BP30
Exception Handling Specification

Edition 2006

Published by Neonseven s.r.l.,
Viale Stazione di Prosecco, 15

34010 Sgonico (Trieste) Italy

For questions on technology, delivery and prices please contact the Neonseven Offices in Italy Sgonico and Gorizia

Attention Please!

The information herein is given to describe certain components and shall not be considered as warranted characteristics.

Terms of delivery and rights to technical change reserved.

We hereby disclaim any and all warranties, including but not limited to warranties of non-infringement, regarding circuits, descriptions and charts stated herein.

Warnings

Due to technical requirements components may contain dangerous substances. For information on the types in question please contact Neonseven.

Neonseven technologies may only be used in life-support devices or systems with the express written approval of Neonseven, if a failure of such technologies can reasonably be expected to cause the failure of that life-support device or system, or to affect the safety or effectiveness of that device or system. Life support devices or systems are intended to be implanted in the human body, or to support and/or maintain and sustain and/or protect human life. If they fail, it is reasonable to assume that the health of the user or other persons may be endangered.

Table of Contents

1 Document Mission/Scope

1.1 Mission

This document contains the specification of the trap and exception handling module.

1.2 Scope

This document is addressed to SW developers who need to understand and customize the behaviour in case of a software crash.

2 List of Acronyms

Abbreviation / Term	Explanation / Definition
BP30	Basic Platform 30
ISR	Interrupt Service Routine
MCU	Micro Controller Unit
NMI	Non-Maskable Interrupt

3 Introduction

This document contains a description of the methods supported by the BP30 platforms to detect and report errors, which are trap handling and exception handling. Furthermore it includes several examples showing how these different debug mechanisms can be used in the effort of isolating and solving potential error scenatios.

4 Architecture

Three different mechanisms are supported for identifying and debugging an abnormal software behaviour:

1. Hardware generated traps
2. Software generated traps
3. Software raised exceptions

The idea is that whenever one of these three error detection mechanisms are invoked, valuable debug information is stored in non-volatile memory and hereafter available for error investigations, also after the mobile has been powered off. The logged data can be retrieved from non-volatile memory using the Phone Tool (the Exceptions part), which also allows clearing such data.

4.1 HW traps

Hardware traps are issued by faults or specific system states that occur runtime. When a hardware trap condition has been detected, the MCU branches to the trap vector location for the respective trap condition. The instruction that caused the trap event is either completed or canceled before the trap-handling routine is entered. Hardware traps are not-maskable and always have a higher priority than any other MCU task. If several hardware trap conditions are detected within the same machine cycle the highest-priority trap is serviced, that is the C166S performs the following actions:
• Push PSW, CSP (in segmented mode) and IP onto the system stack;
• Set MCU level in the PSW register to the highest possible priority level, which disables all interrupts;
• Branch to the trap vector location specified by the trap number of the trap condition.
The C166S distinguishes eight different hardware trap functions, divided in two Classes named A and B.

4.1.1 Class A traps

Class A traps are generated by the high-priority system NMI or by special MCU events such as a software break, or a stack overflow or underflow. Each Class A trap has a dedicated interrupt service routine and its own vector location in the vector table, even though they all share the same priority. After finishing the service routine, the instruction flow must restart from the exact point where it was interrupted, even though this is not advised. If more than one Class A trap occurs at a same time, they are prioritized internally. The NMI trap has the highest priority, and the stack underflow trap has the lowest.
Class A traps are:
• External NMIs: occurs whenever a high-to-low transition on the dedicated NMI is detected
• Stack overflow: occurs when SP becomes lower than STKOV
• Stack underflow: occurs when SP becomes higher than STKUN
• Software Break.

4.1.2 Class B traps

Class B traps are generated by unrecoverable hardware failures. The MCU must immediately start a failure service routine, after which the interrupted instruction flow cannot be restored. All Class B traps have the same trap priority and also share the same vector; therefore it is responsibility of the Software (ISR) to prioritize them. When several Class B traps are active at the same time the corresponding flags in the TFR are set and the common trap service routine is entered.
During the execution of a Class A trap service routine, any Class B trap will not be serviced until the Class A trap service routine is exited.
Class B traps are:
• Undefined opcode: the decoded instruction does not contain a valid operation code
• Protection fault: one protected instruction is executed and the protection is broken
• Illegal word operand access: a word operand (read or write) access is attempted to an odd byte address
• Illegal instruction access: a branch is made to an odd byte address
• Illegal external bus access: an external instruction fetch or a data read or a data write is requested but no external bus configuration has been specified.

4.1.3 Trap storage

Whenever a hardware trap occurs, the following debug information is stored in non-volatile memory:

1. The trap type: 0xAAAA for class A, 0xBBBB for class B
2. The TRAP Flag Register (TRF), a bit field register identifying the hardware trap category:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
NMI	STK OF	STK UF	-	-	-	-	-	UND OPC	-	-	-	PRT FLT	ILL OPA	ILL INA	ILL BUS

3. Unique Identification: not used for hardware traps.
4. The system stack, which is PSW, CSP and IP.
5. Date and time of the occurrence of the trap.
6. 20 bytes of user specified data, that is the content of the array *TRAP_log_data*, filled in by the user as he likes in order to help identifying the trap cause (unpredictable for HW traps).

Once the data mentioned above has been logged to non-volatile memory, the code execution is terminated and the mobile is intentionally switched off to prevent uncontrolled behaviour.

4.2 SW traps

Software generated traps are identified by the SW designer on an implementation time basis in parts of the source code where unrecoverable errors could occur, that is continuing code execution is not advisable. Should the unrecoverable source lines be executed, they can directly invoke a software trap by calling the function

TRAP_envoke_sw_trap with a unique identification number as parameter. Such number is used to point out exactly where the software trap was raised. Note that all identification numbers for the entire system must be defined in the trap header file. Additionally it is possible to store 20 bytes of user selectable data using the array *TRAP_log_data*, which could help finding the source to the problem e.g. state, event etc.

In the example below if the HW_REGISTER is never written then a deadlock scenario occurs.

```
while(!HW_REGISTER)
{
  do_something();
}
```

To prevent the deadlock situation a large counter could be used (see below). Now should the counter reach zero (which means that an unrecoverable error occurred) a software trap will be raised and the faulty spot in the code can be pinpointed from the unique identification number.

```
unsigned int counter = 350;

while(!HW_REGISTER && counter)
{
  do_something();
  if(--counter == 0)
    TRAP_envoke_sw_trap(UNIQUE_IDENTIFICATION);
}
```

When a software trap is invoked (*TRAP_envoke_sw_trap* has been called), the TRAP SW module (trap.c) will store the following debug information in non-volatile memory:

1. The TRAP type: 0xCCCC
2. The TRAP Flag Register (TRF), of no importance for software generated TRAPS.
3. Unique Identification: The link between the identification number and the actual source line which the TRAP was invoked from, can be found in trap.h.
4. The system stack (PSW, CSP, IP).
5. Date and time of the occurrence of the trap.
6. 20 bytes of user specified data, that is the variable *TRAP_log_data*. These optional 20 bytes can be used for identifying the source of the trap.

Once the debug information has been logged to non-volatile memory, the code execution is terminated and the mobile is intentionally switched off to prevent uncontrolled behaviour.

4.3 SW exceptions

Software generated exceptions are identified by the SW designer on an implementation time basis, in parts of the source code where unforeseen but recoverable errors could occur: this means that continuing code execution is possible but further investigation is recommended. Should the unforeseen but recoverable source line(s) be executed, it is possible to invoke a software exception from the involved code section.
Invoking a software generated exception is done by calling the function *TRAP_store_exception*: the parameters are a unique identification number and up to 20 bytes of user selectable data that could help finding out the cause of the abnormal behaviour.

In the switch example below the events x and y are handled normally, but if an unexpected event (default) should occur then in order to understand why an exception (and not a software trap) is raised and code execution is allowed to continue.

```
switch(event)
{
  case x:
```

```
            execute_x();
            break;
          case y:
            execute_y();
            break;
          default:
            TRAP_store_exception(UNIQUE_IDENTIFICATION, log_data_size, &log_data);
            break;
    {
```

When a software exception is invoked the TRAP SW module (trap.c) will store the following debug information in non-volatile memory:

1. The TRAP type: 0xDDDD
2. The TRAP Flag Register (TRF): of no importance for software generated exception.
3. Unique Identification: The link between the identification number and the actual source code line that invoked the exception (can be found in trap.h).
4. The system stack, namely PSW, CSP and IP.
5. Date and time of the occurrence of the Exception.
6. 20 bytes of user specified data. The data passed on in the *TRAP_store_exception* will be saved in non-volatile memory. These optional 20 bytes can be used in the debug effort of finding the source of the problem. In other words you are free to choose the data to log.

Once the data mentioned above has been logged to non-volatile memory, code execution continues as if nothing happened: unlike the HW and SW generated traps, a SW generated exception does not switch off the mobile.
Note that the exceptions are first logged in RAM and eventually stored in NV-RAM only when the mobile is turned off or when a HW or SW trap occurs.

5 Silent reset

Silent reset addresses the problem of handling an unrecoverable error (trap) where the only alternative is to switch off the phone, which is not suitable for a production SW version. Silent reset performs a seamless restart of the MS, which is of course hidden to the user only when the unrecoverable error does not occur during user interaction.

1. When a trap of Class A or Class B is encountered and the associated functions in trap.c is called the SWRw bit of the CGURST register is set before SW reset (SRST).
2. Due to the SRST the system restarts and the startup code (ose_root process) checks the SWRr bit in CGURST and sets the global flag NU_silent_reset_flag accordingly. This flag is then checked by the MMI and SIM driver, which can take action to seamlessly restart the phone.
3. In case the SIM initialization (PIN verification etc) was not carried out before encountering the Silent Reset the SIM driver clears the NU_silent_reset_flag and the MS is switched off.
4. Silent reset handling must be implemented in the MMI, but this is out of the scope of this document.

In order to activate the Silent reset just enable #define SILENT_RESET in system-build/makeoption.mk file and recompile the system.

6 Interface

The TRAP SW module interface is entirely synchronous, which means that it is based on function calls only.

Prototype
void TRAP_class_a_handling(void)
Functional description

This function is the interrupt service routine associated to Calss A hardware traps, which stores the following debug information in non-volatile memory: the trap type, the TFR register, a unique identifier, the system stack, the date and time and the user-defined array TRAP_log_data.
Parameters
None
Return value
None

Prototype
void TRAP_class_b_handling(void)
Functional description
This function is the interrupt service routine associated to Class B hardware traps, which stores essential debug information in non-volatile memory (the same as for Class A traps).
Parameters
None.
Return value
None.

Prototype
void TRAP_envoke_sw_trap(unsigned int id_number)
Functional description
This function is used to envoke a hardware trap from Software, by setting the NMI bit in the Trap Flag Register therefore forcing a a NMI.
Parameters
- id_number: the unique identification number that allows pinpointing the source of the exception and that should be declared in trap.h in order to ensure singleness.

Return value
None.

Prototype
trap_exception_store_type *TRAP_get_exception_store(void)
Functional description
This function returns a pointer to the buffer containing the runtime exceptions that were logged. If no exception was logged, the value will be NULL.
Parameters
None.
Return value
The pointer to the logged exception buffer.

Prototype
void TRAP_store_exception(unsigned int id_number, unsigned char log_data_size, void *log_data)
Functional description
This function stores an exception in RAM, which will eventually be transferred to NV-RAM either at power down or when a hardware trap occurs.
Parameters
- id_number: A unique number identifying the exact source of the abnormal behaviour.
- log_data_size: the number of bytes stored in log_data.
- log_data: The actual exception log data to be storred.

Return value
None.

Prototype
unsigned char TRAP_check_for_silent_reset(void)
Functional description

Used to inform the calling party whether the power-on sequence was the result of a software generated reset or not.

Parameters

None.

Return value

TRUE if Silent reset is the power on cause, FALSE otherwise.

7 References

7.1 External

E-GOLDRadio GSM/GPRS Basesband System – PMB 7870 revision 1.04 – Infineon Techologies.

7.2 Internal

None.

8 Document change report

	Change Reference		Record of changes made to previous released version	
Rev	**Date**	**CR**	**Section**	**Comment**
1.0	22/06/2004	N.A.	Document created	
1.1	14/02/2006	N.A.	Document updated to BP30 Platform	

9 Approval

Revision	Approver(s)	Date	Source/signature
1.0	Stefano Godeas	22/06/2004	Document stored on server
1.1	Stefano Godeas	14/02/2006	Document stored on server

10 Annex 1

The following chapters contain some examples of typical usage of the TRAP functionality for debugging purposes, including how to use the logged information.
Provided that a trap occurred, the debugging procedure requires the following steps:

1. Power up the ME in test mode (press <*>, <#> and ON at the same time for 2-3 seconds).
2. Connect the PC to the mobile via the serial interface and start the PhoneTool, selecting the Exceptions sub-menu.
3. Use the <Get> command to retrieve the Exception logs.

The interpretation and usage of the log data slightly differs between hardware and software generated traps.

10.1 Hardware trap procedure

1. The logged data retrieve by the Phone tool is illustrated in Figure 10-1.

Exception log information:

```
SW version: 00.00:05.09:20.00, DLL version: 1.91n

Exception Number: 1
Trap Class: 0xBBBB
Trap Flag Register: 0x0004
Identification: 0xFFFF
System Stack Word 1: 0xBDA8
             Word 2: 0x0028
             Word 3: 0x0840
Date: 18.04.2001
Time: 16:19
Trap Data:
48 57 20 46 6F 72 63 65 64 20 54 52 41 50 00 00   HW Forced TRAP..
00 00 00 00                                       ....

A Total of 1 exception was read
```

Figure 10-1

The essential information is:

- **Trap Flag Register**. This value indicates the Class type that caused the trap. In the example above the value 0x0004 indicates an Illegal Word Operand Access (ILLOPA) caused the TRAP.
- **System Stack**. The content of the system consists in 3 words: the PSW, the CSP and the IP register respectively. In the example above the system stack holds the Program Counter that is pointing to address 0x28BDA8 (CSP 0x28, IP 0x0840), which is the part of code where the hardware generated error occurred.
- **Trap Data**. This value represents the user selectable data that was store in the variable *TRAP_log_data* when the trap occurred. In the example above the text string “HW Forced TRAP” was written to the variable, but it could just as well contain any kind of values. When dealing with hardware generated traps it is not feasible to predict when the trap will occur, therefore *TRAP_log_data* will not contain anything meaningful; but after the first occurrence it is sometime helpful to store in *TRAP_log_data* additional debug information to be used after the next trap.

2. The linker generated memory map file progsm.map can be parsed to find the code segment pointed to by the CSP and IP (address 0x28BDA8). Note that as illustrated in Figure 10-2 the address 0x28BDA8 is in the range 0x289D10 (start) - 0x28C1D5 (end) therefore it belongs to the section labeled PMI_DI19_6_PR.

Extract from the memory map file of the investigated system:

```
Name                          No. Start      End        Length  Type Algn Comb Mem
-------------------------------------------------------------------------------
:
:
PMI_DI16_2_PR.............. 482 285B4Ah    286B2Dh    000FE4h CODE WORD GLOB ROM
PMI_DI18_5_PR.............. 490 286B2Eh    289D0Fh    0031E2h CODE WORD GLOB ROM
PMI_DI19_6_PR.............. 497 289D10h    28C1D5h    0024C6h CODE WORD GLOB ROM
PMI_GRPH_1_FC.............. 499 28C1D6h    28C7C8h    0005F3h DATA WORD GLOB ROM
T9LANG_1_PR................ 502 28C7CAh    28CA11h    000248h CODE WORD GLOB ROM
:
```

Figure 10-2

3. The “inflicted” code segment PMI_DI19_6_PR has now been identified, but it is also possible to determine the actual line of code causing the problem. Knowing that the problem occurred at address 0x28BDA8 and that the start address of the given code segment is 0x289D10, a subtraction gives an offset of 0x2094 to the actual code line. By compiling the source file holding the particular code segment with the option -s (includes c-code in .lst file) and assembling with option listall (includes address offset in .lst file) it is quite straightforward to pinpoint the exact line using the offset in the *.lst file. This is illustrated below (Figure 10-3).

Extract from the *.lst file originating from the “inflicted” source file:

```
LOC CODE         LINE    SOURCELINE
:
:
208E               13954   _411:
208E 0802          13955           ADD     R0,#02h
                 R 13956           ?SYMB   '',$,96,22
                   13957   ; pmi_di19.c 1276
                   13958   ; pmi_di19.c 1277          TFR |= 0x0004;
                   13959           ?LINE   1277
2090 76D60400      13960           OR      TFR,#04h
                 R 13961           ?SYMB   '',$,102,1
                 R 13962           ?SYMB   '',$,102,2
                   13963   ; pmi_di19.c 1278
                   13964   ; pmi_di19.c 1279          return;
                   13965   ; pmi_di19.c 1280        } /* end pmi_force_trap */
                   13966           ?LINE   1280
2094 06F01600      13967           ADD     R0,#016h
                 R 13968           ?SYMB   '',$,96,0
2098 9830          13969           MOV     R3,[R0+]
:
```

Figure 10-3

4. The EGoldRadio documentation part describing the ILLOPA traps specifies: “The IP value pushed onto the system stack is the address of the instruction following the one which caused the trap”. In other words this means that the instruction causing the error was the one prior to 0x2094 and this address (0x2090) holds an instruction where the TRAP Flag Register is assigned the value 0x04 - deliberately forcing a trap (OR TRF, #02h).

10.2 Software trap procedure

1. The logged data retrieve by the Phone tool is illustrated in Figure 10-4.

Exception log information:

```
SW version: 00.00:05.09:20.00, DLL version: 1.91n

Exception Number: 1
Trap Class: 0xCCCC
Trap Flag Register: 0x8000
Identification: 0x0091
System Stack Word 1: 0x1D6C
              Word 2: 0x003A
              Word 3: 0x0841
Date: 18.04.2001
Time: 16:10
Trap Data:
53 57 20 46 6F 72 63 65 64 20 54 52 41 50 00 00   SW Forced TRAP..
00 00 00 00                                       ....

A Total of 1 exception was read
```

Figure 10-4

The essential information is:

- **System Stack**. The content of the system consists in 3 words: the PSW, the CSP and the IP register respectively. In the example above the system stack holds the Program Counter that is pointing to address 0x28BDA8 (CSP 0x28, IP 0x0840), which is the part of code where the hardware generated error occurred.
- **Trap Data**. The displayed data represents the user selectable data that was store in the variable *TRAP_log_data* when the TRAP or Exception was invoked. In the example above the text string “SW Forced TRAP” was written to the variable, but it could just as well had been variable values etc.
- **Identification**. The identification number identifies the exact place in the source code from where the trap/exception was raised. In the example above the identification number is 0x0091 and the acronym can be identified in the trap header file to be TRAP_HMI_FORCED_TRAP as illustrated in Figure 10-5.

Unique Identification from trap.h:

```
/*****************************************************************/
/*************************               *************************/
/***********************                   ***********************/
/*********************  IDENTIFICATION TAG  *********************/
/******************** SW Traps and Exceptions  ********************/
/*********************  ~~~~~~~~~~~~~~~~~~  *********************/
/***********************                   ***********************/
/*************************               *************************/
/*****************************************************************/
:
:
#define TRAP_NU_EXCEPTION                          142   /* 0x8E */
#define TRAP_IO_UART_ERROR                         143   /* 0x8F */
#define TRAP_NU_DYN_MEM_CHECK_FAIL                 144   /* 0x90 */
#define TRAP_HMI_FORCED_TRAP                       145   /* 0x91 */
#define TRAP_L1D_TEMP_SYNC_TO_LATE                 146   /* 0x92 */
#define TRAP_L1D_FIX_SYNC_TO_LATE                  147   /* 0x93 */
#define TRAP_L1D_TEMP_SYNC_BACK_TO_LATE            148   /* 0x94 */
```

Figure 10-5

The Trap Flag Register and the system stack are not used with respect to SW generated trap or exceptions.

2. The linker generated memory map file progsm.map can be parsed to find the code segment pointed to by the CSP and IP (address 0x28BDA8). Note that as illustrated in Figure 10-2 the address 0x28BDA8 is in the range of 0x289D10 (start address) - 0x28C1D5 (end address) therefore it belongs to the section labeled PMI_DI19_6_PR.

Extract from the memory map file of the investigated system:

```
Name                          No. Start      End        Length   Type Algn Comb Mem
-------------------------------------------------------------------------------------
:
:
PMI_DI16_2_PR.............. 482 285B4Ah    286B2Dh    000FE4h CODE WORD GLOB ROM
PMI_DI18_5_PR.............. 490 286B2Eh    289D0Fh    0031E2h CODE WORD GLOB ROM
PMI_DI19_6_PR.............. 497 289D10h    28C1D5h    0024C6h CODE WORD GLOB ROM
PMI_GRPH_1_FC.............. 499 28C1D6h    28C7C8h    0005F3h DATA WORD GLOB ROM
T9LANG_1_PR................ 502 28C7CAh    28CA11h    000248h CODE WORD GLOB ROM
:
```

Figure 10-6

3. Now it's possible to search through the source code for TRAP_HMI_FORCED_TRAP that will identify the exact place where the trap or exception was raised. This is illustrated below.

Source code where the SW generated TRAP occurred:

```
unsigned char trap_text[] = "SW Forced TRAP";
t_pmi_unicode trap_unicode;

trap_unicode.text      =  (char *)trap_text;
trap_unicode.text_type =  DEFAULT_ALPHABET;

/*------*/
/* Code */
/*------*/
pmi_state = PMI_VERSION;

pmi_clear_display();
pmi_display_text_line (2, LCD_CENTERED_ALIGNED, trap_unicode);
memcpy(&TRAP_log_data[0], &trap_text, 14);
TRAP_envoke_sw_trap(TRAP_HMI_FORCED_TRAP);
```

Figure 10-7

4. From the code section above it is also illustrated how the variable *TRAP_log_data* can be used to store valuable debug information.

11 Annex 2

None.

12 Annex 3

None.