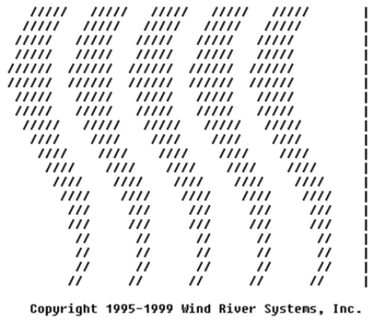
UNIVERSITY OF OSLO

FYS 4220/9220 – 2011 / #5

Real Time and Embedded Data Systems and Computing

Exceptions and Signals


```
T O R N A D O

Development System

Host Based Shell

Version 2.0.2

Copyright 1995-1999 Wind River Systems, Inc.
```

T.B. Skaali, Department of Physics, University of Oslo

Exceptions - definition

- An exception is a generic term for an event that occurs during the execution of a program that disrupts the normal flow of program execution.
- Exceptions can occur at many levels:
  - Hardware/operating system/ language level.
    - Arithmetic exceptions; divide by 0, under/overflow, etc
    - Memory access violations; segment fault, stack over/underflow.
    - Type conversion; illegal values, improper casts.
    - Bounds violations; illegal array indices.
    - Bad references; null pointers.
  - Processor architecture level
    - **Interrupts**
  - Program level.
    - User or system defined exceptions – **signals**
    - **Error return from system calls**
- Exception handling is a ”can of worms” (in other words: tricky stuff)

Classes of Exceptions

- Detected by the environment and raised <u>synchronously</u>; e.g. array bounds error or divide by zero
- Detected by the application and raised <u>synchronously</u>, e.g. the failure of a program-defined assertion check
- Detected by the environment and raised <u>asynchronously</u>; e.g. an exception raised due to the failure of some monitoring mechanism
- Detected by the application and raised <u>asynchronously</u>; e.g. one process may recognise that an error condition has occurred which will result in another process not meeting its deadline or not terminating correctly

Exception handling: general requirements *(ref. B&W)*

- There are a number of general requirements for an exception handling facility:
  - R1: The facility must be simple to understand and use
  - R2: The code for exception handling should not obscure understanding of the program's normal error-free operation
  - R3: The mechanism should be designed so that run-time overheads are incurred only when handling an exception
  - R4: The mechanism should allow the uniform treatment of exceptions detected both by the environment and by the program
  - R5: the exception mechanism should allow recovery actions to be programmed
- Great, easy to write down but not that easily implemented! Some examples follow.

Assembly language: Forced Branch

- Used mainly in assembly languages. Example: programming a microcontroller
  - the typical mechanism is for subroutines to skip return
  - the instruction following the subroutine call is skipped to indicate the presence/absence of an error
  - achieved by incrementing its return address (program counter) by the length of a simple jump instruction
  - where more than one exceptional return is possible, the PC can be manipulated accordingly

```
jsr pc, PRINT_CHAR
jmp IO_ERROR
jmp DEVICE_NOT_ENABLED
# normal processing
```

- Approach incurs little overhead(R3) and enables recovery actions to be programmed(R5). It can lead to obscure program structures and, therefore, violates requirements R1 and R2. R4 also cannot be satisfied

The simple minded approach *(ref. B&W)*

- Unusual return value or error return from a procedure or a function.
- C supports this approach, cf. VxWorks

```
if(function_call(parameters) == ERROR) {
 -- error handling code
} else {
  -- normal return code
};
```

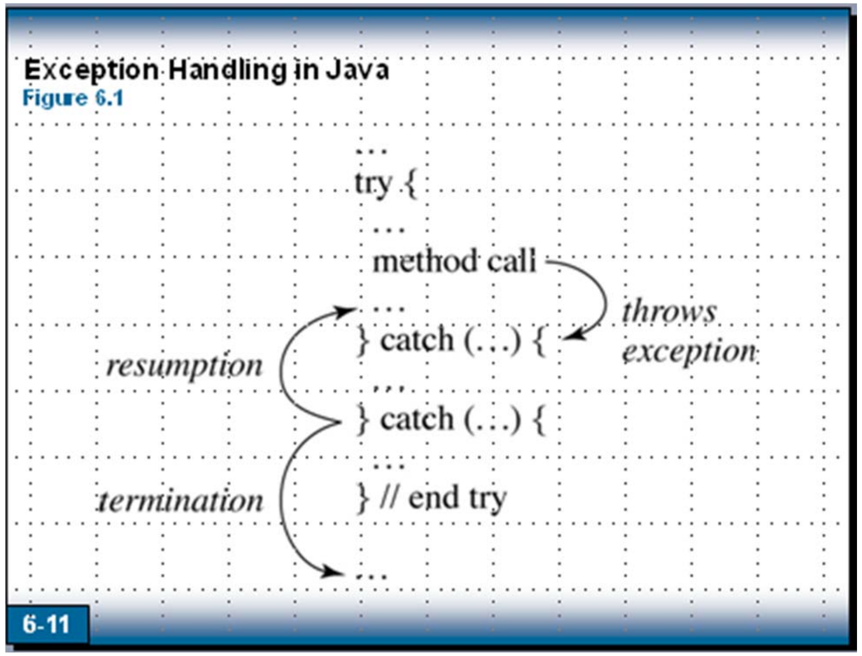
- Meets the simplicity requirement R1 and allows recovery actions to be programmed (R5)
- Fails to satisfy R2, R3 and R4; the code becomes spaghetti (my characterization), it entails overheads every time it is used, and it is not clear how to handle errors detected by the environment

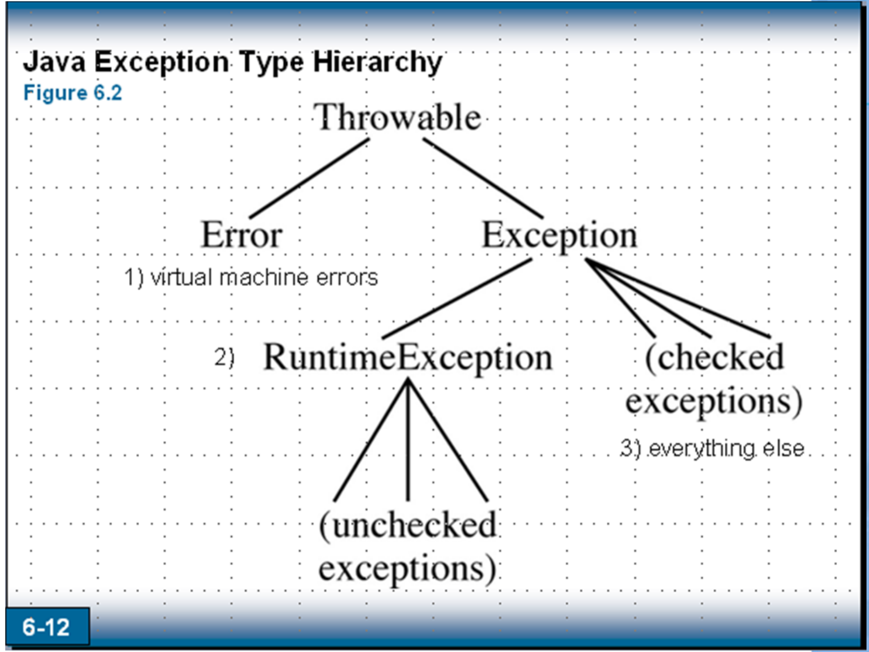
UNIVERSITY
OF OSLO

Language support of exceptions

- Older languages like FORTRAN, C: **none**
- Languages like Ada, C++, Java have added language constructs to facilitate exception handling
  - **try** block
  - **throw** an exception
  - **catch** an exception

Figures from McGraw-Hill Companies, Inc, web





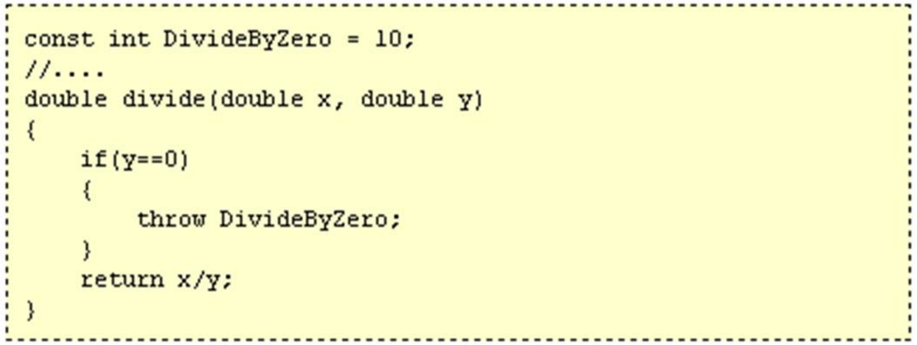
Exception handling in C++

- The designers of C++, Bell labs, extended it with exception handling structures. The commands being used relate closely to the terms used in exception handling. The block of code you want to try starts with specifying the ***try*** command and surrounding the block with curly braces. Within this block, you can throw any occurring errors with the ***throw*** command. You must specify an error and this should be a class but let us ignore that now. Immediately after the ***try***-block is closed, the ***catch***-block starts. Here the error handling code is placed. The piece of pseudo code on next page will show the idea
- We can chain multiple handlers (catch expressions), each one with a different parameter type. Only the handler that matches its type with the argument specified in the throw statement is executed. If we use an ellipsis (...) as the parameter of catch, that handler will catch any exception no matter what the type of the throw exception is. This can be used as a default handler that catches all exceptions not caught by other handlers if it is specified at last. In this case the last handler would catch any exception thrown with any parameter that is neither an int nor a char.

```
 try {
     // code here
    }
catch (int param) { cout << "int exception"; }
catch (char param) { cout << "char exception"; }
catch (...) { cout << "default exception"; }
```

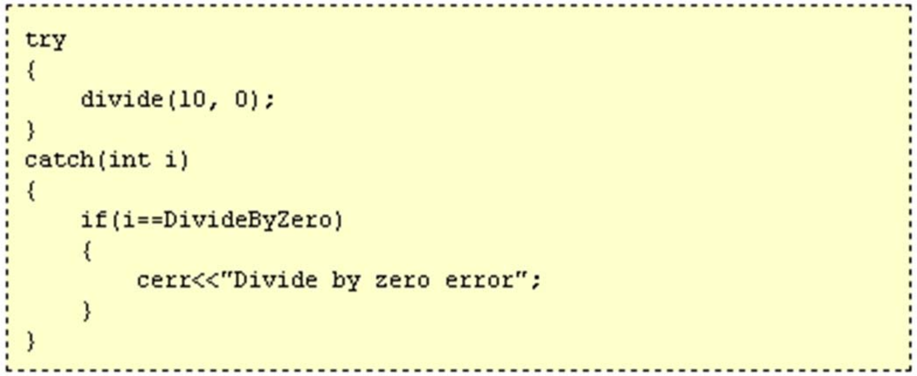
UNIVERSITY OF OSLO

C++ exception *throw* and *catch*



```
const int DivideByZero = 10;
//....
double divide(double x, double y)
{
    if(y==0)
    {
        throw DivideByZero;
    }
    return x/y;
}
```

The function will throw DivideByZero as an exception that can then be caught by an exception-handling catch statement that catches exceptions of type int. The necessary construction for catching exceptions is a try catch system. If you wish to have your program check for exceptions, you must enclose the code that may have exceptions thrown in a try block. For example:



```
try
{
    divide(10, 0);
}
catch(int i)
{
    if(i==DivideByZero)
    {
        cerr<<"Divide by zero error";
    }
}
```

The C++ exception demo implemented under vxWorks

```
//  C++ try/throw/catch on VxWorks
#include <vxWorks.h>
#include <iostream.h>
#include <string>

const int DivideByZero = 10;
double divide (double x, double y)
{
  if ( y == 0)
  {
    throw DivideByZero;
  }
  return x/y;
}

void throw_shit()
{
  try
  {
    divide (10,0);
  }

  catch (int i)
  {
    if (i==DivideByZero)
    {
      cerr <<"Divide by zero error" <<endl;
    }
  }
  cout <<"Shit happens" <<endl;
}
```

Output on console:

```
-> throw_shit
Shit happens
value = 4664368 = 0x472c30 = _cout
```

Output on error device:

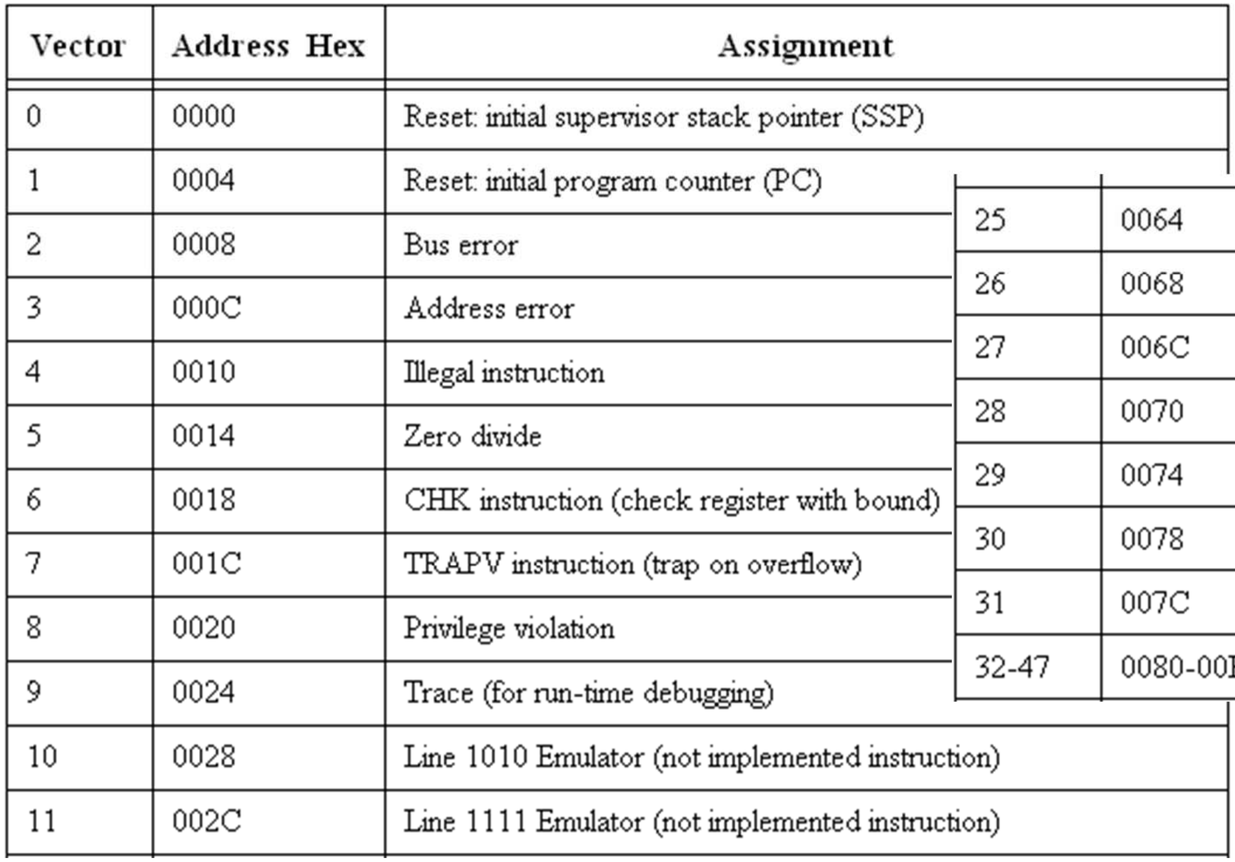
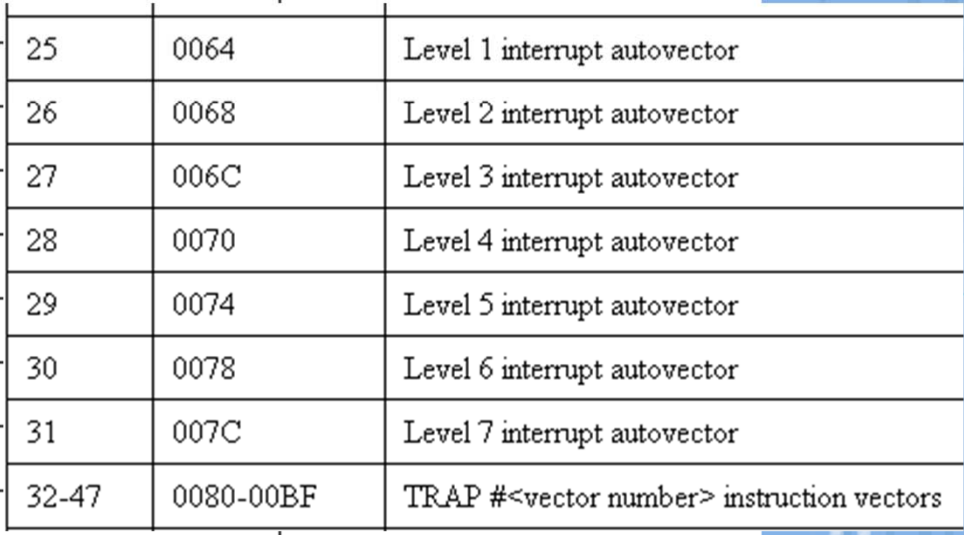
```
Divide by zero error
```

Fine, but what to do after the exception has been catched??

Interrupts

- Interrupts are (in general) hardware signals from external devices which request services, typical for I/O operations.
- When an interrupt occurs the processor will switch to an **interrupt level** which will freeze the execution of user processes. (Normally all user processes are executed on the same hardware level). The number of interrupt levels is architecture dependent.
- Since many devices may be connected to a processor and also to the same interrupt level, a mechanism is required to identify the source of the interrupt
- Interrupts / exceptions are typically hardware bound to a vector value which is used as a lookup key in a exception table. The content of the table are address pointers to the corresponding interrupt service / exception routine.
- A very important number is the **interrupt latency**: the time from the occurence of the interrupt signal until the interrupt service routine is entered.

A simple case study: the MC68K exception vectors

Vector	Address Hex	Assignment
0	0000	Reset: initial supervisor stack pointer (SSP)
1	0004	Reset: initial program counter (PC)
2	0008	Bus error
3	000C	Address error
4	0010	Illegal instruction
5	0014	Zero divide
6	0018	CHK instruction (check register with bound)
7	001C	TRAPV instruction (trap on overflow)
8	0020	Privilege violation
9	0024	Trace (for run-time debugging)
10	0028	Line 1010 Emulator (not implemented instruction)
11	002C	Line 1111 Emulator (not implemented instruction)
25	0064	Level 1 interrupt autovector
26	0068	Level 2 interrupt autovector
27	006C	Level 3 interrupt autovector
28	0070	Level 4 interrupt autovector
29	0074	Level 5 interrupt autovector
30	0078	Level 6 interrupt autovector
31	007C	Level 7 interrupt autovector
32-47	0080-00BF	TRAP #<vector number> instruction vectors

PowerPC440 interrupts and exceptions

- An interrupt is the action in which the processor saves its old context (Machine State Register (MSR) and next instruction address) and begins execution at a pre-determined interrupt-handler address, with a modified MSR.
- Exceptions are the events that may cause the processor to take an interrupt, if the corresponding interrupt type is enabled.
  - Exceptions may be generated by the execution of instructions, or by signals from devices external to the PPC440, the internal timer facilities, debug events, or error conditions.
- The interrupt/exception handling on the PPC440 is much more advanced than on the old MC68K architecture!
  - PPC440 Interrupt classes:
    - Asynchronous or Synchronous
    - Critical or Non-Critical
    - Machine Check interrupts are a special case. They are typical caused by hardware or storage subsystem failure, or by an attempt to access an invalid address.

PPC440 interrupt classes

- Asynchronous:
  - are caused by events that are independent of instruction execution
- Synchronous:
  - are those that are caused directly the execution (or attempted execution) of instructions. They are characterized as Precise or Imprecise interrupts
- Critical and Non-Critical:
  - Certain interrupt types demand immediate attention, even if other interrupt types are currently being processed and have not yet had the opportunity to save the state of the machine (that is, return address and captured state of the MSR). To enable taking a critical interrupt immediately after a non-critical interrupt has occurred (that is, before the state of the machine has been saved), two sets of Save/Restore Register pairs are provided. Critical interrupts use the Save/Restore Register pair CSRR0/CSRR1. Non-Critical interrupts use Save/Restore Register pair.
- PPC440 interrupt architecture:
  - Associated with each kind of interrupt is an *interrupt vector, that is, the address of the initial instruction that is* executed when the corresponding interrupt occurs. Interrupt processing consists of saving a small part of the processor state in certain registers, identifying the cause of the interrupt in another register, and continuing execution at the corresponding interrupt vector location.

UNIVERSITY OF OSLO

Interrupt processing

- An interrupt will be processed either by
  - i) a system driver
  - ii) a user written driver installed in the Operating system
  - iii) For VxWorks: a user C routine connected to the interrupt vector by means of the library call ***intConnect*( )**
- After the interrupt has been processed the normal execution of the interrupted user process is continued
  - Since processing on an processor interrupt level will freeze all activities on the application level the time spent on an interrupt level should be as short as possible

VxWorks ISRs (VxWorks Programmer's Guide)

2.5 Interrupt Service Code

- Hardware interrupt handling is of key significance in real-time systems, because it is usually through interrupts that the system is informed of external events. For the fastest possible response to interrupts, interrupt service routines (ISRs) in VxWorks run in a special context outside of any task's context. Thus, interrupt handling involves no task context switch. The interrupt routines, listed in Table 2-22, are provided in **intLib** and **intArchLib,** latter defines architecture.
- VxWorks includes default interrupt drivers for I/O, network, etc
- **Table 2-22: Interrupt Routines Call Description**

***intConnect*( )**	Connect a C routine to an interrupt vector
***intContext*( )**	Return TRUE if called from interrupt level
***intCount*( )**	Get the current interrupt nesting depth
***intLevelSet*( )**	Set the processor interrupt mask level
***intLock*( )**	Disable interrupts
***intUnlock*( )**	Re-enable interrupts
***intVecBaseSet*( )**	Set the vector base address
***intVecBaseGet*( )**	Get the vector base address
***intVecSet*( )**	Set an exception vector
***intVecGet*( )**	Get an exception vector

Signals

- A **signal** is a limited form of inter-process communication used in Unix, Unix-like, and other POSIX-compliant operating systems. Essentially it is an asynchronous notification sent to a process in order to notify it of an event that occurred. When a signal is sent to a process, the operating system interrupts the process's normal flow of execution. <u>Execution can be interrupted during any non-atomic instruction.</u> If the process has previously registered a **signal handler**, that routine is executed. Otherwise the default signal handler is executed.
  - Typing certain key combinations at the controlling terminal of a running process causes the system to send it certain signals

VxWorks signals

- VxWorks supports a software signal facility.
  - Signals will asynchronously alter the control flow of a task.
  - Any task or Interrupt Service Routine (ISR) can raise a signal for a particular task. The task being signaled immediately suspends its current thread of execution and executes the task-specified signal handler routine the next time it is scheduled to run.
  - The signal handler executes in the receiving task's context and makes use of that task's stack. The signal handler is invoked even if the task is blocked, for instance when waiting for a sempahore or a message queue transfer. Furthermore, an immediate return from a semaphore/message queue routine will take place with an error code. **Therefore, always test the return value!**
  - The *wind* kernel supports two types of signal interface: UNIX BSD-style signals and POSIX-compatible signals.
    - The POSIX-compatible signal interface, in turn, includes both the fundamental signaling interface specified in the POSIX standard 1003.1, and the queued-signals extension from POSIX 1003.1b.
  - In general, signal handlers should be treated like ISRs; no routine should be called from a signal handler that might cause the handler to block.

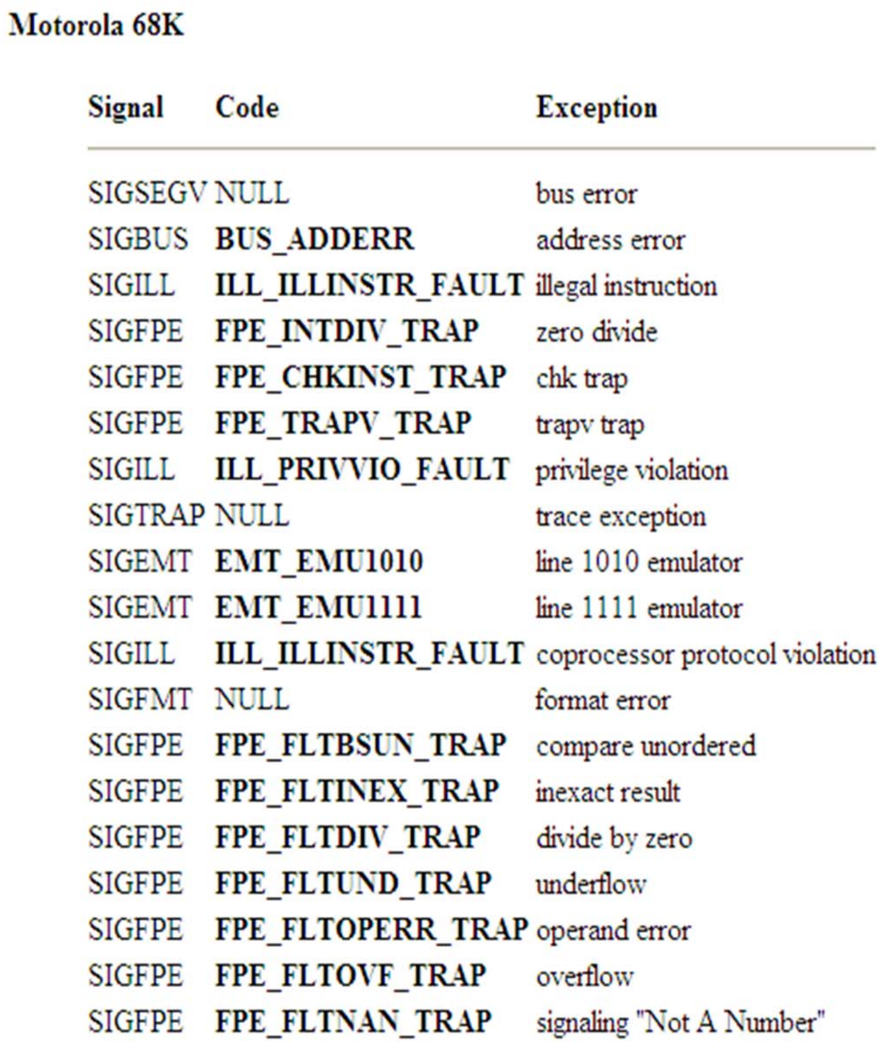
Classes of (VxWorks) signals

- Signals as defined in *signal.h* are listed on the next page. Basically there are two classes:
- i) those orginating from exceptions catched by the processor hardware, like bus errors, and
- ii) those asserted by a process (task), like realtime signals
- Whatever the origin is, the signal will trigger the signal handler provided that the signal has been linked to the handler.
  - A missing signal handler will cause the process (task) to be aborted

VxWorks "*signal.h*"

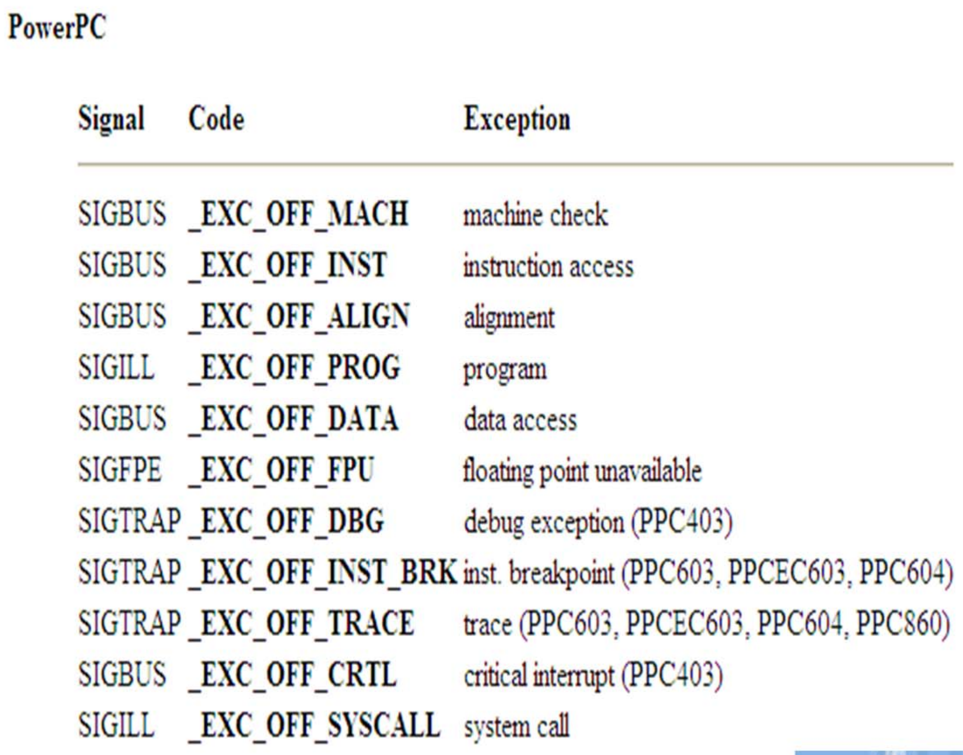
```
/* Signal Numbers:
 *              Required .1 signals         1-13
 *              Job Control signals        14-19 (not implemented but must be defined)
 *              Realtime signals           20-27
 */
#define         SIGHUP      1              /* hangup */
#define         SIGINT      2              /* interrupt */
#define         SIGQUIT     3              /* quit */
#define         SIGILL      4              /* illegal instruction (not reset when caught) */
#define         SIGTRAP     5              /* trace trap (not reset when caught) */
#define         SIGABRT     6              /* used by abort, replace SIGIOT in the future */
#define         SIGEMT      7              /* EMT instruction */
#define         SIGFPE      8              /* floating point exception */
#define         SIGKILL     9              /* kill (cannot be caught or ignored) */
#define         SIGBUS      10             /* bus error */
#define         SIGSEGV     11             /* segmentation violation */
#define         SIGFMT      12             /* STACK FORMAT ERROR (not posix) */
#define         SIGPIPE     13             /* write on a pipe with no one to read it */
#define         SIGALRM     14             /* alarm clock */
#define         SIGTERM     15             /* software termination signal from kill ()*/

#define         SIGSTOP     17             /* sendable stop signal not from tty */
#define         SIGTSTP     18             /* stop signal from tty */
#define         SIGCONT     19             /* continue a stopped process */
#define         SIGCHLD     20             /* to parent on child stop or exit */
#define         SIGTTIN     21             /* to readers pgrp upon background tty read */
#define         SIGTTOU     22             /* like TTIN for output if (tp->t_local&LTOSTOP) */

#define         SIGUSR1     30             /* user defined signal 1 */
#define         SIGUSR2     31             /* user defined signal 2 */

#define         SIGRTMIN    23             /* Realtime signal min */
#define         SIGRTMAX    29             /* Realtime signal max */
```

Hardware exceptions MC68K and PPC



Motorola 68K

Signal	Code	Exception
SIGSEGV	NULL	bus error
SIGBUS	**BUS_ADDERR**	address error
SIGILL	**ILL_ILLINSTR_FAULT**	illegal instruction
SIGFPE	**FPE_INTDIV_TRAP**	zero divide
SIGFPE	**FPE_CHKINST_TRAP**	chk trap
SIGFPE	**FPE_TRAPV_TRAP**	trapv trap
SIGILL	**ILL_PRIVVIO_FAULT**	privilege violation
SIGTRAP	NULL	trace exception
SIGEMT	**EMT_EMU1010**	line 1010 emulator
SIGEMT	**EMT_EMU1111**	line 1111 emulator
SIGILL	**ILL_ILLINSTR_FAULT**	coprocessor protocol violation
SIGFMT	NULL	format error
SIGFPE	**FPE_FLTBSUN_TRAP**	compare unordered
SIGFPE	**FPE_FLTINEX_TRAP**	inexact result
SIGFPE	**FPE_FLTDIV_TRAP**	divide by zero
SIGFPE	**FPE_FLTUND_TRAP**	underflow
SIGFPE	**FPE_FLTOPERR_TRAP**	operand error
SIGFPE	**FPE_FLTOVF_TRAP**	overflow
SIGFPE	**FPE_FLTNAN_TRAP**	signaling "Not A Number"



PowerPC

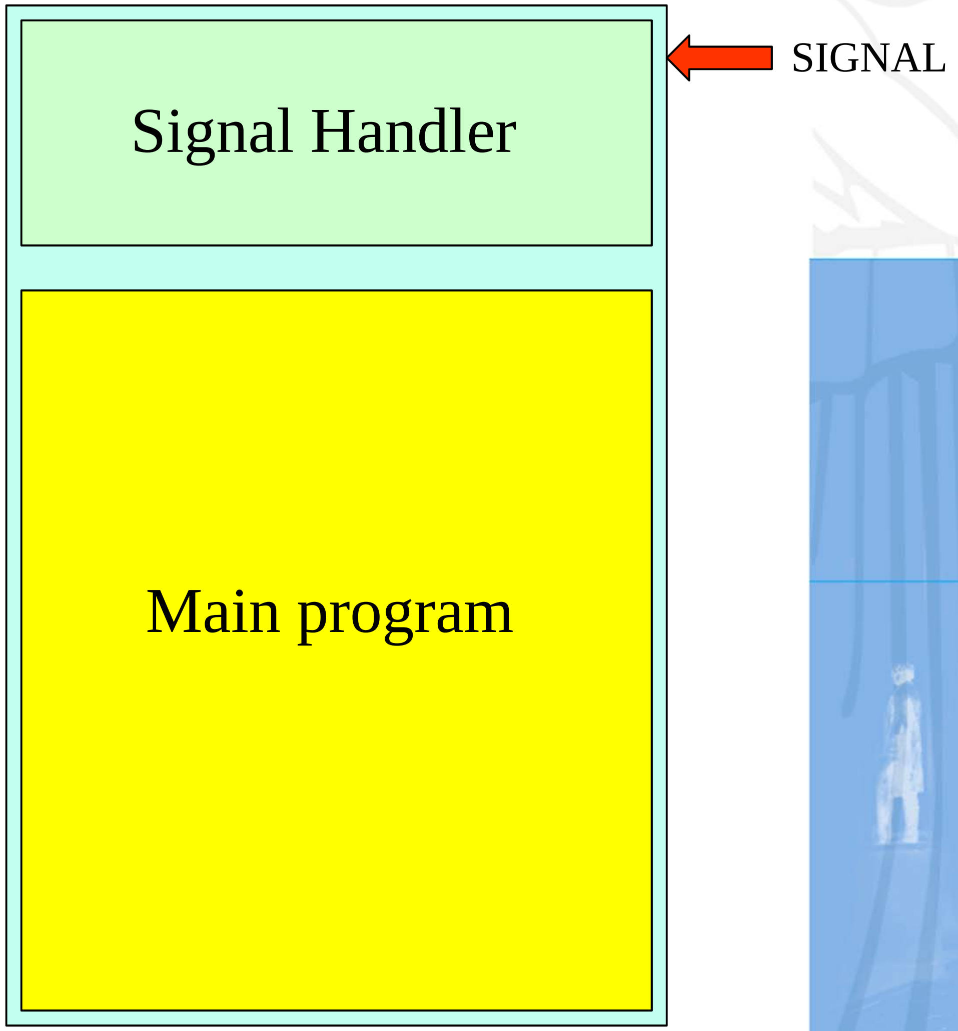
Signal	Code	Exception
SIGBUS	**_EXC_OFF_MACH**	machine check
SIGBUS	**_EXC_OFF_INST**	instruction access
SIGBUS	**_EXC_OFF_ALIGN**	alignment
SIGILL	**_EXC_OFF_PROG**	program
SIGBUS	**_EXC_OFF_DATA**	data access
SIGFPE	**_EXC_OFF_FPU**	floating point unavailable
SIGTRAP	**_EXC_OFF_DBG**	debug exception (PPC403)
SIGTRAP	**_EXC_OFF_INST_BRK**	inst. breakpoint (PPC603, PPCEC603, PPC604)
SIGTRAP	**_EXC_OFF_TRACE**	trace (PPC603, PPCEC603, PPC604, PPC860)
SIGBUS	**_EXC_OFF_CRTL**	critical interrupt (PPC403)
SIGILL	**_EXC_OFF_SYSCALL**	system call

(VxWorks) Signal Handler basics

A signal handler is a routine which is declared by the application program to be a signal handler for the process (task). A signal handler binds to a particular signal with **sigvec()** and **sigaction()** system calls. An application may declare several handlers.

The signal handler is invoked when the signal is asserted. It is executed within the context of the task which has installed the handler. The signal value is the entry value. The handler is executed even if the task is blocked!

How is the communication between the handler and the main program established? For instance, what to do if the signal originates from a division by zero in the main program? We will soon return to this problem.



VxWorks signal calls

Table 2-20: Basic Signal Calls (BSD and POSIX 1003.1b)

POSIX 1003.1b Compatible	UNIX BSD Compatible	Description
***signal*( )**	***signal*( )**	Specify the handler associated with a signal.
***kill*( )**	***kill*( )**	Send a signal to a task.
***raise*( )**	N/A	Send a signal to yourself.
***sigaction*( )**	***sigvec*( )**	Examine or set the signal handler for a signal.
***sigsuspend*( )**	***pause*( )**	Suspend a task until a signal is delivered.
***sigpending*( )**	N/A	Retrieve a set of pending signals blocked from delivery.
***sigemptyset*( ) *sigfillset*( ) *sigaddset*( ) *sigsetmask*( ) *sigdelset*( ) *sigismember*( )**		Manipulate a signal mask
***sigprocmask*( )**	***sigsetmask*( )**	Set the mask of blocked signals.
***sigprocmask*( )**	***sigblock*( )**	Add to a set of blocked signals.

Table 2-21: POSIX 1003.1b Queued Signal Calls

Call	Description
***sigqueue*( )**	Send a queued signal.
***sigwaitinfo*( )**	Wait for a signal.
***sigtimedwait*( )**	Wait for a signal with a timeout.

VxWorks signal handler installation, an incomplete example

- /* signal handler invoked when a signal is sent to the process that the handler belongs to */
- void **taskASigHandler** (int sig, struct siginfo *info, void *pContext)
- {
-   if (sig == SIGUSR1) --- *action #1 whatever that may be* --- ;
-   if (sig == SIGUSR2) --- *action #2 whatever that may be* ---;
-   if (sig == SIGKILL)   --- *can not be caught according to signal.h ;*
- }
- ……………………………………………………………………………………..
- /* main program */
- /* spawn interactively or by another task (process) */
- - - - - - - - - - - - - - - - -
-   struct sigaction Saction;
-   struct siginfo Sinfo;

- /* set up signal handler for accepted signals , see definitions in signal.h   */
-   Saction.sa_sigaction = **taskASigHandler** ;
-   Saction.sa_flags = SA_SIGINFO;   /* because handler needs the siginfo structure as an argument, the SA_SIGINFO flag is set in sa_flags. */
-   sigemptyset (&Saction.sa_mask);
- /* bind the signal handler to following signals */
-   sigaction (SIGUSR1,   &Saction,   NULL);
-   sigaction (SIGUSR2,   &Saction,   NULL);
-   sigaction (SIGKILL,   &Saction,   NULL);
-   -

Now the tricky part: What to do after exceptions?

- In the following we will disregard ”normal” I/O interrupts, they are handled by the appropriate drivers.
- The problem is to make a connection between the exception as catched by the signal handler and the ”application” program that caused the exception
  - In the following we will term as exception also error conditions which are detected directly by the application, for instance an error return from a system call
- Some of the following pages are from the book by Burns and Wellings

The Domain of an Exception Handler

- Within a program, there may be several handlers for particular exceptions
  - One can install a signal handler for several types of exception, as shown in the previous example code, or one can install a handler for each type
- Associated with each handler is a domain which specifies the region of computation during which, if an exception occurs, the handler will be activated
- The accuracy with which a domain can be specified will determine how precisely the source of the exception can be located
- An exception does not necessarily reflects an error, the signal feature is also a valid inter-process communication method.
- However, if the exception is caused by an error, then there a need for a recovery action. “For RT systems, to fail is not an option!”

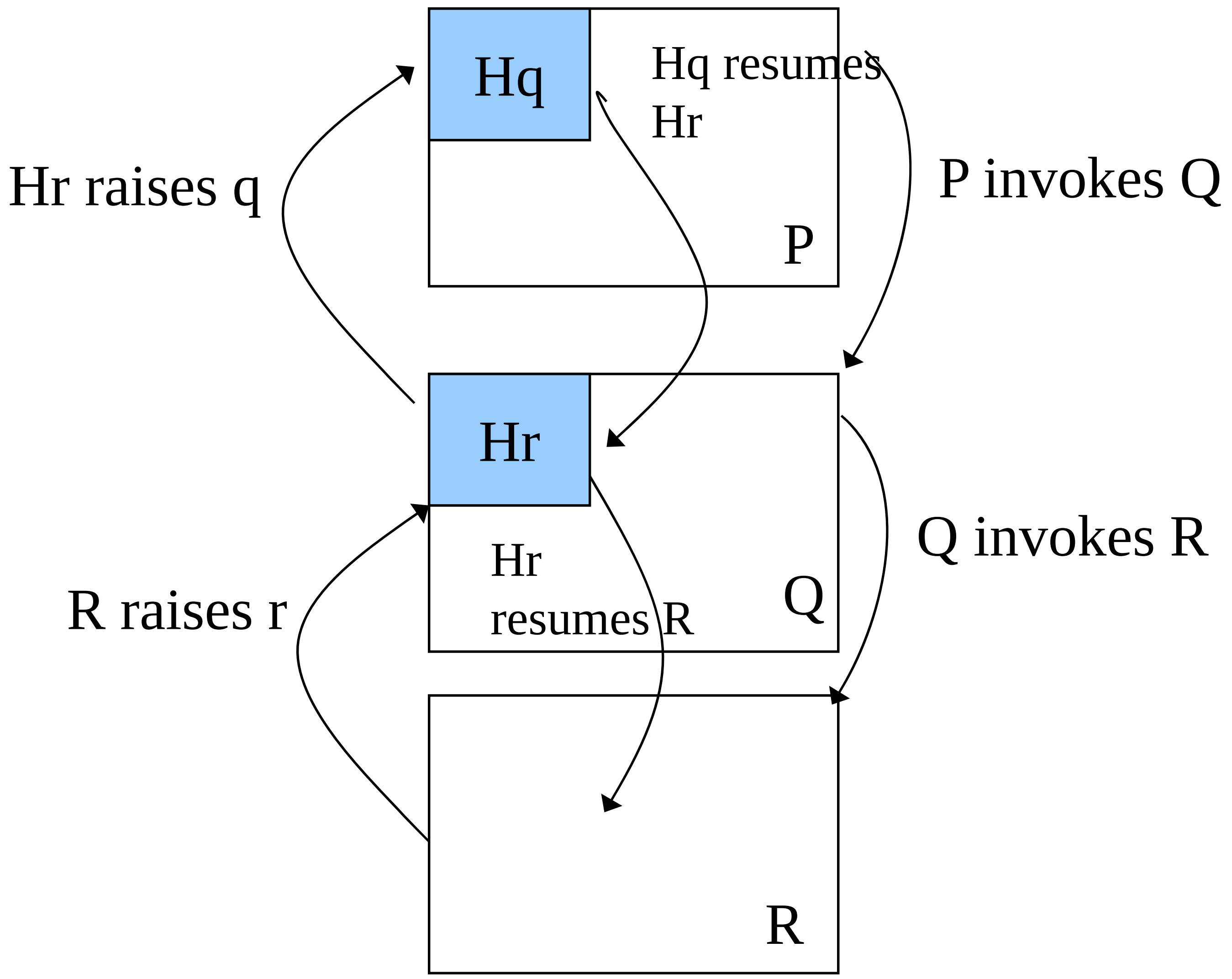
Resumption versus termination model

- Should the invoker (= application) of the exception continue its execution after the exception has been handled ??
- If the invoker can continue, then it may be possible for the handler to cure the problem that caused the exception to be raised and for the invoker to resume as if nothing has happened
- This is referred to as the resumption or notify model
- The model where control is not returned to the invoker is called termination or escape
- A model in which the handler can decide whether to resume the operation which caused the exception, or to terminate the operation; is called the hybrid model

The Resumption Model

- Consider three procedures P, Q and R on next page
- P invokes Q which in turn invokes R.
- R raises an exception which is handled by Q assuming there is no local handler in R.
- The handler for r is Hr.
- In the course of handling r, Hr raises exception q which is handled by Hq in procedure P (the caller of Q).
- Once this has been handled Hr continues its execution and when finished R continues
- Most easily understood by viewing the handler as an implicit procedure which is called when the exception is raised

The Resumption Model



Comments on the Resumption Model

- Problem: it is difficult to repair errors raised by the RTS
- Eg, an arithmetic overflow in the middle of a sequence of complex expressions results in registers containing partial evaluations; calling the handler overwrites these registers
- Implementing a strict resumption model is difficult, a compromise is to re-execute the block associated with the exception handler
- Note that for such a scheme to work, the local variables of the block must not be re-initialised on a retry
- The advantage of the resumption model comes when the exception has been raised asynchronously and, therefore, has little to do with the current process execution

The Termination Model

- In the termination model, when an exception has been raised and the handler has been called, control does not return to the point where the exception occurred
- Instead the block or procedure containing the handler is terminated, and control is passed to the calling block or procedure
- An invoked procedure, therefore, may terminate in one of a number of conditions
- One of these is the normal condition, while the others are exception conditions
- When the handler is inside a block, control is given to the first statement following the block after the exception has been handled

Exception Handling and Operating Systems

- Languages like Ada or Java will usually be executed on top of an operating system
- These systems will detect certain synchronous error conditions, eg, memory violation or illegal instruction
- This will usually result in the process being terminated; however, many systems allow error recovery
- POSIX allows handlers to be called when these exceptions are detected (called signals in POSIX)
- Once the signal is handled, the process is resumed at the point where it was “interrupted” — hence POSIX supports the resumption model
- If a language supports the termination model, the RTSS must catch the error and manipulate the program state so that the program can use the termination model

Summary

- With the resumption model, the invoker of the exception is resumed at the statement after the one at which the exception was invoked
- With the termination model, the block or procedure containing the handler is terminated, and control is passed to the calling block or procedure.
- The hybrid model enables the handler to choose whether to resume or to terminate
- Parameter passing to the handler -- may or may not be allowed

C Exceptions

- C does not define any exception handling facilities
- This clearly limits its in the structured programming of reliable systems
- However, it is possible to provide some form of exception handling mechanism by using the C macro facility
- To implement a termination model, it is necessary to save the status of a program's registers etc. on entry to an exception domain and then restore them if an exception occurs.
- The POSIX facilities of **setjmp** and **longjmp** can be used for this purpose

setjmp *and* ***longjmp***

- **setjmp** saves the program status and returns a 0
- **longjmp** restores the program status and results in the program abandoning its current execution and restarting from the position where **setjmp** was called
- This time **setjmp** returns the values passed by **longjmp**

A naive Exception handling in C

```
/* The simplest error handling based on setjmp() and longjmp() */

#include    "vxWorks.h"
#include    "stdio.h"
#include    "setjmp.h"

jmp_buf    jumper;

int SomeFunction(int a, int b)
{
    if (b == 0) longjmp(jumper, -3); /* can't divide by 0 */
    return a / b;
}

void SomeTask (void)
{
   int result;
   if (setjmp(jumper) == 0)      /* returns here after exception with setjmp() != 0 */
   {
      result = SomeFunction(7, 0);
      /* continue working with result if  OK
      - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
   }
  else
     printf("an error occurred, but what to do now?\n");
}
```

A principle for a more general approach?

- The preceding example relies on a single global variable ***jumper*** of type ***jump_buf*** *.* However, one may need different exception handlers in different functions. So how will SomeFunction() know which *jumper* to use?
- Needed: a dynamically linked list of exception handler records, containing a *jump_buf* struct and whatever additional information which is needed.
- Each function which defines an exception handler adds such a record to the list and removes it from the list when it returns.

The exception handling effectiveness of POSIX operating systems

Koopman, P.; DeVale, J.
Software Engineering, IEEE Transactions on
Volume 26, Issue 9, Sep 2000 Page(s):837 - 848

- Summary (1/2): Operating systems form a foundation for robust application software, making it important to understand how effective they are at handling exceptional conditions. The Ballista testing system was used to characterize the handling of exceptional input parameter values for up to 233 POSIX functions and system calls on each of 15 widely used operating system (OS) implementations. This identified ways to crash systems with a single call, ways to cause task hangs within OS code, ways to cause abnormal task termination within OS and library code, failures to implement defined POSIX functionality, and failures to report unsuccessful operations.
  - Ballista approach: automatic creation and execution of N sets of invalid input data

The exception handling effectiveness of POSIX operating systems

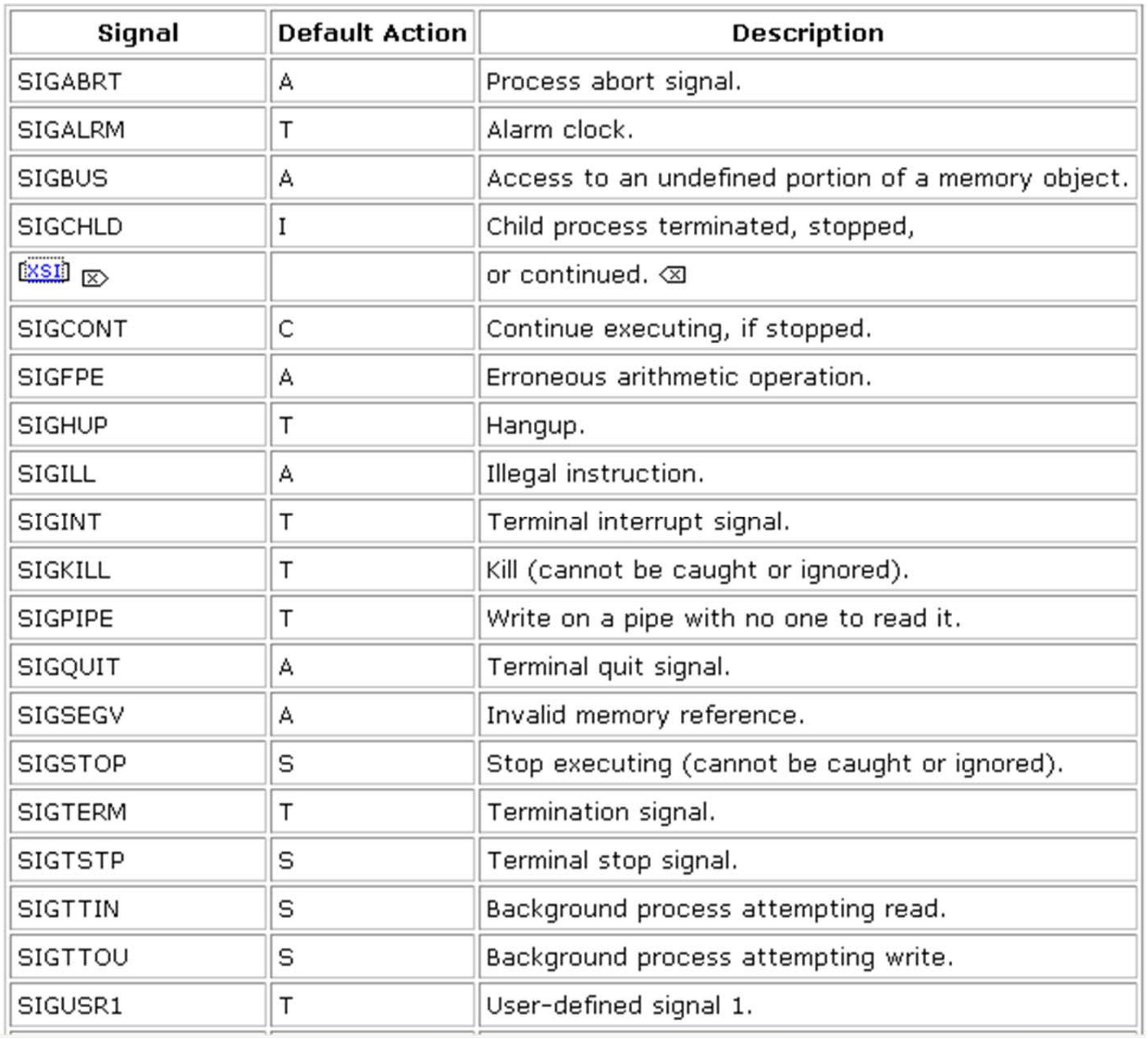
Koopman, P.; DeVale, J.

Software Engineering, IEEE Transactions on

Volume 26, Issue 9, Sep 2000 Page(s):837 - 848

- Summary (2/2):
  - Overall, only 55 percent to 76 percent of the exceptional tests performed generated error codes, depending on the operating system being tested.
  - Approximately 6 percent to 19 percent of tests failed to generate any indication of error despite exceptional inputs.
  - Approximately 1 percent to 3 percent of tests revealed failures to implement defined POSIX functionality for unusual, but specified, situations.
  - Between 18 percent and 33 percent of exceptional tests caused the abnormal termination of an OS system call or library function, and five systems were completely crashed by individual system calls with exceptional parameter values. The most prevalent sources of these robustness failures were illegal pointer values, numeric overflows, and end-of-file overruns

POSIX defined signals (not all of them)



Signal	Default Action	Description
SIGABRT	A	Process abort signal.
SIGALRM	T	Alarm clock.
SIGBUS	A	Access to an undefined portion of a memory object.
SIGCHLD	I	Child process terminated, stopped,
[XSI] ⌦		or continued. ⌫
SIGCONT	C	Continue executing, if stopped.
SIGFPE	A	Erroneous arithmetic operation.
SIGHUP	T	Hangup.
SIGILL	A	Illegal instruction.
SIGINT	T	Terminal interrupt signal.
SIGKILL	T	Kill (cannot be caught or ignored).
SIGPIPE	T	Write on a pipe with no one to read it.
SIGQUIT	A	Terminal quit signal.
SIGSEGV	A	Invalid memory reference.
SIGSTOP	S	Stop executing (cannot be caught or ignored).
SIGTERM	T	Termination signal.
SIGTSTP	S	Terminal stop signal.
SIGTTIN	S	Background process attempting read.
SIGTTOU	S	Background process attempting write.
SIGUSR1	T	User-defined signal 1.

The default actions are as follows:

T	Abnormal termination of the process. The process is terminated with all the consequences of *_exit*() except that the status made available to *wait*() and *waitpid*() indicates abnormal termination by the specified signal.
A	Abnormal termination of the process. [XSI] Additionally, implementation-defined abnormal termination actions, such as creation of a **core** file, may occur.
I	Ignore the signal
S	Stop the process
C	Continue the process, if it is stopped; otherwise, ignore the signal.

In addition, the Real-Time extension of POSIX defines the RTSIG_MAX signals SIGRTMIN to SIGRTMAX

VxWorks task exception handling

- Errors in program code or data can cause hardware exception conditions such as illegal instructions, bus or address errors, divide by zero, and so forth. The VxWorks exception handling package takes care of all such exceptions. The default exception handler suspends the task that caused the exception, and saves the state of the task at the point of the exception. The kernel and other tasks continue uninterrupted. A description of the exception is transmitted to the Tornado development tools, which can be used to examine the suspended task

VxWorks POSIX Queued Signals

- The ***sigqueue*( )** routine provides an alternative to *kill*( ) for sending signals to a task. The important differences between the two are:
- ***sigqueue*( )** includes an application-specified value that is sent as part of the signal. You can use this value to supply whatever context your signal handler finds useful. This value is of type sigval (defined in signal.h); the signal handler finds it in the si_value field of one of its arguments, a structure siginfo_t. An extension to the POSIX *sigaction*( ) routine allows you to register signal handlers that accept this additional argument
- ***sigqueue*( )** enables the queueing of multiple signals for any task. <u>The ***kill*( )** routine, by contrast, delivers only a single signal, even if multiple signals arrive before the handler runs!</u>
- VxWorks includes seven signals reserved for application use, numbered consecutively from **SIGRTMIN**. The presence of these reserved signals is required by POSIX 1003.1b, but the specific signal values are not; for portability, specify these signals as offsets from **SIGRTMIN** (for example, write **SIGRTMIN**+2 to refer to the third reserved signal number)

VxWorks Signal Codes

- Defined in header files, under ….target/h:
  - sigCodes.h
  - signal.h
  - arch/<processor>/…
- How to program signals and exceptions will be demonstrated in lab exercises

Summary I

- All exception handling models address the following issues
  - Exception representation: an exception may, or may not, be explicitly represented in a language
  - The domain of an exception handler: associated with each handler is a domain which specifies the region of computation during which, if an exception occurs, the handler will be activated
  - Exception propagation: when an exception is raised and there is no exception handler in the enclosing domain, either the exception can be propagated to the next outer level enclosing domain, or it can be considered to be a programmer error
  - Resumption or termination model: this determines the action to be taken after an exception has been handled.

Summary II

- With the resumption model, the invoker of the exception is resumed at the statement after the one at which the exception was invoked
- With the termination model, the block or procedure containing the handler is terminated, and control is passed to the calling block or procedure.
- The hybrid model enables the handler to choose whether to resume or to terminate
- Parameter passing to the handler -- may or may not be allowed

Summary III

- It is not unanimously accepted that exception handling facilities should be provided in a language
- The C and the occam2 languages, for example, have none
- To sceptics, an exception is a GOTO where the destination is undeterminable and the source is unknown!
- They can, therefore, be considered to be the antithesis of structured programming
- This is, however, not the view taken by the computer scientists Burns and Wellings!