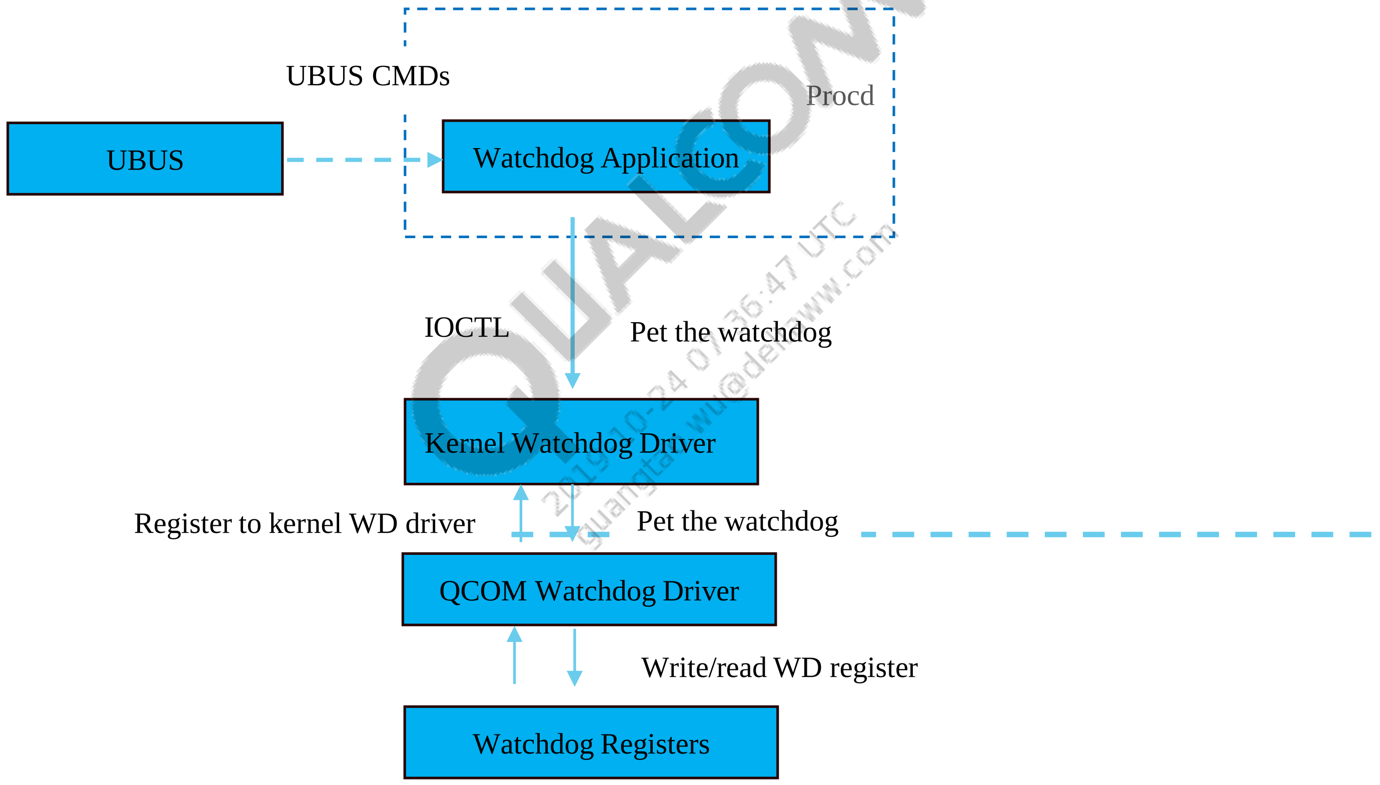
Watchdog Software Architecture



Watchdog SW workflow

- Watchdog application: ‘procd-2015-01-25/watchdog.c ‘
  - The APCS_KPSS_WDT / APCS_WDOG sub-system is used /managed by the standard Linux watchdog code, if the user space daemon in ‘procd-2015-01-25/watchdog.c ‘ does not call qcom_wdt_ping() frequently enough (default every X30 seconds) that are defined in “watchdog.c”), the bite signal is asserted.
- Watchdog driver: qcom-wdt.c
  - qcom_wdt_probe ---> watchdog_register_device.
  - After the above code, /dev/watchdog is created. So user space application can use /dev/watchdog to control the watchdog by ioctl system call. If the user space daemon in ‘procd-2015-01-25/watchdog.c‘ does not call qcom_wdt_ping() frequently enough, the bite signal is asserted.

ubus operation to watchdog in user space

- #To query watchdog status
  - ubus call system watchdog
- #To stop watchdog
  - ubus call system watchdog '{ "stop": true }'
- #To start watchdog
  - ubus call system watchdog '{ "stop": false }'
- #To configure watchdog timeout as 20 seconds (default is 30 seconds)
  - ubus call system watchdog '{ "timeout": 20}'

Related files for watchdog

- Kernel side
  - . /build_dir/target-arm_cortex-a7_uClibc-0.9.33.2_eabi/linux-ipq806x/linux-3.14.43 drivers/watchdog/watchdog_dev.c
  - . /build_dir/target-arm_cortex-a7_uClibc-0.9.33.2_eabi/linux-ipq806x/linux-3.14.43 drivers/watchdog/qcom-watchdog.c
- User space side
  - ./build_dir/target-arm_cortex-a7_uClibc-0.9.33.2_eabi/procd-2015-01-25/watchdog.c

Bark and bite interrupt

- Current IPQ40xx 1.0 CS/1.1 CSU1 QSDK release doesn't support bark interrupt, and the future QSDK release will support kernel handling of watchdog bark interrupts. Related code changes are already published in codeaurora.com, if you want to enable the bark interrupt, it needs to add the following code changes, and rebuild the codes.
  - https://us.codeaurora.org/cgit/quic/qsdk/oss/kernel/linux-msm/tree/drivers/watchdog/qcom-wdt.c?h=coconut_20140924
  - https://us.codeaurora.org/cgit/quic/qsdk/oss/kernel/linux-msm/tree/arch/arm/boot/dts/qcom-ipq40xx.dtsi?h=coconut_20140924
- Related files are,
  - qcom-watchdog.c
    - Add the changes that watchdog requests irq for bark interrupt.
  - qcom-ipq40xx.dtsi
    - Add the interrupt name and interrupt number for bark interrupt in ipq40xx's dts files.
- Bark IRQ is handled in Kernel. However, BITE FIQ is routed to TZ.

Condition to trigger watchdog

- There are two conditions that can trigger APCS KPSS watchdog.
  - Core hang
  - Kernel panic/oop
- The two conditions cause the bite event hander in TZ code to run, and then it will dump the SoC information to ram. It will automatically upload these SoC information from ram to your tftp server, then you can dump it with a python script we provided to find clue where it hang.
- To get the dump file.
  - Configure your board ip and tftp server ip address under uboot.
  - Connect your tftp server to your board and enable tftp server.
  - Wait the hang and the system is rebooted.
  - After the system is rebooted and enters uboot, the dump files will be uploaded by tftp automatically.

Parse the dump files

- parse tools:
  - <qsdk>/build_dir/host/linux-ramdump-parser-v2/linux-ramdump-parser-v2
- Command to dump the file:

```
python ramparse.py --ram-file file <crash-dump-location>/EBICS0.bin
0x80000000 0x8fffffff --vmlinux <qsdk-location-vmlinux>/openwrt-ipq806xvmlinux.elf
--phys-offset 0x80000000 --force-hardware 4018 --gdb-path
<qsdk-location-gdb>/arm-openwrt-linux-gdb --nm-path <qsdk-location-nm>/armopenwrt-
linux-nm --parse-debug-image --everything --outdir <output-files-location>
```

Watchdog clock

- Watchdog runs with 32KHZ Clock. For setting the timeout for bark and bite timer, it needs the timeout to multiple the watchdog clock rate to get the value for the bark and bite time registers.

Configure bite and bark timeout

- BITE/BARK timeout is configurable, and bite timeout is configured by ubus command. For example, set the bite timeout,
  - ubus call system watchdog '{ "timeout": 20 }'
- Currently, the bark timeout is default to be the bite timeout subtract 1, and therefore the bark timeout is 19 seconds in this case.

WDOG expiry tests

- Make sure all 4 CPU are loaded to test watchdog. I have attached “dead.c”, and that can make system to be dead, and then the watchdog will expires. We does not support per core WDOG currently, this can be implemented in SW.

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kthread.h>
#include <linux/spinlock_types.h>

static DEFINE_SPINLOCK(test_lock);

static int process_test(void *info)
{

  spin_lock(&test_lock);
  while(1);
  return 0;
}
```

WDOG expiry tests – Cont.

```
static int __init init_3mb_alloc(void)
{
        struct task_struct *p1, *p2, *p3, *p4;
        printk("MODULE\tINITIALIZED\n");

        p1 = kthread_create(process_test, NULL, "TEST_CPU_1_THREAD");
        kthread_bind(p1, 0);

        p2 = kthread_create(process_test, NULL, "TEST_CPU_2_THREAD");
        kthread_bind(p2, 1);

        p3 = kthread_create(process_test, NULL, "TEST_CPU_3_THREAD");
        kthread_bind(p3, 2);

        p4 = kthread_create(process_test, NULL, "TEST_CPU_4_THREAD");
        kthread_bind(p4, 3);

        wake_up_process(p1);
        wake_up_process(p2);
        wake_up_process(p3);
        wake_up_process(p4);

        return 0;
}
```

WDOG expiry tests – Cont.

```
static void __exit exit_3mb_alloc(void)
{
    printk("MODULE\tTERMINATED\n");
}

module_init(init_3mb_alloc);
module_exit(exit_3mb_alloc);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("QUALCOMM");
```

GCNT Watchdog

GCNT_WDOG

The GCNT_WDOG sub-system is used exclusively by the Trustzone /QSEE code, there is no interaction with the scm calls present in the normal kernel code. TZ will configure the xPU to protect GCNT Watchdog from accessing in kernel or user space.

Register	Addr	Type	Retention?
GCNT_RESET	0x4AA000	Write	NO
GCNT_CTL_REG	0x4AA004	Read/Write	NO
GCNT_STATUS_REG	0x4AA008	Read	NO
GCNT_BARK_VAL_REG	0x4AA00C	Read/Write	NO
GCNT_BITE_VAL_REG	0x4AA010	Read/Write	NO

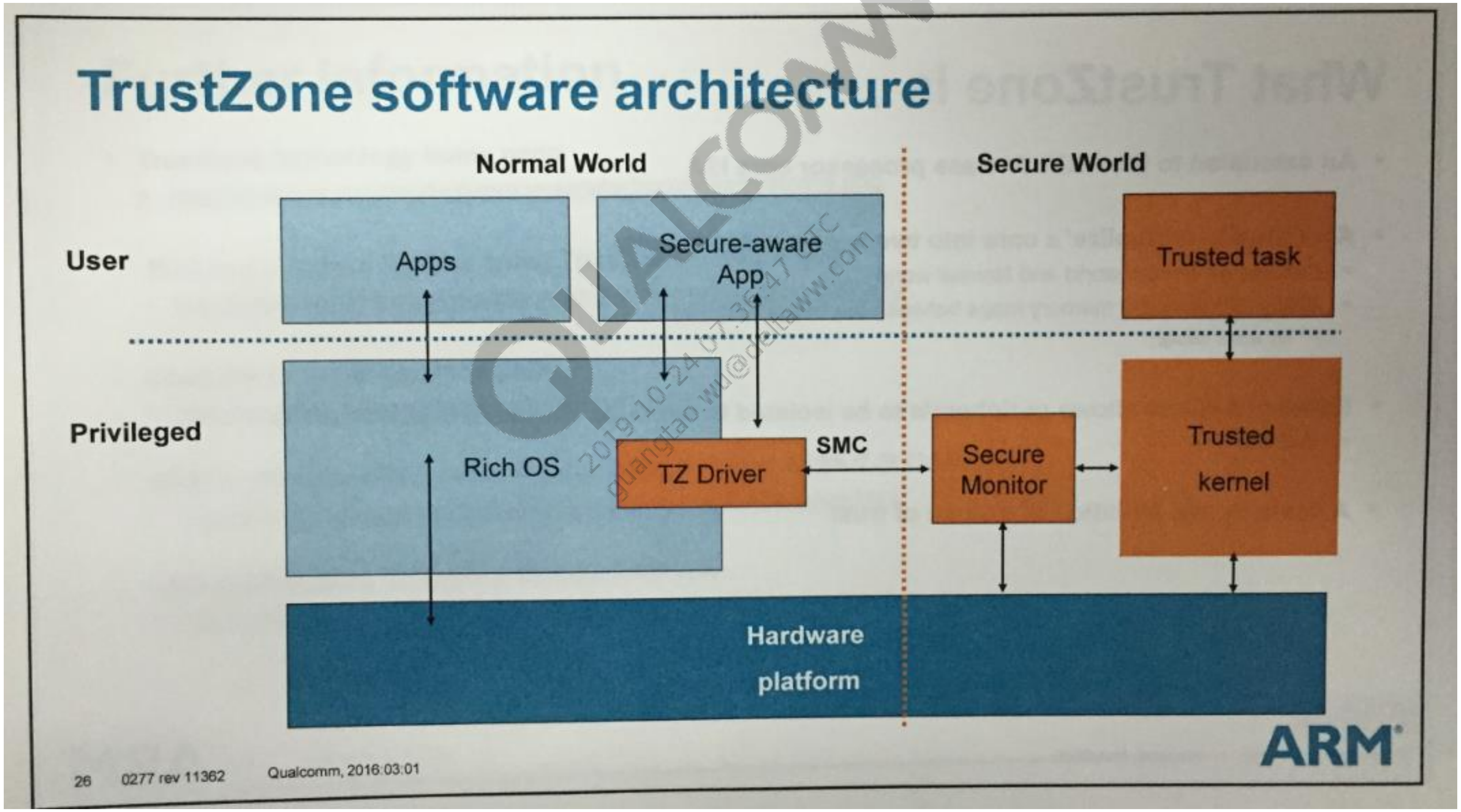
CNT Watchdog clock

- The GCNT watchdog counter increments at a constant 32,000 Hz rate, and this clock is derived from the 48MHz crystal reference.

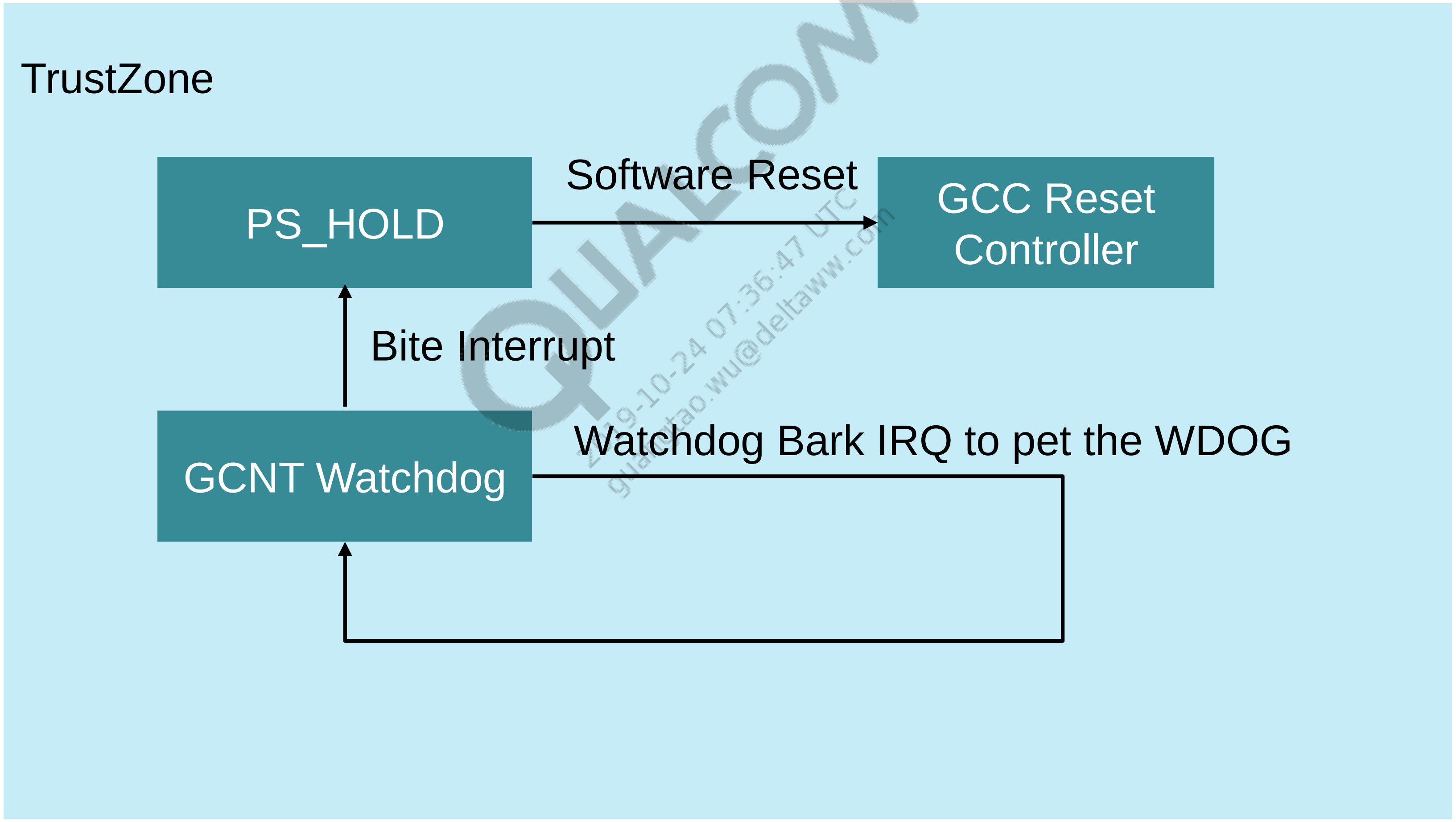
GCNT Watchdog Flow

- At power up, the GCNT watchdog counter is disabled. Trustzone needs to enable the watchdog counter to let it run. When the watchdog counter reaches the bark value, a bark signal is asserted. Normally the bark signal is an interrupt to SW which will in turn write the reset register of the watchdog timer to reset the counter. This is the so-called pet operation. If Trustzone is stuck, the watchdog counter will continue to run after the bark event and eventually reach the bite value. The bite signal is also asserted. The bite signal output by the GCNT watchdog causes the SoC internal reset signal to be asserted, causing a complete reset of the chip.

TZ SW Arch


TrustZone software architecture
Normal World
Secure World
User
Privileged
Apps
Secure-aware App
Trusted task
Rich OS
TZ Driver
SMC
Secure Monitor
Trusted kernel
Hardware platform
ARM
26
0277 rev 11362
Qualcomm, 2016.03.01

GCNT Watchdog In Tz



Reset flow In IPQ40xx

Related Registers for reset flow for IPQ40xx

- SDI represents the below:
- TCSR_RESET_DEBUG_SW_ENTRY: 0x01940000
- Reset: 0x00000000

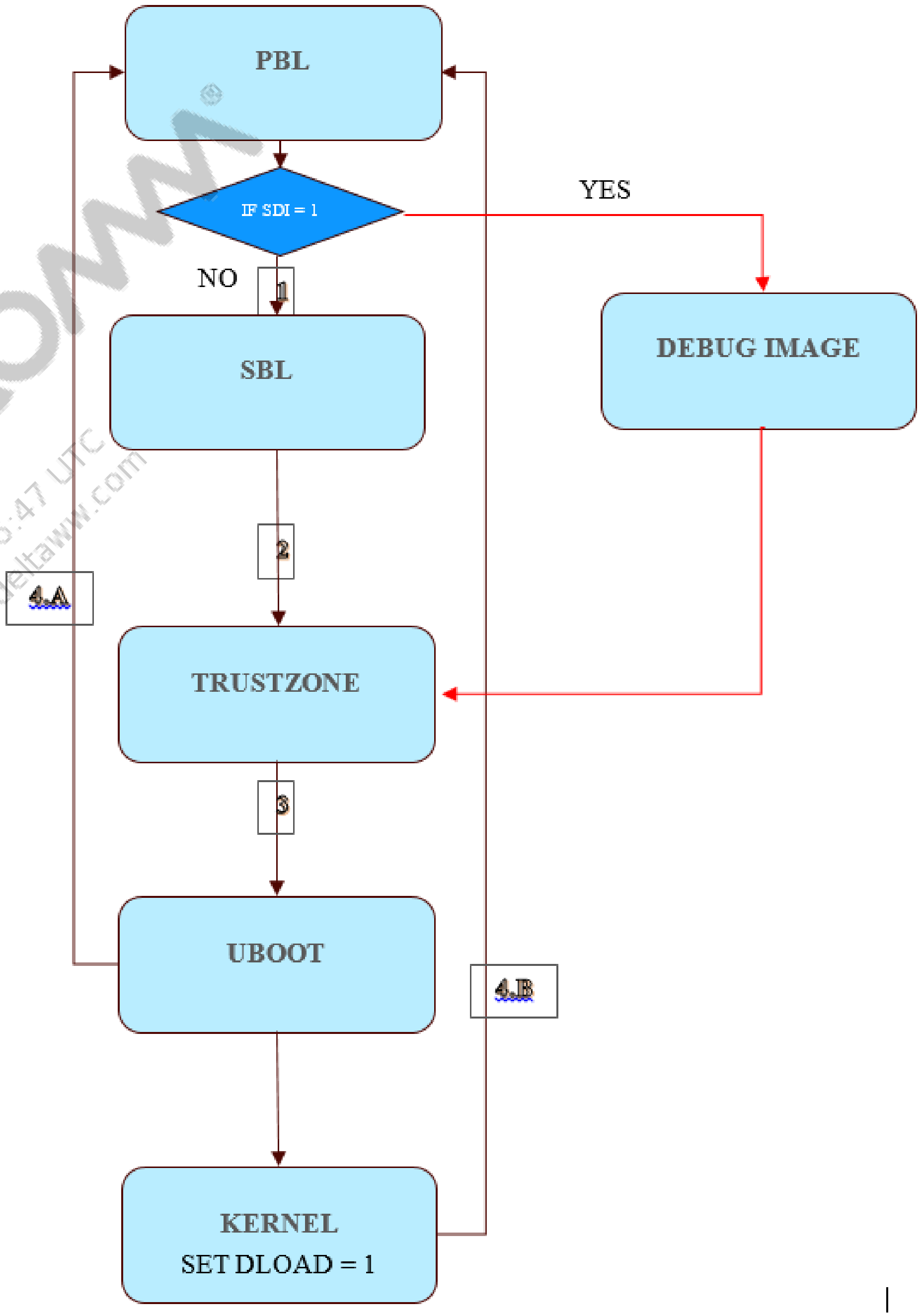
Bits	Name	Description
0	RESET_DEBUG_SW_ENTRY	This register is used to check if the PBL jumps to debug image or SBL. 0 = Disable SDI(Secure Debug Image) 1 = Enable SDI

- DLOAD represents the below:
- TCSR_BOOT_MISC_DETECT: 0x0193D100
- Reset: 0x00000000

Bits	Name	Description
4:0	DLOAD_DETECT	This register is used by PBL to check cookie and enter DLOAD mode. This register is reset only at chip POR (Power Reset) and retains value across all the other resets. 0x0 = Disable Crashdump 0x10 = Enable Crashdump

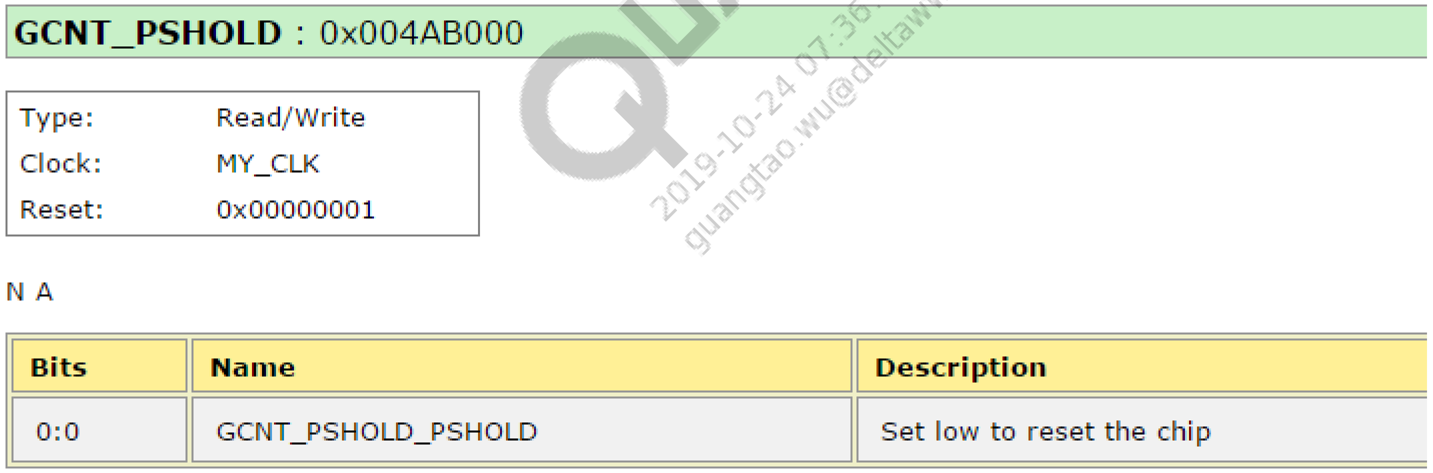
NORMAL BOOT

- The system start executes PBL and then passes to SBL
- SBL is executed and passes to TZ
- TZ sets SDI to 0x1 and DLOAD to 0x1 and passes on to uboot
  - In UBOOT, upon “reset” command, clears DLOAD and SDI and reboots.
  - In Kernel, upon “reboot” command, clears DLOAD and SDI and reboots.
- Note:
  - Debug image brings the DDR out of se refresh. Debug image with the help of makes sure the GPR (General Purpos Registers) and cache contents are sav properly



NORMAL BOOT

- Normal Reset in IPQ40xx is to set GCNT_PSHOLD including
  - System reboot in Kernel.
  - System reset in Uboot.



GCNT_PSHOLD : 0x004AB000

Type:	Read/Write
Clock:	MY_CLK
Reset:	0x00000001

N A

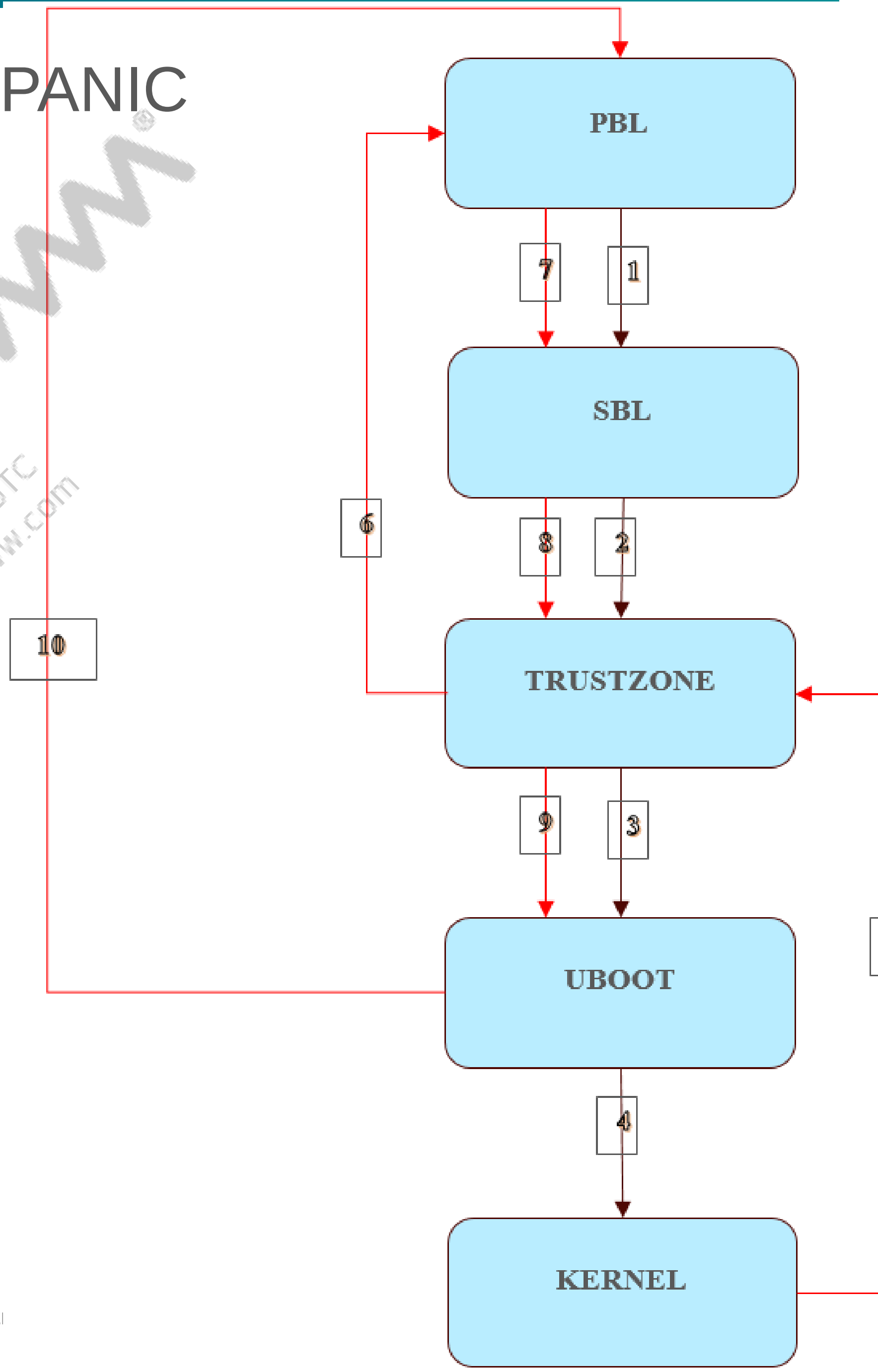
Bits	Name	Description
0:0	GCNT_PSHOLD_PSHOLD	Set low to reset the chip

APCS_WDOG EXPIRY / KERNEL PANIC

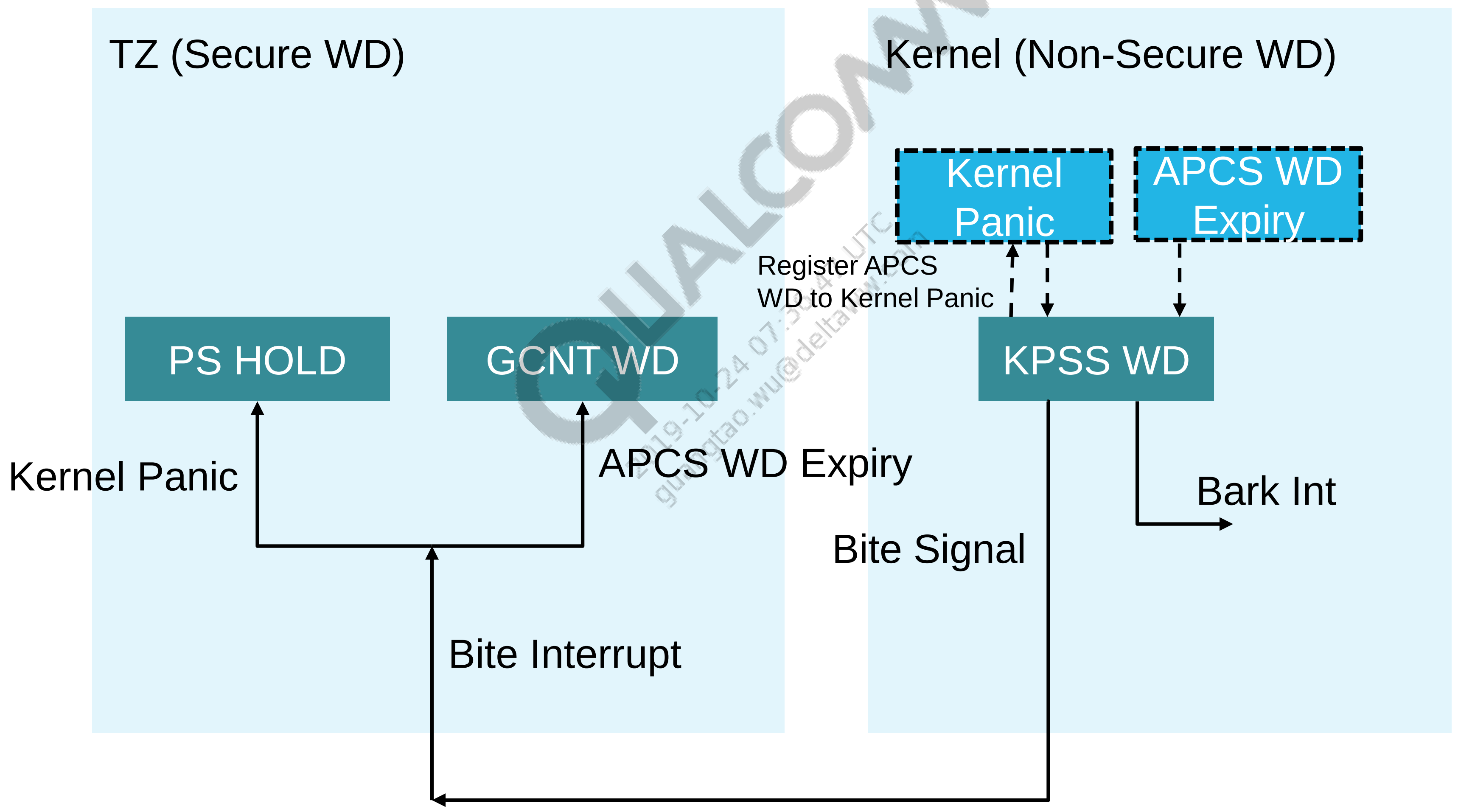
CONTROL FLOW:

1. The system start executes PBL and then passes to SBL
2. SBL is executed and passes to TZ
3. TZ sets SDI to 0x1 and DLOAD to 0x1 and passes on to uboot
4. UBOOT executes and passes on to kernel. Kernel sets SDI to 0x1
5.
    - Upon APCS_WDOG expiry, HW sends FIQ to TZ
    - Upon Kernel Panic, the kernel sets the SDI to 0x0 and causes APCS_WDOG bite. This causes the HW to send FIQ to TZ
6. TZ collects context dump and reboot using
    - GCNT_WDOG for APCS_WDOG expiry
    - PS_HOLD for kernel panic

   TZ then clears the SDI
7. PBL is executed and passes on to SBL
8. SBL is executed and passes on to TZ
9. TZ is executed and passes on to UBOOT
10. UBOOT collects the crashdump, clears the DLOAD and reboots.



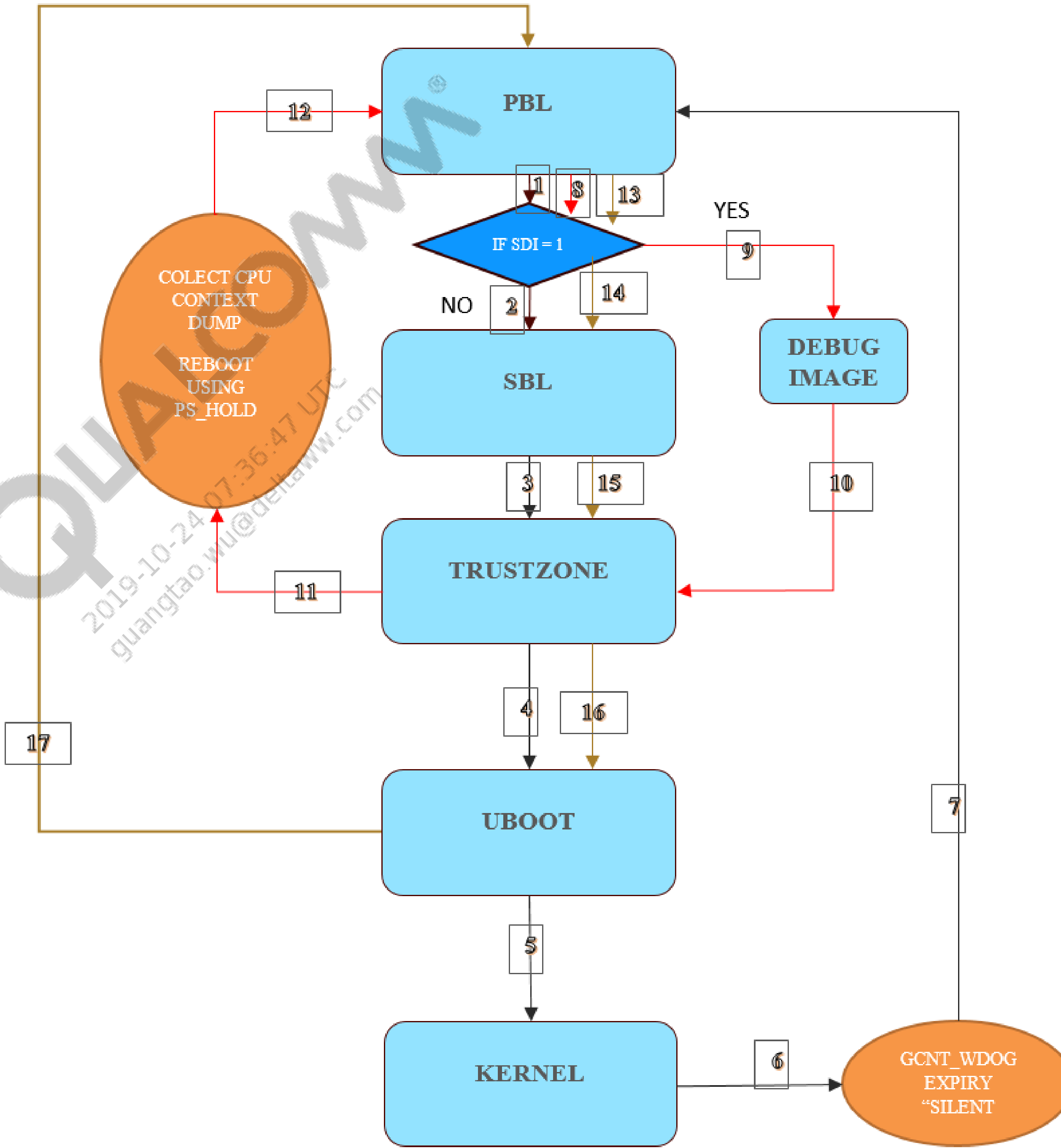
WDOG flow for APCS WDOG Expiry and Kernel Panic



SILENT REBOOT

CONTROL FLOW:

1. The system start executes PBL
2. Checks for the SDI value and passes to SBL
3. SBL is executed and passes to TZ
4. TZ sets SDI to 0x1 and DLOAD to 0x1 and passes on to uboot
5. UBOOT executes and passes on to kernel. Kernel sets SDI to 0x1
6. HW WDOG expiry
7. The system gets rebooted
8. PBL is executed and checks if SDI is 0x1
9. Since SDI is 0x1, the control is passes to the debug image
10. Debug image is executed and the control is passed to TZ
11. TZ collects CPU context dump
12. Clears SDI and reboots using PS_HOLD
13. PBL is executed
14. The control passes to SBL since SDI is 0x0
15. SBL is executed and passes on to TZ
16. TZ is executed and passes on to UBOOT
17. UBOOT collects the crashdump, clears the DLOAD and reboots



CPU Context dump info

- UBOOT collects the crashdump , and tftp “EBICS0.BIN” to remote PC.
- EBICS0.BIN is the whole DDR memory content when the Watchdog bite event is triggered. In DK0.1, the EBICS0.bin’s size is 256M bytes.
- The whole DDR memory content (EBICS0.BIN) is stored in the uboot and the starting address is from 0x80000000.
- the CPU context is stored from 0x87b00000 (DLOAD_MODE_DUMP_BASE), and we can use the CPU context for last reboot to find clue what to lead to the crash.

CPU context for watchdog bites.

```
/* Dump data type */
typedef struct
{
   uint32 version;
   uint32 magic;
   char name[DUMP_DATA_TYPE_NAME_SZ];
   uint64 start_addr;
   uint64 len;
}dump_data_type

typedef struct
{
   /* Status fields */
   uint32 status[4];
   /* Context for all CPUs */
   sysdbg_cpu_ctxt_regs_type cpu_regs;
   /* Secure Context - Not used  */
   sysdbg_cpu_ctxt_regs_type __reserved3;
}sysdbgCPUCtxtType;
```

```
(IPQ40xx) # md 0x87b00000 1024
87b00000: 00000003 42445953 00005a54 00000000
87b00010: 00000000 00000000 00000000 00000000
87b00020: 00000000 00000000 87b000e0 00000000
87b00030: 00000000 00000000 00000003 42445953
87b00040: 00005a54 00000000 00000000 00000000
87b00050: 00000000 00000000 00000000 00000000
87b00060: 87b003e0 00000000 00000000 00000000
87b00070: 00000003 42445953 00005a54 00000000
87b00080: 00000000 00000000 00000000 00000000
87b00090: 00000000 00000000 87b006e0 00000000
87b000a0: 00000000 00000000 00000003 42445953
87b000b0: 00005a54 00000000 00000000 00000000
87b000c0: 00000000 00000000 00000000 00000000
87b000d0: 87b009e0 00000000 00000000 00000000
87b000e0: 00000000 00000000 00000000 00000000
87b000f0: ffffffed 00000000 00000000 00000000    .......
87b00100: 0f5b7000 00000000 00000000 00000000
87b00110: c0814008 00000000 c0814000 00000000
```

FAQs

1. How to let IPQ40xx HW watchdog timeout?

- Comment the watchdog reset in function qcom_wdt_ping() in "drivers/watchdog/qcom-wdt.c", and watchdog will timeout.

```
--- a/drivers/watchdog/qcom-wdt.c
+++ b/drivers/watchdog/qcom-wdt.c
@@ -116,7 +130,7 @@ static int qcom_wdt_ping(struct watchdog_device *wdd)
 {
    struct qcom_wdt *wdt = to_qcom_wdt(wdd);

-   writel(1, wdt->wdt_reset);
+   //writel(1, wdt->wdt_reset);
    return 0;
 }
```

- Issue the stop timeout ubus command
  - ubus call system watchdog '{ "stop": true }'

2. How to disable IPQ4019 APCS_KPSS watchdog timer?

“devmem2 0x0B017008 w 0x0” will disable the APCS_KPSS watchdog.

3. How to program the watchdog timeout?

- Issue the following commands to change the timeout to 20 seconds.
  - ubus call system watchdog '{ "timeout": 20 }'

4. How to parse watchdog dump data?

- Please follow the below command to parse dump data.

  -
    ```
    ramparse.py --ram-file C:\Users\LAdmin\Desktop\136530\EBICS0.BIN 0x80000000 0x8fffffff --vmlinux C:\Users\LAdmin\Desktop\136530\openwrt-ipq806x-vmlinux.elf --phys-offset 0x80000000 --force-hardware 4018 --gdb-path C:\SysGCC\arm-elf\bin\arm-elf-gdb.exe --nm-path C:\SysGCC\arm-elf\bin\arm-elf-nm.exe --parse-debug-image --everything --outdir C:\Users\LAdmin\Desktop\136530\rpoutput
    ```

5. What’s the difference between APCS_KPSS_WDT and GCNT_WDOG?

- The APCS_KPSS_WDT timers is used by Linux kernel, however the GCNT_WDOG sub-system is used exclusively by the Trustzone /QSEE code, there is no interactions with the normal kernel codes.

6. How can the GCNT Watchdog be disabled?

- GCNT watchdog is used by TZ and there are no API’s available right now to disable it from Kernel/user space. TZ will configure the xPU to protect GCNT Watchdog.

7. How can I prevent the Watchdog from interfering with my JTAG debug session?

- Clear bit 0 - GCNT_WDOG_WDOG_WDOG_EN in GCNT_CTL_REG: 0x004AA004 register.
  - D.S ZSD:0x4AA004 %LE %Long 0x80000004
- Clear bit 0 - APCS_WDOG_CTL : 0x0B017008 register
  - D.S ZSD:0xB017008 %LE %Long 0x00000000

8. How can I program (on my production line) the QFPROM fuse bit that disables the watchdog disable pin?

- By default, WDOG is disabled in OTP (Fuse Name: WDOG_EN, QFPROM Row: 6, Bit Number in Row: 12)
- OTP fuse can be blown from uboot in production line. Please refer to Section 8.1 Fuse blower tool in IPQ40xx Security Design User Guide 80-Y8950-22 Rev. E

9. What’s exact behavior of the WATCHDOG_EN and boot config pins (GPIO15)?

- IPQ40x9 WDOG can be enabled through OTP (QFPROM Row 6 bit 12) or through GPIO 15 Pull up. By Default, watchdog is disabled in OTP. So OEM can enable it through OTP (fuse this bit to 1) or GPIO15. Either one of them should be enabled for Watchdog to work. In all our reference design, GPIO15 is pulled high. The decision is made by HW based on OTP/Boot Config GPIO and SW cannot override it.

10. The tables below are the boot configuration GPIOs for enabling watchdog on IPQ40x8 and IPQ40x9. Why does IPQ40x8 have a WDOG DIS-able pin & IPQ40x9 have a WDOG EN-able pin?

IPQ40x8: To enable Watchdog, GPIO61 should be pulled-down.

GPIO61	I	Watchdog disable. Only valid in native mode.
		0 Watchdog enabled
		1 Watchdog disabled

IPQ40x9: To enable Watchdog, GPIO15 should be pulled-up.

GPIO15	I	Watchdog enable
		0 Watchdog disable
		1 Watchdog enable

The description is correct. The difference is due to the different type of pads used for GPIO61 and GPIO15. The default internal pull state should enable the watchdog when there is no external pull. GPIO61 has the traditional definition. But the GPIO15 only has pull up capability internally. So, the polarity is swapped.

11. What does ‘only valid in native mode’ mean for IPQ40x8 WDOG DIS-able pin?

- Native mode means the normal function mode of the chip. The other modes are test, analog test and TIC. The description means the boot-strap value of this pin is repurposed in other modes. But in fact, this watchdog enable/disable is not really useful in the 3 test modes.

12. How can disable crash dump when GCNT Watchdog / APCS watchdog expires and system reboots?

- We can disable crash dump when GCNT Watchdog / APCS watchdog expires and system reboots.
  - We reserve memory for crash dump usage (11 MB), which can be used by customers if they decide to disable it in production builds.
  - For disabling crash dump, please follow the procedures as below.
- To disable Crashdump
  - In the <qsdk>/qca/src/uboot-1.0/include/configs/ipq40xx_cdp.h file, remove the definition of the macro CONFIG_QCA_APPSBL_DLOAD to disable crashdump.
- To check if the crashdump is disabled successfully, enter the below command in kernel
  - cat /proc/iomem

Cont.

- By default memory is reserved as below
  - 80000000-86ffffff : System RAM
  - 80208000-807d55a3 : Kernel code
  - 80806000-8090e657 : Kernel data
  - 88000000-8fffffff : System RAM

- After the crashdump is disabled, ensure that memory is reserved as below
  - 80000000-87afffff : System RAM
  - 80208000-807d55a3 : Kernel code
  - 80806000-8090e657 : Kernel data
  - 88000000-8fffffff : System RAM

- Inferred from the above that the memory reserved for crashdump 0x87000000 of size 0xB00000 is made free when the dump is disabled.

Questions?

You may also submit questions to: https://support.cdmatech.com