Freescale Semiconductor
Application Note

Document Number: AN3584
Rev. 0, 5/2008

MPC5510 Family Low Power Features

by: Christopher Platt and Carl Culshaw
Applications Engineering
Microcontroller Division
Austin, Texas, USA and East Kilbride, Scotland, UK

1 Introduction

The performance requirements in the automotive body and gateway applications space have increased significantly over the past few years. Freescale has addressed this need by introducing a family of Power Architecture devices specifically targeted at applications where low power is a critical requirement.

The MPC5510 family builds on the success of the MPC55xx family popular in powertrain and chassis systems. It adds many features specifically aimed at reducing the overall power during normal operation and modes when the device is stopped. This document discusses new features in detail and how to best use them in an application to minimize the overall power used by the device.

Contents

freescale™ semiconductor

2 Power Problem

Power is made up of two separate components: run current (dynamic current) and stop current (static current).

Dynamic Current — This is the main current prevalent during normal operating conditions. The mechanism responsible for this current is basically due to charging and discharging the gates of the millions of MOS transistors that switch on and off as the device operates. The switching currents adhere to the equation:

$$I = \frac{dQ}{dt} = C\frac{dv}{dt} \qquad \textit{Eqn. 1}$$

Thus as technology advances and transistors become smaller, the gates become smaller resulting in lower gate capacitance. The lower capacitance means lower switching currents and hence an overall reduced dynamic run current.

On the other side of the equation, because switching speeds are increasing (dv/dt) this leads to an increased dynamic current.

Fortunately, the reduced current due to lower capacitance wins out over the increased current due to faster switching leading to a lower overall dynamic power. Unfortunately, as applications become more complex and demanding, the performance requirements and hence frequencies are increasing, leading to an overall increase in dynamic power.

Static Current — This current is present when the transistors of the device are not switching and are hence in a static state. The main mechanism causing this current is leakage due to the finite resistance that exists between power and ground if power is applied to a CMOS circuit.

This leakage current is highly dependent on the threshold voltage of the transistor. As technology becomes smaller, supply voltage levels are scaled lower. To improve circuit speed, the threshold voltages are also decreased, which results in an exponential increase in sub-threshold leakage current.

In summary, as technology shrinks ever smaller, even though dynamic power trends lower, the required increase in performance means that any savings made by technology are neutralized by the increased frequencies. So, overall power is trending upwards.

Couple this fact with the dramatic increase in static leakage inherent as we move to smaller geometries. We have an overall picture of increased dynamic power and a static component becoming a larger component of the overall current.

3 MPC5510 Family Low Power Feature Summary

Because the power requirements are showing an upward trend, the MPC5510 family has a number of features that are specifically designed to minimize power:

- Clock management
  - The ability to stop clocks on a per module basis
  - Optimized clock tree design
  - Ability to divide down the system clock to peripherals
  - Ability to gate off or divide down the clock to the core
- Onboard clocking
  - Part can self-clock without a PLL
    - 16 MHz internal oscillator
    - 32 kHz internal oscillator (low power mode only)
- Power gating
  - Literally removing power to large areas of silicon
- Active well biasing (in stop mode)
- Dual core
  - Able to run at lower speed to achieve the same throughput.

All the above features are discussed in this document.

4 MPC5510 Power Modes

The MPC5510 family incorporates three modes of operation:

- Run mode
  - Device in this mode is running normally and executing main application code.
  - The estimated room temp current in this mode is 120 mA at 66 MHz with all peripherals and both cores running.
- Stop mode

In this mode, all of the main clocks on the device stop similar to traditional stop modes on many microcontroller devices such as the S12X family.

The estimated room temp current in this mode is 250 μA.

- Sleep mode
  - This mode is new to the Power Architecture family of devices. In this mode, power is literally removed from large areas of silicon to reduce static leakage currents. There are multiple sleep modes on the MPC5510 family that differ only in the amount of SRAM that remains powered during this mode.
  - The estimated room temp current for this mode with 8K RAM remaining powered is 20 μA.

A tabular summary of modes listing what remains powered during each mode is displayed in Table 1.

Table 1. Modes Listing What Remains Powered

		Low Power Modes		
Features	**Normal Operation (Run Mode)**	**Stop CRP_PSCR[STOP]**	**Sleep 0–4 CRP_PSCR[SLEEP]**	
Standard cell logic (z0/z1 cores, DMA, peripherals, etc.)	Powered up, running	Powered up, halted (well biased)	Powered down	
Flash	Optionally disabled	Disabled (exit SW cannot be in flash)	Powered down	
SRAM powered	All SRAM powered	All SRAM powered	Sleep 0: 0 KB Sleep 1: 8 KB Sleep 2: 16 KB Sleep 3: 32 KB Sleep 4: 64 KB Sleep 5: 80 KB	SRAM powered
Output pads	Active	States maintained	States maintained	Output pads
Input pads	Active	Enabled for wakeup	Enabled for wakeup	Input pads
API, RTC clocks	Optionally enabled	Optionally enabled	Optionally enabled	API, RTC clocks

5 Clock, Reset, and Power Module (CRP)

A key element of the MPC5510 family is the CRP module. This module is new to this family of devices. The CRP module forms the heart of the low-power functionality on the device and consists of:

- Input isolation block
  - Allows the inputs from external blocks to be driven to known states if the logic driving the inputs is powered down.
- Pad keeper circuitry
  - Wakeup and power status block
  - Clock and reset control block
  - Low power state machine

The primary function of this module is to maintain all control logic that requires power if other portions of the device are powered down in sleep modes. This module does not contain clock and reset generation logic or any voltage regulator, power gating, or LVI (low voltage interrupt) circuits. This module contains the logic necessary to control these elements during transitions between the various available modes.

The CRP module is the heart of the low power mode control. It is always powered up, even in sleep mode while most of the device is power gated off.

6 Low Power Entry

The sequence of events to enter a power saving mode is listed below. The following sequence assumes that low power exit criteria are already set up and that any wakeup routines are resident in SRAM or flash. Some example code detailing entry into and exit from LPM is included in Appendix D, "Example Code."

- Disable DMA and FlexRay bus masters (if applicable, depending on family member)
- Halt all modules (SIU_HLT), verify HLTACK
- Shut down RTI (optional)
- Set system clock to 16 MHz IRC
  - This clock drives CRP module
- Disable PLL
- Shut down XOSC (optional)
- Select sleep or stop
  - CRP_PSCR(SLEEP:STOP)
- Execute wait instruction on any active core
- CRP module now controls power down sequence

1 and 2 disable DMA and FlexRay bus masters and halt all modules

The SIU_HLT register contains halt bits for all of the MPC5510 modules.

Table 2. HLTACK Register Field Descriptions

Field	Description
0–31 HLTACK[0:31]	Halt Flags. Each bit corresponds to a separate module, as mapped below. 0 e200z1 1 e200z0 2 FLEXRAY 3 DMA 4 Reserved 5 Reserved 6 NPC 7 EBI 8 EQADC 9 MLB 10 EMIOS200 11 Reserved 12 IIC_A 13 PIT 14 FLEXCAN_F 15 FLEXCAN_E 16 FLEXCAN_D 17 FLEXCAN_C 18 FLEXCAN_B 19 FLEXCAN_A 20 DSPI_D 21 DSPI_C 22 DSPI_B 23 DSPI_A 24 ESCI_H 25 ESCI_G 26 ESCI_F 27 ESCI_E 28 ESCI_D 29 ESCI_C 30 ESCI_B 31 ESCI_A

Writing a 1 to the relevant bit enables the halt logic built into the module to perform a controlled shutdown of the module rather than just simply stopping the clock to the module. This allows a module currently performing a task to complete the task and stop gracefully (such as a DSPI in the middle of transmitting a message). Each module also has a corresponding flag in the HLTACK (halt acknowledge) register.

When a module is requested to halt, and has completed its halt sequence, it sets the appropriate halt acknowledge flag in the HLTACK register. In this way, software can easily monitor the progress of any modules requested to halt.

When entering a low power mode, because all modules are going to be halted, writing $FFFFFFFF to the 32-bit SIU_HLT register and then waiting for the flags to assert appears to be sufficient. However, on the MPC5510 family, the DMA and the FlexRay modules (if on device) do not contain the halt logic necessary to perform a graceful halt. This is mainly because of the complex nature of these particular modules. Hence if either of these two modules is being used, the user must take care of shutting them down.

NOTE

The halt bits in the SIU_HLT register do not directly affect the FlexRay and DMA modules. These modules must still be set. The set action sets the halt flags for these modules. All bus masters must be halted prior to low-power mode entry. The DMA and the FlexRay module are bus masters. Failure to set the halt bits for these modules prevents the device from entering low-power mode.

6.1 Shutdown RTI

The RTI circuit is unaffected by the SIU_HLT bits or the MDIS (module disable). It must be manually shut down by user software. Because the RTI function is mapped to PIT0, this is accomplished by writing a 0 to bit 31 of the PITEN register.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	0	0	0	0	0	0	0	PEN8	PEN7	PEN6	PEN5	PEN4	PEN3	PEN2	PEN1	PEN0
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Field	Description
0–22	Reserved.
23–31 PEN*n*	Timer Enable Bit 0 Timer is disabled 1 Timer is active

Figure 1. PIT Timer Enable Register (PITEN)

6.2 Set System Clock to 16 MHz IRC

While the MPC5510 is in low-power mode, any operation of the CRP module is clocked by the 16 MHz IRC. Thus prior to LPM (low power mode) entry the system clock must be set to the 16 MHz IRC. This is achieved by writing 00 to the SYSCLKCEL bits in the SIU_SYSCLK register. If the 16 MHz IRC is not selected prior to execution of the HALT command, the 16 MHz IRC is automatically started by the system to clock the CRP module. In this case, whichever clock source that was clocking the system prior to sleep mode entry (most likely PLL/XOSC) is still active and using power.

Access: User read-only

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R	SYSCLKSEL		SYSCLKDIV		SWT CLKSEL	0	0	0	0	0	0	0	0	0	0	0
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	LPCLKDIV7		LPCLKDIV6		LPCLKDIV5		LPCLKDIV4		LPCLKDIV3		LPCLKDIV2		LPCLKDIV1		LPCLKDIV0	
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Field	Description
0–1 SYSCLKSEL	System Clock Select. The SYSCLKSEL bit selects the source for the system clock. 00 System clock supplied by 16 MHz IRC 01 System clock supplied by XOSC 10 System clock supplied by PLL 11 Reserved (defaults to 16 MHz IRC)

Figure 2. System Clock Register (SIU_SYSCLK)

6.3 Disable the PLL (Stop Only)

In sleep mode, the power to the PLL is removed and hence the PLL stops automatically. However, if entering stop mode, the user must take care of disabling the PLL themselves. Writing 000 to the CLKCFG bits in the ESYNCR1 register disables the PLL.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R	1	CLKCFG[2:0]			0	0	0	0	0	0	0	0	EPREDIV[3:0]			
W																
Reset	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	0	0	0	0	0	0	0	0	EMFD[7:0]							
W																
Reset	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	1

Field	Description
0	Reserved. Note: This bit is set to 1 on reset and always reads as 1.
1–3 CLKCFG[2:0]	Clock Configuration. The CLKCFG[2:0] bits are a writable version of the MODE, PLLSEL, and PLLREF bits in the SYNSR. These change the clock mode, after reset has negated, via software. CLKCFG[2:0] maps directly to MODE, PLLSEL, and PLLREF to control the system clock mode. (For more detailed information, refer to the “Frequency Modulated Phase Locked Loop (FMPLL)” chapter in the MPC5510 reference manual.) CLKCFG[2:0] = 0b101 can produce an unpredictable clock output. The ESYNCR2[LOLRE] and ESYNCR2[LOCRE] must be set to 0 before changing the PLL mode, so that a reset is not immediately generated upon the write to CLKCFG[2:0].
4–11	Reserved.

Figure 3. FMPLL Synthesizer Status Register (SYNSR)

6.4 Shut Down the External Oscillator Circuit (XOSC)

Care needs to be taken to disable the external oscillator circuit prior to LPM entry. In stop and sleep modes the output from the XOSC is disabled. Leaving it running inadvertently is simply a waste of power.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R	0	0	0	0	0	0	0	0	0	0	0	0	32KIRC EN	XOSC EN	0	32KOSC EN
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	TRIM32IRC[0:7]								TRIMIRC[0:7]							
W																
Reset	1	1	0	1	1	1	1	1	1	0	0	0	1	1	1	1

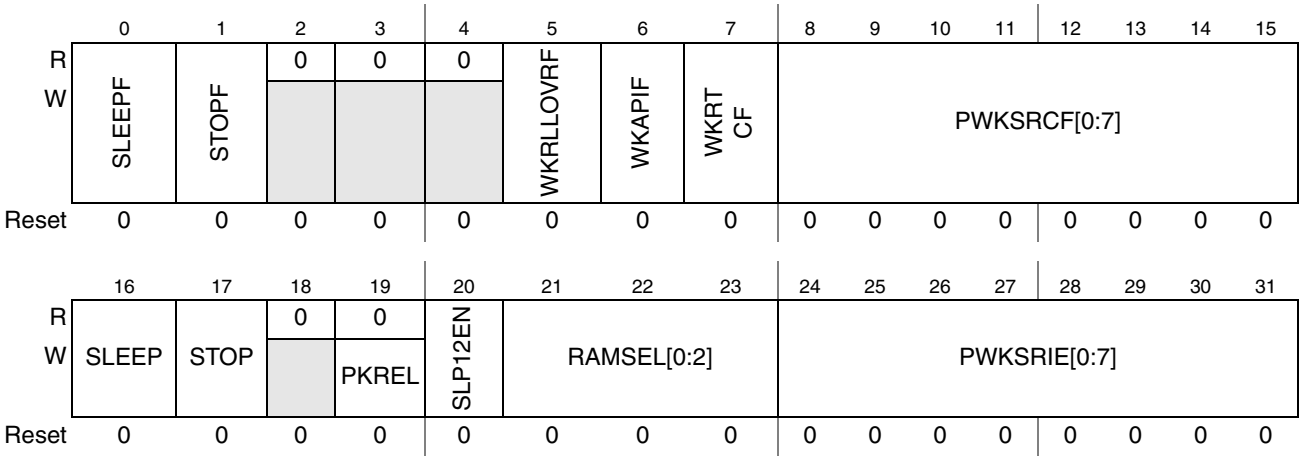
Field	Description
13 XOSCEN]	External Oscillator Enable. The XOSCEN bit continuously enables the external oscillator or puts it in a lower power state when a low-power mode is entered. 0 XOSC disabled, but crystal remains powered to reduce start up time after low-power mode exit. 1 XOSC enabled

Figure 4. Clock Source Register (CRP_CLKSRC)

XOSC is disabled by writing a 0 to the XOSCEN bit in the CRP_CLKSRC register.

6.5 Select Sleep or Stop Mode and Execute WAIT Instruction

The final elements of entry into LPM is to select whether the device is required to enter stop or sleep mode, and then execute the WAIT instruction.



Field	Description
21–23 RAMSEL[0:2]	RAM Selects. The RAMSEL bits select which ram configuration retains power during the sleep mode. 000 All RAMs powered down 001 8K RAM retains power (0x4000_0000 – 0x4000_1FFF) 010 16K RAM retains power (0x4000_0000 – 0x4000_3FFF) 011 32K RAM retains power (0x4000_0000 – 0x4000_7FFF) 110 64K RAM retains power (0x4000_0000 – 0x4000_FFFF) 111 80K RAM retains power (0x4000_0000 – 0x4001_3FFF)

Figure 5. Power Status and Control Register (CRP_PSCR)

The required LPM is specified by writing a 1 to bit 16 or 17 of the CRP_PSCR register. If sleep mode is selected, the user must also specify the amount of RAM to remain powered after sleep mode is entered but write the relevant value in the RAMSEL bits of the same register.

After the WAIT command is executed, control of the device is handed over to the CRP module. If the device is entering sleep mode, the I/O state is held and power gating circuits are enabled to remove power to most of the device. Power is however retained in the CRP module and any RAM specified as RAM to remain powered in the CRP_PSCR register, and any clocks that are running.

If the device is entering stop mode, the system clocks are stopped, the flash is disabled, and an active well bias is applied to the standard cell logic to reduce leakage currents.

Appendix A, “Low Power Entry Sleep,” and Appendix B, “Low Power Entry Stop,” are flowcharts to logically illustrate the sleep and stop entry sequences.

7 Low-Power Mode Exit

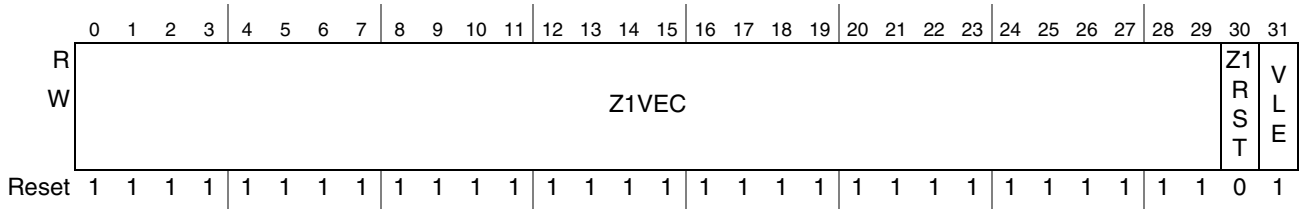
Exit from low-power mode is caused by a reset (power on reset or external reset pin) or the occurrence of a pre-defined wakeup event that was specified prior to entry into LPM.

The programmable wakeup events are discussed in more detail in the next section.

Because sleep and stop modes are inherently different in that large areas on the device are powered down in sleep but not in stop, the exit from sleep and stop modes are managed differently.

7.1 Sleep Mode Exit

If a pre-defined wakeup event occurs during sleep mode, the 16 MHz IRC is enabled (if not already enabled). When this pre-defined event stabilizes, it begins to clock the CRP module. The CRP module then executes the control sequence to bring the part out of sleep mode. After the device is in a stable voltage and clocking state (or both) the e200z1 or e200z0 core begin code execution from the address contained in the CRP_Z1VEC or CRP_Z0VEC registers.



Field	Description
30 Z1RST	Controls the assertion of RESET to the Z1 core. Writes to this bit cause the Z1 to immediately enter/exit reset. Reads of this bit indicate if the core is being held in reset. 0 Z1 not in reset 1 Z1 in reset

Figure 6. Reset Vector Register (CRP_Z1VEC)

This register is set up prior to LPM entry. It resides in the CRP and retains its power and hence value during LPM. What determines whether a core begins execution or not is the RST bit (bit 30) of the CRP_Z1(0)VEC register.

If the RST bit is set to a 0 the core begins execution at the address location specified by Z1(0)VEC. If the RST bit is set to a 1 the core does not execute code from Z1(0) VEC because it is held in reset.

After being released from reset, the core begins execution from the location specified. It is not allowed for both cores to be held in a reset state. If one core is in reset and the other core tries to put itself into reset, then the internal logic prevents this from happening.

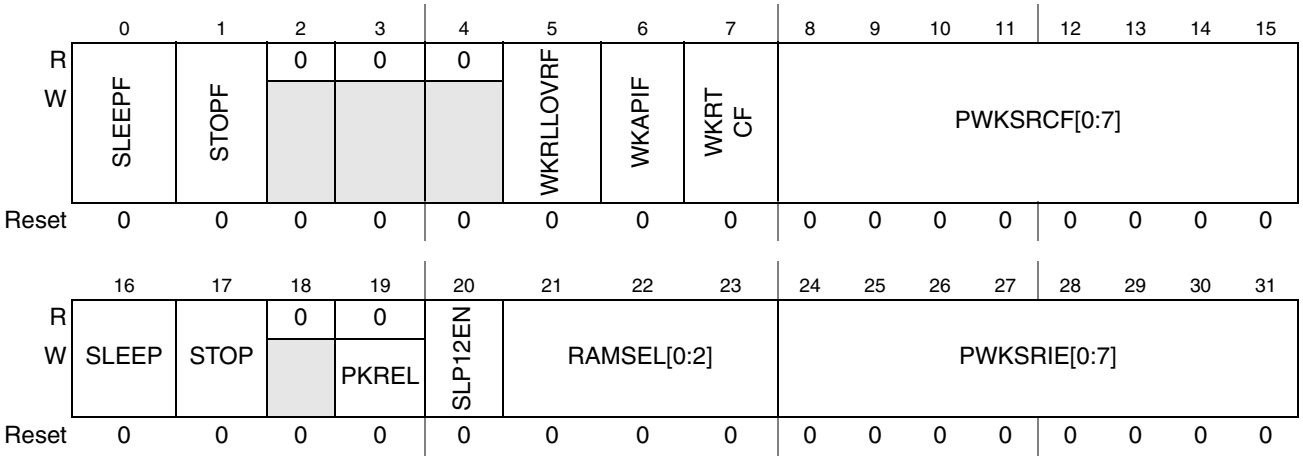


Figure 7. Power Status and Control Register (CRP_PSCR)

If multiple wakeup sources were enabled, user code located at the vector address is now able to determine which source caused the wakeup by reading the flags (bits 5–15) in the CRP_PSCR.

User code must also release the state of the pads. Upon entry into sleep mode, the state of the pads is held in their current state. On exiting sleep mode the pad state is still held. This pad state can be released only by user software writing a 1 to the PKREL bit in CRP_PSCR.

Here are some important things to remember while exiting sleep mode (also refer to Appendix A, “Low Power Entry Sleep,” and Appendix B, “Low Power Entry Stop”).

Because most of the device is powered down, all logic except that in the CRP module is reset. Hence any context or register states are lost and revert back to their reset state.

Therefore:

- The user must re-initialize the watchdog circuit.
- The user must re-initialize any RAM that did not remain powered during sleep mode.
- The user must take care not to access the flash until approximately 120 µs after the wakeup event occurred. (If fast recovery is enabled, see below).
- The pads, pad functions, and pin multiplexing scheme must be reconfigured by user software.

7.1.1 Fast Recovery

On exiting from sleep mode, an automatic reset sequence is generated that allows time for the flash memory to recover and stabilize so that program execution from flash can occur. This reset sequence takes 2400 (if PLL is not enabled) or 9600 (if PLL is enabled) cycles. However, the MPC5510 family also has a fast recovery feature that allows the reset sequence to be shortened to only 64 cycles. To use this feature, ensure that exit code is located in RAM that remains powered and that the Z1(0)VEC register is pointing to it. Execution can then occur within 64 cycles.

Fast recovery is enabled by setting the FASTREC bit in the CRP_RECPTR register.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	RECPTR																														FASTREC	0
W																																
Reset	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0

Figure 8. Reset Recovery Pointer (CRP_RECPTR)

It must be noted that the rest of the bits in this register have no purpose and because this register resides in the CRP module, it could thus be used by a user to store data not lost during a low power entry.

7.2 Stop Mode Exit

After a pre-defined wakeup event has occurred, exit from stop mode is essentially a two-stage process.

First, the clocks become stabilized and are applied to the device. After this has happened the core remains in the wait state until it receives an interrupt.

Some important things while exiting stop are:

- Even though power is not gated in stop mode, the flash is disabled to reduce leakage in the control circuits. This means that any ISR (interrupt service routine) must reside in RAM to enable immediate execution. Additionally, the flash must not be accessed until it has stabilized (around 70 μs).
- Unlike sleep mode where the user must release the state of the pads manually, the I/O states out of stop mode are automatically released by hardware.
- Fast recovery is not available from stop mode.

8 Wakeup Sources

As discussed previously, there are four ways to wake the device from stop or sleep mode:

- RTC counter match
- RTC counter rollover
- API counter match
- External pin transition (positive, negative, or edge)

8.1 Low Power Timer Wakeups

The RTC counter match, RTC counter rollover, and API counter match wakeup sources are all related to the low power timer system on the MPC5510.

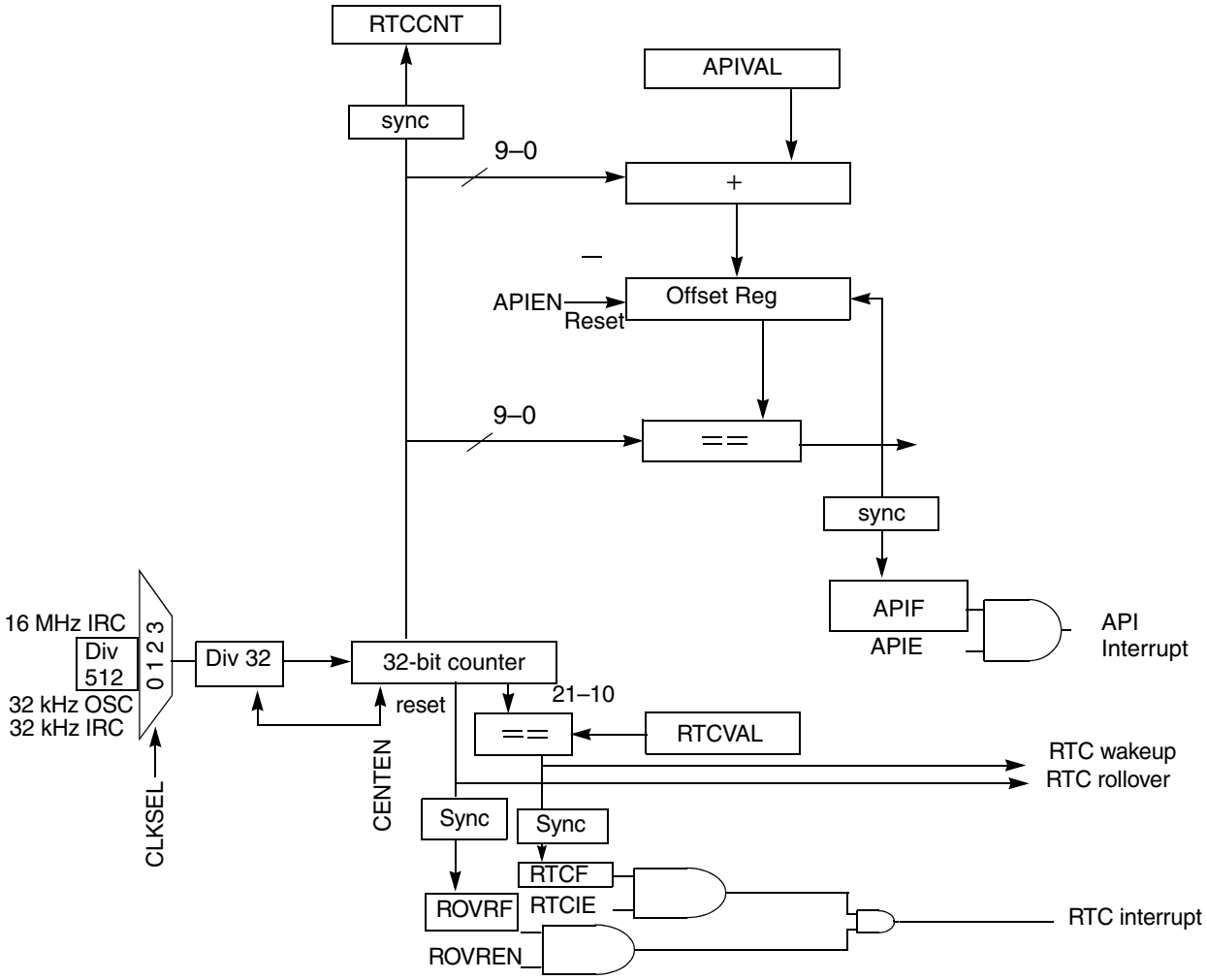


Figure 9. RTC Block Diagram

As can be seen in Figure 9, the RTC and the API share a common 32-bit, free-running counter. The counter clock source is selectable from the 16 MHz IRC, 32 kHz IRC, or 32 kHz OSC. There is a fixed 32-bit divide going into the counter that provides a 1 ms resolution for the 32 kHz sources. The 16 MHz IRC is another source, and this can be the full 16 MHz or passed through a fixed divide-by-512 counter. This fixed divider again provides a 1 ms resolution on the 32-bit counter. This 1 ms resolution provides up to 1.5 months time period.

In choosing to use the RTC or the API, the user needs to trade off the length of time the timer may have to wait before wakeup against resolution:

- The RTC system with its 12-bit compare gives a 1 s to 1 hour timeout with a 1 s resolution.
- The API system with a 10-bit compare supports wakeup intervals 1 ms to 1 s.

The compare values of these timers can be changed while they are running although clearly if the part is in LPM, there is no mechanism to achieve this.

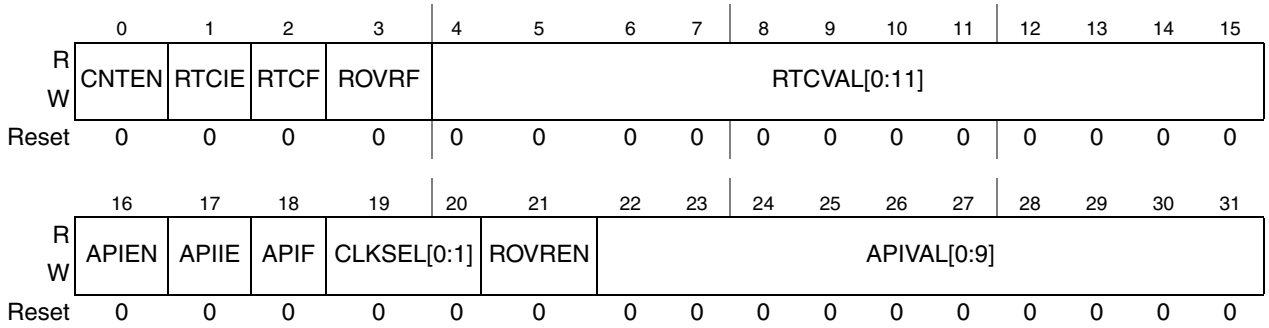


Figure 10. RTC Status and Control Register (CRP_RTCSC)

There are separate fields within the RTC status and control register (CRP_RTCSC) to control the operation and parameters for the RTC and API systems. The clock source select bits and initial values are all assignable as is the rollover enable for the RTC and the individual enable bits for the counters. Bits are available to enable interrupts for each of the mechanisms as well as flag bits that are set if an event occurs.

For these mechanisms to wake the device from LPM they must be both enabled and assigned as a wakeup source.

8.1.1 RTC/API Clock Source

In sleep and stop modes, most clocks are stopped. However, there always has to be a reference clock running to feed the RTC/API counter to enable wakeups. If the RTC or API is not configured as a wakeup source, and the only wakeup source is on an external pin transition, there must still be one of the internal clocks running to latch the pin transition. In short, in LPM the user must always have a clock running.

There are three choices of clocks to drive the low power circuitry in LPM. The clock choice is made in the CLKSEL bits (19–20) of the CRP_RTCSC register as shown above.

- 16 MHz IRC: CLKSEL = 10
- 32 kHz IRC: CLKSEL = 00
- 32 kHz XOSC: CLKSEL = 01

The choice again comes down to a trade-off between accuracy, power, and recovery time.

The lowest power option is the 32 kHz IRC that consumes around only 1 µA. The disadvantage is that the clock is accurate to around only 10% deviation. If a pin transition occurs immediately after a clock edge, the edge won't be latched and acted upon until the next clock edge. At 32 kHz, the clock is over 30 µs. That may not be a problem in most applications.

The next lowest power option is using the 32 kHz external oscillator facility. This option consumes around 3 µA of current but requires the addition of a 32 kHz crystal. The advantage is that this is a much more accurate clock than the internal IRC and could potentially be used to track "time of day."

The 16 MHz IRC is the most accurate internal clock available (<5%), consuming a higher current of 160 µA. This is always used as the default system clock from low power mode, hence allowing near instantaneous recovery. This capability allows registers, modules, etc., to be configured while waiting for the external crystal to stabilize. Alternatively, it supports rapid code execution during short run cycles, enabling the device to return to an LPM much more quickly.

8.2 External Pin Wakeup

It is possible to assign up to eight pins (from a possible total of 64) to wake the part from LPM should a transition occur at the specified pin.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	0	WKPSEL 7			0	WKPSEL 6			0	WKPSEL 5			0	WKPSEL 4			0	WKPSEL 3			0	WKPSEL 2			0	WKPSEL 1			0	WKPSEL 0		
W																																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 11. Wakeup Pin Source Select Register (CRP_WKPINSEL)

Each of the eight pins are assigned in the wakeup pin source select (CRP_WKPINSEL) register from the total group of 64. A partial list of the possible wakeup sources is shown below.

Table 3. Wakeup Source Selects

	111	110	101	100	011	010	001	000	1xxx
WKPSEL0	PG11	PD15	PD10	PD0	PC2	PB13	PA4	PA0	Reserved
WKPSEL1	PJ12	PG15	PD14	PD13	PC4	PB15	PA5	PA1	Reserved
WKPSEL2	PF10	PD6	PD1	PC0	PB8	PB5	PA6	PA2	Reserved
WKPSEL3	PG6	PD8	PD7	PC5	PC1	PB14	PB10	PA3	Reserved
WKPSEL4	PH5	PH4	PG12	PG5	PD12	PD5	PB9	PB6	Reserved
WKPSEL5	PH8	PH7	PG10	PF12	PE2	PD2	PB7	PA7	Reserved
WKPSEL6	PH9	PG13	PG7	PF14	PF13	PD9	PD3	PC6	Reserved
WKPSEL7	PH6	PG14	PG9	PF15	PF11	PD11	PD4	PB12	Reserved

For example, if the user requires PG11 to be assigned as a wakeup pin, they write b0111 to bits 28–31 of the CRP_WKPINSEL register. Again, as with the timers, the user must first assign and then enable. Selecting the sources as above simply determines which pins are to be assigned if the wakeup function for the pins is in fact enabled.

8.3 Wakeup Source Enable

Up until this point, we have configured and assigned potential wakeup sources.

Sources are disabled by default and are only finally enabled as wakeup sources in the wakeup source enable (CRP_WKSE) register shown below.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R W	WKPDET7		WKPDET6		WKPDET5		WKPDET4		WKPDET3		WKPDET2		WKPDET1		WKPDET0	
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	0	0	0	0	0	RTCOVR EN	RTCWK EN	APIWK EN	0	0	0	0	0	0	0	WKCLK SEL
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 12. Wakeup Source Enable Register (CRP_WKSE)

The RTCOVREN, APIWKEN, and RTCWKEN bits enable the RTC rollover, API, and RTC match wakeups, respectively.

Each of the wakeup pins is enabled in the 2-bit WKPDETx fields:

- 00 — Disables the pin as a wakeup source
- 01 — Enables the pin to wake the part on a positive edge
- 01 — Enables the pin to wake the part on a negative edge
- 11 — Enables the pin to wake the part on any edge

8.4 Wakeup Source Determination

After a wakeup event takes place, assuming that more than one wakeup source was enabled, user software can quickly determine what the source of the wakeup event was by reading the wakeup flags in the power status and control register (CRP_PSCR).

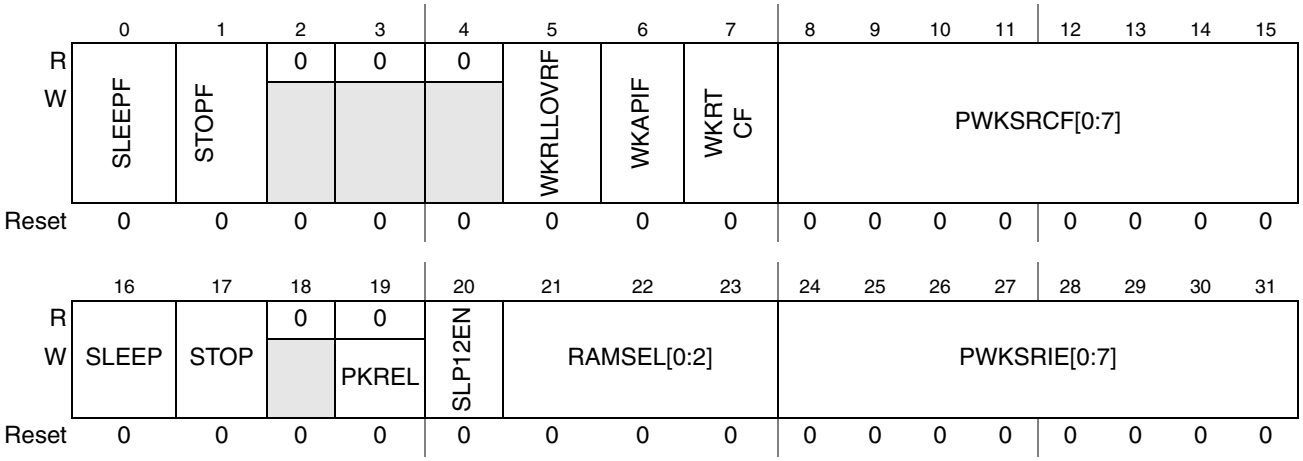


Figure 13. Power Status and Control Register (CRP_PSCR)

Bits 5, 6, and 7 indicate which of the RTC and API match and RTC rollover was responsible for the exit from LPM. Bits 8–15 indicate which of the eight possible wakeup pins was responsible for the LPM exit.

9 Conserving Run Time Power

There are various ways to slow down or turn off clocks on the MPC5510 family to various modules or groups of modules. Because dynamic run current is a direct result of clocking gates (capacitive charging and discharging), anything that can be done to eliminate unneeded clocking or slow down clocks has a dramatic effect on current. For example, slowing a clock to a group of gates (module) down to half frequency reduces the dynamic current by half.

9.1 Disabling Modules

The simplest way to conserve run power is to disable any unused modules on the device. All modules are disabled by default, so rather than disabling unused modules, the user must simply not enable them.

Each module on the MPC5510 family contains an MDIS bit within its control register structure. The MDIS bit is disabled by default. This default state means clocking is disabled to the non-memory-mapped portions of the module. The memory-mapped portions, such as control registers, related to the module operation continue to be accessible.

If a module is enabled and functioning but unlikely to be used for a significant period of time, the user software may choose to disable the module temporarily. To achieve this, the user could simply clear the MDIS bit within the module.

This action however, has the effect of immediately stopping the clocks to the module. This is not necessarily a good thing to happen because the user has no visibility as to what was happening when the module suddenly stopped.

For instance, if a DSPI is transmitting data and the clocks stopped to the module, the data is not transmitted. In short, setting the MDIS bit to stop a module is a very clumsy solution. This type of action is typically used where there is a more urgent requirement to stop all activity instantaneously.

9.1.1 Halt

A much better way of disabling an unused module is to use the halt function. In this way, system clock gating is forced via the centralized halt mechanism. The SIU_HLT register's bits corresponding to individual modules are configured to determine which modules are clock gated.

The HLT bits are used to drive the stop inputs to the modules. After the module completes a clean shutdown, the module asserts the stop acknowledge handshake. The stop acknowledge is visible in the SIU_HLTACK read-only register bits. The modules are individually controlled and halted.

This mechanism is described in more detail in Section 6, "Low Power Entry."

9.2 Slowing Down Clocks

9.2.1 System Clock

The system clock on the MPC5510 family has clock dividers present that can be programmed to create a system clock from the selected clock source divided by one, two, four, or eight. This feature can be used if a system is going into a period where the full performance of the device may not be fully necessary. The divider is set in the SYSCLKDIV bits within the SIU system clock register.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R	SYSCLKSEL		SYSCLKDIV		SWT CLK SEL	0	0	0	0	0	0	0	0	0	0	0
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R	LPCLKDIV7		LPCLKDIV6		LPCLKDIV5		LPCLKDIV4		LPCLKDIV3		LPCLKDIV2		LPCLKDIV1		LPCLKDIV0	
W																
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 14. System Clock Register (SIU_SYSCLK)

- 00 = Div 1
- 01 = Div 2
- 10 = Div 4
- 11 = Div 8

9.2.2 Peripheral Clocks

Peripheral clock dividers on the MPC5510 family provide a mechanism to reduce run current when it is not necessary to run the peripherals at the full system clock frequency.

Each peripheral on the device does not have its own divider to divide down the system clock. The peripherals are grouped together in predefined sets and share a common divider. Within the SIU_SYSCLK register (above), the LPCLKDIVx bits control the clock divide value for each specific group of modules. Table 4 shows how the groups are defined.

Table 4. LPCLKDIV Module Groups

LPCLKDIV (n)	Modules
0	FlexCAN_A DSPI_A
1	ESCI_A IIC_A PIT_RTI
2	FlexCAN_B-F
3	DSPI_B-C
4	ESCI_B-H
5	eMIOS
6	Reserved
7	Reserved

For example, if a user wishes to run group 1 (ESCI_A, IIC_A, PIT_RTI) at one fourth of the system clock frequency, they write 10 to bits 28 and 29 of SIU_SYSCLK register.

Aspects that need to be considered by the user when scaling down clocks to a group of modules include:

- Adjusting the module function, prescalers or protocol timings that may be affected by the reduction in module frequency
- Register accesses within the module is proportionally longer
- Other functions such as DMA and interrupts

9.3 System Clock Choice

There are three possible clock sources available for use as the main system clock on the MPC5510 family:

- 16 MHz IRC
- PLL
- XOSC

Essentially, the choice of clock source is a trade off between performance, accuracy, and power.

The choice of system clock is made in the SIU_SYSCLK register (shown above) by writing the SYSCLKSEL bits.

- 00 selects the internal 16 MHz IRC as the clock source, which is also the default system clock source out of reset. It has a frequency deviation of ≤ 5% and consumes around 160 µA of current.

- 01 selects XOSC to be the system clock source. If this is selected, the system is clocked by the external clock source provided at the EXTAL pin (PLL bypass mode) or a crystal clock reference attached to the XTAL and EXTAL pins. A frequency of between 4 and 40 MHz must be used as a reference. The exact consumption varies. It depends on the crystal type used but is typically in the 500 μA range.
- 10 selects the PLL as the system clock. The PLL reference can be an external crystal or an external clock source. The PLL consumes between 1 mA to 3 mA depending on the frequency. A reference of between 4 and 40 MHz must be provided. The PLL must be used to run the part at frequencies above 40 MHz.

10 Use Cases

With the low power flexibility offered by the MPC55xx range, the user is able to select from a number of different scenarios to optimize their particular system. The following use cases have been documented as potential implementations of a variety of application scenarios:

- Periodically use eMIOS to control a PWM function
- Periodically measure three analog inputs (fast sensors)
- Periodically measure three analog inputs (slow sensors)
- Full O/S constantly available and achieve lowest possible power
- Execute full O/S periodically
- Periodically trigger an external watchdog communication
- Periodically trigger LIN communication
- Intermittently wakeup in response to CAN activity

Each use case must be taken as an example approach only. The ideal solution may well be a combination of several use cases.

The relevant sheets are attached for a convenient reference. However, please check the latest MPC5516 data book before arriving at any final conclusions.

10.1 Use Case Solutions

The following application guidelines is assumed in all use cases, unless explicitly stated otherwise:

- Clock only single core and hold second core in reset
- Leave all other modules in default disabled and non-clocked state

10.2 Use Case Factors and Assumptions

The following assumptions are made in all use cases, unless explicitly stated otherwise:

- 20 μs Vreg recovery time
- 10 μs Vreg shutdown time
- Ambient temperature full run current consumed during the regulator startup and shutdown phases
- Current consumption figures taken from MPC5516 data book, rev 0.1

NOTE

All use cases assume that full run current is consumed at the instant that the regulator is switched on. In fact this is not the case, because the majority of the device is power-gated off until the regulator has stabilized.

10.3 List of Use Cases

10.3.1 Use Case 1: Periodically Use eMIOS to Control a PWM Function

Scenario:

- Periodically use eMIOS to control a PWM function
- Timing of PWM not critical (for example a lighting application)
- PWM function active for approximately 80 µs in every 10 ms

Potential solution:

- Use sleep -> run -> sleep approach
- Use sleep and retain 8K of RAM
- Use API to wake periodically every 10 ms and transition into run
- Clock the API with the on-chip 32 kHz IRC (very-low-power IRC)
- Use 16 MHz IRC for fast execution, accurate enough for this example
- Execute limited code from RAM at 16 MHz bus speed
- Clock eMIOS from 16 MHz IRC

Factors:

- 80 µs execution time (includes PWM initialization and operation)

Assumptions:

- Accuracy of PWM activity not critical

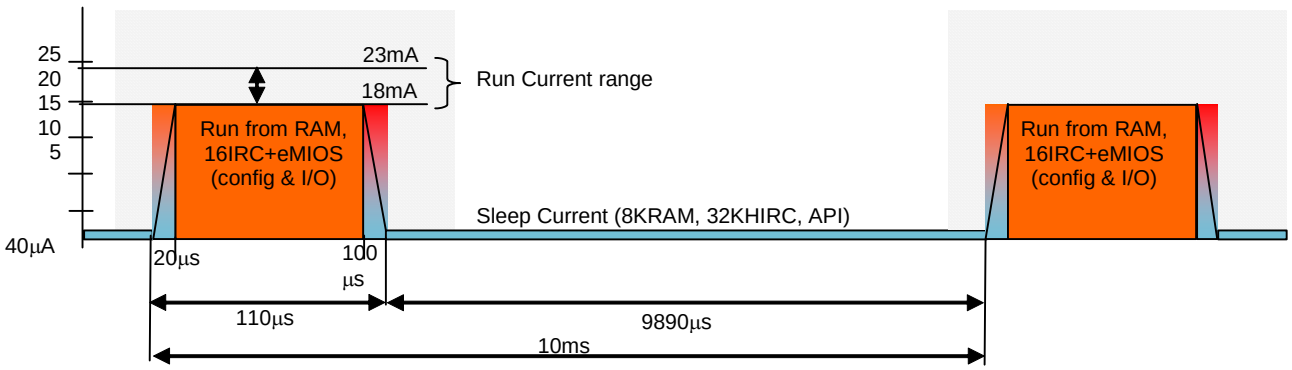


Figure 15. Use Case 1

Average current consumption calculations (based on above duty cycle):

MIN: I_{ave} = (40*9890 + 18000*110)/10000 = 238 µA

MAX: I_{ave} = (40*9890 + 23000*110)/10000 = 293 µA

10.3.2 Use Case 2: Periodically Measure Three Analog Inputs (Fast Sensors)

Scenario:

Periodically:

- Measure three analog inputs using ADC (for example, temperature sensors)
- Check the state of five port inputs
- Perform these checks every 10 ms
- Only continue full power-up if values breech pre-defined conditions
- Timing and absolute accuracy of initial measurements not critical

Potential solution:

- Use sleep -> run -> sleep approach
- Use sleep and retain 8K of RAM
- Use API to wake periodically every 10 ms and transition into run
- Clock the API with the on-chip 32 kHz IRC (very-low-power IRC)
- Use 16 MHz IRC for fast execution, accurate enough for this example
- Execute limited code from RAM at 16 MHz bus speed
- Clock ADC from 16 MHz IRC
- Run conversions in ADC fast mode (leave ADC bypass V_{cap} off)
- Read I/O lines during ADC conversion period
- Return to sleep unless pre-defined conditions exceeded

Factors:

- 10 µs execution time (includes ADC settling and measurement of three inputs)

Assumptions:

- Sensors instantly available for reading (see use case 3 for slower sensor availability)

Diagram:

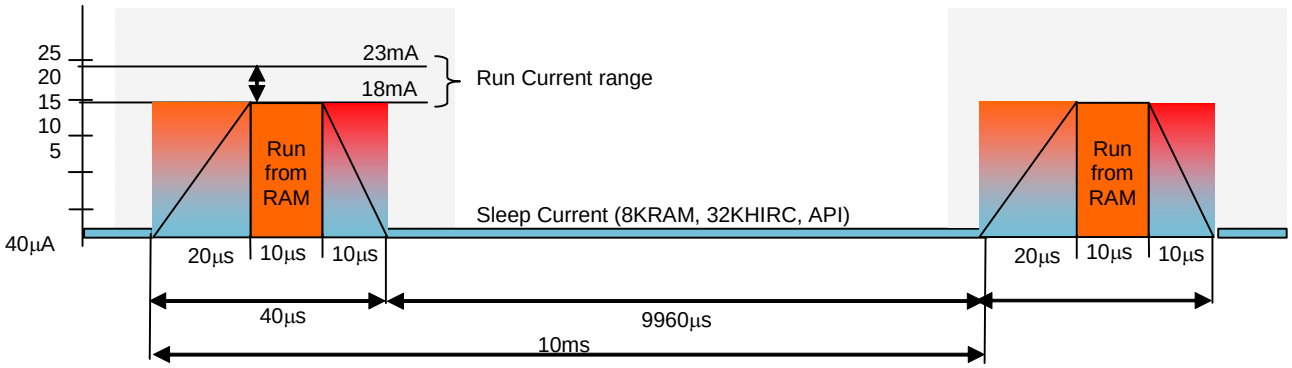


Figure 16. Use Case 2

Average current consumption calculations (based on above duty cycle):

MIN: I_{ave} = (40*9960 + 18000*40)/10000 = 111 µA

MAX: I_{ave} = (40*9960 + 23000*40)/10000 = 132 µA

10.3.3 Use Case 3a: Periodically Measure Three Analog Inputs (Slow Sensors — Solution 1)

Scenario:

- Identical to use case 2, except that the sensors are slower to provide a stable reading
- Periodically:
  - Measure three analog inputs and check values (for example, temperature sensors)
  - Check the state of five port inputs and check values
  - Only continue full power-up if values breech pre-defined conditions
- Timing and absolute accuracy of initial measurements not critical
- ADC function active for approximately 9 µs (3 µs per conversion) in every 10 ms

Potential solution:

- Use sleep -> run -> sleep -> run approach
- Use sleep and retain 8K of RAM
- Use API to wake periodically every 10 ms and transition into run
- Clock the API with the on-chip 32 kHz IRC (very-low-power IRC)
- Use 16 MHz IRC for fast execution, accurate enough for this example application (for example, temperature monitoring)
- Execute limited code from RAM at 16 MHz bus speed
- Set the I/O state to enable/power the external temperature sensors
- Set the API to wake the device after the sensors have stabilized (example 2 ms)
- Return to sleep—the external I/O states is automatically latched and continues to be driven

- Recover again from sleep into run (via the API) after the sensors have stabilized
- Clock ADC from 16 MHz IRC
- Run conversions in ADC fast mode (leave ADC bypass V_{cap} off)
- Read all required I/O lines during ADC conversion period
- Shut down sensor
- Return to sleep unless pre-defined conditions exceeded

Factors:

- 10 µs execution time (includes ADC settling and measurement of three inputs)

Assumptions:

- Sensors not instantly available for reading (see use case two for faster sensor availability)
- Sensor settling time of 2 ms

Diagram:

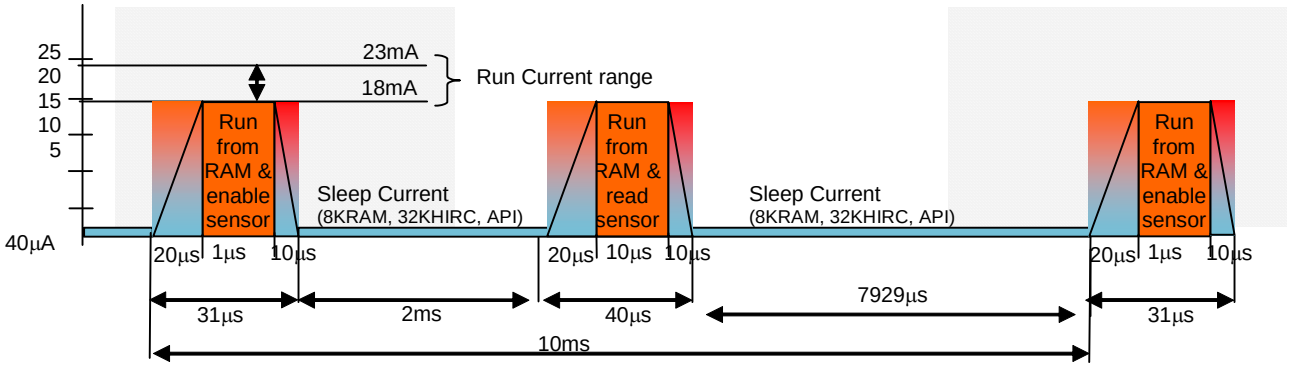


Figure 17. Use Case 3a

Average current consumption calculations (based on above duty cycle):

MIN: I_{ave} = (40*9929 + 18000*71)/10000 = 168 µA

MAX: I_{ave} = (40*9929 + 23000*71)/10000 = 203 µA

10.3.4 Use Case 3b: Periodically Measure Three Analog Inputs (Slow Sensors — Solution 2)

Scenario:

- As per use case 3a

Potential solution:

- The solution to 3b is identical, except the stop state is used instead of sleep during the sensor stabilization period
- Use sleep -> run -> stop -> run approach
- Return to sleep unless pre-defined conditions exceeded

Factors:

- 20 µs Vreg recovery time
- 10 µs Vreg shutdown time
- 10 µs execution time (includes ADC settling and measurement of three inputs)

Assumptions:

- Sensors not instantly available for reading (see use case two for faster sensor availability)
- Sensor settling time of 2 ms

Diagram:

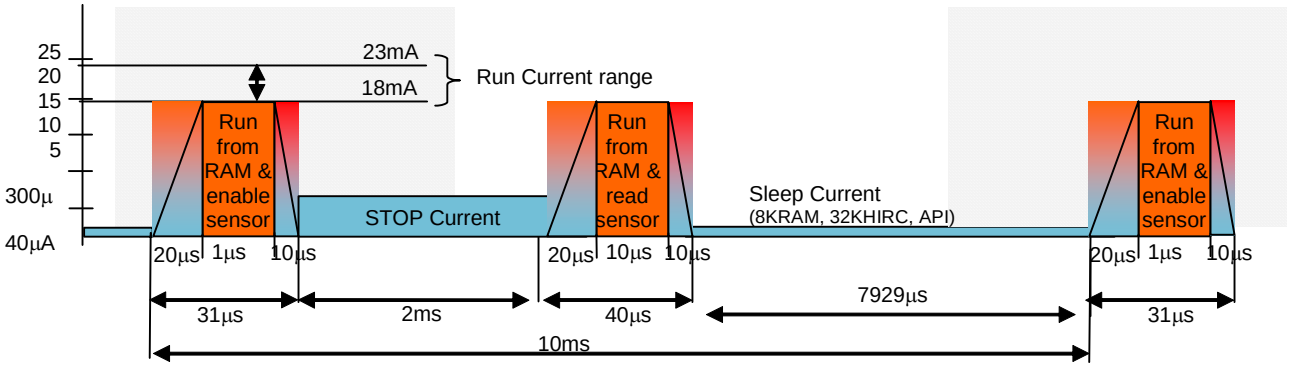


Figure 18. Use Case 3b

Average current consumption calculations (based on above duty cycle):

MIN: I_{ave} = (40*7929 + 18000*71 + 2000*300)/10000 = 220 µA

MAX: I_{ave} = (40*9929 + 23000*71 + 2000*300)/10000 = 263 µA

10.3.5 Use Case 4: Full O/S Constantly Available and Achieve Lowest Possible Power

Scenario:

- Full O/S constantly available, while achieving lowest possible power
- Maintain full core context at all times
- Not required to always execute code operation

Potential solution:

- Use stop -> run -> stop approach
- Use stop and automatically retain all RAM
- All modules and full device context retained during stop
- Transition between run -> stop modes still requires regulator time delays (stop mode regulator smaller than run regulator)
- Use 16 MHz XOSC for code execution, maintain XOSC operation in stop

- Execute full code from flash at 16 MHz bus speed
- Only enable and clock modules required by application
- Leave all other modules in default disabled and non-clocked state

Factors:

- 10 ms execution time (allowing 10 ticks on a 1 ms tick operating system)
- 90 ms non-execution time (in stop)

Assumptions:

- Ignore regulator switching time in this example (swamped by OS)

Diagram:

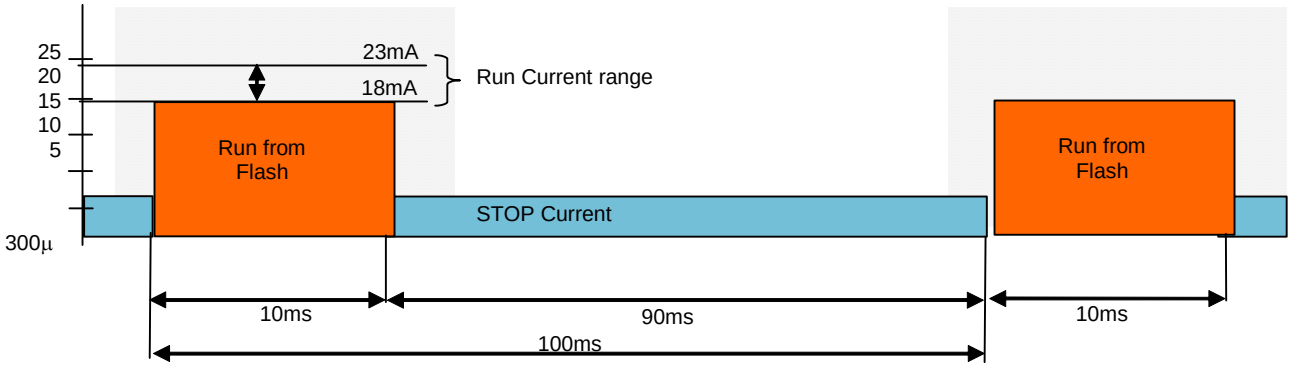


Figure 19. Use Case 4

Average current consumption calculations (based on above duty cycle):

MIN: I_{ave} = (300*90 + 18000*10)/100 = 2070 µA

MAX: I_{ave} = (300*90 + 23000*10)/100 = 2570 µA

10.3.6 Use Case 5: Execute Full O/S Periodically

Scenario:

- Execute full O/S periodically and achieve lowest possible power
- Not required to always maintain full core context
- Not required to always execute code operation

Potential solution:

- Use sleep -> run -> sleep approach
- Perform full context save of MCU and registers before entering sleep
- Perform full context restore of MCU and registers while exiting sleep
- Select appropriate amount of RAM required in sleep
- Consider copying relevant OS sections into RAM to facilitate faster recovery/execution

Factors:

- 10 ms execution time (allowing 10 ticks on a 1 ms tick operating system)
- 90 ms non-execution time (in sleep)

Assumptions:

- Ignore regulator switching time in this example (swamped by OS)
- Full context save takes < 1 ms for > 10,000 instructions @ 16 MHz
- Full context restore takes approximately 0.75 ms for > 10,000 instructions @ 16 MHz bus speed
- >10,000 instructions assumes that all registers, peripherals, CAN message buffers, etc., are being stored—clearly the actual amount stored/restored is application dependent

Diagram:

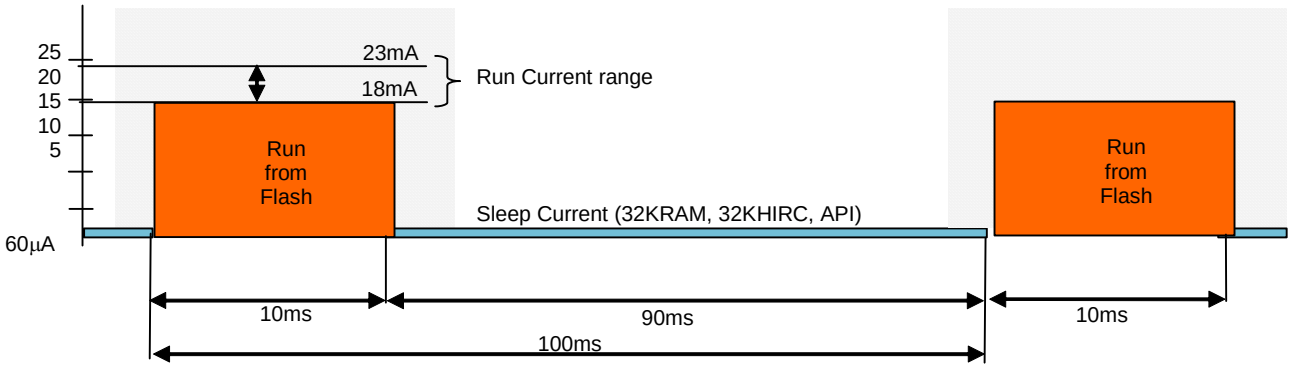


Figure 20. Use Case 5

Average current consumption calculations (based on above duty cycle):

MIN: I_{ave} = (60*90 + 18000*10)/100 = 1854 µA

MAX: I_{ave} = (60*90 + 23000*10)/100 = 2354 µA

10.3.7 Use Case 6: Periodically Trigger an External Watchdog Communication

Scenario:

- Periodically use API to trigger an external watchdog communication
- During wakeup event also sample various pins
- MCU is SPI master; external watchdog is SPI slave
- Timing of SPI allows internal 16 MHz IRC to be used
- SPI communication active for approximately 40 µs
- Initialization and I/O reading active for approximately 80 µs
- Watchdog needs to be accessed every 500 ms

Potential solution:

- Use sleep -> run -> sleep approach
- Use sleep and retain 8K of RAM
- Use API to wake periodically every 500 ms and transition into run
- Clock the API with the on-chip 32 kHz IRC (very-low-power IRC)
- Use 16 MHz IRC for fast execution, accurate enough for this example
- Execute limited code from RAM at 16 MHz bus speed
- Clock SPI from 16 MHz IRC

Factors:

- 120 µs execution time (includes pin sampling, SPI initialization, and operation)

Assumptions:

- Accuracy of SPI activity not critical

Diagram:

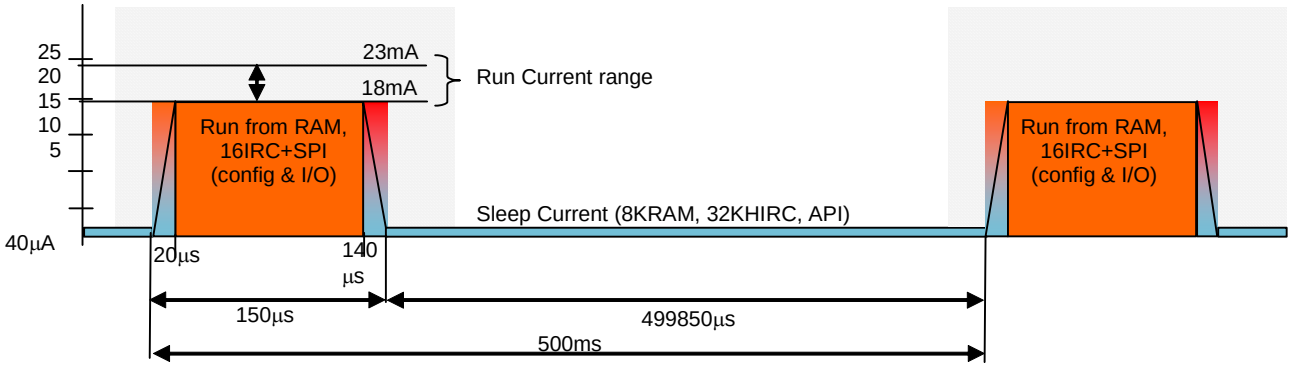


Figure 21. Use Case 6

Average current consumption calculations (based on above duty cycle):

MIN: I_{ave} = (40*499850 + 18000*150)/500000 = 45 µA

MAX: I_{ave} = (40*499850 + 23000*150)/500000 = 47 µA

10.3.8 Use Case 7a: Periodically Trigger LIN Communication (Solution 1)

Scenario:

- Periodically use API to trigger LIN communication
- During wakeup event also sample various pins
- MCU is LIN master
- LIN communication requires < 0.5% clock tolerance, hence use XOSC
- Five LIN frames, each eight bytes in length, transmitted at 19,200 baud
- Total LIN communication is 45 ms (assumes 9 ms per frame with no additional wait/break conditions)

- Initialization and I/O reading active for approximately 80 µs
- LIN communication required every 500 ms (for example, reading alarm status)

Potential solution:

- Use sleep -> run -> sleep approach
- Use sleep and retain 8K of RAM
- Use API to wake periodically every 500 ms and transition into run
- Clock the API with the on-chip 32 kHz IRC (very-low-power IRC)
- Use 16 MHz IRC for fast execution, accurate enough for executing initialization code
- Start the XOSC
- Execute code from flash at 16 MHz bus speed
- Clock SCI from XOSC when stable

Factors:

- 120 µs XOSC stabilization time

Assumptions:

- Initialization / I/O reading / Regulator stabilization is ignored in this use case (negligible)

Diagram:

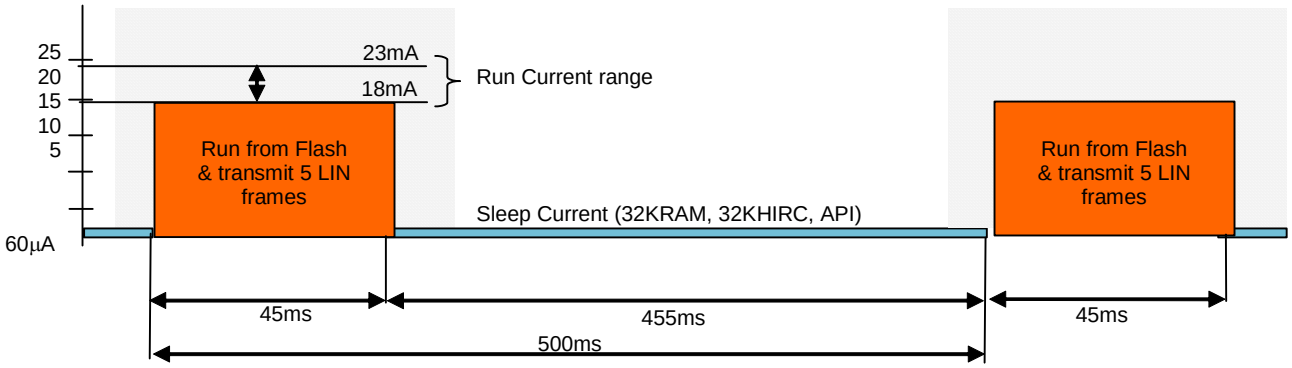


Figure 22. Use Case 7a

Average current consumption calculations (based on above duty cycle):

MIN: I_{ave} = (60*455 + 18000*45)/500 = 1674 µA

MAX: I_{ave} = (60*455 + 23000*45)/500 = 2125 µA

10.3.9 Use Case 7b: Periodically Trigger LIN Communication (Solution 2)

Scenario:

- As per use case 7a, except that more low power intelligence used during the LIN communication

- As before, five LIN frames, each eight bytes in length, transmitted at 19,200 baud
- Total LIN communication remains at 45 ms (assumes 9 ms per frame with no additional wait/break conditions)
- LIN communication required every 500 ms (for example, reading alarm system status)

Potential solution:

- Use sleep -> run -> sleep approach
- Use sleep and retain 8K of RAM
- Use API to wake periodically every 500 ms and transition into run
- Clock the API with the on-chip 32 kHz IRC (very-low-power IRC)
- Use 16 MHz IRC for fast execution, accurate enough for executing initialization code
- Start the XOSC
- Execute code from flash at 16 MHz bus speed
- Set up DMA to transmit five LIN frames
- Set LIN / DMA mode on the SCI to support the automatic transmission/reception of frames
- Configure data to be transmitted
- Configure a second DMA channel for data reception
- Clock system clock (and hence peripherals) from XOSC while stable
- Use clock divider chain to reduce SCI clock speed
- Execute wait mode on main core during LIN communication (rely on autonomous actions of DMA/SCI)

Factors:

- 120 µs XOSC stabilization time

Assumptions:

- Initialization / I/O reading / regulator stabilization is ignored in this use case (negligible)

Diagram:

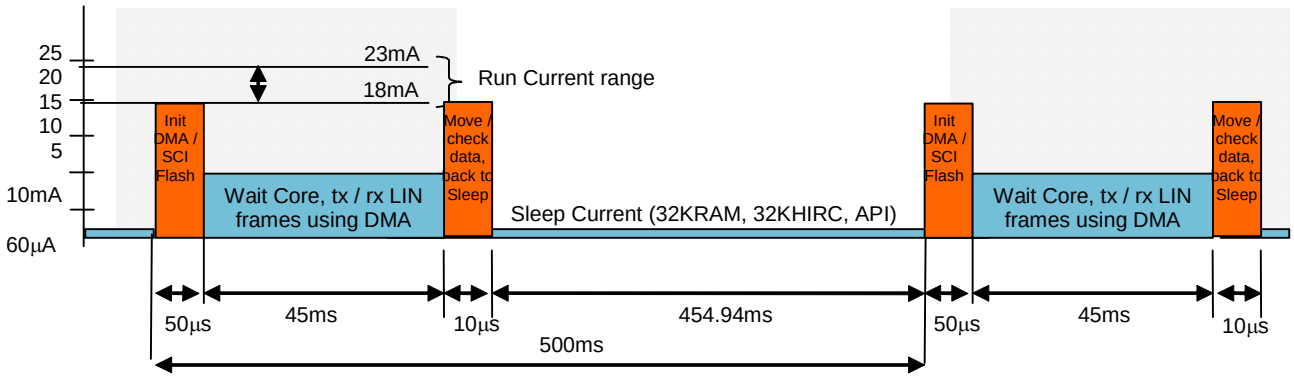


Figure 23. Use Case 7b

<<<get new copy of figure from authors>>>

Average current consumption calculations (based on above duty cycle):

MIN: I_{ave} = (60*454.94 + 18000*0.060 + 10000*45)/500 = 957 μA

MAX: I_{ave} = (60*454.94 + 23000*0.060 + 10000*45)/500 = 957 μA

10.3.10 Use Case 8: Intermittently Wake in Response to CAN Activity

Scenario:

- Intermittently wake in response to CAN activity
- Assume CAN wakeup once in 500 ms for average calculation
- Three standard message CAN frames, each containing eight data bytes, transmitted at 125 Kbps
- Acceptable to miss first message (though must wake)
- Capture second and third CAN messages
- CAN communication requires < 0.5% clock tolerance, hence use XOSC
- Total CAN communication is 2.5872 ms (assumes 3x0.86 ms frames plus 3x24 μs 3-bit interframe space)

Potential solution:

- Use sleep -> run -> sleep approach
- Use sleep and retain 8K of RAM
- Set CAN Rx line as input wakeup pin
- Use 16 MHz IRC for fast execution, accurate enough for executing initialization code (including CAN module initialization)
- Start the XOSC
- Execute code from flash at 16 MHz bus speed
- Clock CAN from XOSC when stable

Factors:

- 120 μs XOSC stabilization time

Assumptions:

- During first CAN message wakeup, start XOSC, initialize CAN module

Diagram:

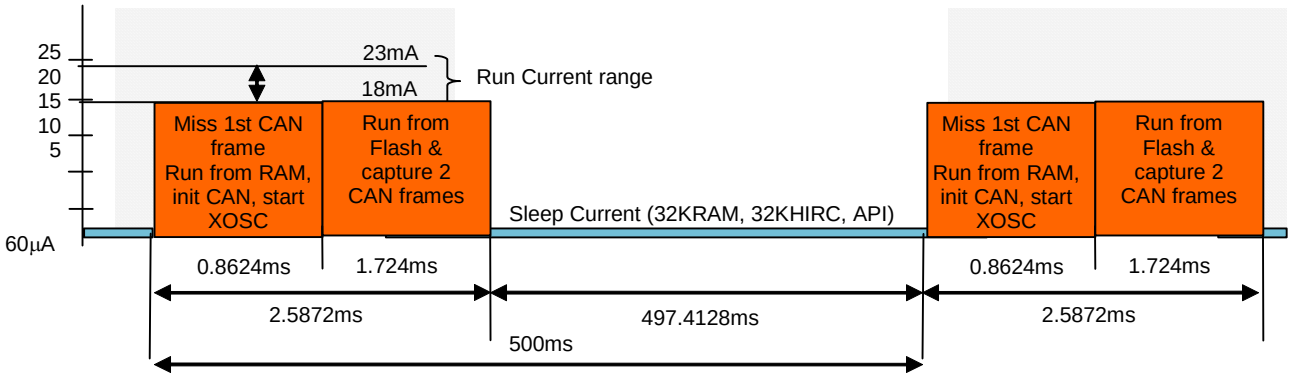


Figure 24. Use Case 8

Average current consumption calculations (based on above duty cycle):

MIN: I_{ave} = (60*497.42 + 18000*2.58)/500 = 152.57 µA

MAX: I_{ave} = (60*497.42 + 23000*2.58)/500 = 178.37 µA

10.4 Use Case Analysis

In use case 1 and use case 2 it has been assumed that the full run current is consumed during the regulator startup and shutdown phases. In fact this is not the case, because the majority of the device is power-gated off until the regulator has stabilized. This factor must be considered while trying to optimize the choice of solution.

Use case 3a differs from case 2 only in that the external temperature sensor takes a predefined time to settle. Hence the suggested strategy is to first of all recover from sleep, quickly power the sensor, and then return to sleep while waiting for the sensor to stabilize. This is possible because during sleep the I/O states continue to be driven.

After the predetermined settling period, use the API to again wake the device, but this time the sensors are ready for reading. The alternative is to transition into stop, which is illustrated in use case 3b. Arguably a simpler implementation, but the average current consumption increases by approximately 30%. This could be significant in some applications with an aggressive power budget.

The final ideal solution depends on the settling time of the sensors.

Use case 4 (stop–run–stop) appears to be preferable to use case 5 (sleep–run–sleep), because there is little additional current and it is a simpler environment.

However, if the duty cycle changes to 10 ms run, 990 ms stop/sleep, then the preference might also change:

Use case 4 with 10 ms / 990 ms duty cycle:

MIN: I_{ave} = (300*990 + 18000*10)/1000 = 477 µA

MAX: I_{ave} = (300*990 + 23000*10)/1000 = 527 µA

Use case 5 with 10 ms / 990 ms duty cycle:

MIN: I_{ave} = (60*990 + 18000*10)/1000 = 240 μA

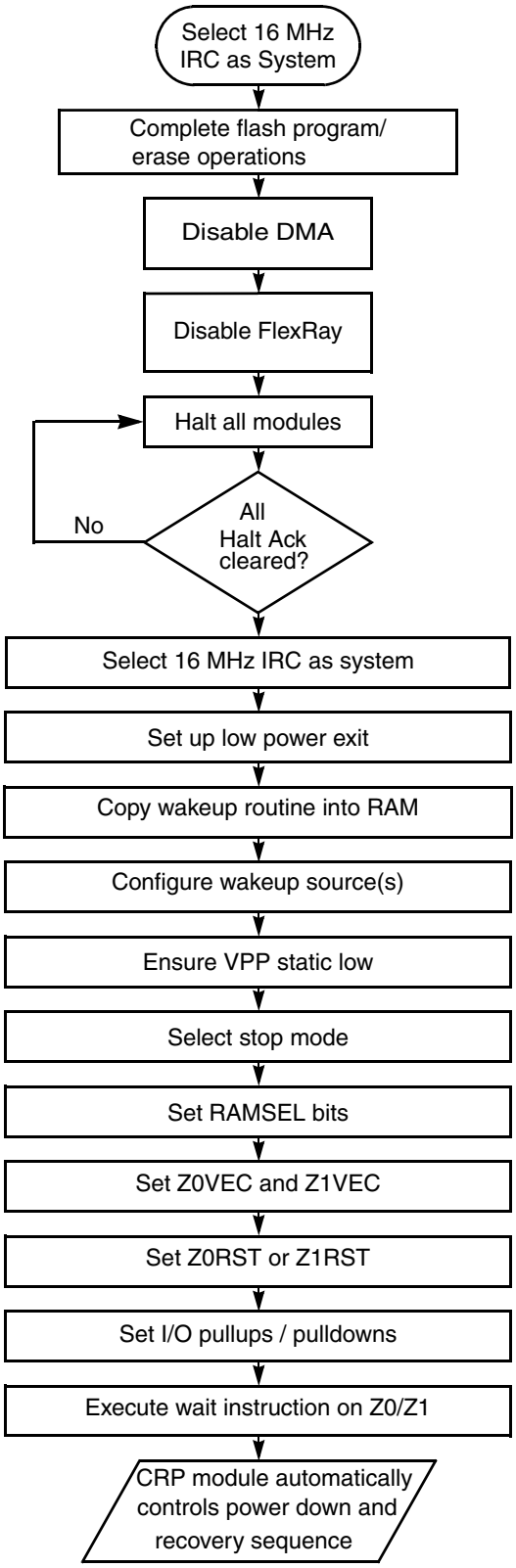
MAX: I_{ave} = (60*990 + 23000*10)/1000 = 290 μA

With this duty cycle clearly the potential reductions to be made are much more significant, because the average consumption can almost be halved. Hence in this example the user might decide to opt for the added complexity of the store/restore approach, but gain a significant current improvement.

In use case 5 it is indicated that over 10,000 instructions are required to perform the store/restore function. This assumes that all registers, peripherals, CAN message buffers, etc., are being stored. The actual amount stored/restored is application-dependent, and the system must be partitioned accordingly during design.

If the most appropriate design is used, then this allows only the necessary core context, peripherals, data structures, etc., to be stored/restored, hence reducing the time taken and the capacity needed to perform the action.

Appendix A Low Power Entry Sleep



All flash write/erase access must be completed before entering sleep

Clean up DMA activity, then disable and check

Clean up FlexRay queue, then disable and check

Write HALT bits to all modules in SIU_HLT register

Verify all HLT_FLG bits are clear before proceeding

IRC is lowest consumption system clock source — can also be allowed to operate in sleep (Main XOSC will automatically be disabled in sleep)

Wakeup routine must be in RAM because flash is disabled in sleep

Set up RTC / API for timed periodic recovery and / or select wakeup pins

Force VPP low externally for lowest current consumption

Select stop mode via CRP_PSCR
No effect until WAIT instruction executed

Select amount of RAM required to be retained in sleep mode (either 8K, 16K, 32K, 64K, or 80K)

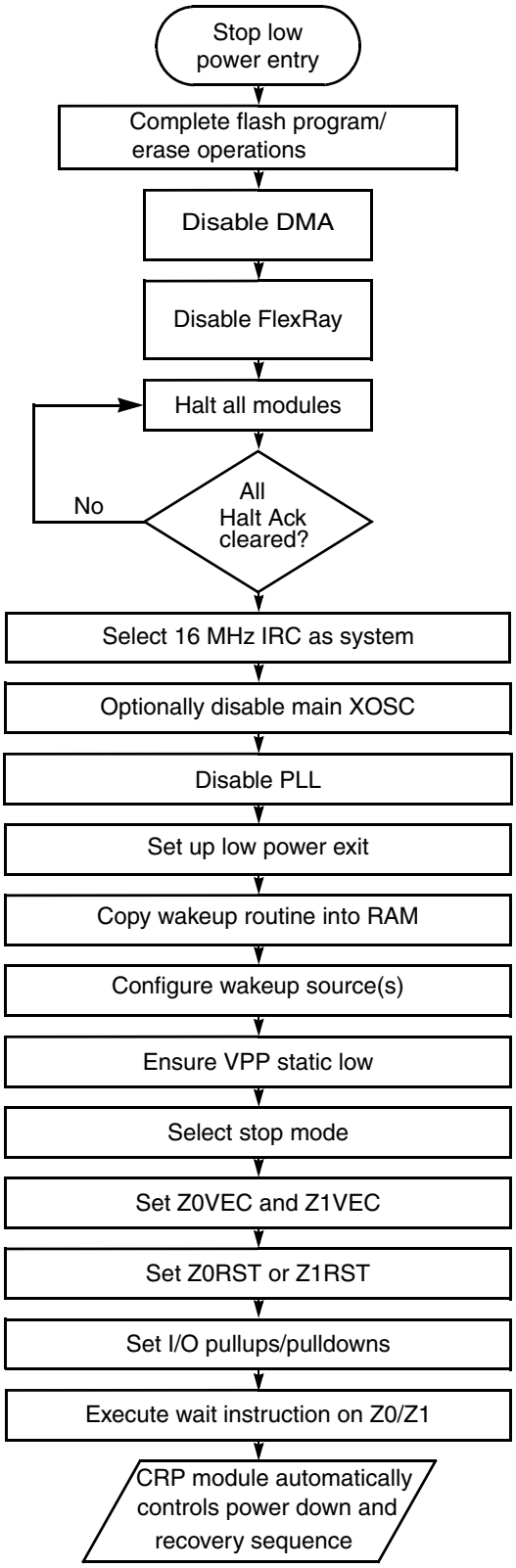
Point core vector(s) to recovery routines

If one core not being used for sleep recovery then hold in reset

Pull down where possible for lowest current consumption

Last user-controlled operation — device now enters sleep

Appendix B Low Power Entry Stop



All flash write/erase access must be completed before entering sleep

Clean up DMA activity, then disable and check

Clean up FlexRay queue, then disable and check

Write HALT bits to all modules in SIU_HLT register

Verify all HLT_FLG bits are clear before proceeding

IRC is lowest consumption system clock source — can also be allowed to operate in sleep

Main XOSC can be allowed to continue running in stop (set XOSCENSTOP bit)

PLL cannot run in stop

Flash powered down in stop, so still necessary to place recovery routine in RAM

Set up RTC/API for timed periodic recovery and/or select wakeup pins

Force VPP low externally for lowest current consumption

Select amount of RAM required to be retained in sleep mode (either 8K, 16K, 32K, 64K, or 80K)

Point core vector(s) to recovery routines

If one core not being used for stop recovery then hold in reset

Pull down where possible for lowest current consumption

Last user-controlled operation — device now enters stop

Appendix C MPC5516 Electrical Reference Data

Ref	Characteristic	Symbol	Typical[1] @25°C Ambient	Typical @70°C Ambient	Max –40°C to 150°C Junction	Unit
Equations	$I_{Total} = I_{DDE} + I_{PP} + I_{DDA} + I_{DDR}$ $I_{DDE} = I_{DDE1} + I_{DDE2} + I_{DDE3}$ $I_{DDR} = I_{DD33} + I_{DDSYN} + I_{Flash} + I_{DDF} + I_{DD}$					
1	$V_{DDE(1,2,3)}$ Current $V_{DDE(1,2,3)}$ @ 3.0 V–5.5 V	I_{DDE}				
	Static[2]		1 Note[3]	3 Note[3]	30 Note[3]	µA mA
2	V_{PP} Current	I_{PP}				
	V_{PP} @ 0 V		1	1	1	µA
	V_{PP} @ 5.25 V					
	Sleep Mode Stop Mode Standby, Read Program/Erase RWW		15 15 1 3 5	20 20 1 3 5	30 30 2 3 5	µA µA mA mA mA
3	V_{DDA} Current V_{DDA} @ 4.5 V–5.25 V	I_{DDA}				
	Sleep Mode, Stop Mode Run Mode[4]		11 1	15 1	25 2	µA mA
	Optional: 32KIRC Optional: 32KOSC Optional: 16MIRC (Sleep/Stop Modes) Optional: ADC Active (Run Mode)		1 1 100 5	1 1 150 5	1 3 200 6	µA µA µA mA
4	V_{DD33} Current V_{DD33} @ 3.0 V–3.6 V[5]	I_{DD33}				
	Static[1] Dynamic[3]		1 Note[3]	1 Note[3]	1 Note[3]	µA mA
5	V_{DDSYN} Current V_{DDSYN} @ 3.0 V–3.6 V[5]	I_{DDSYN}				
	Sleep Mode, Stop Mode Run Mode[6]		1 1	5 5	20 20	µA µA
	Optional: XOSC (Sleep and Stop Modes) Optional: XOSC (Run Mode) Optional: PLL (Run Mode)		500 0.8 3	600 2 5	900 3 6	µA mA mA

Ref	Characteristic	Symbol	Typical[1] @25°C Ambient	Typical @70°C Ambient	Max –40°C to 150°C Junction	Unit
6	V_{Flash} Current V_{Flash} @ 3.0 V–3.6 V[5]	I_{Flash}				
	Sleep Mode		1	10	50	µA
	Stop Mode		1	10	50	µA
	Standby		1	1	2	mA
	Read		1	1	2	mA
	Program/Erase		1	1	2	mA
	RWW		1	1	2	mA
7	V_{DDF} Current V_{DDF} @ 1.35 V–1.65 V[5]	I_{DDF}				
	Sleep Mode		1	1	50	µA
	Stop Mode		1	1	50	µA
	Standby		1	1	2	mA
	Read		1	1	2	mA
	Program/Erase		1	1	2	mA
	RWW		1	1	2	mA
8	V_{DD} Current V_{DD} @ 1.35 V–1.65 V[5]	I_{DD}				
	Sleep Mode		15	30	120	µA
	Stop Mode		300	600	2000	µA
	Run Mode (Static)		10	15	20	mA
	Run Mode (Maximum @ 16 MHz)		35	40	45	mA
	Run Mode (Maximum @ 66 MHz)		100	110	120	mA
	Optional: RTC/API		1	1	3	µA
	Optional: Each 8K RAM Block (Sleep Mode)		0.8	7	45	µA

1 Typical — nominal voltage levels and functional activity. Max — maximum voltage levels and functional activity.

2 Static state of pins is when input pins are disabled or not being toggled and driven to a valid input level, output pins are not toggling or driving against any current loads, and internal pull devices are disabled or not pulling against any current loads.

3 Dynamic current from pins is application specific and depends on active pull devices, switching outputs, output capacitive loads, output current loads, and switching inputs. Refer to *MPC5510 Microcontroller Family Data Sheet* for more information.

4 16MIRC optionally disabled in sleep and stop modes, always active in run mode. 32KIRC and 32KOSC optionally enabled/disabled independent of mode. ADC always disabled in sleep and stop modes, optionally enabled in run mode. Base sleep and stop modes assume 16MIRC, 32KIRC, and 32KOSC are disabled. Base run mode assumes 32KIRC, 32KOSC, and ADC are disabled.

5 Voltage generated from internal regulator.

6 XOSC optionally enabled in sleep and stop modes (oscillator remains running from crystal but XOSC clock output disabled); XOSC optionally enabled in run mode. PLL only optionally enabled in run mode. Base sleep and stop modes assume XOSC is disabled. Base run mode assumes XOSC and PLL are disabled.

Appendix D Example Code

```
/*************************************************************************/
/* Code to enter and exit Low Power Modes.   */
/*************************************************************************/

#include "mpc5516.h"
#include "typedefs.h"

#define PULL_DOWN0
#define LPM_SLEEP0

extern execute_wait(void);    /*  defined in execute_wait.s */
extern sleep_recovery(void);  /*               "                          */
extern void _start(void);     /* defined  in     z1_crt0.s      */

void enter_low_power_mode(uint32_t LPM);
void Enable_all_internal_pull_devices ( uint32_t pull_select);
void wake_up(void);
void release_padkeepers(void);
void delay(void);

int8_t counter;

int32_t test[76];

/*************************************************************************************/
/* define the z0 code start location. This address is where the z0 start code is forced in
z0_crt0.s  */
/* using the.org assembler directive.                                                */
/*************************************************************************************/
#define Z0_ENTRY_POINT 0x00002000


void main()
{
      int i;

      /* RAM CODE; setup MMU, disable watchdog, set value on port pin PD12, turn on PD12 as
      output, release padkeepers, setup stack , delay, and branch using link register to the */
      /* function enter_low_power_mode() residing in FLASH.  While going to sleep, Z1VEC will
      be written with the  starting address of the array "test" in RAM */
      /* delay is currently set to 100uS approx */

      int32_t temp[76] =
      {0x7162E000,0x7160C000,0x7D709BA6,0x7178E000,0x7160C400,0x7D719BA6,0x7168E000,0x7160C02
      8,0x7D729BA6,0x7168E000,0x7160C03F,0x7D739BA6,0x7C0004AC,0x00017C00,0x07A40001,0x7162E0
      01,0x7160C000,0x7D709BA6,0x7178E000,0x7160C500,0x7D719BA6,0x717FE7F0,0x7160C02A,0x7D729
      BA6,0x717FE7F0,0x7160C03F,0x7D739BA6,0x7C0004AC,0x00017C00,0x07A40001,0x7162E002,0x7160
      C000,0x7D709BA6,0x7178E000,0x7161C100,0x7D719BA6,0x7160E000,0x7160C020,0x7D729BA6,0x716
      0E000,0x7160C03F,0x7D739BA6,0x7C0004AC,0x00017C00,0x07A40001,0x73FFE7F4,0x4806AB0F,0x74
      C03E30,0xBB0FE801,0x73E8E000,0x807F2007,0x907F73FF,0xE7FE73F0,0xC63C907F,0x73FFE7FE,0x7
      3F0C0B8,0x480770E0,0xC20CB07F,
      0x73FFE7FE,0x73F8C060,0x480770E2,0xC000D07F,0x480770E0,0xC1482A07,0xE6032407,0xE8FD7028
      ,0xE0017034,0xC00071A8,0xE00071AF,0xC7F07040,0xE0017050,0xC0001801,0x06C04807,0x70E0C07
      0,0x00970004,0xAAAAAAAA};
```

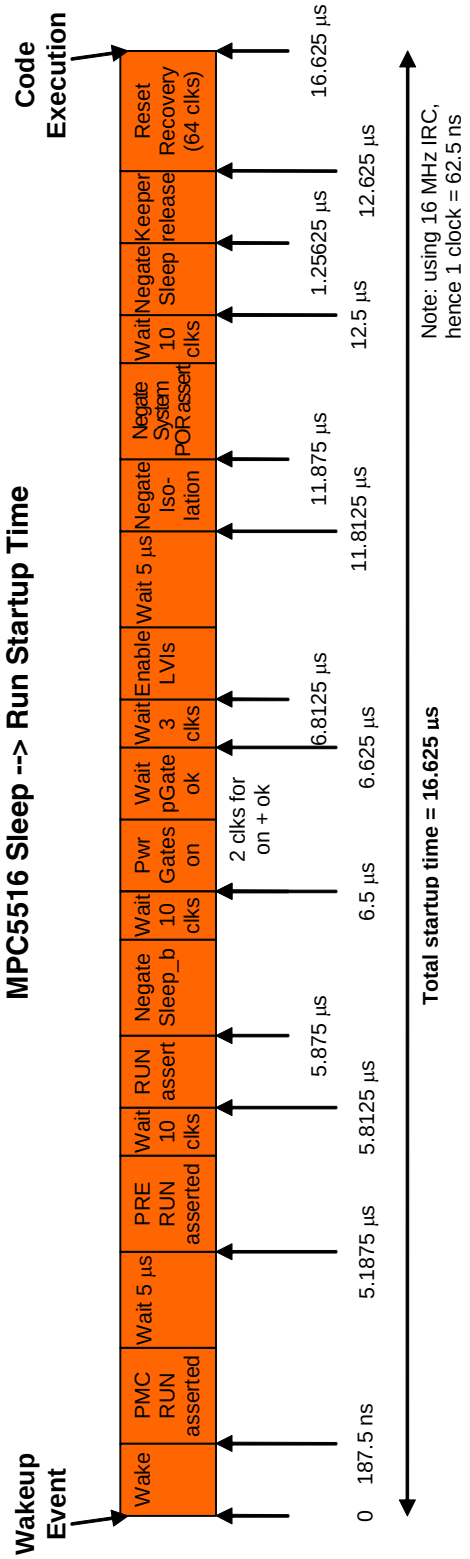
```
	MCM.SWTCR.B.SWE = 0;/*disable watchdog*/
	counter                                                                 = 0;

	/* Peripherals can be "init"ed here. */
	{
		SIU.GPDO[60].R  = counter;              /* Start at high for starters */
		SIU.PCR[60].R   = 0x020c;          /* Setup PD12 for output;  max slew rate */
	}

	/* Copy "temp" onto "test" */
	for(i = 0; i < 76; i++)
	{
		test[i] = temp[i];
	}

	/* Your main  run code here. This test code toggles port pin PD12  */
	{
		SIU.GPDO[60].R  = ~(SIU.GPDO[60].R );
		delay();
		SIU.GPDO[60].R  = ~(SIU.GPDO[60].R );
		delay();
		SIU.GPDO[60].R  = ~(SIU.GPDO[60].R );
		delay();
		SIU.GPDO[60].R  = ~(SIU.GPDO[60].R );
		delay();
		SIU.GPDO[60].R  = ~(SIU.GPDO[60].R );
		delay();
		SIU.GPDO[60].R  = ~(SIU.GPDO[60].R );
		delay();
	}

	/* Setup wakeup source(API) and 32 kHz IRC  */
	{
		CRP.CLKSRC.B.KIRCEN = 1;                            /* enable 32K IRC */
		CRP.RTCSC.R = 0x8000000A;   /* set value for API : count of 0x0A for a 10 ms
	pulse width. */
		CRP.RTCSC.R = 0x8000800A;   /* enable counter,  API  with 32 kHz  IRC  */
		CRP.WKSE.B.APIWKEN  = 1;              /* select API as wake up source*/
	}


	/* Enter sleep mode */
	{
		enter_low_power_mode(LPM_SLEEP);
	}

	} /* END of Main */

/******************************************************************************
FUNCTION        : enter_low_power_mode
PURPOSE                   : Configures part for stop mode and executes wait instruction
INPUTS NOTES    : LPM_STOP goes to stop mode and LPM_SLEEP goes to sleep mode
RETURNS NOTES   : If LPM_STOP enabled, then function will return a PASS if if successfully
                                        recovers from stop mode and ISR.  If LPM_SLEEP
enabled, Z1VEC register
                                        will be written to beginning of __start routine and
it will begin execution
```

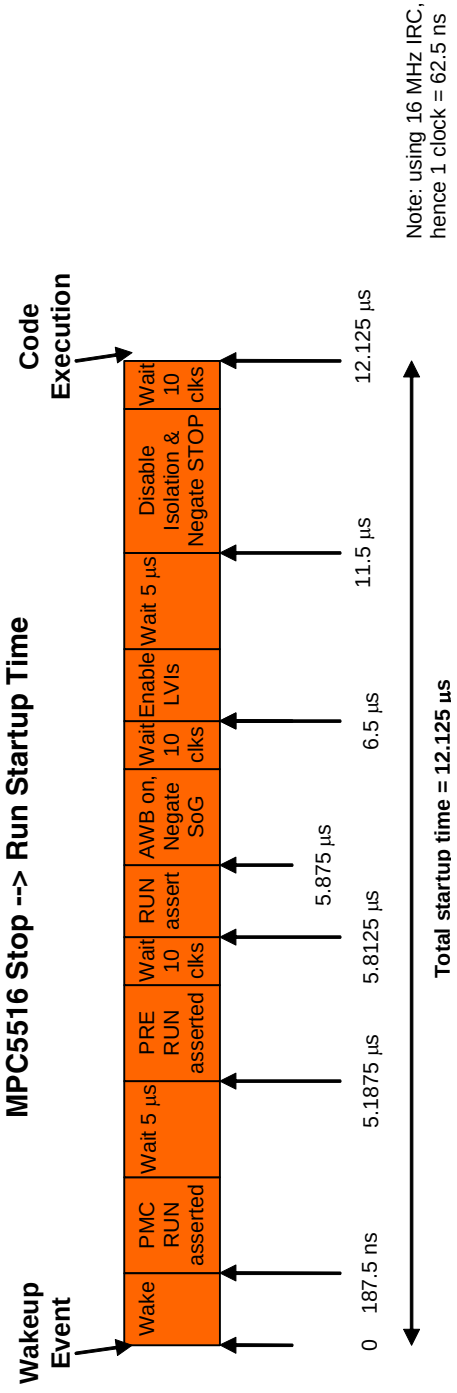
```
                                                there.
GENERAL NOTES   : Writes to CRP stop/sleep mode, halts all modules, executes wait.
*********************************************************************************************
*/
void enter_low_power_mode (uint32_t LPM)
{

       {
               counter++;
               SIU.GPDO[60].R = counter;
       }


       {
               CRP.PSCR.B.SLEEP  = 1;    /* Set the sleep bit; following a WAIT instruction,
       the device will go to sleep */
               CRP.PSCR.B.STOP12EN = 1;/* enable the 1.2V internal regulator when in sleep mode
       only */
               CRP.PSCR.B.RAMSEL   = 0x6;/* 0x1 8k, 0x2 16k, 0x3 32k, 0x6 64k -- RAMs maintain
       power */
       }



       /*********************************************************************/
       /* NOTE: Z1VEC.R must be forced to be near the beginning of a 4k boundary if  */
       /* the Z1 core is to be used when coming out of sleep so that there is adequate      */
       /* room for code to reside in the 4k boundary that sets back up the MMU               */
       /* This is done by using the .org directive to ensure that the  sleep_recovery         */
       /* function starts from a 4k aligned address.  This address could be location in      */
       /* RAM or FLASH. 1 is added to the recovery address to ensure that the VLE bit   */
       /* is set in Z1VEC.R  NOTE(2): Z0 can also be used and it does not need the   4k    */
       /* alignment because it doesn't go through the MMU.                                    */
       /*********************************************************************/
       {
               vuint32_t temp = 0;
               temp                                                = (uint32_t)(test) + 1;
               CRP.Z1VEC.R = temp;
               CRP.RECPRTR.B.FASTREC = 1;  /* turn on fast recovery from sleep */
       }


       /*********************************************************************************/
       /* halt all modules except Z1 and Z0. Note DMA and FlexRay must have had a controlled
       shut down */
       /* before writing to the halt register (because the Halt bits will immediately stop these
       two modules)    */
           /* The timeout value needs to be asjusted to the system requirements to reflect the
       time needed to  */
       /* complete pending module activities i.e., CAN transmissions, SPI transmissions etc.*/
       /*********************************************************************************/
       {
               vuint32_t timeout = 0;
               vuint32_t xtemp = 1;
               SIU.HLT.R = 0x3FFFFFFF;
               while((SIU.HLTACK.R != 0x3FFFFFFF) && (timeout<3000))
               {
                                                               xtemp = SIU.HLTACK.R;
                                                                       timeout++;
```

```
                }
                CRP.Z0VEC.B.Z0RST = 1;      /* put Z0 in reset if not used for wakeup */
        }

           /* Enable_all_internal_pull_devices() is called to eliminate floating inputs in sleep
        */
        Enable_all_internal_pull_devices (PULL_DOWN);

        execute_wait();  /* Part enters sleep here; returns to" ZOVEC" or " Z1VEC"  */
        }

/*************************************************************************
FUNCTION        : Enable_all_internal_pull_devices
PURPOSE            : to pull up or pull down all I/O pins except Port Pin PD12.
INPUTS NOTES    : Pass in PULL_UP to enable the pull ups and PULL_DOWN to enable the pull downs..
RETURNS NOTES   : none
GENERAL NOTES   :
*************************************************************************/
void Enable_all_internal_pull_devices ( uint32_t pull_select)
{
        /************************************************************************/
        /* Pull DOWN all pins on EVB for the lowest currents [except EVTI .  EVTI (PCR[80]) */
           /* has to be pulled up because the EVB has it  pulled high(Nexus enable)
        */
        /************************************************************************/
        uint32_t i=0;
        for(i = 0; i <= 59; i++)
        {
                SIU.PCR[i].R = 0x0002 + pull_select;
        }

        /* Skipping PCR[60] which is Port Pin PD12 */

        for(i = 61; i <= 145; i++)
        {
                SIU.PCR[i].R = 0x0002 + pull_select;
        }
        SIU.PCR[80].R = 0x3; /* Pull up enabled due to EVTI pull up on EVB     */

}

/*********************************************************************/
/* In the execute_wait.s file, the recovery routine "sleep_recovery" configures  */
/* the MMU, sets up the stack and then calls this function.                                */
/*********************************************************************/
void wake_up(void)
{
           /*************************************************************/
        /* WatchDog is automatically reset (re-enabled)  when exiting sleep,      */
        /* either service the WatchDog or turn it off. Here it is turned off.    */
        /*************************************************************/
        {
                MCM.SWTCR.B.SWE = 0;                                    /*disable watchdog*/
        }

        counter++;              /* A variable in RAM; gets incremented each time this function
        is called */
```

```
	/*************************************************************/
	/* Sample  code after waking up from sleep;  here we are toggling PD12  */
	/* each time we exit sleep. Incrementing a byte variable "counter"          */
	/* and writing it to the port register.                                     */
	/* NOTE: Pad keepers maintain the state of the outputs in sleep but the */
	/* SIU registers are reset in sleep,so setup SIU registers before           */
	/* releasing the padkeepers.                                                */
	/* Also, all peripheral reinitializations [except the CRP] must be done    */
	/* because they were powered off during sleep.                              */
	/*************************************************************/
	{
		SIU.GPDO[60].R = counter;
		SIU.PCR[60].R  = 0x020c;
		release_padkeepers();
	 }

	{
		/* code that needs to be run after waking up from sleep can */
		/* be put here */
	}

	enter_low_power_mode(LPM_SLEEP); /* back into sleep */
	}


void disable_watchdog(void)
{
	MCM.SWTCR.B.SWE = 0;/*disable watchdog*/
}

void release_padkeepers(void)
{
	CRP.PSCR.B.PKREL = 1;
}

void delay()
{
	unsigned int temp = 0;
	while(temp < 0xA)
	{
		temp++;
	}
}
```

Appendix E Startup Times in Clock Cycles




MPC5516 Stop --> Run Startup Time
Wakeup Event
Code Execution
Wake
PMC RUN asserted
Wait 5 µs
PRE RUN asserted
Wait 10 clks
RUN assert
AWB on, Negate SoG
Wait 10 clks
Enable LVIs
Wait 5 µs
Disable Isolation & Negate STOP
Wait 10 clks
0
187.5 ns
5.1875 µs
5.8125 µs
5.875 µs
6.5 µs
11.5 µs
12.125 µs
Total startup time = 12.125 µs
Note: using 16 MHz IRC, hence 1 clock = 62.5 ns

How to Reach Us:

Home Page:
www.freescale.com

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

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
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 Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Number: AN3584
Rev. 0
5/2008

freescale™
semiconductor