Freescale Semiconductor
Engineering Bulletin

Document Number:EB770
Rev. 1, August 2012

Managing Low-Power Mode Errata in MPC56xx Devices

Contents

1 Introduction

This engineering bulletin is intended to help users implement a software workaround for low-power mode errata that affect Freescale's MPC56xx devices. This includes errata e3570, e3584, e3950, and e5099. These errata affect many MPC56xx devices when executing from flash memory and entering a low-power mode (LPM) where the flash memory will be in low-power or power-down configuration. Different errata exist depending on the available modes on each specific device.

There is a window of two to four clock cycles at the beginning of the Run mode to LPM transition during which a WKUP (wakeup) or interrupt will generate an exception state due to the flash memory not being available at the Run mode re-entry. For e200z0 cores this will be either a checkstop reset or machine check interrupt if the Machine Check Enable bit of the core's Machine State Register is set (MSR[ME] = 1). For e200z4 cores, there is no checkstop state and a machine check interrupt will occur.

1.1 The failing sequence

1. The Mode Entry module (MC_ME) requests that the flash memory enter the power-down/low-power state.
2. Synchronisation logic between MC_ME and the flash memory results in the Mode Entry Global Status flags for Code and Data Flash (ME_GS[S_DFLA/S_CFLA])

not being updated until the flash memory signals back to MC_ME that the low-power transition is fully complete.

3. A 2-4 clock cycle window therefore exists where the flash memory may be in the power-down/low-power state but MC_ME has not yet been signalled that the transition has completed.
4. During this window, if a WKPU or interrupt is received, MC_ME presumes the flash memory is available and allows the core to execute from flash memory, which is not available. The bus error from the flash memory results in the core entering the checkstop state.
5. To resynchronise the correct flash memory state at MC_ME, the flash memory needs to re-enter the intended power-down/low-power mode and then return to Normal mode.

1.2 Workaround options

This issue can be handled in one of these ways if the checkstop reset must be avoided:

- Workaround #1 configures the application to generate a machine check interrupt instead of a checkstop reset and to handle the machine check interrupt in RAM. This approach deals with the problem only if it occurs.
- Workaround #2 configures the MCU to avoid the conditions that can cause the checkstop reset or machine check interrupt.
- Workaround #3 configures the MCU to manage the machine check exception at flash and you must then wait until the flash successfully reinitializes (MPC5645S only).

There is no requirement to implement any of these three workarounds where the LPM transition is handled by an e200z0 core. In the event that no workaround is implemented, parts in this configuration core will reset and follow the normal recovery process from a checkstop condition. In the e200z0 case, if the use of the part is unlikely to generate the conditions necessary to activate the issue — for example, the WKPU is triggered infrequently and beyond the vulnerable window, or if the time taken to reset is acceptable within the application — the problem can be acknowledged and addressed without implementing any of these workarounds.

For parts managing the LPM transition using an e200z4 core, the checkstop reset state is not available and a machine check interrupt will occur.

1.3 Abbreviations

The abbreviations listed below are used within this document.

Table 1. Abbreviations

API	Autonomous Periodic Interrupt
APIVAL	API Value
EE	External Interrupt Enable bit (in MSR)
FIRC	Fast Internal RC clock
Flash	CFLASH or DFLASH (wherever application code resides)
IVOR	Interrupt Vector Offset Register
IVPR	Interrupt Vector Prefix Register
LPM	Low-Power mode (for example, HALT, STOP, and STANDBY modes)
MAV	MCAR Address Valid
MC_ME	Mode Entry module
MCAR	Machine Check Address Register
MCSR	Machine Check Syndrome Register

Table continues on the next page...

Table 1. Abbreviations (continued)

MCU	Microcontroller Unit
ME	Machine Check Enable bit (in MSR)
MSR	Machine State Register
RAM	Random Access Memory
RTC	Real-Time Clock
RTCC	Real-Time Clock Control register
RUN	Run mode (for example, DRUN, RUN[0:3])
WKPU	Wakeup Unit
WKUP	Wakeup signal

2 Re-creating the issue

The issue can be recreated by using the Autonomous Periodic Interrupt (API) with a short timeout value. The steps to set up the conditions are:

1. Configure API.
    a. Select FIRC as RTC/API clock source.
    b. Disable divide by 512 and divide by 32.
    c. Select RTC_RTCC[APIVAL] for a small value. It is best to try a sweep from 4–30 to find the particular value that causes the checkstop state (note the APIVAL will vary due to variance of FIRC).
2. Select API as a WKPU source (WKUP0).
3. Enter Stop mode.
4. At the API WKPU event a checkstop event will be triggered when the MCU is woken for the case when the flash memory is not available.

Figure 1 shows the MCU entering and exiting Stop mode using the increasing timeout for RTC_RTCC[APIVAL]. Note that:

1. The checkstop reset is generated for RTC_RTCC[APIVAL] = 10.
2. For small API timeout values the MCU returns to Run mode in a short time because the flash memory has not entered power-down mode.
3. For the particular sample (MPC5607B, 1M03Y, XAA1043) used to capture this data, an RTC_RTCC[APIVAL] of 9 and 10 generated the checkstop reset.

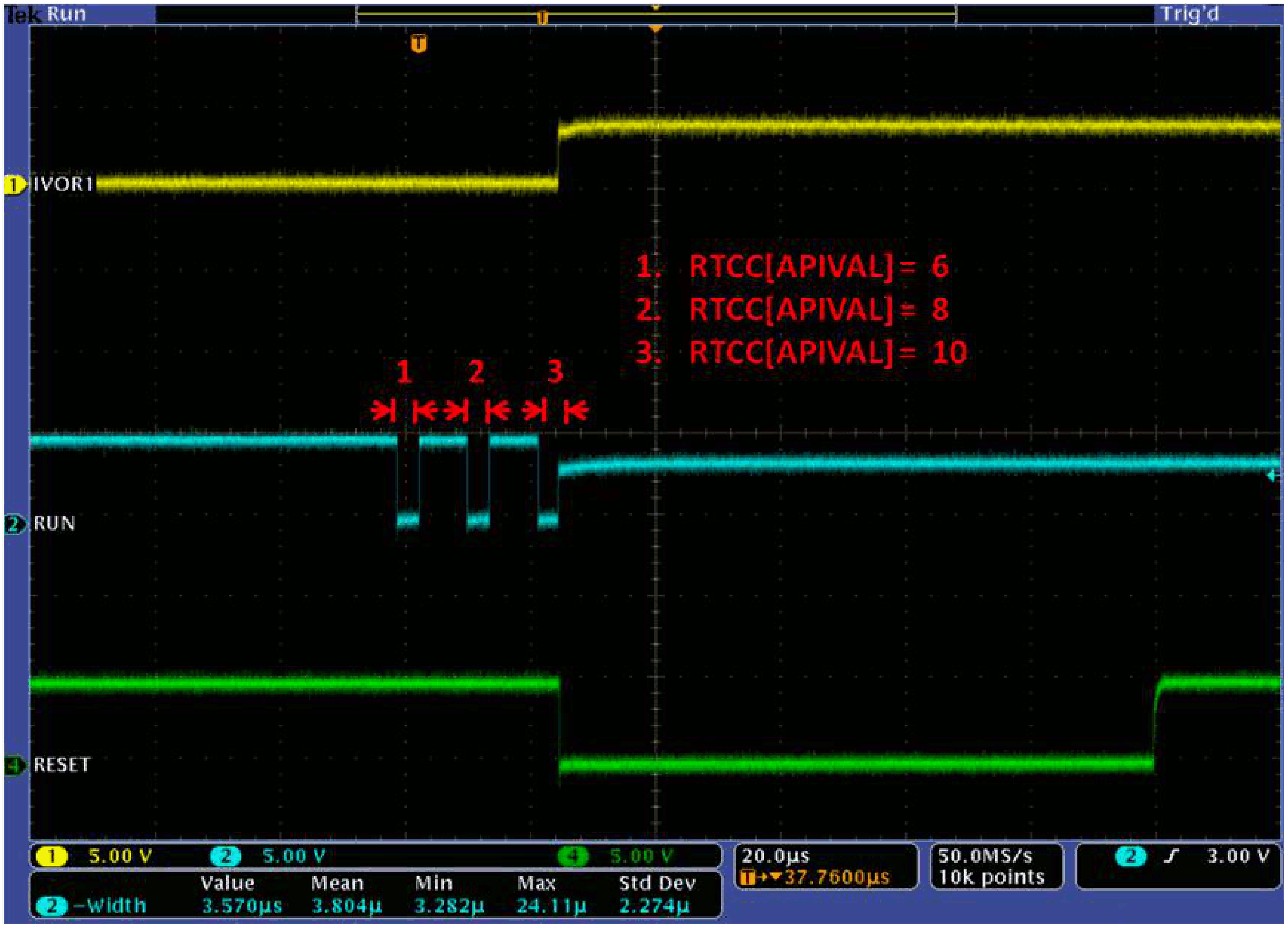


Figure 1. Sweeping WKPU timeout to cause checkstop reset

Figure 2 shows an API timeout that is large enough to allow correct signalling between flash memory and MC_ME, avoiding the checkstop reset condition. Note the MCU remains in Stop mode for a longer period because the flash memory has entered power-down state and more time is required for it to return to Normal mode. The flash memory power-up time ($T_{FLAPDEXIT}$) is specified in the data sheet.

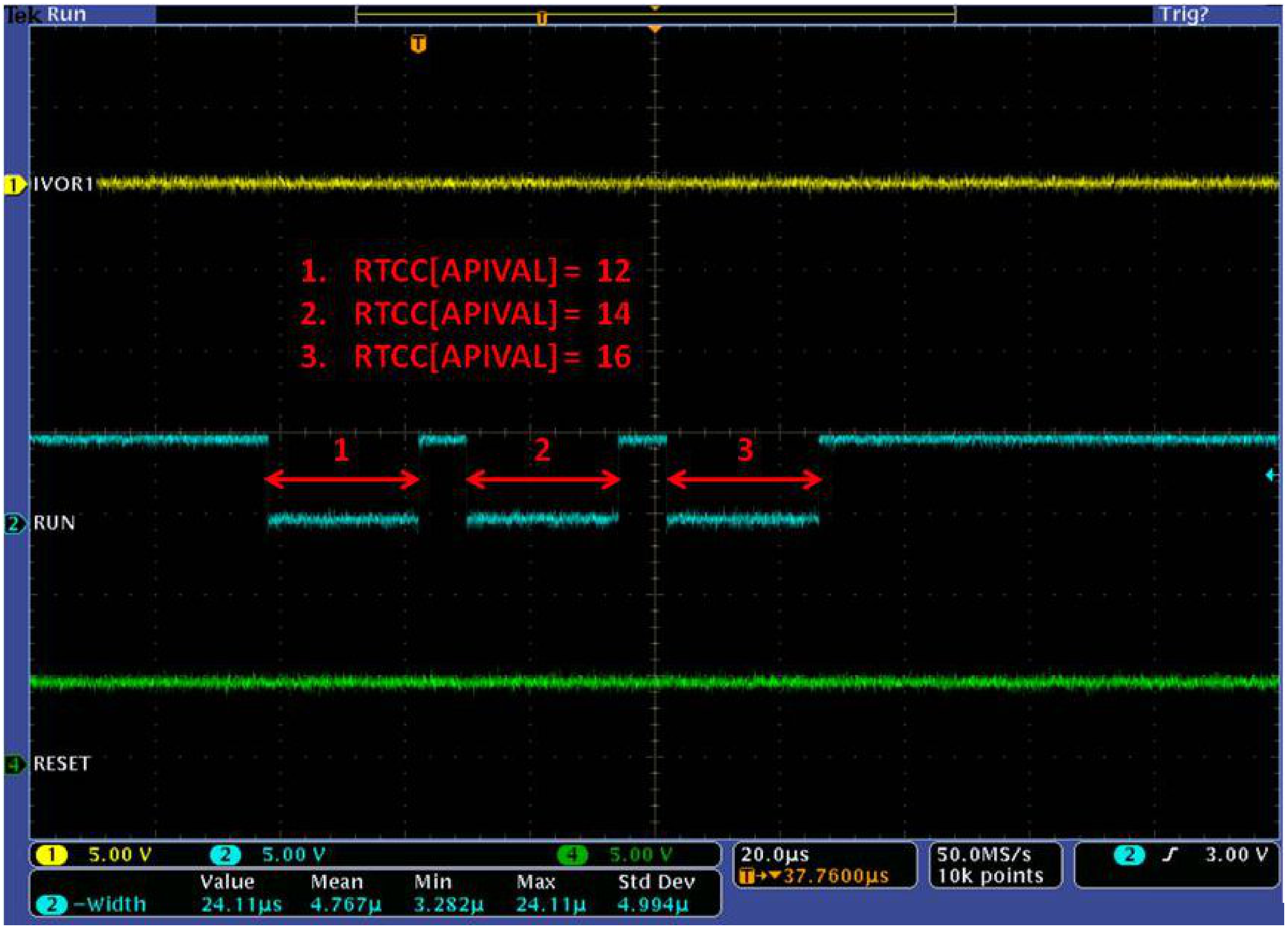


Figure 2. WKPU timeout above that required to cause reset

3 Workaround 1

The premise of Workaround 1 is to deal with the issue as it occurs: that is, enter and exit LPM as normal executing application code from flash memory. However, in the event the MCU exits LPM and the flash memory is not available, the next instruction fetch to flash memory will result in the core checkstop state as a result of a bus error. The checkstop state will generate a checkstop reset unless the machine check exception is enabled at the core. In the case of this problem occurring and the machine check exception being enabled, the application software can handle this event if the exception handler code is placed in RAM.

3.1 Enable Machine Check Exception

The machine check enable (ME) is set at the machine state register field MSR[ME]. The following assembly code reads the MSR and sets the ME field:

```
mfmsr r(X)
e_or2i r(X), 0x1000
mtmsr r(X)
```

On capturing the machine check for the outlined condition the machine check syndrome register will be set to 0x0000_0010 to indicate "Read Bus Error on Instruction Fetch" — MCSR[BUS_IRERR].

NOTE

Enabling the machine check interrupt at MSR[ME] only applies for e200z0 cores (for example, MPC560xB/C). For the e200z4 cores (such as MPC564xB/C) the checkstop state is not supported and therefore the bus error condition will always generate the machine check interrupt — therefore MSR[ME] need not be set. For cores other than e200z0 and e200z4, consult the relevant Zen core reference manual for checkstop feature implementation.

3.2 Copy IVOR vector table to RAM

Prior to LPM entry the interrupt vector table must be available in RAM. For this the user has two options.

The first option is to have a permanent copy of the vector table in RAM. This can be configured at startup during RAM initialization. The second option is to copy the interrupt vector table to RAM prior to every LPM entry. Option 1 has the benefit of reduced runtime because the vector table is only copied once. However, this will impact code size because a section of RAM will have to be permanently dedicated to the interrupt vector table. Option 2 will allow more flexible use of the RAM — however, moving code at every LPM entry will have a substantial software overhead. Therefore choosing the best option depends on which factor is more important in a particular situation. Select option 1 to optimize for execution time, or select option 2 to optimize for code size.

3.3 Update IVPR to point to base address of interrupt vector table

The Interrupt Vector Prefix Register (IVPR) is used during interrupt processing for determining the starting address of the software handler used to handle an interrupt. The hard-wired interrupt vector offset value for a particular interrupt type is added to the value held in the interrupt vector prefix register (IVPR) to form the instruction address from which execution is to begin.

Note that the machine check interrupt IVOR 1 has an offset of 0x010 — therefore the machine check interrupt code will reside at address {IVPR[0:19], 0x010}. IVPR can be set using the following assembly code:

```
e_lis r(x),  {Interrupt Vector Table Base Adresss[0:15]}
e_or2i r(x), {Interrupt Vector Table Base Adresss[16:31]}
mtIVPR r(X)
```

NOTE

e200z4 cores have dedicated offset registers for all IVORs, and therefore the machine check interrupt code should reside at address {IVPR[0:19], IVOR1}.

3.4 Machine Check Interrupt Routine

When the machine check interrupt has been asserted as a result of entering RUN mode after LPM exit and the flash memory is not available, the flash memory will have to be re-enabled prior to continuing the regular software flow. It is only possible to re-enable the flash memory by placing it in power-down/low-power mode and then setting it for normal mode. This involves two Run mode changes

The first mode change is Run[flash = power-down/low-power] followed by Run[flash = normal]. The selection of power-down/low-power mode is dependent on the LPM flash memory configuration. For example:

1. If LPM[flash = power-down] was selected, then mode changes of Run[flash = power-down] and Run[flash = normal] are required.
2. If LPM[flash = low-power] was selected, then mode changes of Run[flash = low-power] and Run[flash = normal] are required.

3.5 Example code for Machine Check Interrupt

Prior to the execution of the following example code, the core register context will have to be stored (also known as prolog), and after execution the core register context will have to be restored (epilog). Note that the entire interrupt routine could be written in assembly code to optimize for code size and speed of execution, and that in this code, ME refers to the Mode Entry module, MC_ME.

1. To recover from LPM[flash = Power-down]

```
ME.RUN[0].R = 0x00150010;          /* FLASH power down */
ME.MCTL.R   = 0x40005AF0;          /* Mode & Key */
ME.MCTL.R   = 0x4000A50F;          /* Mode & Key inverted */
/* Wait for mode change to complete - FLASH power down */
while(ME.GS.B.S_MTRANS == 1);
ME.RUN[0].R = 0x001F0010;          /* FLASH normal */
ME.MCTL.R   = 0x40005AF0;          /* Mode & Key */
ME.MCTL.R   = 0x4000A50F;          /* Mode & Key inverted */
/* Wait for mode change to complete - FLASH normal */
while(ME.GS.B.S_MTRANS == 1);
```

2. To recover from LPM[flash = Low-power]

```
ME.RUN[0].R = 0x001A0010;          /* FLASH low power */
ME.MCTL.R   = 0x40005AF0;          /* Mode & Key */
ME.MCTL.R   = 0x4000A50F;          /* Mode & Key inverted */
/* Wait for mode change to complete - FLASH power down */
while(ME.GS.B.S_MTRANS == 1);
ME.RUN[0].R = 0x001F0010;          /* FLASH normal */
ME.MCTL.R   = 0x40005AF0;          /* Mode & Key */
ME.MCTL.R   = 0x4000A50F;          /* Mode & Key inverted */
/* Wait for mode change to complete - FLASH normal */
while(ME.GS.B.S_MTRANS == 1);
```

3.6 Considerations for Workaround 1

The user will have to decide whether to implement interrupt handlers in RAM for interrupt vectors other than IVOR1. The Zen core architecture does help as it limits the interrupt options. When the machine check interrupt is executed the following occurs:

- MSR[EE] = 0, external interrupts (IVOR4) are automatically disabled. Therefore the user does not need to implement interrupt handlers for all peripherals.
- MSR[ME] = 0, machine check interrupt (IVOR1) is automatically disabled. Therefore the user does not need to be concerned with another checkstop state because this will automatically generate a checkstop reset. However, the user does have the option to re-enable MSR[ME] and handle the second interrupt.

The user can decide to execute interrupt handlers for the remaining core interrupts but it is suggested that for this workaround if another interrupt occurs the interrupt routine executes a software reset. This strategy helps to limit the size of code required in RAM to implement the interrupt vector table and interrupt service routines.

3.7 Timing Impact for Workaround 1

When the machine check interrupt is called, the flash memory is already in power-down or low-power mode. Therefore the first mode change to configure the flash memory is immediate. The second mode change to set the flash memory to normal mode from power-down or low-power mode takes the time as specified in the data sheet. The time for the flash memory to return to normal mode is the same whether it is initiated by the machine check interrupt or by the MCU WKPU module.

Therefore there is little impact to the application runtime. The only overhead to the software is the number of cycles it takes to implement the machine check interrupt handler. If the handler is written in assembly code and is optimized to perform the minimum save and restore of core registers, the cycle count is less than 200 clock cycles (12.5 µS).

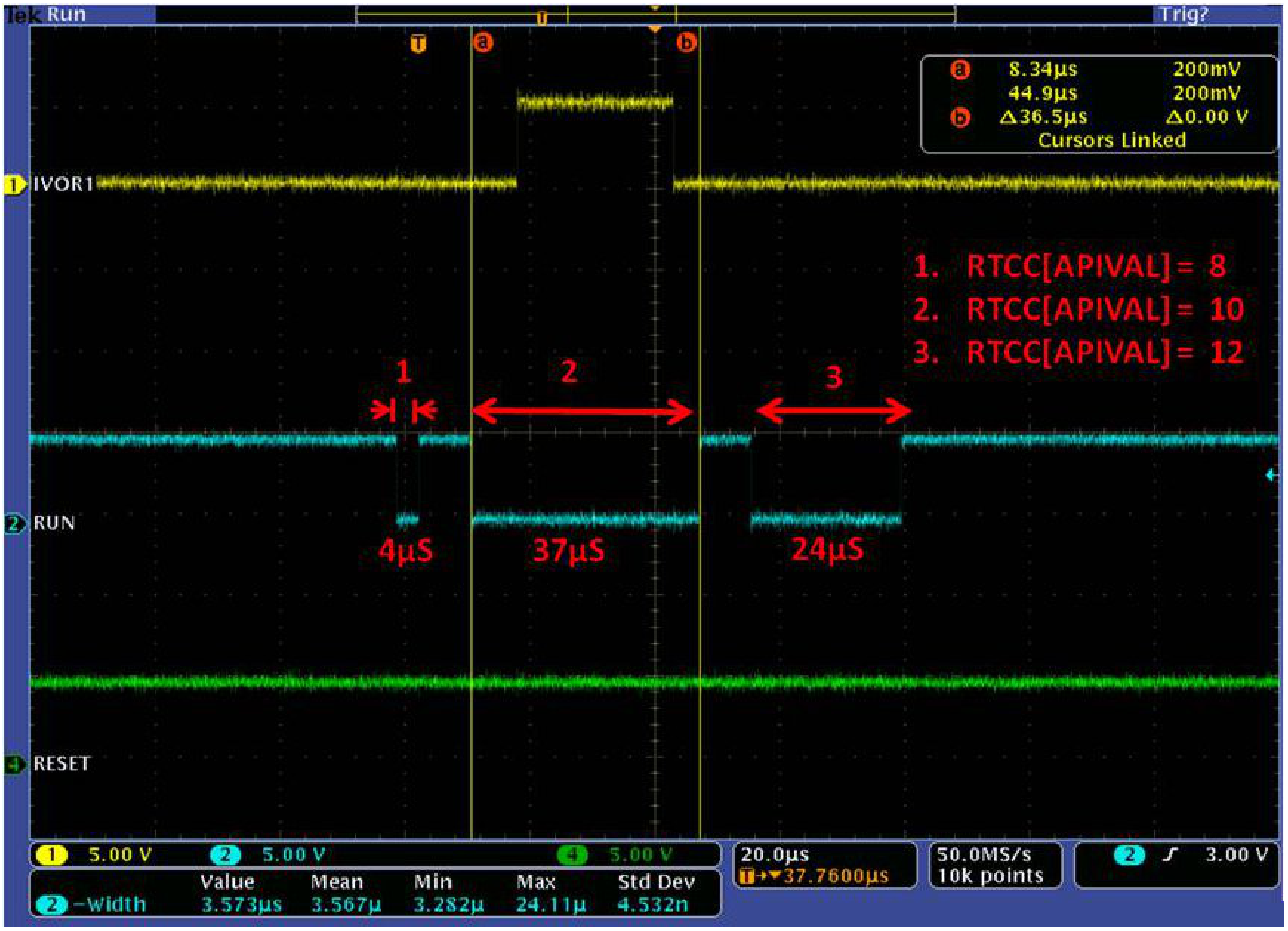


Figure 3. Timing impacts for workaround 1

Figure 3 shows the following timings:

1. API WKUP returns the MCU to Run mode before the flash memory is powered down.
2. API WKUP returns the MCU to Run mode after the flash memory is powered down. The first instruction in Run mode causes a machine check interrupt and IVOR1 is executed.
3. API WKUP returns the MCU to Run mode after the flash memory is powered down. The flash memory is returned to normal mode prior to the first instruction in Run mode.

NOTE

The difference in time between low-pulse 1 and low-pulse 3 in Figure 3 is the delay for the flash memory to exit power-down mode, defined as $T_{FLAPDEXIT}$ in the data sheet.

NOTE

The difference in time between low-pulse 2 and low-pulse 3 in Figure 3 is not a delay for the flash memory to enter or exit power-down mode, it is the execution time for the machine check exception handler.

3.8 Optimised Machine Check Interrupt

The scope trace in Figure 4 is similar to the one in Figure 3. However, section 2 is shorter because the machine check interrupt has been optimized to minimum core context save/restore. The software overhead is reduced to 100 clock cycles (8 µS) so that fewer than 100 instructions are required to implement the IVOR1 interrupt handler.

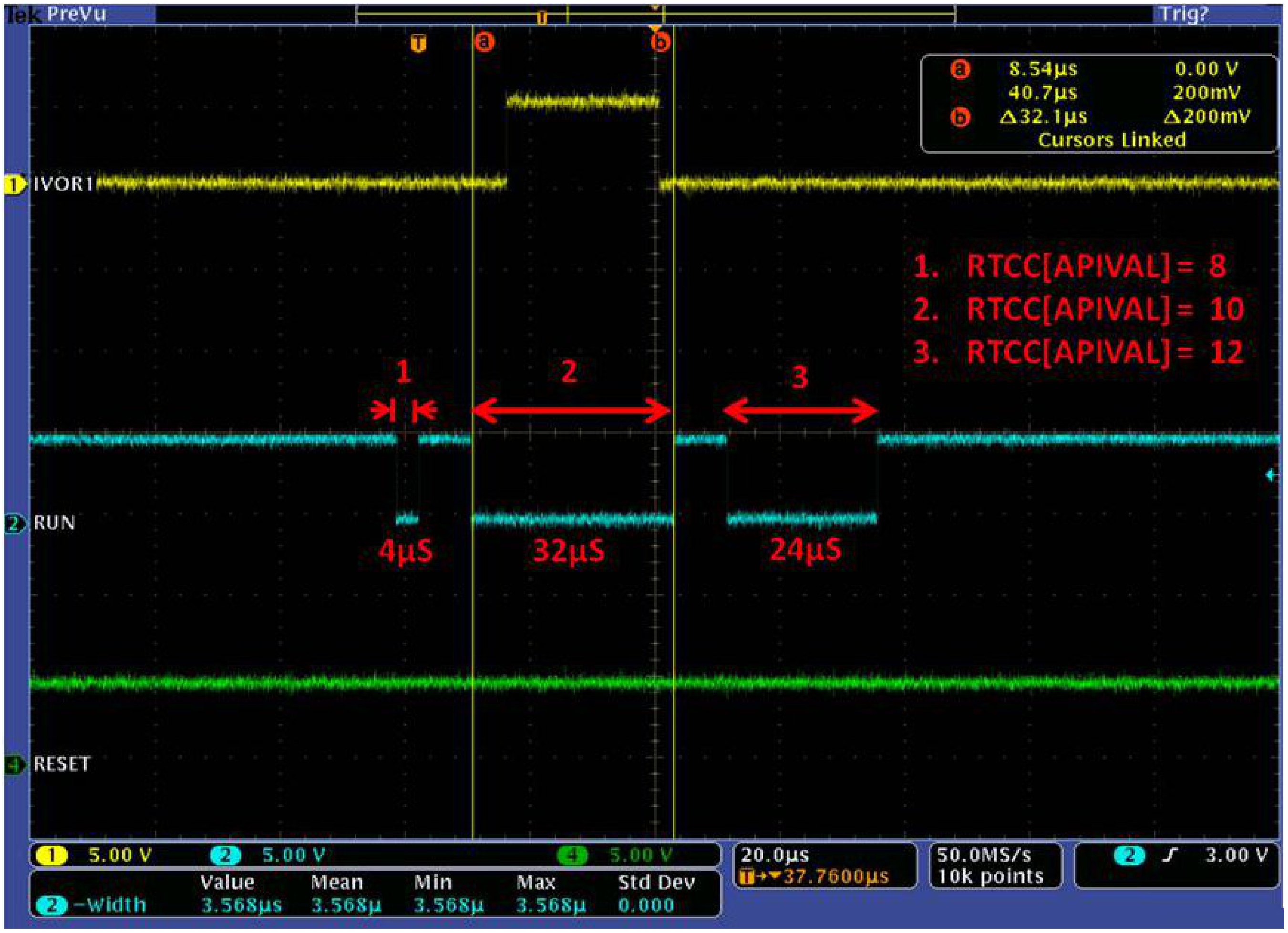


Figure 4. Optimised Machine Check Interrupt timing

4 Workaround 2

The application can be configured to avoid the checkstop reset or machine check interrupt. To do this the code initiating the LPM transition has be executed in RAM. By this means, if the LPM mode is exited prior to the flash memory being returned to normal mode, any instruction fetch will be from the available RAM rather than the unavailable flash memory. The application software can then re-initialize the flash memory while executing safely in RAM. When the flash memory is ready, code execution can then return to the flash memory.

4.1 Interrupt Handling

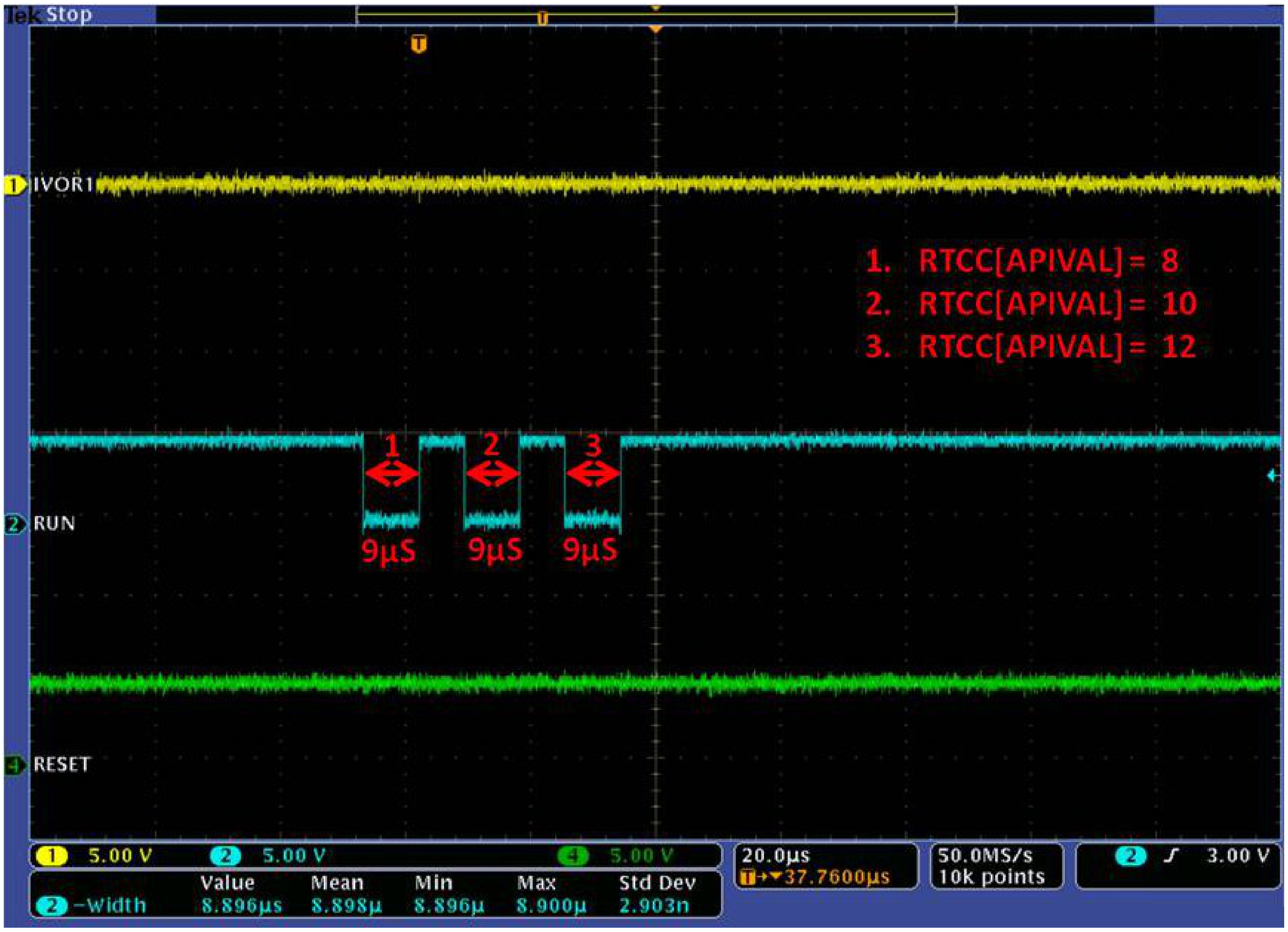
While executing from RAM while the flash memory is not available, the core is not able to handle any core interrupts. Peripheral interrupts (IVOR4) can be disabled at the core by clearing the external interrupt field at the machine status register (MSR[EE] = 0). However, all other core interrupts will result in a checkstop reset as the core will not be able to access the interrupt vector table located in the flash memory.

4.2 Timing Impact for Workaround 2

The idea is to match the post-LPM Run mode flash memory configuration with the LPM flash configuration, so that exit from LPM is as quick as possible. Once running in RAM, the flash memory can be configured to Normal mode.

1. API timeout = 8. The flash memory does not get to Low-power mode but the flash memory is power-cycled after wakeup to ensure correct flash memory status. This very early wakeup would normally wake the MCU before the flash memory power-down mode is entered, so the return to Run mode now takes longer.
2. API timeout = 10. This is the problem case where flash memory enters Low-power mode and an early wakeup would result in the flash memory not being available. For workaround 2 the MCU is executing from RAM ,so there is no issue and the flash memory is power-cycled to ensure the correct status.
3. API timeout = 12. For this case the flash memory enters Low-power mode and returns to normal mode prior to re-entry to Run mode. This later wakeup would always result in the flash memory entering power-down state, so there is little overhead for this case.

Because Workaround 2 will always power-down the flash memory and return it to normal mode, the time to recover from LPM mode is constant for the cases illustrated in Figure 5 and Figure 6.



```
           Value      Mean     Min      Max      Std Dev
2 -Width   8.896µs    8.898µ   8.896µ   8.900µ   2.903n
```

Figure 5. STOP[Flash = Low-power] → RUN[Flash = Low-power] → RUN[Flash = Normal]

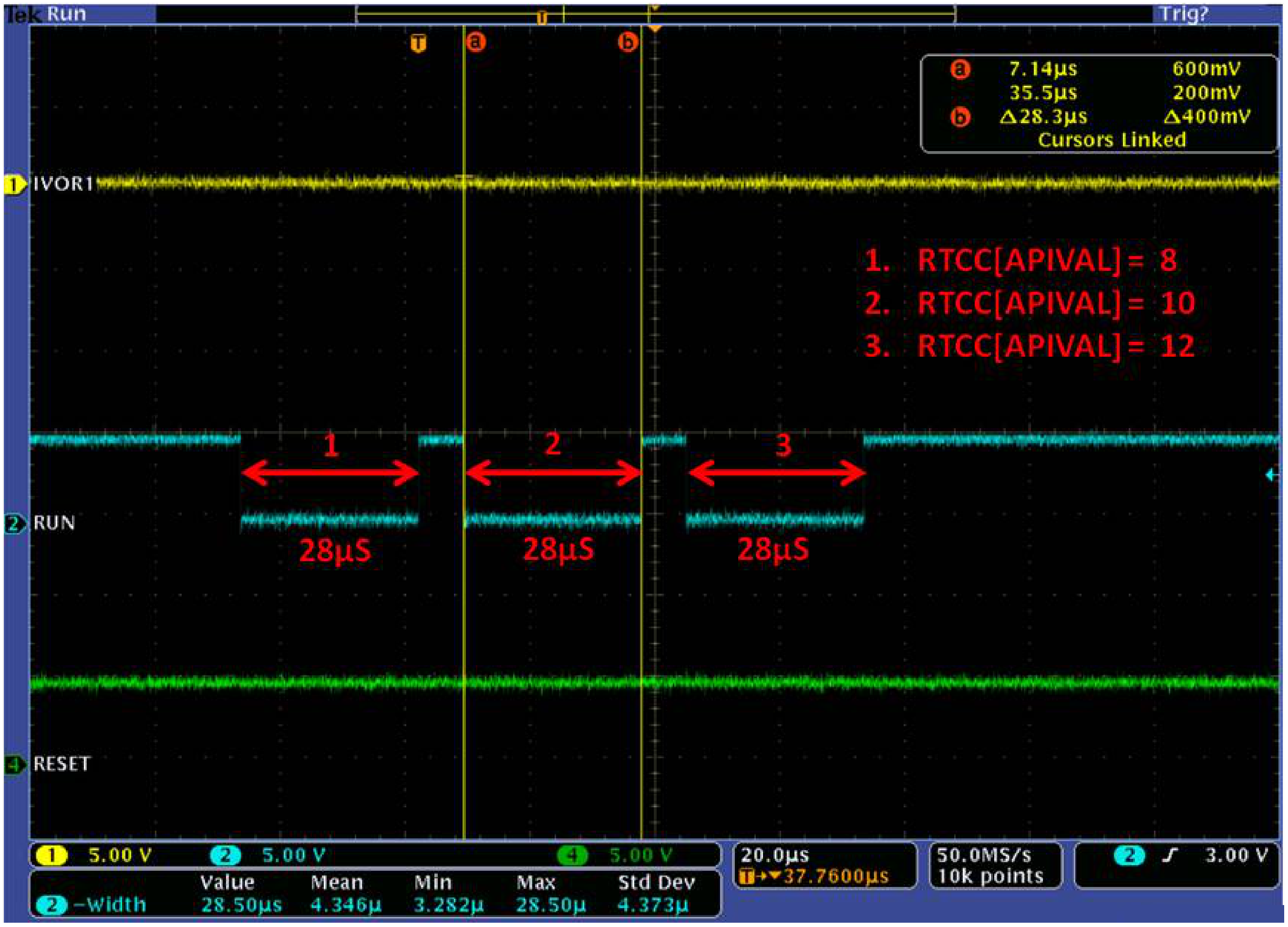


Figure 6. STOP[Flash = Power-down] → RUN[Flash = Power-down] → RUN[Flash = Normal]

4.3 Software flow

1. Prior to LPM mode entry reques,t branch to code execution in RAM while flash memory is still in normal mode.
2. Set ME_RUNx_MC[flash] = 0b01 (power-down) or 0b10 (low-power).
3. Set ME_<LPM>_MC[flash] = ME_RUNx_MC[flash].
4. Disable peripheral interrupts (IVOR4) at the core (MSR[EE] = 0) to prevent attempted flash memory access for interrupt vector table.
5. Enter LPM .
6. At wakeup or interrupt from STOP/HALT, MCU enters Run mode, executing from RAM with flash memory in low-power or power-down mode as per the ME_RUNx_MC configuration from step 2.
7. After the LPM request, set ME_RUNx_MC[flash] = 0b11 (normal).
8. Re-enter Run mode.
9. Wait for transition to Run mode to complete (ME_GS[S_MTRANS] = 0).
10. Re-enable peripheral interrupts (IVOR4) at the core (MSR[EE] = 1) .
11. Return to code execution in flash memory.

NOTE

Steps 2, 7, and 8 from the above list are not required for MPC5645S devices.

5 Workaround 3 (MPC5645S only)

For MPC5645S devices it is not necessary to run a work around from RAM when a machine check is present due to a premature low power mode abort, you can prepare code to verify the flash status in the machine check exception subroutine. MPC5645S devices make a few attempts until the flash is successfully restarted and the user code is executed, after this you can return to run from the flash again.

5.1 Software flow

1. Prior to LPM mode entry exception routine must be prepared and can be stored in flash.
2. Enter LPM
3. At wakeup or interrupt from STOP or HALT mode, the MCU enters Run mode, and an exception is triggered. The MCU then attempts to access the flash for first time, at this point the flash has not been completely initialized and another exception is triggered. The MCU will re-attempt to start the flash module, it repeats this until the flash is ready. After the MCU succeeds to access the flash, the user code from the exception routine is reached.
4. You may now return to code execution in flash memory (application program).

6 Considerations for e200z4 Cores

For those MCUs that have an e200z4 core, such as the MPC564xBC and MPC5645S, the checkstop state is not implemented. Therefore the checkstop reset will not be generated for the case when flash memory is unavailable and is accessed at exit from LPM. For the z4 cores the machine check interrupt (IVOR1) will be executed regardless of the state of MSR[ME]. Also, the flags at the Machine Check Syndrome Register (MCSR) are write ‘1’ to clear and must be cleared prior to exiting the IVOR1 handler, or else the IVOR1 will be executed again.

NOTE

MCSR on these cores will read 0x00090010 = MCSR[MAV (MCAR Address Valid), IF (Instruction Fetch Error Report), BUS_IRERR (Read bus error on Instruction fetch or linefill)].

Appendix A Sample code

The code within this appendix allows for testing devices and exception handlers for this issue.

The sample application was built for the MPC5604B device using the XPC56XXEVB main board and expansion board, and the Green Hills Compiler. Testing and using this code with other devices will require consideration of at least the header file referenced, but also the registers, low-power modes, and GPIO ports available on the target device.

To use this code, the user must ensure the following blocks are placed in RAM (workarounds 1 and 2 only):

1. Interrupt Vector Table
2. Prolog & Epilog for IVOR1 interrupt handler MachineCheck_ISR()
3. Function MachineCheck_ISR()
4. Function Enter_STOP_from_RAM()

```
#include "..\header\MPC5604B_0M27V_0102.h"

/*=======================================================================
Compile options
=======================================================================*/
/* Configure FLASH for Low Power Mode (LPM)       */
/* 1 = Power-Down, 2 = Low-Power, 3 = Normal (on) */
```

```
#define FLASH_LPM  1
#define FLASH_RUN  3
/* Configure the API value sweep. STOP mode entered for each value */
#define API_START  8
#define API_STOP   14
#define API_STEP   2

/*==============================================================================
Declare start of RAM code section (Greenhills compiler)
==============================================================================*/
#pragma ghs section text ".code"

/*==============================================================================
Function description - Located in RAM the Machine Check (IVOR1) interrupt
service routine disables Flash for the target Run mode and enters Run mode to
action the Flash Power Down mode. This is followed by enabling the Flash for
the next target Run mode and entering the Run mode to enable the Flash for
Normal mode.  This routine resynchronises the Flash status at the Mode Entry
module ME_GS[S_DFLA/S_CFLA].
Note a context store/restore of the core registers will be required in
conjunction with this function.
==============================================================================*/
void MachineCheck_ISR(void)
{
  SIU.GPDO[68].R = 1;             /* Set GPIO to flag start of Workaround */

  /* Disable FLASH to correct status at ME_GS[S_CFLA] */
  ME.RUN[0].B.CFLAON = FLASH_LPM; /* Config CFLASH for LPM */
  ME.RUN[0].B.DFLAON = FLASH_LPM; /* Config DFLASH for LPM */
  ME.MCTL.R = 0x40005AF0;         /* Mode & Key */
  ME.MCTL.R = 0x4000A50F;         /* Mode & Key inverted */
  while(ME.GS.B.S_MTRANS == 1);   /* Wait for mode change to complete */

  /* Enable FLASH to correct status at ME_GS[S_CFLA] */
  ME.RUN[0].B.CFLAON = 3;         /* Enable CFLASH */
  ME.RUN[0].B.DFLAON = 3;         /* Enable DFLASH */
  ME.MCTL.R = 0x40005AF0;         /* Mode & Key */
  ME.MCTL.R = 0x4000A50F;         /* Mode & Key inverted */
  while(ME.GS.B.S_MTRANS == 1);   /* Wait for mode change to complete */

  SIU.GPDO[68].R = 0;             /* Clear GPIO to flag end of Workaround */
}

/*==============================================================================
Function description - Located in RAM the function is used in Workaround 2 and 3
to ensure the flash is in the correct mode at low power mode exit, in this case
STOP mode exit. The flash is turned off before going into low power down mode.
After wake-up the Flash is configured to be in Normal mode for the target mode and the
target mode is entered. This routine ensures ME_GS[S_DFLA/S_CFLA] shows the correct
condition of the Flash. After this routine code execution can resume from Flash.
==============================================================================*/
void Enter_STOP_from_RAM(void)
{
  /* If  the MPC5645S is used comment the following two lines for RUN[0] */
  ME.RUN[0].B.CFLAON = FLASH_LPM; /* CFLASH config for RUN mode after STOP exit  */
  ME.RUN[0].B.DFLAON = FLASH_LPM; /* DFLASH config for RUN mode after STOP exit  */

  /*If the MPC5645S is used replace two lines above for Run[0] */
  /*ME.RUN[0].B.CFLAON = FLASH_RUN;   uncomment this line for MPC5645S     */
  /*ME.RUN[0].B.CFLAON = FLASH_RUN;   uncomment this line for MPC5645S     */

  ME.STOP0.B.CFLAON  = FLASH_LPM; /* CFLASH config for STOP mode */
  ME.STOP0.B.DFLAON  = FLASH_LPM; /* DFLASH config for STOP mode */

  ME.MCTL.R = 0xA0005AF0;         /* STOP Mode & Key */
  RTC.RTCC.B.CNTEN    = 1;        /* RTC counter enable */
  ME.MCTL.R = 0xA000A50F;         /* STOP Mode & Key inverted */
  while(ME.GS.B.S_MTRANS==1) {};  /* Wait for mode entry to complete */
  /* At STOP mode exit code will resume from here */
  ME.RUN[0].B.CFLAON = 3;         /* Enable CFLASH, not required for MPC5645S     */
  ME.RUN[0].B.DFLAON = 3;          /* Enable CFLASH, not required for MPC5645S     */
  ME.MCTL.R = 0x40005AF0;          /* Mode & Key, not required for MPC5645S        */
  ME.MCTL.R = 0x4000A50F;        /* Mode & Key inverted, not required for MPC5645S */
  while(ME.GS.B.S_MTRANS == 1);    /* Wait for mode change to complete */
}

#pragma ghs section text=default
/*===============================================================================
Declare end of RAM code section (Greenhills compiler)
All code from here will be placed in CFlash
===============================================================================*/

/*===============================================================================
Function description - The following function configures the core to manage
the Checkstop condition as a Machine Check interrupt (IVOR1). The IVPR and the
Interrupt Vector Base address are set to an area in RAM for Workaround 1. Also
the Machine Check interrupt is enabled at MSR.
===============================================================================*/
asm MachineCheck_init(void)
{
  /* Set IVPR to vector table base address in RAM */
  e_lis r5, __IV_ADDR@h
  e_or2i r5, __IV_ADDR@l
  mtIVPR r5
   /* Set ME to enable Machine Check Interrupt */
  e_li r8, 0x1000
  mtmsr r8
}

/*===============================================================================
Function description - The following function configures the core to handle
the Checkstop condition as a Machine Check interrupt (IVOR1) for Workaround 3.
Machine Check interrupt is enabled at MSR.
===============================================================================*/
asm MachineCheck_Workaround_3_init(void)
{
  /* Set ME to enable Machine Check Interrupt */
  e_li r8, 0x1000
  mtmsr r8
}


/*===============================================================================
Function description - The following function configures the GPIO to help the
user follow the software flow.
===============================================================================*/
void GPIO_init(void)
{
  uint32_t delay;

  SIU.GPDO[68].R = 0; /* low */
  SIU.GPDO[69].R = 1; /* high */
  SIU.PCR[68].R = 0x0200; /* PCR68 (LED 1) - toggled at machine check interrupt */
  SIU.PCR[69].R = 0x0200; /* PCR69(LED 2) - set high in RUN, low in STOP */
  /* Delay to offset GPIO configuration from mode changes */
  for(delay=0;delay<0xFFFF;delay++);
}

/*===============================================================================
Function description - The following function configures the API WKUP with API
timeout value - APIVAL
===============================================================================*/
void API_WKUP_init(uint32_t APIVAL)
{
  /* Config API */
  RTC.RTCC.B.CNTEN    = 0;       /* Clear RTC counter */
  RTC.RTCC.B.CLKSEL   = 2;        /* FIRC */
  RTC.RTCC.B.DIV512EN = 0;        /* Divide API clock by 512 */
  RTC.RTCC.B.DIV32EN  = 0;        /* Divide API clock by 32 */
  RTC.RTCC.B.APIVAL   = APIVAL;   /* 8 Hz x 5 seconds */
  RTC.RTCC.B.APIEN    = 1;        /* API enable */

  /* Config WKUP */
  WKUP.WISR.R    = 0xFFFFFFFF;  /* Clear all WKUP flags          */
  WKUP.WRER.R    = 0x00000001;  /* Enable WKUP for wakeup event  */
  WKUP.IRER.R    = 0x00000001;  /* Enable WKUP interrupt         */
  WKUP.WIREER.R = 0x00000001;  /* Enable Rising Edge of WKUP    */
  WKUP.WIPUER.R = 0xFFFFFFFF;  /* Enable all pullups on WKUP    */
}
/*==============================================================================
Function description - The following function configures the API for LPM WKPU.
An API timeout value is loaded into this function - APIVAL - and STOP mode is
entered.  A port toggle helps the user follow the software flow.
==============================================================================*/
void LPM_Checkstop(uint32_t APIVAL)
{
  API_WKUP_init(APIVAL);

  /* Clear GPIO to mark STOP mode entry */
  SIU.GPDO[69].R = 0;

  ME.STOP0.B.CFLAON  = FLASH_LPM; /* CFLASH config for STOP mode */
  ME.STOP0.B.DFLAON  = FLASH_LPM; /* DFLASH config for STOP mode */

  /* Enter STOP & enable API counter */
  ME.MCTL.R = 0xA0005AF0;         /* STOP Mode & Key */
  RTC.RTCC.B.CNTEN    = 1;        /* RTC counter enable */
  ME.MCTL.R = 0xA000A50F;         /* STOP Mode & Key inverted */
  while(ME.GS.B.S_MTRANS==1) {};  /* Wait for mode entry to complete */

  /* Set GPIO to mark STOP mode exit */
  SIU.GPDO[69].R = 1;
}

/*==============================================================================
Function description - The following function configures the API for LPM WKPU.
An API timeout value is loaded into this function - APIVAL.  To enter STOP mode
the code branches out to RAM to implement Workaround 2.
A port toggle helps the user follow the software flow.
==============================================================================*/
void LPM_Checkstop_RAM(uint32_t APIVAL)
{
  API_WKUP_init(APIVAL);

  /* Clear GPIO to mark STOP mode entry */
  SIU.GPDO[69].R = 0;

  /* Enter STOP & enable API counter */
  Enter_STOP_from_RAM();

  /* Set GPIO to mark STOP mode exit */
  SIU.GPDO[69].R = 1;
}

/*==============================================================================
Function description - The following function enables all peripherals for RUN0
and enters RUN0. This will allow a transition to STOP mode at some point later.
==============================================================================*/
void RUN0_all_peripheral_on(void)
{
  ME.MER.R      = 0x000025FD; /* enable all modes */
  ME.RUNPC[0].R = 0x000000FE; /* Peripheral ON in every RUN mode */
  /* Enter in RUN0 mode to update */
  ME.MCTL.R = 0x40005AF0;        /* RUN0 Mode & Key */
  ME.MCTL.R = 0x4000A50F;        /* RUN0 Mode & Key */
  while(ME.GS.B.S_MTRANS==1) {};  /* Wait for mode entry to complete */
}

/*==============================================================================
Function description - The following function disables the Software Watchdog
```

```
Timer so that the user does not have to service the watchdog.
==============================================================================*/
void DISABLE_WATCHDOG()
{
  SWT.SR.R = 0x0000c520; /* key */
  SWT.SR.R = 0x0000d928; /* key */
  SWT.CR.R = 0x8000010A; /* disable WEN */
}

/*==============================================================================
Function description - The following function validates Workaround 1.  Stop is
entered a number of times using a range of API values to wake-up.  The API
values are selected centred around the time at which the errata manifests.
==============================================================================*/
void Workaround_1(void)
{
  uint32_t apival_sweep;

  for(apival_sweep=API_START; apival_sweep<API_STOP; apival_sweep=apival_sweep+API_STEP)
  {
    MachineCheck_init();
    LPM_Checkstop(apival_sweep);
  }
}

/*==============================================================================
Function description - The following function validates Workaround 2. Stop is
entered a number of times using a range of API values to wake-up.  The API
values are selected centred around the time at which the errata manifests.
==============================================================================*/
void Workaround_2(void)
{
  uint32_t apival_sweep;

  /* Disable External Interrupts as jumping to RAM execution and interrupt vector
     table in Flash so may not be available at low power mode exit */
  asm(" wrteei 0");

  for(apival_sweep=API_START; apival_sweep<API_STOP; apival_sweep=apival_sweep+API_STEP)
  {
    LPM_Checkstop_RAM(apival_sweep);
  }

  /* Enable Eternal Interrupts */
  asm(" wrteei 1");
}

/*==============================================================================
Function description - The following function validates Workaround 3 and is
only for MPC5645S devices. The MCU enters Stop mode several times using a range of
API values to wake. The API values are selected around the time the errata manifests.
This procedure is similar to Workaround_2 with the difference that LPM is entered from the
flash. When the checkstop is present the MCU executes Workaround_3_ISR
in the flash, it will fail a few times until the flash completes re-initialization.
==============================================================================*/
void Workaround_3(void)
{
  uint32_t apival_sweep;

  /* Disable External Interrupts, because jumping to RAM execution and the interrupt vector
table in the Flash may not be available at low power mode exit */
  asm(" wrteei 0");

  for(apival_sweep=API_START; apival_sweep<API_STOP; apival_sweep=apival_sweep+API_STEP)
  {
      MachineCheck_Workaround_3_init();
      LPM_Checkstop(apival_sweep);
  }
  while(1); /* this code should not be reached if checkstop is triggered */
}
```

```
/*==============================================================================
Function description - The following service routine Machinecheck_Workaround_3_ISR  is
entered after the flash re-starts, after checkstop, before this function is executed some
attempts are performed by the MCU to execute it, this first causes some illegal memory
accesses, then after the flash completes re-initialization this function is successfully
executed.
Prolog & Epilog for IVOR1 interrupt handler for Machinecheck_Workaround_3_ISR
is required.
==============================================================================*/
void Machinecheck_Workaround_3_ISR(void)
{
   while(1 == ME.GS.B.S_MTRANS);

   /* user strategy: RESET/ DIAGNOSE/ RETURN TO APPLICATION SECTION */

}

/*==============================================================================
Function description - The following function generates a Checkstop Reset for
the case when STOP is exited and the Flash is not available. Stop is entered
many times using a range of API values to wake the MCU.  The API values are
selected centred around the time at which the errata manifests.
==============================================================================*/
void No_Workaround(void)
{
  uint32_t apival_sweep;

  for(apival_sweep=API_START; apival_sweep<API_STOP; apival_sweep=apival_sweep+API_STEP)
  {
    LPM_Checkstop(apival_sweep);
  }
}

/*==============================================================================
Function description - The following function generates four test cases
1. No_Workaround()    - generates the Checkstop Reset condition
2. Workaround_1()     - generates Machine Check interrupt
3. Workaround_2()     - executes STOP mode entry from RAM

The following case is exclusive for the MPC5645S.
4. Workaround 3       - executes the exception subroutine from flash,
                        waiting until the flash recovers after a few flash access
                        attempts.
User observations:
PCR68 (LED 1)         - toggled at machine check interrupt
PCR69 (LED 2)         - set high in RUN, low in STOP
==============================================================================*/
int main(void)
{
  DISABLE_WATCHDOG();
  RUN0_all_peripheral_on();
  GPIO_init();

  /* User to select function to call */
  No_Workaround();
  //Workaround_1();
  //Workaround_2();
  //Workaround_3(); /* stored in flash*/
  while(1) {}; /* trap here */
} /* main */
```

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
support.asia@freescale.com

Document Number: EB770
Rev. 1, August 2012