights, trademarks, or other intellectual property rights. If the Software is not accompanied by a license agreement and you do not otherwise have a written agreement with Cypress governing the use of the Software, then Cypress hereby grants you under its copyright rights in the Software, a personal, non-exclusive, nontransferable license (without the right to sublicense) (a) for Software provided in source code form, to modify and reproduce the Software solely for use with Cypress hardware products, only internally within your organization, and (b) to distribute the Software in binary code form externally to end users (either directly or indirectly through resellers and distributors), solely for use on Cypress hardware product units. Cypress also grants you a personal, non-exclusive, nontransferable, license (without the right to sublicense) under those claims of Cypress's patents that are infringed by the Software (as provided by Cypress, unmodified) to make, use, distribute, and import the Software solely to the minimum extent that is necessary for you to exercise your rights under the copyright license granted in the previous sentence. Any other use, reproduction, modification, translation, or compilation of the Software is prohibited.

CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS DOCUMENT OR ANY SOFTWARE, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Cypress reserves the right to make changes to this document without further notice. Cypress does not assume any liability arising out of the application or use of any product or circuit described in this document. Any information provided in this document, including any sample design information or programming code, is provided only for reference purposes. It is the responsibility of the user of this document to properly design, program, and test the functionality and safety of any application made of this information and any resulting product. Cypress products are not designed, intended, or authorized for use as critical components in systems designed or intended for the operation of weapons, weapons systems, nuclear installations, life-support devices or systems, other medical devices or systems (including resuscitation equipment and surgical implants), pollution control or hazardous substances management, or other uses where the failure of the device or system could cause personal injury, death, or property damage ("Unintended Uses"). A critical component is any component of a device or system whose failure to perform can be reasonably expected to cause the failure of the device or system, or to affect its safety or effectiveness. Cypress is not liable, in whole or in part, and Company shall and hereby does release Cypress from any claim, damage, or other liability arising from or related to all Unintended Uses of Cypress products. Company shall indemnify and hold Cypress harmless from and against all claims, costs, damages, and other liabilities, including claims for personal injury or death, arising from or related to any Unintended Uses of Cypress products.

Cypress, the Cypress logo, Spansion, the Spansion logo, and combinations thereof, PSoC, CapSense, EZ-USB, F-RAM, and Traveo are trademarks or registered trademarks of Cypress in the United States and other countries. For a more complete list of Cypress trademarks, visit cypress.com. Other names and brands may be claimed as property of their respective owners.