Volume

1

Architectural Specific Data

Linux-S/390 & z/Architecture Reference

LINUX FOR S/390 AND Z/ARCHITECTURE

Architecture Specific Reference

Last updated: 2003-07-28

Table of Contents

DEBUGGING ON LINUX FOR 390 AND Z/ARCHITECTURE 3
- S/390 and z/Architecture Register Set 3
  - Program Status Word (PSW) 4
- Prefix Page 6
- Address Spaces on Linux 7
- Address Spaces on Linux for S390 and z/Architecture 8
  - Virtual Addresses on S/390 and z/Architecture 9
- The Linux for S390 Kernel Task Structure 10
- Register Usage and Stack-Frames on Linux for S390 11
  - Overview 11
  - Glossary 11
- S/390 and z/Architecture Register Usage 13
- Stack Frame Layout 15
- A sample program with comments 16
  - Comments on the function test 16
  - Comments on the function main 17
  - New Compiler Changes 17
  - 64 bit z/Architecture code disassembly 17
- Compiling programs for debugging on Linux for S390 and z/Architecture 18
- Figuring out gcc compile errors 19
- Debugging Tools 21
  - objdump 21
- strace 22
- Performance Debugging 25
  - Using top to find out where processes are sleeping in the kernel 25
  - The time command 25
- Debugging under VM 26
  - Useful VM debugger commands 27
  - Tracing particular processes 30
  - Stack tracing under VM 38
- S/390 and z/Architecture I/O Overview 40
  - General Concepts 42
  - Common 390 Devices 43
  - Debugging IO on S390 under VM 43
  - Other Common VM Device Related Commands 44
- gdb on S390 45
  - Invocation 45
  - Online help 45
  - Assembly 45
  - Viewing and modifying variables 46
  - Modifying execution 46
  - breakpoints 47
  - User defined functions/macros 47
  - Other hard to classify stuff 47
  - Hints 48
  - Stack chaining in gdb by hand 48
  - Disassembling instructions without debug information 49
  - For more information 49
- Examining Core Dumps 49
- ldd 51
- Debugging shared libraries 51
- Debugging modules 51
- The proc file system 52
  - Some driver debugging techniques 54
- Miscellaneous Techniques 55
  - Starting points for debugging scripting languages etc. 55
  - SysRq 56
  - References 56

S/390 DEBUGGING FACILITY 57
- Design 57
  - Example 58
- Kernel Interfaces 58
  - debug_register 58
  - debug_unregister 59
  - debug_set_level 59
  - debug_event 59
  - debug_int_event 59
  - debug_text_event 59
  - debug_sprintf_event 60
  - debug_exception 60
  - debug_int/long_exception 60
  - debug_text_exception 60
  - debug_sprintf_exception 61
  - debug_register_view 61
  - debug_unregister_view 61
- Predefined views 61
- Examples 61
  - hex_ascii + raw-view 62
  - sprintf-view 62
  - sprintf-view 63
- ProcFS Interface 63
  - Example – Viewing the Debug Log 63
  - Example - Changing the debug level 63
- Flushing Debug Areas 64
- lcrash Interface 64
- Investigating raw memory 64
- Predefined Views 64
  - Defining views 65
  - Example 66

COMMON I/O LAYER 68
- Command line parameters 68

/proc entries 69
/proc/subchannels 69
/proc/deviceinfo/ 69
/proc/cio_ignore 69
/proc/s390dbf/cio_*/ (S/390 debug feature) 70
/proc/irq_count 70
/proc/chpids 70

CHANNEL DEVICE LAYER 72
Chandev Arguments 74
Glossary 74
Commonly Used Options 74
Power User Options 75

COMMON DEVICE SUPPORT 78
General Information 79
Overview of CDS interface concepts 79
Miscellaneous Support Routines 99
Special Console Interface Routines 100

DASD DEVICE DRIVER 102
Usage 102
Low-level format 102
Make a filesystem 103
Bugs 103
TODO-List 103

TAPE SUPPORT 104
Tape driver features 104
Tape character device front-end 104
Tape block device front-end 104
Tape block device example 105
TODO List 105
BUGS 105

3270 DISPLAY SYSTEM SUPPORT 107
INTRODUCTION 107
OPERATION 107

XPRAM 110
Features 110
Limitations 110
Configuration option 110
Module name 110
Kernel parameter syntax 110
Example 111
Module parameter syntax 111
Example 112
Usage 112

CISCO CLAW SUPPORT 113

IUCV 115
iucv_accept 115
Parameters 115
Returns 116
iucv_connect 116
Parameters 116
Returns 117
iucv_purge 117
Parameters 118
Returns 118
iucv_query_maxconn 118
Parameters 118
Returns 118
iucv_query_bufsize 118
Parameters 118
Returns 118
iucv_quiesce 118
Parameters 118
Returns 119
iucv_receive 119
Parameters 119
Returns 120
iucv_receive_array 120
Parameters 120
Returns 121
iucv_reject 121
iucv_reply 121
Parameters 121
Returns 122
iucv_reply_array 122
Parameters 122
Returns 123
iucv_reply_prmmsg 123
Parameters 123
Returns 123
iucv_resume 123
Parameters 123
Returns 124
iucv_send 124
Parameters 124
Returns 124
iucv_send2way 124
Parameters 125
Returns 125
iucv_send2way_array 125
Returns 125
Returns 126
iucv_send_array 126
Parameters 126
Returns 127
iucv_send2way_prmmsg 127
Parameters 127
Returns 127
iucv_send2way_prmmsg_array 128
Parameters 128
Returns 128
iucv_send_prmmsg 128
Parameters 129
Returns 129
iucv_setmask 129
Parameters 129
iucv_sever 129
Parameters 130
Returns 130
Iucv_register_program 130
Parameters 130
Returns 130
Notes 130
iucv_unregister_program 131
Parameters 131
Returns 131

Debugging on Linux for 390 and z/Architecture

by
Denis Joseph Barrow (djbarrow@de.ibm.com, barrow_dj@yahoo.com)
Copyright © 2000 IBM Deutschland Entwicklung GmbH, IBM Corporation

This document is intended to give an good overview of how to debug Linux for S390 and z/Architecture. It isn't intended as a complete reference and not a tutorial on the fundamentals of C and assembly, it doesn't go into 390 IO in any detail. It is intended to compliment the following books.

- Enterprise Systems Architecture/390 Reference Summary SA22-7209-01 and the
- Enterprise Systems Architecture/390 Principles of Operation SA22-7201-05 and any other worthwhile references you get.

It is intended like the Enterprise Systems Architecture/390 Reference Summary to be printed out and used as a quick cheat sheet self help style reference when problems occur.

S/390 and z/Architecture Register Set

The current architectures have the following registers.

ESA/390	z/Architecture
16 32 bit General propose registers (r0-r15 or gpr0-gpr15) used for arithmetic and addressing	16 64 bit General propose registers (r0-r15 or gpr0-gpr15) used for arithmetic and addressing
16 32 bit Control registers (cr0-cr15 kernel usage only) used for memory management, interrupt control, debugging control etc.	16 64 bit Control registers (cr0-cr15 kernel usage only) used for memory management, interrupt control, debugging control etc.
16 Access registers (ar0-ar15) not used by normal programs but potentially could be used as temporary storage. Their main purpose is their 1 to 1 association with general-purpose registers and is used in the kernel for copying data between kernel user address spaces. Note ar0 (and ar1 on z/Architecture) is currently used by the pthread library as a pointer to the current running thread's private area.	
16 64 bit floating point registers (fp0-fp15) IEEE and HFP floating point format compliant on G5 upwards and a Floating point control register (FPC)	

4 64 bit registers (fp0, fp2, fp4 and fp6) HFP only on older machines.	N/A

Note: Linux (currently) always uses IEEE and emulates G5 IEEE format on older machines, (provided the kernel is configured for this).

Program Status Word (PSW)

The PSW is the most important register on the machine it is 64 bits on S/390 and 128 bits on z/Architecture, and serves the roles of a program counter (PC), condition code register, memory space designator.

In IBM standard notation I am counting bit 0 as the MSB. It has several advantages over a normal program counter in that you can change address translation and program counter in a single instruction. To change address translation, for example, switching address translation off requires that you have a logical=physical mapping for the address at which you are currently running.

Bit		**Value**
S/390	**zArch**	
0	0	Reserved (must be 0 otherwise specification exception occurs.
1	1	Program Event Recording 1 PER enable. PER is used to facilitate debugging e.g. single stepping.
2-4	2-4	Reserved (must be 0).
5	5	Dynamic address translation 1=DAT on.
6	6	Input/Output interrupt Mask
7	7	External interrupt Mask used primarily for inter-processor signaling and clock interrupts.
8-11	8-11	PSW Key used for complex memory protection mechanism not used under Linux
12	12	Architecture selection: 1 on S/390; 0 on z/Architecture
13	13	Machine Check Mask 1=enable machine check interrupts
14	14	Wait State set this to 1 to stop the processor except for interrupts and give time to other LPARS used in CPU idle in the kernel to increase overall usage of processor resources.

15	15	Problem state (if set to 1 certain instructions are disabled) all Linux user programs run with this bit 1 (useful info for debugging under VM).
16-17	16-17	Address Space Control 00 Primary Space Mode when DAT on. The Linux kernel currently runs in this mode: CR1 is affiliated with this mode and points to the primary segment table origin etc. 01 Access register mode this mode is used in functions to copy data between kernel and userspace. 10 Secondary space mode not used in Linux however CR7 the register affiliated with this mode is and this and normally CR13=CR7 to allow us to copy data between kernel and user space. We do this as follows: We set ar2 to 0 to designate its affiliated gpr (gpr2) to point to primary=kernelspace. We set ar4 to 1 to designate its affiliated gpr (gpr4) to point to secondary=home=user space and then essentially do a `memcopy(gpr2,gpr4,size)` to copy data between the address spaces. The reason we use home space for the kernel and don't keep secondary space free is that code will not run in secondary space. 11 Home Space Mode all user programs run in this mode. It is affiliated with CR13.
18-19	18-19	Condition codes (CC)
20	20	Fixed point overflow mask if 1=FPU exceptions for this event occur (normally 0)
21	21	Decimal overflow mask if 1=FPU exceptions for this event occur (normally0)
22	22	Exponent underflow mask if 1=FPU exceptions for this event occur (normally 0)
23	23	Significance Mask if 1=FPU exceptions for this event occur (normally 0)
24-31	24-30	Reserved Must be 0.
	31-32	Extended Addressing Mode; Basic Addressing mode. Used to set addressing mode 00 – 24 bit 01 – 31 bit 11 – 64 bit
32		1=31 bit addressing mode 0=24 bit addressing mode (for backward compatibility). Linux always runs with this bit set to 1
33-64		Instruction address.

	33-63	Reserved. Must be 0.
	64-127	Address: In 24-bit mode bits 64-103=0; bits 104-127=Address In 31-bit mode bits 64-96=0; bits 97-127=Address Note: unlike 31-bit mode on S/390, bit 96 must be zero when loading the address with LPSWE otherwise a specification exception occurs. LPSW is fully backward compatible.

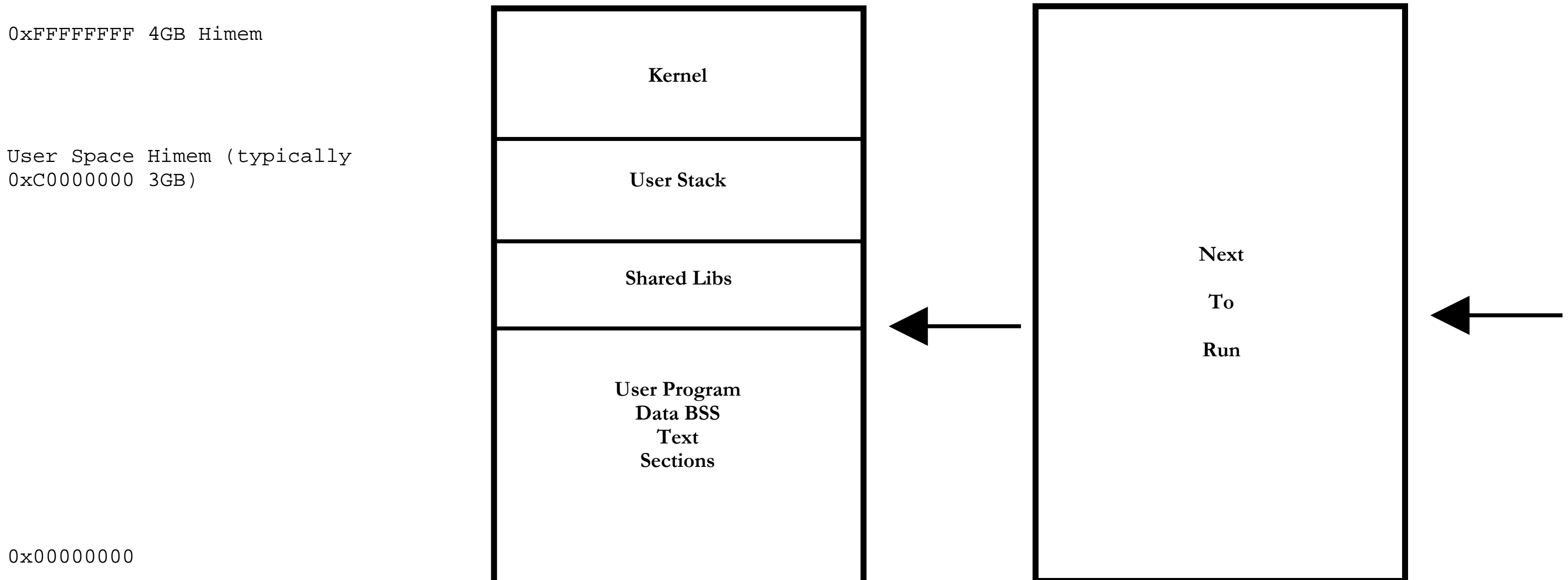
Prefix Page

This per CPU memory area is too intimately tied to the processor not to mention. It exists between the real addresses 0-4096 on S/390 and 0-8192 on z/Architecture, and is exchanged with a 1 page on S/390 or 2 pages on z/Architecture, in absolute storage by the set prefix instruction in Linux's startup. This page is mapped to a different prefix for each processor in an SMP configuration. Bytes 0-512 (200 hex) on S/390 and 0-512; 4096-4544; 4604-5119 currently on z/Architecture, are used by the processor itself for holding such information as exception indications and entry points for exceptions.

There is a gap on z/Architecture between 0xc00-0x1000 that Linux uses for per processor global variables. The closest thing to this on traditional architectures is the interrupt vector table. This is a good thing and does simplify some of the kernel coding however it means that we now cannot catch stray NULL pointers in the kernel without hard coded checks.

Address Spaces on Linux

The traditional Intel Linux is approximately mapped as follows:



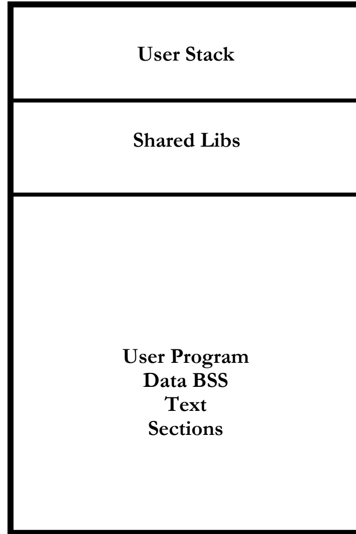
Now it is easy to see that on Intel it is quite easy to recognize a kernel address as being one greater than user space high memory (in this case 0xC0000000). Addresses of less than this are the ones in the current running program on this processor (if an SMP box). If using the virtual machine (VM) as a debugger it is quite difficult to know which user process is running, as the address space you are looking at could be from any process in the run-queue. Thankfully you normally get lucky as address spaces don't overlap that and you can recognize the code at by cross-referencing with a dump made by `objdump` (more about that later).

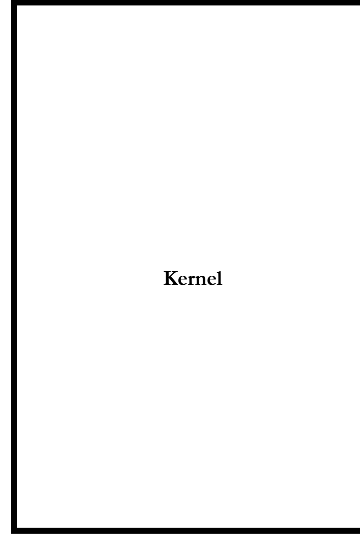
The limitation of Intel's addressing technique is that the Linux kernel uses a very simple real address to virtual addressing technique of Real Address=Virtual Address-User Space Himem. This means that on Intel the kernel Linux can typically only address Himem=0xFFFFFFFF-0xC0000000=1GB and this is all the RAM these machines can typically use. They can lower User Himem to 2GB or lower and thus be able to use 2GB of RAM however this shrinks the maximum size of User Space from 3GB to 2GB they have a no win limit of 4GB unless they go to 64 Bit. On S/390 our limitations and strengths make us slightly different. For backward compatibility (because of the PSW address hi bit which indicates whether we are in 31 or 24 bit mode) we are only allowed use 31 bits (2GB) of our 32 bit addresses. However, we use entirely separate address spaces for the user and kernel. This means we can support 2GB of non-extended RAM, and more with the extended memory management swap device and 4TB of physical memory currently on z/Architecture.

Address Spaces on Linux for S390 and z/Architecture

S/390 is, for historical reasons, a 31-bit system that means it can only address 2GB of storage. However, thanks to the multiple address space facilities of the architecture we are able to use entirely different address spaces for user and kernel. On S/390 our addressing scheme is as follows:

0x7FFFFFFF 2GB Himem

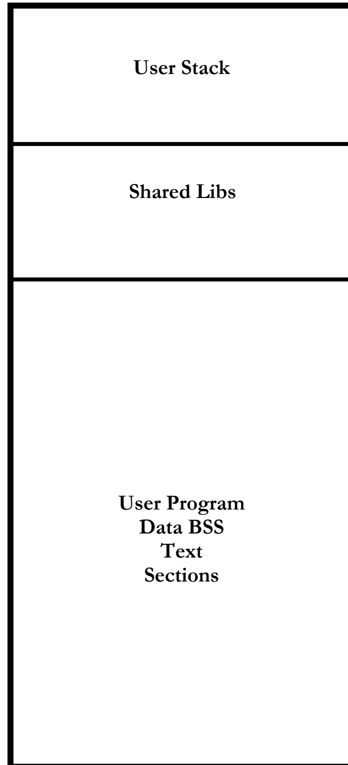


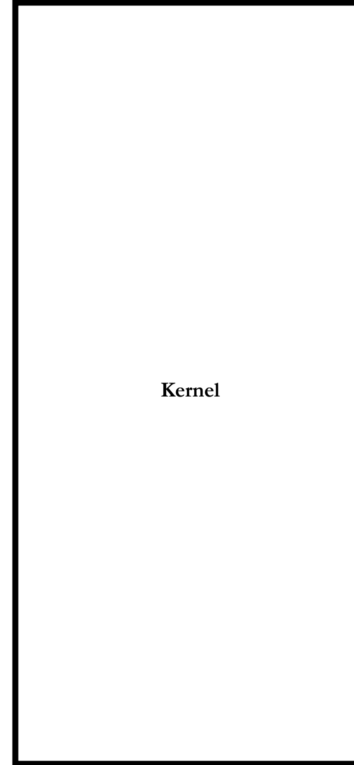


0x00000000

Under z/Architecture the structure is:

0x3FFFFFFFFFF 4TB Himem





0x00000000

This also means that we need to look at the PSW problem state bit or the addressing mode to decide whether we are looking at user or kernel space.

Virtual Addresses on S/390 and z/Architecture

A virtual address on S/390 is made up of 3 parts.

1. The SX (segment index, roughly corresponding to the PGD and PMD in Linux terminology): bits 1-11.
2. The PX (page index, corresponding to the page table entry (PTE)) bits 12-19.
3. The remaining bits BX (the byte index is the offset into a page) bits 20-31.

On z/Architecture in Linux we currently make up an address from 4 parts:

1. The region index (RX) bits 0-32 of which we use bits 22-32.
2. The segment index (SX): bits 33-43.
3. The page index (PX): bits 44-51.
4. The byte index (BX): bits 52-63.

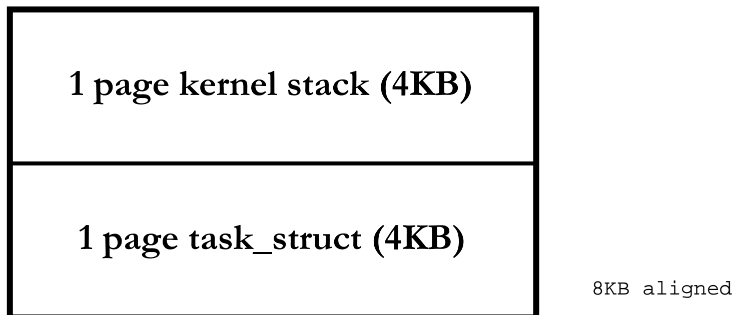
Notes:

1. S/390 has no PMD so the PMD is really the PGD also. A lot of this stuff is defined in `pgtable.h`.
2. Also seeing as S/390's page indexes are only 1K in size (bits 12-19 x 4 bytes per PTE), we use 1 page to make the best use of memory by updating 4 segment indices each time we mess with a PMD and use offsets 0, 1024, 2048 and 3072 in this page as for our segment indexes. On z/Architecture our page indexes are now 2K in size (bits 12-19 by 8 bytes per PTE). We do a similar trick but only mess with 2 segment indices each time we mess with a PMD.
3. z/Architecture supports up to a massive 5-level page table lookup, however currently we can only use 3 on Linux (as this is all the generic kernel currently supports). This may change in the future. This 3-tier structure allows us to access 4TB of virtual storage per process. To do this we use a region-third-table designation type in our address space control registers.

The Linux for S390 Kernel Task Structure

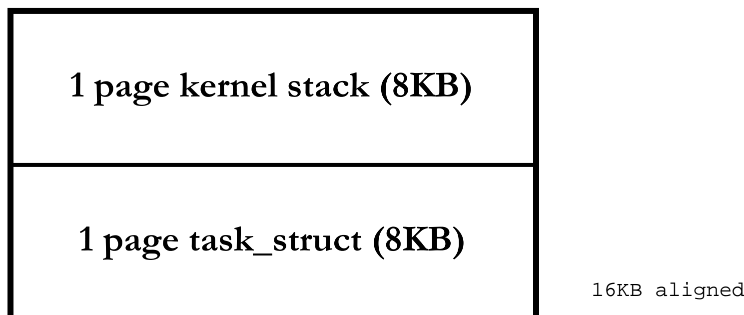
Each process/thread under Linux for S390 has its own kernel `task_struct` defined in `linux/include/linux/sched.h`. The S390 on initialization and resuming of a process on a cpu sets the `__LC_KERNEL_STACK` variable in the spare prefix area for this CPU (which we use for per processor globals). The kernel stack pointer is intimately tied with the task structure for each processor as follows.

1. S/390 –



8KB aligned

2. z/Architecture –



16KB aligned

What this means is that we don't need to dedicate any register or global variable to point to the current running process and can retrieve it with the following very simple construct for S/390 and one very similar for z/Architecture:

```
static inline struct task_struct * get_current(void)
{
 struct task_struct *current;

       __asm__("lhi %0,-8192\n\t"
               "nr  %0,15"
               : "=r" (current) );

       return current;
}
```

That is, just and'ing the current kernel stack pointer with the mask -8192. Thankfully because Linux doesn't have support for nested IO interrupts and our devices have large buffers can survive interrupts being shut for short amounts of time we don't need a separate stack for interrupts.

Register Usage and Stack-Frames on Linux for S390

Overview

This is the code that `gcc` produces at the top and the bottom of each function. It usually is fairly consistent and similar from function to function and if you know its layout you can probably make some headway in finding the ultimate cause of a problem after a crash without a source level debugger.

Note: To follow stack-frames requires knowledge of C or Pascal and limited knowledge of one assembly language.

Please note that there are some differences between the S/390 and z/Architecture stack layouts as the latter didn't have to maintain compatibility with older linkage formats.

Glossary

alloca	This is a built in compiler function for runtime allocation of extra space on the caller's stack that is obviously freed up on function exit. For example, the caller may choose to allocate nothing of a buffer of 4k if required for temporary purposes. It generates very efficient code (a few cycles) when compared to alternatives like `malloc()`.
automatics	These are local variables on the stack, that is, they aren't in registers and they aren't static.
back-chain	This is a pointer to the stack pointer before entering a framed functions (see frameless function) prologue got by de-referencing the address of the current stack pointer, i.e. got by accessing the 32/64 bit value at the stack pointers current location.
base-pointer	This is a pointer to the back of the literal pool that is an area just behind each procedure used to store constants in each function.
call-clobbered	The caller probably needs to save these registers if there is something of value in them, on the stack or elsewhere before making a call to another procedure so that it can restore it later.
epilogue	The code generated by the compiler to return to the caller.
frameless-function	A frameless function in Linux is one that does nOt need more than the

	register save area (96 bytes for S/390; 160 on z/Architecture) given to it by the caller. A frameless function never: ▪ Sets up a back chain. ▪ Calls alloca. ▪ Calls other normal functions ▪ Has automatics.
GOT-pointer	This is a pointer to the global-offset-table in ELF (Executable Linkable Format, Linux's most common executable format). All globals and shared library objects are found using this pointer.
lazy-binding	ELF shared libraries are typically only loaded when routines in the shared library are actually first called at runtime.
procedure-linkage-table	This is a table found from the GOT which contains pointers to routines in other shared libraries which can't be called to by easier means.
prologue	The code generated by the compiler to set up the stack frame.
outgoing-args	This is extra area allocated on the stack of the calling function if the parameters for the callee's cannot all be put in registers, the same area can be reused by each function the caller calls.
routine-descriptor	A COFF executable format based concept of a procedure reference actually being 8 bytes or more as opposed to a simple pointer to the routine. This is typically defined as follows: Routine Descriptor offset 0=Pointer to Function Routine Descriptor offset 4=Pointer to Table of Contents. The table of contents/TOC is roughly equivalent to a GOT pointer and it means that shared libraries etc. can be shared between several environments each with their own TOC.
static-chain	This is used in nested functions a concept adopted from pascal by gcc not used in ansi C or C++ (although quite useful), basically it is a pointer used to reference local variables of enclosing functions. You might come across this stuff once or twice in your lifetime. For example, the function below should return 11 though gcc may get upset and toss warnings about unused variables.

	``
	int FunctionA(int a)
	{
	 int b;
	
	 FunctionC(int c)
	 {
	    b=c+1;
	 }
	
	 FunctionC(10);
	 return(b);
	}
	``

S/390 and z/Architecture Register Usage

r0	used by syscalls/assembly	call-clobbered
r1	used by syscalls/assembly	call-clobbered
r2	argument 0 / return value 0	call-clobbered
r3	argument 1 / return value 1 (if long long on S/390)	call-clobbered
r4	argument 2	call-clobbered
r5	argument 3	call-clobbered
r6	argument 4	saved
r7	pointer-to arguments 5 to ...	saved
r8	this and that	saved
	this and that	saved
r10	static-chain (if nested function)	saved
r11	frame-pointer (if function used alloca)	saved
r12	got-pointer	saved

r13	base-pointer	saved
r14	return-address	saved
r15	stack-pointer	saved
f0	r9	call-clobbered
f2	argument 1	call-clobbered
f4	z/Architecture argument 2	saved
f6	z/Architecture argument 3	saved
The remaining floating points f1,f3,f5 f7-f15 are call-clobbered.		

Notes:

1. The only requirement is that registers that are used by the callee are saved. For example, the compiler is perfectly capable of using r11 for purposes other than a frame a frame pointer if a frame pointer is not needed.

2. In functions with variable arguments. For example, `printf` the calling procedure is identical to one without variable arguments and the same number of parameters. However, the prologue of this function is somewhat more hairy owing to it having to move these parameters to the stack to get `va_start`, `va_arg` and `va_end` to work.

3. Access registers are currently unused by gcc but are used in the kernel. Possibilities exist to use them at the moment for temporary storage but it isn't recommended.

4. Only 4 of the floating point registers are used for parameter passing as older machines such as G3 only have only 4 and it keeps the stack frame compatible with other compilers. However, with IEEE floating-point emulation under Linux on the older machines you are free to use the other 12.

5. A `long long` or `double` parameter cannot be the first 4 bytes in a register and the second four bytes in the outgoing args area. It must be purely in the outgoing args area if crossing this boundary.

6. Floating-point parameters are mixed with outgoing args on the outgoing args area in the order they are passed in as parameters.

7. Floating-point arguments 2 and 3 are saved in the outgoing args area for z/Architecture

Stack Frame Layout

S/390	z/Architecture	Description
0	0	back chain (a 0 here signifies end of back chain)
4	8	eos (end of stack, not used on Linux for S390 used in other linkage formats)
8	16	glue used in other linkage formats for saved routine descriptors etc.
12	24	glue used in other linkage formats for saved routine descriptors etc.
16	32	scratch area
20	40	scratch area
24	48	saved r6 of caller function
28	56	saved r7 of caller function
32	64	saved r8 of caller function
36	72	saved r9 of caller function
40	80	saved r10 of caller function
44	88	saved r11 of caller function
48	96	saved r12 of caller function
52	104	saved r13 of caller function
56	112	saved r14 of caller function
60	120	saved r15 of caller function

64	128	saved f4 of caller function
72	136	saved f6 of caller function
80		undefined
96	160	outgoing args passed from caller to callee
96+x	160+x	possible stack alignment (8 bytes desirable)
96+x+y	160+x+y	alloca space of caller (if used)
96+x+y+z	160+x+y+z	automatics of caller (if used)
0		back-chain

A sample program with comments

Comments on the function test

1. It didn't need to set up a pointer to the constant pool gpr13 as it isn't used (:-( ).
2. This is a frameless function and no stack is bought.
3. The compiler was clever enough to recognize that it could return the value in r2 as well as use it for the passed in parameter (:-)).
4. The basr (branch and save) trick works as follows the instruction has a special case with r0 (r0 with some instruction operands is understood as the literal value 0, some RISC architectures also do this). So now we are branching to the next address and the address new program counter is in r13, so now we subtract the size of the function prologue we have executed + the size of the literal pool to get to the top of the literal pool

```
0040037c int test(int b)
{ # Function prologue below
 40037c: 90 de f0 34        stm    %r13,%r14,52(%r15)   # Save registers r13 & r14
 400380: 0d d0              basr   %r13,%r0             # Set up pointer to constant pool using
 400382: a7 da ff fa        ahi    %r13,-6              # basr trick
        return(5+b);
 # Huge main program
 400386: a7 2a 00 05        ahi    %r2,5                # add 5 to r2
 # Function epilogue below
 40038a: 98 de f0 34        lm     %r13,%r14,52(%r15)   # restore registers r13 & 14
 40038e: 07 fe       br     %r14                        # return
}
```

Comments on the function main

The compiler did this function optimally (8-))

```
Literal pool for main.
400390: ff ff ff ec  .long 0xffffffec
main(int argc,char *argv[])
{ # Function prologue below
 400394: 90 bf f0 2c        stm    %r11,%r15,44(%r15)   # Save necessary registers
 400398: 18 0f              lr     %r0,%r15             # copy stack pointer to r0
 40039a: a7 fa ff a0        ahi    %r15,-96             # Make area for callee saving
 40039e: 0d d0              basr   %r13,%r0             # Set up r13 to point to
 4003a0: a7 da ff f0        ahi    %r13,-16             # literal pool
 4003a4: 50 00 f0 00        st     %r0,0(%r15)          # Save backchain
      return(test(5));                                  # Main Program Below
 4003a8: 58 e0 d0 00        l      %r14,0(%r13)         # load relative address of test from
 # literal pool
 4003ac: a7 28 00 05        lhi    %r2,5                # Set first parameter to 5
 4003b0: 4d ee d0 00        bas    %r14,0(%r14,%r13)    # jump to test setting r14 as return
                                                        # address using branch & save instruction.
 # Function Epilogue below
 4003b4: 98 bf f0 8c        lm     %r11,%r15,140(%r15)  # Restore necessary registers.
 4003b8: 07 fe              br     %r14                 # return to do program exit
}
```

New Compiler Changes

```
main(int argc,char *argv[])
{
  4004fc:     90 7f f0 1c          stm    %r7,%r15,28(%r15)
  400500:     a7 d5 00 04          bras   %r13,400508 <main+0xc>
  400504:     00 40 04 f4          .long  0x004004f4
# compiler now puts constant pool in code to so it saves an instruction
  400508:     18 0f                lr     %r0,%r15
  40050a:     a7 fa ff a0          ahi    %r15,-96
  40050e:     50 00 f0 00          st     %r0,0(%r15)
return(test(5));
  400512:     58 10 d0 00          l      %r1,0(%r13)
  400516:     a7 28 00 05          lhi    %r2,5
  40051a:     0d e1                basr   %r14,%r1
# compiler adds 1 extra instruction to epilogue this is done to
# avoid processor pipeline stalls owing to data dependencies on g5 &
# above as register 14 in the old code was needed directly after being loaded
  # by the lm %r11,%r15,140(%r15) for the br %14.
  40051c:     58 40 f0 98          l      %r4,152(%r15)
  400520:     98 7f f0 7c          lm     %r7,%r15,124(%r15)
  400524:     07 f4                br     %r4
}
```

Hartmut (our compiler developer) also has been threatening to take out the stack backchain in optimized code as this also causes pipeline stalls: you have been warned.

64 bit z/Architecture code disassembly

If you understand the stuff above you'll understand the stuff below too so I'll avoid repeating myself and just say that some of the instructions have g's on the end of them to indicate they are 64 bit and the stack offsets are a bigger. The only other difference you'll find between 32 and 64 bit is that we now use f4 and f6 for floating point arguments on 64 bit.

```
00000000800005b0 <test>:
```

```
int test(int b)
{
return(5+b);
    800005b0: a7 2a 00 05          ahi    %r2,5
    800005b4: b9 14 00 22          lgfr   %r2,%r2 # downcast to integer
    800005b8: 07 fe                br     %r14
    800005ba: 07 07                bcr    0,%r7
}

00000000800005bc <main>:
main(int argc,char *argv[])
{
    800005bc: eb bf f0 58 00 24    stmg   %r11,%r15,88(%r15)
    800005c2: b9 04 00 1f          lgr    %r1,%r15
    800005c6: a7 fb ff 60          aghi   %r15,-160
    800005ca: e3 10 f0 00 00 24    stg    %r1,0(%r15)
return(test(5));
800005d0:     a7 29 00 05          lghi   %r2,5
# brasl allows jumps > 64k & is overkill here bras would do fine
    800005d4: c0 e5 ff ff ff ee    brasl  %r14,800005b0 <test>
    800005da: e3 40 f1 10 00 04    lg     %r4,272(%r15)
    800005e0: eb bf f0 f8 00 04    lmg    %r11,%r15,248(%r15)
    800005e6: 07 f4                br     %r4
}
```

Compiling programs for debugging on Linux for S390 and z/Architecture

`-gdwarf-2` now works it should be considered the default debugging format for s/390 and z/Architecture as it is more reliable for debugging shared libraries, normal -g debugging works much better now Thanks to the IBM java compiler developers bug reports.

Make sure that the `gcc` is compiling and linking with the `-g` flag on this generates plain old gnu stabs, do not use `-ggdb`, `-gxcoff+` or any other silly option these other options more than likely don't work (we haven't tested them), `-gstabs` is supposed to add extra extensions to the debugging info for debugging c++ we haven't got round to testing this yet.

This is typically done adding/appending the flags `-g` or `-gdwarf-2` to the `CFLAGS` and `LDFLAGS` variables `Makefile` of the program concerned.

If using `gdb` and you would like accurate displays of registers and stack traces compile without optimization: That is, make sure that there is no `-O2` or similar on the `CFLAGS` line of the `Makefile` and the emitted gcc commands, obviously this will produce worse code (not advisable for shipment) but it is an aid to the debugging process.

This aids debugging because the compiler will copy parameters passed in registers onto the stack so backtracking and looking at passed in parameters will work, however some larger programs that use inline functions will not compile without optimization.

Debugging with optimization has since much improved after fixing some bugs, please make sure you are using gdb-5.0 or later developed after November 2000.

Figuring out gcc compile errors

If you are getting a lot of syntax errors compiling a program and the problem isn't blatantly obvious from the source. It often helps to just preprocess the file; this is done with the `-E` option in `gcc`. What this does is that it runs through the very first phase of compilation (compilation in `gcc` is done in several stages and `gcc` calls many programs to achieve its end result) with the `-E` option `gcc` just calls the preprocessor (`cpp`). The c preprocessor does the following, it joins all the files included together recursively (`#include` files can `#include` other files) and also the c file you wish to compile. It puts a fully qualified path of the included files in a comment and it does macro expansion. This is useful for debugging because:

1. You can double check whether the files you expect to be included are the ones that are being included (for example, double check that you aren't going to the i386 `asm` directory).

2. Check that macro definitions are not clashing with `typedefs`,

3. Check that definitions are not being used before they are being included.

4. Helps put the line emitting the error under the microscope if it contains macros.

For convenience the Linux kernel's makefile will do preprocessing automatically for you by suffixing the file you want built with `.i` (instead of `.o`). For example, from the Linux directory type:

```
make arch/s390/kernel/signal.i
```

This will build:

```
s390-gcc -D__KERNEL__ -I/home1/barrow/linux/include -Wall -Wstrict-prototypes -O2 -fomit-
frame-pointer
-fno-strict-aliasing -D__SMP__ -pipe -fno-strength-reduce -E arch/s390/kernel/signal.c
arch/s390/kernel/signal.i
```

Now look at `signal.i` you should see something like:

```
# 1 "/home1/barrow/linux/include/asm/types.h" 1
typedef unsigned short umode_t;
typedef __signed__ char __s8;
typedef unsigned char __u8;
typedef __signed__ short __s16;
typedef unsigned short __u16;
```

If instead you are getting errors further down. For example, `unknown instruction: 2515 "move.l"` or better still `unknown instruction:2515 "Fixme not implemented yet, call Martin"` you are probably are attempting to compile some code meant for another architecture or code that is simply not implemented, with a fixme statement stuck into the inline assembly code so that the author of the file now knows he has work to do. To look at the assembly emitted by gcc just before it is about to call `gas` (the gnu assembler) use the `-S` option. Again for your convenience the Linux kernel's Makefile will hold your hand and do all this donkey work for you also by building the file with the `.s` suffix. For example, from the Linux directory type:

```
make arch/s390/kernel/signal.s
s390-gcc -D__KERNEL__ -I/home1/barrow/linux/include -Wall -Wstrict-prototypes -O2 -fomit-
frame-pointer
```

```
-fno-strict-aliasing -D__SMP__ -pipe -fno-strength-reduce -S arch/s390/kernel/signal.c
-o arch/s390/kernel/signal.s
```

This will output something like, (please note the constant pool and the useful comments in the prologue to give you a hand at interpreting it).

```
.LC54:
       .string "misaligned (__u16 *) in __xchg\n"
.LC57:
       .string "misaligned (__u32 *) in __xchg\n"
.L$PG1: # Pool sys_sigsuspend
.LC192:
       .long -262401
.LC193:
       .long -1
.LC194:
       .long schedule-.L$PG1
.LC195:
       .long do_signal-.L$PG1
       .align 4
       .globl sys_sigsuspend
       .type sys_sigsuspend,@function
sys_sigsuspend:
#        leaf function 0
#        automatics 16
#        outgoing args 0
#        need frame pointer 0
#        call alloca 0
#        has varargs 0
#        incoming args (stack) 0
#        function length 168
         STM  8,15,32(15)
         LR   0,15
         AHI  15,-112
         BASR 13,0
.L$CO1: AHI   13,.L$PG1-.L$CO1
         ST   0,0(15)
         LR   8,2
         N    5,.LC192-.L$PG1(13)
```

Adding `-g` to the above output makes the output even more useful. For example, typing:

```
make CC:="s390-gcc -g" kernel/sched.s
```

Which compiles:

```
s390-gcc -g -D__KERNEL__ -I/home/barrow/linux-2.3/include -Wall -Wstrict-prototypes -O2 -
fomit-frame-pointer -fno-strict-aliasing -pipe -fno-strength-reduce   -S kernel/sched.c -
o kernel/sched.s
```

This also outputs stabs (debugger) information, from which you can find out the offsets and sizes of various elements in structures. For example, the stab for the structure

```
struct rlimit {
       unsigned long rlim_cur;
       unsigned long rlim_max;
};
```

is

```
.stabs "rlimit:T(151,2)=s8rlim_cur:(0,5),0,32;rlim_max:(0,5),32,32;;",128,0,0,0
```

From this stab you can see that:

- rlimit_cur starts at bit offset 0 and is 32 bits in size
- rlimit_max starts at bit offset 32 and is 32 bits in size.

Debugging Tools

This section examines the tools available to debug your Linux system.

objdump

This is a tool with many options the most useful being (if compiled with `-g`).

```
objdump—source <victim program or object file <victims debug listing
```

The whole kernel can be compiled like this (doing this will make a 17MB kernel and a 200 MB listing) however you have to strip it before building the image using the strip command to make it a more reasonable size to boot it.

A source/assembly mixed dump of the kernel can be done with the line:

```
objdump—source vmlinux vmlinux.lst
```

Also if the file isn't compiled `-g` this will output as much debugging information as it can (for example, function names). However, this is very slow as it spends lots of time searching for debugging info, the following self-explanatory line should be used instead if the code isn't compiled `-g`.

```
objdump—disassemble-all—syms vmlinux vmlinux.lst
```

as it is much faster.

As hard drive space is valuable most of us use the following approach.

1. Look at the emitted PSW on the console to find the crash address in the kernel.
2. Look at the file `System.map` (in the linux directory) produced when building the kernel to find the closest address less than the current PSW to find the offending function.
3. Use `grep` or similar to search the source tree looking for the source file with this function if you don't know where it is.
4. Rebuild this object file with `-g` on, as an example suppose the file was `/arch/s390/kernel/signal.o`
5. Assuming the file with the erroneous function is `signal.c.` Move to the base of the Linux source tree
6. Erase the existing object file: `rm /arch/s390/kernel/signal.o`

7. Build the object file: `make /arch/s390/kernel/signal.o`

8. Watch the `gcc` command line emitted

9. Type it in again or alternatively cut and paste it on the console adding the `-g` option.

10. `objdump—source arch/s390/kernel/signal.o signal.lst`

11. This will output the source and the assembly intermixed, as the snippet below shows. This will unfortunately output addresses which aren't the same as the kernel ones you should be able to get around the mental arithmetic by playing with the—`adjust-vma` parameter to `objdump`.

```
extern inline void spin_lock(spinlock_t *lp)
{
a0: 18 34           lr     %r3,%r4
a2: a7 3a 03 bc     ahi    %r3,956
__asm__ __volatile(" lhi 1,-1\n"
a6: a7 18 ff ff     lhi    %r1,-1
aa: 1f 00           slr    %r0,%r0
ac: ba 01 30 00     cs     %r0,%r1,0(%r3)
b0: a7 44 ff fd     jm     aa <sys_sigsuspend+0x2e
saveset = current-blocked;
b4: d2 07 f0 68     mvc    104(8,%r15),972(%r4)
b8: 43 cc
return (set-sig[0] and mask) != 0;
}
```

- If debugging under VM go down to that section in the document for more info.

I now have a tool that takes the pain out of `--adjust-vma`, and you are able to do something like:

```
make /arch/s390/kernel/traps.lst
```

And it automatically generates the correctly relocated entries for the text segment in `traps.lst`. This tool is now standard in Linux distributions in `scripts/makelst`.

strace

Q. WHAT IS IT?

A. It is a tool for intercepting calls to the kernel and logging them to a file and on the screen.

Q. What use is it?

A. You can use it to find out what files a particular program opens.

EXAMPLE 1

If you wanted to know does ping work but didn't have the source:

```
strace ping -c 1 127.0.0.1
```

Then look at the man pages for each of the syscalls below, (In fact this is sometimes easier than looking at some spaghetti source which conditionally compiles for several architectures). Not everything that it throws out needs to make sense immediately.

Just looking quickly you can see that it is making up a RAW socket for the ICMP protocol. Doing an `alarm(10)` for a 10 second timeout and doing a `gettimeofday()` call before and after each read to see how long the replies took, and writing some text to stdout so the user has an idea what is going on.

```
socket(PF_INET, SOCK_RAW, IPPROTO_ICMP) = 3
getuid() = 0
setuid(0) = 0
stat("/usr/share/locale/C/libc.cat", 0xbffff134) = -1 ENOENT (No such file or directory)
stat("/usr/share/locale/libc/C", 0xbffff134) = -1 ENOENT (No such file or directory)
stat("/usr/local/share/locale/C/libc.cat", 0xbffff134) = -1 ENOENT (No such file or
directory)
getpid() = 353
setsockopt(3, SOL_SOCKET, SO_BROADCAST, [1], 4) = 0
setsockopt(3, SOL_SOCKET, SO_RCVBUF, [49152], 4) = 0
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(3, 1), ...}) = 0
mmap(0, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x40008000
ioctl(1, TCGETS, {B9600 opost isig icanon echo ...}) = 0
write(1, "PING 127.0.0.1 (127.0.0.1): 56 d"..., 42PING 127.0.0.1 (127.0.0.1): 56 data
bytes
) = 42
sigaction(SIGINT, {0x8049ba0, [], SA_RESTART}, {SIG_DFL}) = 0
sigaction(SIGALRM, {0x8049600, [], SA_RESTART}, {SIG_DFL}) = 0
gettimeofday({948904719, 138951}, NULL) = 0
sendto(3, "\10\0D\201a\1\0\0\17#\2178\307\36"..., 64, 0, {sin_family=AF_INET,
sin_port=htons(0), sin_addr=inet_addr("127.0.0.1")}, 16) = 64
sigaction(SIGALRM, {0x8049600, [], SA_RESTART}, {0x8049600, [], SA_RESTART}) = 0
sigaction(SIGALRM, {0x8049ba0, [], SA_RESTART}, {0x8049600, [], SA_RESTART}) = 0
alarm(10) = 0
recvfrom(3, "E\0\0T\0005\0\0@\1|r\177\0\0\1\177"..., 192, 0,
{sin_family=AF_INET, sin_port=htons(50882), sin_addr=inet_addr("127.0.0.1")}, [16]) = 84
gettimeofday({948904719, 160224}, NULL) = 0
recvfrom(3, "E\0\0T\0006\0\0\377\1\275p\177\0"..., 192, 0,
{sin_family=AF_INET, sin_port=htons(50882), sin_addr=inet_addr("127.0.0.1")}, [16]) = 84
gettimeofday({948904719, 166952}, NULL) = 0
write(1, "64 bytes from 127.0.0.1: icmp_se"...,
5764 bytes from 127.0.0.1: icmp_seq=0 ttl=255 time=28.0 ms
```

EXAMPLE 2

```
strace passwd 2>&1 | grep open
```

Produces the following output:

```
open("/etc/ld.so.cache", O_RDONLY) = 3
open("/opt/kde/lib/libc.so.5", O_RDONLY) = -1 ENOENT (No such file or directory)
open("/lib/libc.so.5", O_RDONLY) = 3
open("/dev", O_RDONLY) = 3
open("/var/run/utmp", O_RDONLY) = 3
open("/etc/passwd", O_RDONLY) = 3
open("/etc/shadow", O_RDONLY) = 3
open("/etc/login.defs", O_RDONLY) = 4
open("/dev/tty", O_RDONLY) = 4
```

The `2>&1` is done to redirect stderr to stdout and `grep` is then filtering this input through the pipe for each line containing the string open.

EXAMPLE 3

NOW WE ARE GETTING SOPHISTICATED: TELNETD CRASHES ON AND I DON'T KNOW WHY

1. Replace the following line in `/etc/inetd.conf`: `telnet stream tcp nowait root /usr/sbin/in.telnetd -h` with `telnet stream tcp nowait root /blah`

2. Create the file `/blah` with the following contents to start tracing `telnetd`

3. ```
   #!/bin/bash
   /usr/bin/strace -o/t1 -f /usr/sbin/in.telnetd -h
   ```

4. `chmod 700 /blah` to make it executable only to root

5. `killall -HUP inetd or ps aux | grep inetd`. get inetd's process id and `kill -HUP inetd` to restart it.

IMPORTANT OPTIONS

- `-o` is used to tell `strace` to output to a file: in our case `t1` in the root directory

- `-f` is to follow children: For example, in our case above `telnetd` will start the login process and subsequently a shell like bash. You will be able to tell which is which from the process ID's listed on the left hand side of the `strace` output.

- `-p<pid>` will tell `strace` to attach to a running process, yup this can be done provided it isn't being traced or debugged already and you have enough privileges, the reason 2 processes cannot trace or debug the same program is that `strace` becomes the parent process of the one being debugged and processes (unlike people) can have only one parent.

However the file `/t1` will get big quite quickly to test it `telnet 127.0.0.1`. Now look at what files `in.telnetd execve`'d

```
413 execve("/usr/sbin/in.telnetd", ["/usr/sbin/in.telnetd", "-h"], [/* 17 vars /]) = 0
414 execve("/bin/login", ["/bin/login", "-h", "localhost", "-p"], [/ 2 vars */]) = 0
```

Whey it worked!

OTHER HINTS

If the program is not very interactive (that is, not much keyboard input) and is crashing in one architecture but not in another you can do an `strace` of both programs under as identical a scenario as you can on both architectures outputting to a file then. Do a `diff` of the two traces using the diff program. That is:

```
diff output1 output2
```
```

Now maybe you'll be able to see where the call paths differed, this is possibly near the cause of the crash.

MORE INFORMATION

Look at man pages for `strace` and the various `syscalls`. For example, `man strace`, `man alarm`, and `man socket`.

Performance Debugging

`gcc` is capable of compiling in profiling code just add the `-p` option to the `CFLAGS`, this obviously affects program size and performance. This can be used by the `gprof` gnu profiling tool or the `gcov` the gnu code coverage tool (code coverage is a means of testing code quality by checking if all the code in an executable in exercised by a tester).

Using top to find out where processes are sleeping in the kernel

To do this:

- Copy the `System.map` from the root directory where the Linux kernel was built to the `/boot` directory on your Linux machine.
- Start `top`
- Now type `fU<return>`
- You should see a new field called `WCHAN`, which tells you where each process is sleeping here is a typical output.

```
6:59pm  up 41 min,  1 user,  load average: 0.00, 0.00, 0.00
28 processes: 27 sleeping, 1 running, 0 zombie, 0 stopped
CPU states:  0.0% user,  0.1% system,  0.0% nice, 99.8% idle
Mem:   254900K av,   45976K used,  208924K free,       0K shrd,   28636K buff
Swap:       0K av,       0K used,       0K free                    8620K cached

  PID USER     PRI  NI  SIZE  RSS SHARE WCHAN     STAT  LIB %CPU %MEM   TIME COMMAND
  750 root      12   0   848  848   700 do_select S       0  0.1  0.3   0:00 in.telnetd
  767 root      16   0  1140 1140   964           R       0  0.1  0.4   0:00 top
    1 root       8   0   212  212   180 do_select S       0  0.0  0.0   0:00 init
    2 root       9   0     0    0     0 down_inte SW      0  0.0  0.0   0:00 kmcheck
```

The time command

Another related command is the time command that gives you an indication of where a process is spending the majority of its time: For example,

```
time ping -c 5 nc
outputs
real   0m4.054s
user   0m0.010s
sys    0m0.010s
```

Debugging under VM

Addresses and values in the VM debugger are always hex never decimal. Address ranges are of the format `<HexValue1-<HexValue2` or `<HexValue1.<HexValue2`. For example, the address range 0x2000 to 0x3000 can be described as 2000-3000 or 2000.1000.

The VM Debugger is case insensitive.

VM's strengths are usually other debuggers weaknesses you can get at any resource no matter how sensitive (for example, memory management resources, change address translation in the PSW. For kernel hacking you will reap dividends if you get good at it.

The VM Debugger displays operators but not operands, probably because some of it was written when memory was expensive and the programmer was probably proud that it fitted into 2k of memory and the programmers didn't want to shock hardcore VM'ers by changing the interface :-). Also, the debugger displays useful information on the same line and the author of the code probably felt that it was a good idea not to go over the 80 columns on the screen.

As some of you are probably in a panic now this isn't as unintuitive as it may seem as the 390 instructions are easy to decode mentally and you can make a good guess at a lot of them as all the operands are nibble (half byte aligned). If you have an `objdump` listing also it is quite easy to follow. If you don't have an `objdump` listing keep a copy of the ESA Reference Summary and look at between pages 2 and 7 or alternatively the ESA principles of operation. For example, even I can guess that:

```
0001AFF8' LR 180F CC 0
```

is a (load register) `lr r0,r15`

Also it is very easy to tell the length of a 390 instruction from the 2 most significant bits in the instruction (not that this info is really useful except if you are trying to make sense of a hexdump of code):

Bits	Instruction Length
00	2 Bytes
01	4 Bytes
10	4 Bytes
11	6 Bytes

The debugger also displays other useful info on the same line such as the addresses being operated on destination addresses of branches and condition codes. For example:

```
00019736' AHI        A7DAFF0E                     CC 1
000198BA' BRC        A7840004      -> 000198C2'   CC 0
000198CE' STM        900EF068      0FA95E78       CC 2
```

Useful VM debugger commands

I suppose I'd better mention this before I start to list the current active traces do

```
Q TR
```

There can be a maximum of 255 of these per set (more about trace sets later). To stop traces issue:

```
TR END
```

To delete a particular breakpoint issue

```
TR DEL <breakpoint number>
```

The PA1 key drops to CP mode so you can issue debugger commands. Doing alt-c (on my 3270 console at least) clears the screen. Hitting b <enter> comes back to the running operating system from CP mode (in our case Linux).

It is typically useful to add shortcuts to your PROFILE EXEC file if you have one (this is roughly equivalent to autoexec.bat in DOS or .profile in Linux). Here are a few from mine:

```
/* this gives me command history on issuing f12 */
set pf12 retrieve
/* this continues */
set pf8 imm b
/* goes to trace set a */
set pf1 imm tr goto a
/* goes to trace set b */
set pf2 imm tr goto b
/* goes to trace set c */
set pf3 imm tr goto c
```

INSTRUCTION TRACING

Setting a simple breakpoint:

```
TR I PSWA <address>
```

To debug a particular function try:

- TR I R <function address range>
- TR I on its own will single step.
- TR I DATA <MACHINE-CODE> <OPTIONAL RANGE> will trace for particular mnemonics. For example, TR I DATA 4D R 0197BC.4000 will trace for BAS'es (opcode 4D) in the range 0197BC.4000. If you were inclined you could add traces for all branch instructions and suffix them with the run prefix so you would have a backtrace on screen when a program crashes.

- `TR BR <INTO OR FROM>` will trace branches into or out of an address. For example, `TR BR INTO 0` is often quite useful if a program is getting awkward and deciding to branch to 0 and crashing as this will stop at the address before in jumps to 0.

- `TR I R <address range> RUN cmd d g` single steps a range of addresses but stays running and displays the general-purpose registers on each step.

DISPLAYING AND MODIFYING REGISTERS

- `D G` will display 32-bits of all the gprs. Use `D GG` to display the full 64-bits.

- `D X` will display all the control registers

- `D AR` will display all the access registers

- `D AR4-7` will display access registers 4 to 7

- `CPU ALL D G` will display the GRPS of all CPUS in the configuration

- `D PSW` will display the current PSW

- `ST PSW 2000` will put the value 2000 into the PSW and probably crash your machine.

- `D PREFIX` will display the prefix register

DISPLAYING MEMORY

To display memory that was mapped using the current PSW's mapping try:

```
D <range>
```

To make VM display a message each time it hits a particular address and continue try:

- `D I<range>` will disassemble/display a range of instructions.

- `ST addr <32 bit word>` will store a 32 bit aligned address

- `D T<range>` will display the EBCDIC in an address (if you are that way inclined)

- `D R<range>` will display real addresses, without DAT, but with prefixing.

There are other complex options to display if you need to get at say home space but are in primary space the easiest thing to do is to temporarily modify the PSW to the other addressing mode, display the stuff and then restore it.

If you want to issue a debugger command without halting your virtual machine with the PA1 key, then try prefixing the command with `#CP`:

```
#cp tr i pswa 2000
```

Also suffixing most debugger commands with RUN will cause them not to stop just display the mnemonic at the current instruction on the console. If you have several breakpoints you want to put into your program and you get fed up of cross referencing with `System.map` you can do the following trick for several symbols.:

```
grep do_signal System.map
```

This emits the following among other things:

```
0001f4e0 T do_signal
```

Now you can do:

```
TR I PSWA 0001f4e0 cmd msg * do_signal
```

This sends a message to your console each time `do_signal` is entered. (As an aside I wrote a perl script once that automatically generated a REXX script with breakpoints on every kernel procedure. This isn't a good idea because there are thousands of these routines and VM can only set 255 breakpoints at a time, you nearly had to spend as long pruning the file down as you would enter the messages by hand). However, the trick might be useful for a single object file.

On Linux's 3270 emulator x3270 there is a very useful option under the file menu: Save Screens In File this is very good of keeping a copy of traces.

From CMS help <command name> will give you online help on a particular command: e.g.

```
HELP DISPLAY
```

Also CMS has a file called PROFILE EXEC which automatically gets called on startup of CMS (like autoexec.bat). Keeping on a DOS analogy session, CP has a feature similar to doskey, it may be useful for you to use PROFILE EXEC to define some keystrokes: e.g.

```
SET PF9 IMM B
```

This does a single step in VM on pressing F8.

```
SET PF10  ^
```

This sets up the ^ key, which can be used for `^c` (ctrl-c), `^z` (ctrl-z) which cannot be typed directly into some 3270 consoles.

```
SET PF11 ^-
```

This types the starting keystrokes for a sysrq see SysRq below.

```
SET PF12 RETRIEVE
```

This retrieves command history on pressing F12.

Sometimes in VM the display is set up to scroll automatically this can be very annoying if there are messages you wish to look at. To stop do this:

```
TERM MORE 255 255
```

This will nearly stop automatic screen updates, however it will cause a denial of service if lots of messages go to the 3270 console, so it would be foolish to use this as the default on a production machine.

Tracing particular processes

The kernels text segment is intentionally at an address in memory that it will very seldom collide with text segments of user programs (thanks Martin), this simplifies debugging the kernel. However it is quite common for user processes to have addresses that collide. This can make debugging a particular process under VM painful under normal circumstances as the process may change when doing a

```
TR I R <address range>.
```

Thankfully after reading VM's online help I figured out how to debug I particular process.

Your first problem is to find the STD (segment table designation) of the program you wish to debug. There are several ways you can do this. Here are a few:

1. Locating value of CR13 when main is invoked:

   - `objdump—syms <program to be debugged> | grep main`

     (This will get the address of main in the program)

   - `tr i pswa <address of main>`

   - Start the program, if VM drops to CP on what looks like the entry point of the main function this is most likely the process you wish to debug.

   - Now do a `D X13` (on S/390) or `D XG13` (on z/VM). For 31 bit the STD is bits 1-19 (the STO segment table origin) and 25-31 (the STL segment table length) of CR13.

   - Now type:

     ```
     TR I R STD <CR13's value> 0.7fffffff
     ```

     For example:

     ```
     TR I R STD 8F32E1FF 0.7fffffff
     ```

2. Intercepting the setting of the STD:

   - `TR STORE INTO STD <CR13's value> <address range>`

3. Navigating `/proc`. This method is more complex but could be quite convenient if you aren't updating the kernel much and so your kernel structures will stay constant for a reasonable period of time).

- `grep task /proc/<pid>/status`. From this you should see something like:

  `task: 0f160000 ksp: 0f161de8 pt_regs: 0f161f68`

- This now gives you a pointer to the task structure. Now `make CC:="s390-gcc -g" kernel/sched.s` to get the `task_struct` stabinfo. (`task_struct` is defined in `include/linux/sched.h`).

- Now we want to look at `task->active_mm->pgd`. On my machine the `active_mm` in the task structure stab is:

  `active_mm:(4,12),672,32`

  Its offset is 672/8=84=0x54, the pgd member in the mm_struct stab is pgd:(4,6)=*(29,5),96,32. So its offset is 96/8=12=0xc

- So we'll: `hexdump -s 0xf160054 /dev/mem | more`

  (That is, `task_struct+active_mm` offset) to look at the `active_mm` member:

  `f160054 0fee cc60 0019 e334 0000 0000 0000 0011`

- `hexdump -s 0x0feecc6c /dev/mem | more` (That is, `active_mm+pgd` offset):

  `feecc6c 0f2c 0000 0000 0001 0000 0001 0000 0010`

- Now do:

  `TR I R STD <pgd|0x7f> 0.7fffffff` (that is, the 0x7f is added because the pgd only gives the page table origin and we need to set the low bits to the maximum possible segment table length.)

  `TR I R STD 0f2c007f 0.7fffffff` (S/390) or

  `TR I R STD <pgd|0x7> 0.ffffffffffffffff` (z/Architecture)

TRACING PROGRAM EXCEPTIONS

If you get a crash which says something like illegal operation or specification exception followed by a register dump You can restart Linux and trace these using the `tr prog <range | value>` option.

The most common ones you will normally be tracing for is:

1	Operation exception
2	Privileged operation exception
4	Protection exception

5	Addressing exception
6	Specification exception
10	Segment translation exception
11	Page translation exception

The full list of these is in the ESA Reference Summary. For example:

- `tr prog 10` will trace segment translation exceptions.
- `tr prog` on its own will trace all program interruption codes.

TRACE SETS

On starting VM you are initially in the `INITIAL` trace set. You can do a `Q TR` to verify this. If you have a complex tracing situation where you wish to wait for instance till a driver is open before you start tracing I/O, but know in your heart that you are going to have to make several runs through the code till you have a clue what's going on.

What you can do is:

```
TR I PSWA <Driver open address>
```

Enter `b` to continue until the breakpoint is reached. Now do your:

```
TR GOTO B
TR IO 7c08-7c09 (or whatever and trace tour IO)
```

To got back to the initial trace set do:

```
TR GOTO INITIAL
```

Now the `TR I PSWA <Driver open address>` will be the only active breakpoint again.

TRACING LINUX SYSCALLS UNDER VM

Syscalls are implemented on Linux for S390 by the Supervisor call instruction (SVC). There are 256 possibilities of these as the instruction is made up of a 0x0a operation code and the second byte being the syscall number. They are traced using the simple command:

```
TR SVC <Optional value or range>
```

The syscalls are defined in `linux/include/asm-s390/unistd.h.` For example, to trace all file opens just do:

```
TR SVC 5 (as this is the syscall number of open)
```

- To find out how many CPUs you have use the `Q CPUS` command.
- To find the CPU that the current VM debugger commands are being directed at use `Q CPU`.
- To change the current CPU at which the commands are being directed then issue the `CPU <cpuid>` command.
- To issue a command to all CPUs prefix the command with `CPU ALL`.
- If you are running on a guest with several CPUs and you have an I/O related problem but cannot follow the flow of code, then (if you know the problem is not SMP-related) do the following:
    - Issue `shutdown –h now` to terminate Linux
    - `Q CPUS` to find out how many CPUs you have.
    - Detach all CPUs except CPU 0 by issuing `DETACH CPU 01-`*nn* and boot linux again
    - `DEFINE CPU 01-<`*nn*`>` will make your guest's CPUs available again.
- `TR SIGP` will trace inter-processor signaling instructions.

PRODUCING TRACE OF SYSTEM FLOW

VM's ability to trace branch operations allows the production of system flow data. The output can (and is) quite voluminous but can be made human-friendly by the following process.

1. FTP the `System.map` to the VM user who will process the flow data or allow the EXEC to FTP it for you each time.
2. Create the trace set that will produce a printout of all branch operations. You can start the trace prior to booting Linux or prior to running your daemon or application:

   ```
   #CP TR BR PRINT
   ```

3. Redirect the output of the command to a user who will process the data:

   ```
   #CP SP P <user>
   ```

4. When you are ready to process the data, end the trace and close the print file:

   ```
   #CP TR END
   #CP CLOSE P
   ```

5. Run the following EXEC to process the trace:

   ```
   /* */
   parse upper arg Option .
   signal on SYNTAX
   MaxSym = 0
   ```

```
Cache. = ''
if Option <> '' then
       Stage  = ''
else
Stage  = '| nlocate /memset/',
              '| nlocate /memcpy/',
              '| nlocate /memcmp/',
              '| nlocate /update_wall_time/',
              '| nlocate /printk/',
              '| nlocate /strcpy/',
              '| nlocate /strncpy/',
              '| nlocate /strcmp/',
              '| nlocate /strncmp/',
              '| nlocate /strchr/',
              '| nlocate /strlen/',
              '| nlocate /ExternalException/',
              '| nlocate /External+/',
              '| nlocate /do_timer/',
              '| nlocate /vsprintf/',
              '| nlocate /set_bit/',
              '| nlocate /free_pages/',
              '| nlocate /mem_init/'
'PIPE (name READ_MAP)',
       '| ftp ftp://<user>:<password>@<host.domain>/linux/System.map binary',
       '| xlate from 437 to 1047',
       '| deblock c',
       '| drop 1',
       '| strip',
       '| locate 1',
       '| nfind U' ||,
       '| spec w1 x2c 1 w1-* nw',
       '| stem Map.'
'PIPE (name READ_TRACE end ?)',
       '| reader',
       '| mctoasa',
       '| spec 2-* 1',
       '| a: locate /BASR/',
       '| b: faninany',
       '| spec w2-4 1 w6 nw',
       '| stem Trace.',
       '? a:',
       '| c: locate /LPSW/',
       '| b:',
       '? c:',
       '| locate / 07FE /',
       '| b:'
do I_Trace = 1 to Trace.0
       if ((I_Trace // 5000) = 0) then
              say '...'I_Trace
       parse var Trace.I_Trace From Branch BrType To
       if (Cache.From = '') then
       do
              FromSym = GET_ADDR(From,'F')
              Cache.From = FromSym
       end
       else
              FromSym = Cache.From
       if (Cache.To = '') then
       do
              ToSym   = GET_ADDR(To,'T')
```

```
            Cache.To = ToSym
        end
        else
            ToSym = Cache.To
        select
            when BrType = '0D1E' then
                Branch = '<---'
            when BrType = '0DEF' then
                Branch = '--->'
            when BrType = '07FE' then
                Branch = '<---'
            otherwise
                Branch = '--->'
        end
        Flow.I_Trace = FromSym LEFT(Branch,8) ToSym
end
Flow.0 = Trace.0
'PIPE (name WRITE_FLOW end ?)',
        '| stem Flow.',
        '| spec w1 1.'MaxSym 'w2-* nw',
        Stage,
        '| spec 1-* 5',
        '| > LINUX FLOW A'
exit
GET_ADDR:
        parse arg VAddr,Type
        parse var VAddr Addr"'"
        XAddr = X2C(Addr)
        Target    = Map.0
        NewTarget = 1
        Disp      = Map.0 % 2
        LastLow   = Target
        do forever
            parse var Map.Target 1 XSym 5 . . Symbol
            if XAddr = XSym then
            do
                LastLow = Target
                leave
            end
            if XAddr < XSym then
            do
                NewTarget = Target - Disp
                if (LastLow = NewTarget) then
                    leave
                LastLow   = NewTarget
            end
            else
                NewTarget = Target + Disp
            Disp = Disp % 2
            if (Disp < 1) then
                Disp = 1
            Target = NewTarget
        end
        parse var Map.LastLow 1 XSym 5 . . Symbol
        Disp = X2D(Addr) - C2D(XSym)
        if Disp <> 0 then
            Symbol = Symbol'+'D2X(Disp)
        if ((LENGTH(Symbol) > MaxSym) & (Type = 'F')) then
            MaxSym = LENGTH(Symbol)
return Symbol
```

```
SYNTAX:
       say 'Error:' ERRORTEXT(Rc) 'at line' Sigl
       say SOURCELINE(Sigl)
       trace ?r; nop
exit -1
```

SMP SPECIFIC COMMANDS

- To find out how many CPUs you have:

  `Q CPUS`

- To find the cpu that the current cpu VM debugger commands are being directed at do:

  `Q CPU`

- To change the current CPU VM debugger commands are being directed at do:

  `CPU <desired cpu no>`

- On an SMP guest issue a command to all CPUs try prefixing the command with `CPU ALL`.

- To issue a command to a particular CPU, try `CPU <cpu number>.` For example:

  `CPU 01 TR I R 2000.3000`

- If you are running on a guest with several cpus and you have an I/O related problem and cannot follow the flow of code but you know it isn't SMP related. Then from the bash prompt issue:

  `shutdown -h now or halt.`

  Do a `Q CPUS` to find out how many CPUs you have; detach each one of them from your virtual machine except CPU 0 by issuing:
  `DETACH CPU 01-<number of CPUs in configuration>`

  and re-boot Linux.

- `TR SIGP` will trace inter processor signal processor instructions.

HELP FOR DISPLAYING ASCII TEXT

If you are running z/VM or VM/ESA 2.4.0 with the latest service the DISPLAY and STORE commands have been enhanced to support the display and storing of ASCII text. For example:

`#CP D TX0.100`

This will display 256 bytes of storage in hex and ASCII format.

For other version of VM, text cannot be displayed in ASCII under the VM debugger (I love EBDIC too), I have written this little program which will convert a command line of hex digits to ASCII text which can be compiled under Linux and you can copy the hex digits from your x3270 terminal to your xterm if you are debugging from a Linux box.

This is quite useful when looking at a parameter passed in as a text string under VM (unless you are good at decoding ASCII in your head). For example, consider tracing an open syscall:

```
TR SVC 5
```

We have stopped at a breakpoint

```
000151B0' SVC 0A05 - 0001909A' CC 0
```

Use D P SVC to check the SVC old PSW in the prefix area and see was it from user-space (for the layout of the prefix area consult page18 of the ESA 390 Reference Summary if you have it available).

```
SVC 0005  20 OLD 070C2000 800151B2   60  NEW 04080000 8001909A
```

The problem state bit wasn't set and it's also too early in the boot sequence for it to be a user-space SVC if it was we would have to temporarily switch the PSW to user space addressing so we could get at the first parameter of the open in gpr2. To display the parameter:

```
D 0.20;BASE2
V00014CB4 2F646576 2F636F6E 736F6C65 00001BF5
V00014CC4 FC00014C B4001001 E0001000 B8070707
```

Alternatively you can do the more elegant

```
D 0.20;BASE2
```

BASE2 tells VM to use GPR2 as the base register.

Now copy the text till the first 00 hex (which is the end of the string) to an xterm and do hex2ascii on it:

```
hex2ascii 2F646576 2F636F6E 736F6C65 00
```

The resulting output is:

```
Decoded Hex:=/ d e v / c o n s o l e 0x00
```

We were opening the console device. You can compile the code below yourself for practice :-),

```
/*
* hex2ascii.c
* a useful little tool for converting a hexadecimal command line to ascii
*
* Author(s): Denis Joseph Barrow (djbarrow@de.ibm.com,barrow_dj@yahoo.com)
* © 2000 IBM Deutschland Entwicklung GmbH, IBM Corporation.
*/

#include <stdio.h
int main(int argc,char *argv[])
{
 int cnt1,cnt2,len,toggle=0;
 int startcnt=1;
 unsigned char c,hex;

if(argc1&&(strcmp(argv[1],"-a")==0))
 startcnt=2;
printf("Decoded Hex:=");
for(cnt1=startcnt;cnt1<argc;cnt1++)
{
```

```
 len=strlen(argv[cnt1]);
 for(cnt2=0;cnt2<len;cnt2++)
 {
  c=argv[cnt1][cnt2];
  if(c='0'&&c<='9')
c=c-'0';
if(c='A'&&c<='F')
c=c-'A'+10;
if(c='a'&&c<='F')
c=c-'a'+10;
  switch(toggle)
  {
case 0:
    hex=c<<4;
    toggle=1;
    break;

   case 1:
    hex+=c;
    if(hex<32||hex127)
    {
     if(startcnt==1)
      printf("0x%02X ",(int)hex);
     else
      printf(".");
    }
    else
    {
     printf("%c",hex);
     if(startcnt==1)
      printf(" ");
    }
    toggle=0;
    break;
   }
  }
 }
 printf("\n");
}
```

Alternatively, the following CMS PIPELINE will achieve the same thing:

```
/* */
parse arg XString
'PIPE (name E2A)',
'| var Xstring',
   '| change / //',         /* Remove any blanks within string */
   '| spec 1-* x2c 1',      /* Convert graphic hex to binary   */
   '| xlate from 437 to 1047',    /* Choose the code-page you prefer */
'| cons'
```

Stack tracing under VM

Here are the tricks I use 9 out of 10 times it works pretty well.

This can happen when an exception happens in the kernel and the kernel is entered twice if you reach the NULL pointer at the end of the back chain you should be able to sniff further back if you follow the following tricks.

1. A kernel address should be easy to recognize since it is in primary space and the problem state bit isn't set and also The Hi bit of the address is set.

2. Another backchain should also be easy to recognize since it is an address pointing to another address approximately 100 bytes or 0x70 hex behind the current stackpointer.

Here is some practice.

- Boot the kernel and hit PA1 at some random time

- `d g` to display the gprs, this should display something like

```
GPR 0  = 00000001 00156018 0014359C 00000000
GPR 4  = 00000001 001B8888 000003E0 00000000
GPR 8  = 00100080 00100084 00000000 000FE000
GPR 12 = 00010400 8001B2DC 8001B36A 000FFED8
```

- Note that GPR14 is a return address but as we are real men we are going to trace the stack. Display 0x40 bytes after the stack pointer:

```
D 0.40;BASEF
V000FFED8 000FFF38 8001B838 80014C8E 000FFF38
V000FFEE8 00000000 00000000 000003E0 00000000
V000FFEF8 00100080 00100084 00000000 000FE000
V000FFF08 00010400 8001B2DC 8001B36A 000FFED8
```

- Ah now look at what is in sp+56 (sp+0x38) this is 8001B36A our saved r14 if you look above at our stackframe and also agrees with GPR14. Now backchain:

```
d 000FFF38.40
```

- We now are taking the contents of SP to get our first backchain.

```
V000FFF38 000FFFA0 00000000 00014995 00147094
V000FFF48 00147090 001470A0 000003E0 00000000
V000FFF58 00100080 00100084 00000000 001BF1D0
V000FFF68 00010400 800149BA 80014CA6 000FFF38
```

  This displays a 2nd return address of 80014CA6

- Now do `d 000FFFA0.40` for our 3rd backchain

```
V000FFFA0 04B52002 0001107F 00000000 00000000
V000FFFB0 00000000 00000000 FF000000 0001107F
V000FFFC0 00000000 00000000 00000000 00000000
V000FFFD0 00010400 80010802 8001085A 000FFFA0
```

  Our 3rd return address is 8001085A. As the 04B52002 looks suspiciously like rubbish it is fair to assume that the kernel entry routines for the sake of optimization do not set up a backchain.

- Now look at `System.map` to see if the addresses make any sense:

  `grep -i 0001b3 System.map`

  Outputs among other things:

  `0001b304 T cpu_idle`

  So `8001B36A` is `cpu_idle+0x66` (quiet the CPU is asleep, don't wake it!)

- Next:

  `grep -i 00014 System.map`

  Produces among other things

  `00014a78 T start_kernel`

  So `0014CA6` is `start_kernel+0x22e`

- Then:

  `grep -i 00108 System.map`

  This produces:

  `00010800 T _stext`

  So `8001085A` is `_stext+0x5a`

- Congratulations you've performed your first backchain!

S/390 and z/Architecture I/O Overview

I am not going to give a course in 390 I/O architecture as this would take me quite a while and I'm no expert. Instead I'll give a 390 IO architecture summary for Dummies if you have the ESA principles of operation available read this instead. If nothing else you may find a few useful keywords in here and be able to use them on a web search engine like Altavista to find more useful information.

Unlike other bus architectures modern 390 systems do their I/O using mostly fiber optics and devices such as tapes and disks can be shared between several mainframes, also S390 can support up to 65536 devices while a high end PC based system might be choking with around 64. Here is some of the common I/O terminology.

Subchannel	This is the logical number most I/O commands use to talk to an I/O device there can be up to 0x10000 (65536) of these in a configuration typically there is a few hundred. Under VM for simplicity they are allocated contiguously, however on the native hardware they are not they typically stay consistent between boots provided no new hardware is inserted or removed. Under Linux for 390 we use these as IRQ's and also when issuing an I/O command (CLEAR SUBCHANNEL, HALT SUBCHANNEL, MODIFY SUBCHANNEL, RESUME SUBCHANNEL, START SUBCHANNEL, STORE SUBCHANNEL and TEST SUBCHANNEL) we use this as the ID of the device we wish to talk to. The most important of these instructions are START

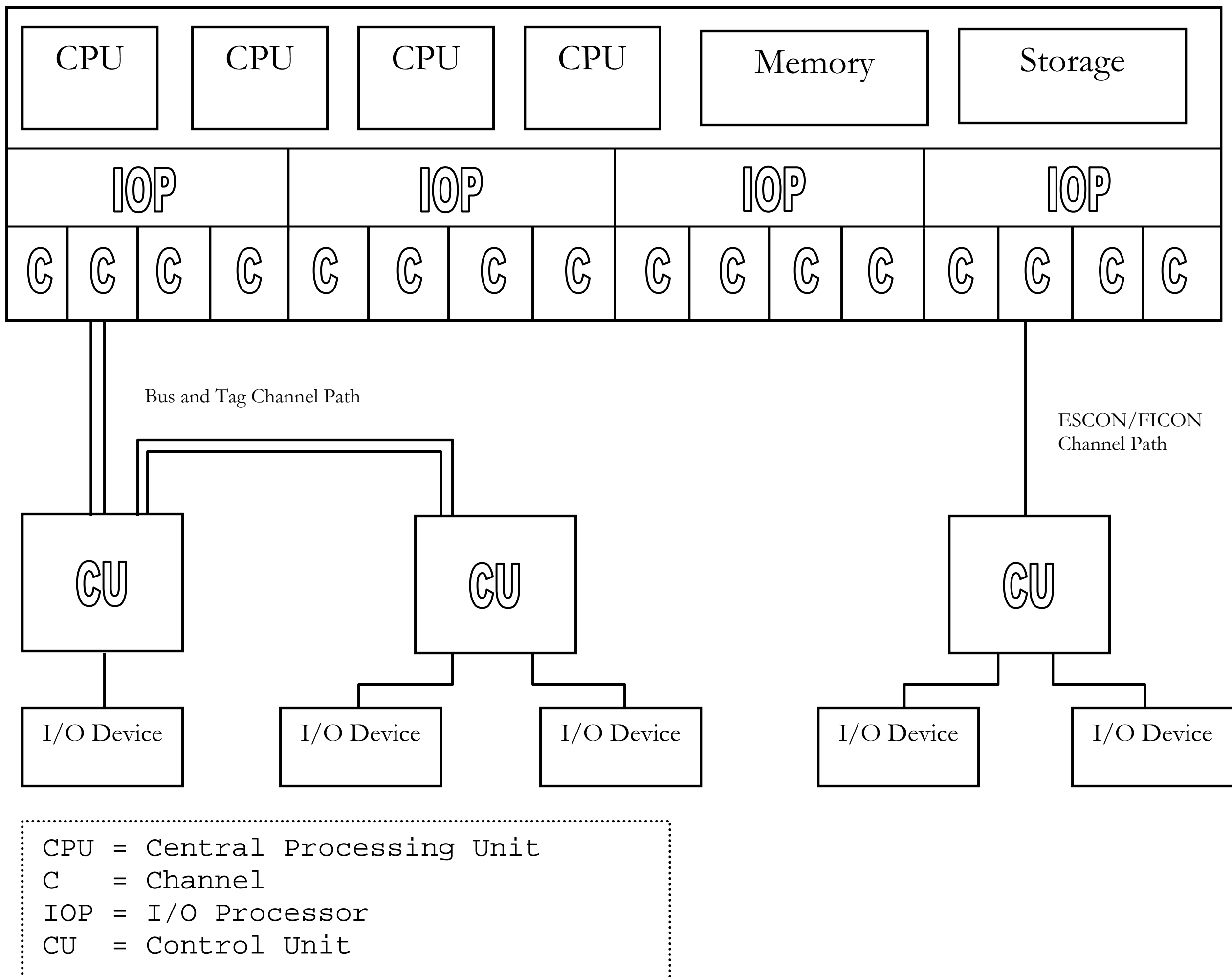
	SUBCHANNEL (to start IO), TEST SUBCHANNEL (to check whether the IO completed successfully), and HALT SUBCHANNEL (to kill IO). A subchannel can have up to 8 channel paths to a device this offers redundancy if one is not available.
Device Number	This number remains static and Is closely tied to the hardware, there are 65536 of these also they are made up of a CHPID (Channel Path ID, the most significant 8 bits) and another LSB 8 bits. These remain static even if more devices are inserted or removed from the hardware, there is a 1 to 1 mapping between Subchannels and Device Numbers provided devices are not inserted or removed.
Channel Control Words	CCWS are linked lists of instructions initially pointed to by an operation request block (ORB), which is initially given to Start Subchannel (SSCH) command along with the subchannel number for the I/O subsystem to process while the CPU continues executing normal code. These come in two flavors, Format 0 (24 bit for backward) compatibility and Format 1 (31 bit). These are typically used to issue read and write (and many other instructions) they consist of a length field and an absolute address field. For each IO typically get 1 or 2 interrupts one for channel end (primary status) when the channel is idle and the second for device end (secondary status). Sometimes you get both concurrently; you check how the I/O went on by issuing a TEST SUBCHANNEL at each interrupt, from which you receive an Interruption response block (IRB). If you get channel and device end status in the IRB without channel checks etc. your I/O probably went okay. If you did not, you probably need to examine the IRB and extended status word etc. If an error occurs, then more sophisticated control units have a facility known as concurrent sense. This means that if an error occurs Extended sense information will be presented in the Extended status word in the IRB if not you have to issue a subsequent SENSE CCW command after the test subchannel.
TPI	Test pending interrupt can also be used for polled IO but in multitasking multiprocessor systems it isn't recommended except for checking special cases (i.e. non-looping checks for pending I/O etc.).
STSCH and MSCH	Store Subchannel and Modify Subchannel can be used to examine and modify operating characteristics of a subchannel (e.g. channel paths).
Sysplex	S390's Clustering Technology
QDIO	S390's new high speed IO architecture to support devices such as gigabit Ethernet, this architecture is also designed to be forward compatible with up and coming 64 bit machines.

General Concepts

Input Output Processors (IOP's) are responsible for communicating between the mainframe CPU's and the channel and relieve the mainframe CPU's from the burden of communicating with IO devices directly, this allows the CPU's to concentrate on data processing.

IOP's can use one or more links (known as channel paths) to talk to each IO device. It first checks for path availability and chooses an available one, then starts (and sometimes terminates IO). There are two types of channel path ESCON and the Parallel IO interface.

I/O devices are attached to control units. Control units provide the logic to interface the channel paths and channel path I/O protocols to the I/O devices. They can be integrated with the devices or housed separately and often talk to several similar devices (typical examples would be RAID controllers or a control unit which connects to 1000 3270 terminals).



The 390 I/O systems come in 2 flavors the current 390 machines support both the older 360 and 370 interface, sometimes called the parallel I/O interface, sometimes called Bus-and Tag and sometimes Original Equipment Manufacturers Interface (OEMI).

This byte wide parallel channel path/bus has parity and data on the "Bus" cable and control lines on the "Tag" cable. These can operate in byte multiplex mode for sharing between several slow devices or burst mode and monopolize the channel for the whole burst. Up to 256 devices can be addressed on one of these cables. These cables are about one inch in diameter. The maximum un-extended length supported by these cables is 125 Meters but this can be extended up to 2km with a fiber optic channel extended such as a 3044. The maximum burst speed supported is 4.5 megabytes per second however some really old processors support only transfer rates of 3.0, 2.0 and 1.0 MB/sec. One of these paths can be daisy chained to up to 8 control units.

IBM introduced ESCON, which is fiber optic based, in 1990. It uses 2 fiber optic cables and uses either LEDs or lasers for communication at a signaling rate of up to 200 megabits/sec. As 10 bits are transferred for every 8 bits of information this drops to 160 megabits/sec and to 18.6 Megabytes/sec once control information and CRC are added. ESCON only operates in burst mode.

ESCONs typical maximum cable length is 3km for the LED version and 20km for the laser version known as XDF (extended distance facility). Using an ESCON director, which triples the above-mentioned ranges, can further extend this. Unlike Bus and Tag as ESCON is serial. It uses packet switching architecture. The standard Bus and Tag control protocol is however present within the packets. Up to 256 devices can be attached to each control unit that uses one of these interfaces.

A new fiber architecture has been released by IBM called FICON which improves on the performance of ESCON.

Common 390 Devices

- Network adapters typically OSA2, 3172's, 2116's and OSA-E gigabit Ethernet adapters
- Consoles 3270 and 3215 (a TTY emulated under Linux for a line mode console)
- DASD's direct access storage devices (otherwise known as hard disks)
- Tape Drives.
- CTC (Channel to Channel Adapters), ESCON or Parallel Cables used as a very high-speed serial link between 2 machines. We use 2 cables under Linux to do a bi-directional serial link.

Debugging IO on S390 under VM

Now we are ready to go on with I/O tracing commands under VM. First, a few self-explanatory queries:

```
Q OSA
Q CTC
Q DASD
```

If I have sufficient privileges Q OSA may return the state of all OSAs on the processor. Using Q V OSA will return the status of those on my machine:

```
OSA 7C08 ON OSA 7C08 SUBCHANNEL = 0000
OSA 7C09 ON OSA 7C09 SUBCHANNEL = 0001
```

```
OSA 7C14 ON OSA 7C14 SUBCHANNEL = 0002
OSA 7C15 ON OSA 7C15 SUBCHANNEL = 0003
```

Now using the device numbers returned by this command we will trace the Io starting up on the first devices 7c08 and 7c09. In our simplest case we can trace the start subchannels:

```
TR SSCH 7C08-7C09
```

Or the halt subchannels

```
TR HSCH 7C08-7C09
```

You can also trace MSCH's, STSCH's, but I think you can guess the rest.

Ingo's favorite trick is tracing all the I/O's and CCWS and spooling them into the reader of another VM guest so he can ftp the logfile back to his own machine. I'll do a small bit of this and give you a look at the output.

1. Spool stdout to VM reader

   ```
   SP PRT TO [<another user> | *]
   ```

2. Fill reader with the trace

   ```
   TR IO 7c08-7c09 INST INT CCW PRT RUN
   ```

3. Start up Linux

4. Finish the trace

   ```
   TR END
   ```

5. Close the reader

   ```
   C PRT
   ```

6. List reader contents

   ```
   RDRLIST
   ```

7. Copy it to minidisk

   ```
   RECEIVE / LOG TXT A1 ( replace
   ```

8. FILELIST and press F11 to look at it. You should see something like:

   ```
   00020942' SSCH B2334000 0048813C CC 0 SCH 0000 DEV 7C08
   CPA 000FFDF0 PARM 00E2C9C4 KEY 0 FPI C0 LPM 80
   CCW 000FFDF0 E4200100 00487FE8 0000 E4240100 ........
   IDAL 43D8AFE8
   IDAL 0FB76000
   00020B0A' I/O DEV 7C08 - 000197BC' SCH 0000 PARM 00E2C9C4
   00021628' TSCH B2354000 00488164 CC 0 SCH 0000 DEV 7C08
   CCWA 000FFDF8 DEV STS 0C SCH STS 00 CNT 00EC
   KEY 0 FPI C0 CC 0 CTLS 4007
   00022238' STSCH B2344000 00488108 CC 0 SCH 0000 DEV 7C08
   ```

Other Common VM Device Related Commands

These commands are listed only because they have been of use to me in the past and may be of use to you too. For more complete information on each of the commands use `HELP <command>` from CMS.

Command	Description
`DET <devno range> [<guest>]`	Detach devices from a guest
`ATT <defno range> [<guest>]`	Attach devices to a guest
`READY <devno>`	Fake a device-end interrupt from a device
`VARY ON PATH <path> TO <devno range>`	Vary on the path to devices (VM administrator command)
`VARY OFF PATH <path> FROM <devno range>`	Vary the path offline to devices (VM Administrator command)
`Q CHPID <path>`	Display state us of devices using this channel path
`D SCHIB <subchannel>`	Display the subchannel information block for the virtual device
`DEFINE CTC <devno>`	Define a virtual Channel-to-Channel connection (a pair of devices is required for use by Linux)
`COUPLE <devno> <userid> <remote devno>`	Join a virtual device to a virtual device owned by `<userid>`
`DEF VFB-<blocksize> <subchannel> <blocks>`	Define a virtual disk (VM ramdisk)
`LINK <userid> <devno1> <devno2> <mode> <password>`	Share a disk between multiple guests

gdb on S390

Note, compiling for debugging with `gdb` works better without optimization (see “Compiling programs for debugging on Linux for S390 and z/Architecture” on page 18).

Invocation

```
gdb <victim program <optional corefile>
```

Online help

help: gives help on commands. For example:

```
help
help display
```

Note gdb’s online help is very good and we advise you to use it.

Assembly

- `info registers`: displays registers other than floating point.

- `info all-registers`: displays floating points as well.
- `disassemble`: disassembles. For example,

```
disassemble (specifying no parameters will disassemble the current function)
disassemble $pc $pc+10
```

Viewing and modifying variables

- `print` or `p`: displays variable or register. For example:

  `p/x $sp` will display the stack pointer
- `display`: prints variable or register each time program stops. For example:

  `display/x $pc` will display the program counter
  `display argc`
- `undisplay`: undo's display's
- `info breakpoints`: shows all current breakpoints
- `info stack`: shows stack back trace (if this doesn't work too well, I'll show you the stacktrace by hand below).
- `info locals`: displays local variables.
- `info args`: display current procedure arguments.
- `set args`: will set argc and argv each time the victim program is invoked.

```
set <variable=value>
set argc=100
set $pc=0
```

Modifying execution

- `step`: steps *n* lines of source code `step` with no value steps 1 line.
- `next`: like step except this will not step into subroutines
- `stepi`: steps a single machine code instruction.
- `nexti`: steps a single machine code instruction but will not step into subroutines.
- `finish`: will run until exit of the current routine
- `run`: (re)starts a program
- `cont`: continues a program
- `quit`: exits gdb.

breakpoints

- `break`: sets a breakpoint. For example:

```
break main
break *$pc
break *0x400618
```

- `rbr`: Set a breakpoint for all functions matching REGEXP. This is really useful for large programs. For example:

```
rbr 390
```

  Will set a breakpoint with all functions with 390 in their name.

- `info breakpoints`: lists all breakpoints

- `delete`: delete breakpoint by number or delete them all. For example:

  `delete 1` will delete the first breakpoint
  `delete` will delete them all

- `watch`: This will set a watchpoint (usually hardware assisted). This will watch a variable till it changes. For example:

  `watch cnt,` will watch the variable cnt till it changes. As an aside unfortunately gdb's, architecture independent watchpoint code is inconsistent and not very good. Watchpoints usually work but not always.

- `info watchpoints`: Display currently active watchpoints

- `condition`: Specify breakpoint number N to break only if COND is true. Usage is `condition N COND`, where `N` is an integer and `COND` is an expression to be evaluated whenever breakpoint N is reached. This is a particularly another useful command.

User defined functions/macros

- `define`: Define a macro/function. This is very useful, simple and powerful. Usage: `define <name> <list of commands> end`. Examples which you should consider putting into .gdbinit in your home directory:

```
define d
stepi
disassemble $pc $pc+10
end

define e
nexti
disassemble $pc $pc+10
end
```

Other hard to classify stuff

- `signal` *n*: sends the victim program a signal. For example, `signal 3` will send a `SIGQUIT`.

- `info signals`: what gdb does when the victim receives certain signals.

- `list`: Examples:

`list`	lists current function source
`list 1,10`	list first 10 lines of current file
`list test.c:1,10`	

- directory: Adds directories to be searched for source if gdb cannot find the source. Note it is a bit sensitive about slashes. For example, to add the root of the filesystem to the searchpath do:

```
directory //
```

- `call <function>`: This calls a function in the victim program, this is pretty powerful. For example:

```
(gdb) call printf("hello world")
```

Outputs:

```
$1 = 11
```

You might now be thinking that the line above didn't work, something extra had to be done.

```
(gdb) call fflush(stdout)
hello world$2 = 0
```

As an aside the debugger also calls `malloc` and `free` under the hood to make space for the `"hello world"` string.

Hints

1. Command completion works just like `bash` (if you are a bad typist like me this really helps). For example, type `br <TAB>`

2. If you have a debugging problem that takes a few steps to recreate put the steps into a file called `.gdbinit` in your current working directory if you have defined a few extra useful user defined commands put these in your home directory and they will be read each time `gdb` is launched. A typical .gdbinit file might be:

```
break main
run
break runtime_exception
cont
```

Stack chaining in gdb by hand

- This is done using the same trick described for VM.

  - 31-bit

    ```
    p/x (*($sp+56))&0x7fffffff get the first backchain.
    ```

  - 64-bit

    ```
    p/x (*(long *)(***long ***)$sp+112))get the first backchain.
    ```

  This outputs:

  ```
  $5 = 0x528f18
  ```

(On my 31-bit machine that is.)

- Now you can use:

  ```
  info symbol (*($sp+56))&0x7fffffff
  ```

  You might see something like:

  `rl_getc + 36 in section .text` telling you what is located at address 0x528f18

- Now do:

  ```
  p/x (*(*$sp+56))&0x7fffffff
  ```

  This outputs:

  ```
  $6 = 0x528ed0
  ```

- Now do:

  ```
  info symbol (*(*$sp+56))&0x7fffffff
  rl_read_key + 180 in section .text
  ```

- Now do:

  ```
  p/x (*(**$sp+56))&0x7fffffff
  ```

  and so on. Another good trick to look at addresses on the stack if you've somehow lost the backchain is.

  ```
  x/500xa $sp
  ```

  This displays anything the name of any known functions above the stack pointer for 500 bytes.

Disassembling instructions without debug information

`gdb` typically complains if there is a lack of debugging symbols in the disassemble command with "No function contains specified address." To get around this do:

x/<*number of lines to disassemble*>xi <*address*>

For example:

```
x/20xi 0x400730
```

For more information

From your Linux box do:

`man gdb` or `info gdb`

Examining Core Dumps

A core dump is a file generated by the kernel (if allowed) which contains the registers, and all active pages of the program which has crashed. From this file gdb will allow you to look at the registers and stack trace and memory of the program as if it just crashed on your system. It is usually called core and created in the current working directory. This is very useful in that a customer can mail a core dump to a technical support department and the technical support department can reconstruct what happened. Provided the have an identical copy of this program with debugging symbols compiled in and the source base of this build is available.

In short it is far more useful than something like a crash log could ever hope to be.

In theory all that is missing to restart a core dumped program is a kernel patch that will do the following.

1. Make a new kernel task structure
2. Reload all the dumped pages back into the kernels memory management structures.
3. Do the required clock fixups
4. Get all files and network connections for the process back into an identical state (really difficult).
5. A few more difficult things I have not thought of.

WHY HAVE I NEVER SEEN ONE?

Probably because you haven't used the command to allow core dumps:

```
ulimit -c unlimited
```

Now do the following to verify that the limit was accepted:

```
ulimit -a
```

A SAMPLE CORE DUMP

- To create this I'm going to do:

  ```
  ulimit -c unlimited
  gdb
  ```

  To launch `gdb` (my victim application).

- Now be bad and do the following from another telnet/xterm session to the same machine:

  ```
  ps -aux | grep gdb
  kill -SIGSEGV <gdb's pid >
  ```

  Alternatively, if you have the `killall` command, use:

  ```
  killall -SIGSEGV gdb
  ```

- Now look at the core dump.

  ```
  ./gdb ./gdb core
  ```

  The following will be displayed:

  ```
  GNU gdb 4.18
  Copyright 1998 Free Software Foundation, Inc.
  GDB is free software, covered by the GNU General Public License, and you are
  welcome to change it and/or distribute copies of it under certain conditions.
  Type "show copying" to see the conditions.
  There is absolutely no warranty for GDB. Type "show warranty" for details.
  This GDB was configured as "s390-ibm-linux"...
  Core was generated by './gdb'.
  ```

```
Program terminated with signal 11, Segmentation fault.
Reading symbols from /usr/lib/libncurses.so.4...done.
Reading symbols from /lib/libm.so.6...done.
Reading symbols from /lib/libc.so.6...done.
Reading symbols from /lib/ld-linux.so.2...done.
#0 0x40126d1a in read () from /lib/libc.so.6
Setting up the environment for debugging gdb.
Breakpoint 1 at 0x4dc6f8: file utils.c, line 471.
Breakpoint 2 at 0x4d87a4: file top.c, line 2609.
(top-gdb) info stack
#0 0x40126d1a in read () from /lib/libc.so.6
#1 0x528f26 in rl_getc (stream=0x7ffffde8) at input.c:402
#2 0x528ed0 in rl_read_key () at input.c:381
#3 0x5167e6 in readline_internal_char () at readline.c:454
#4 0x5168ee in readline_internal_charloop () at readline.c:507
#5 0x51692c in readline_internal () at readline.c:521
#6 0x5164fe in readline (prompt=0x7ffff810 \177~Aÿ~Aøx\177~Aÿ~A÷~AØ\177~Aÿ~Aøx~AÀ")
at readline.c:349
#7 0x4d7a8a in command_line_input (prompt=0x564420 "(gdb) ", repeat=1,
annotation_suffix=0x4d6b44 "prompt") at top.c:2091
#8 0x4d6cf0 in command_loop () at top.c:1345
#9 0x4e25bc in main (argc=1, argv=0x7ffffdf4) at main.c:635
```

ldd

This is a program that lists the shared libraries that a library needs. Note you also get the relocations of the shared library text segments that help when using `objdump --source`. For example:

```
ldd ./gdb
```

Outputs:

```
libncurses.so.4 = /usr/lib/libncurses.so.4 (0x40018000)
libm.so.6 = /lib/libm.so.6 (0x4005e000)
libc.so.6 = /lib/libc.so.6 (0x40084000)
/lib/ld-linux.so.2 = /lib/ld-linux.so.2 (0x40000000)
```

Debugging shared libraries

Most programs use shared libraries, however it can be very painful when you single step instruction into a function like `printf` for the first time and you end up in functions like `_dl_runtime_resolve`. This is the `ld.so` doing lazy binding; lazy binding is a concept in ELF where shared library functions are not loaded into memory unless they are actually used. This is great for saving memory but a pain to debug.

To get around this either re-link the program `-static` or exit `gdb`; type `export LD_BIND_NOW=true` (this will stop lazy binding), and restart the `gdb`'ing the program in question.

Debugging modules

As modules are dynamically loaded into the kernel their address can be anywhere to get around this use the -m option with `insmod` to emit a load-map that can be piped into a file if required.

The proc file system

This is a file system created by the kernel with files which are created on demand by the kernel if read, or can be used to modify kernel parameters. It is a powerful concept. For example:

```
cat /proc/sys/net/ipv4/ip_forward
```

On my machine this outputs:

```
0
```

This tells me that `ip_forwarding` is not on. To switch it on I can do:

```
echo 1 /proc/sys/net/ipv4/ip_forward
cat it again:
cat /proc/sys/net/ipv4/ip_forward
1
```

That is, IP forwarding is now on.

There is a lot of useful info in here best found by going in and having a look around, so I'll take you through some entries I consider important.

- All the processes running on the machine have there own entry defined by `/proc/<pid>`.

  - Let us have a look at the `init` process:

    ```
    cd /proc/1
    cat cmdline
    init [2]
    ```

  - Now look at the file descriptor entries:

    ```
    cd /proc/1/fd
    ```

  This contains numerical entries of all the open files.

  - The storage map for the process may be examined:

    ```
    cat /proc/29/maps
    00400000-00478000 r-xp 00000000 5f:00 4103 /bin/bash
    00478000-0047e000 rw-p 00077000 5f:00 4103 /bin/bash
    0047e000-00492000 rwxp 00000000 00:00 0
    40000000-40015000 r-xp 00000000 5f:00 14382 /lib/ld-2.1.2.so
    40015000-40016000 rw-p 00014000 5f:00 14382 /lib/ld-2.1.2.so
    40016000-40017000 rwxp 00000000 00:00 0
    40017000-40018000 rw-p 00000000 00:00 0
    40018000-4001b000 r-xp 00000000 5f:00 14435 /lib/libtermcap.so.2.0.8
    4001b000-4001c000 rw-p 00002000 5f:00 14435 /lib/libtermcap.so.2.0.8
    4001c000-4010d000 r-xp 00000000 5f:00 14387 /lib/libc-2.1.2.so
    4010d000-40111000 rw-p 000f0000 5f:00 14387 /lib/libc-2.1.2.so
    40111000-40114000 rw-p 00000000 00:00 0
    40114000-4011e000 r-xp 00000000 5f:00 14408 /lib/libnss_files-2.1.2.so
    4011e000-4011f000 rw-p 00009000 5f:00 14408 /lib/libnss_files-2.1.2.so
    7fffd000-80000000 rwxp ffffe000 00:00 0
    ```

Showing us the shared libraries init uses where they are in memory and memory access permissions for each virtual memory area.

- `/proc/1/cwd` is a soft link to the current working directory.
- `/proc/1/root` is the root of the file system for this process.
- `/proc/1/mem` is the current running process' memory, which you can read and write to like a file. `strace` uses this sometimes as it is a bit faster than the rather inefficient `ptrace` interface for peeking at DATA.
- Use `cat /proc/1/status` to display the status of the process:

```
Name: init
State: S (sleeping)
Pid: 1
PPid: 0
Uid: 0 0 0 0
Gid: 0 0 0 0
Groups:
VmSize: 408 kB
VmLck: 0 kB
VmRSS: 208 kB
VmData: 24 kB
VmStk: 8 kB
VmExe: 368 kB
VmLib: 0 kB
SigPnd: 0000000000000000
SigBlk: 0000000000000000
SigIgn: 7fffffffd7f0d8fc
SigCgt: 00000000280b2603
CapInh: 00000000fffffeff
CapPrm: 00000000ffffffff
CapEff: 00000000fffffeff
User PSW: 070de000 80414146
task: 004b6000 tss: 004b62d8 ksp: 004b7ca8 pt_regs: 004b7f68
User GPRS:
00000400 00000000 0000000b 7ffffa90
00000000 00000000 00000000 0045d9f4
0045cafc 7ffffa90 7fffff18 0045cb08
00010400 804039e8 80403af8 7ffff8b0
User ACRS:
00000000 00000000 00000000 00000000
00000001 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
Kernel BackChain CallChain BackChain CallChain
004b7ca8 8002bd0c 004b7d18 8002b92c
004b7db8 8005cd50 004b7e38 8005d12a
004b7f08 80019114
```

Showing among other things memory usage and status of some signals and the processes' registers from the kernel `task_structure` as well as a backchain that may be useful if a process crashes in the kernel for some unknown reason.

DEBUG FEATURE

Some of the drivers now support a "debug feature" in `/proc/s390dbf`. See "S/390 Debugging Facility" on page 57 for more information. For example, to switch on the lcs "debug feature":

```
echo 5 >/proc/s390dbf/lcs/level
```

After an error occurs issue:

```
cat /proc/s390dbf/lcs/sprintf >logfile
```

The file "`logfile`" now contains some information that may help technical support resolve a problem in the field.

HIGH LEVEL DEBUGGING NETWORK DRIVERS

The `ifconfig` command is a quite useful. It gives the current state of network drivers. If you suspect your network device driver is dead one way to check is type `ifconfig <network device>`. For example,

```
ifconfig tr0
tr0       Link encap:16/4 Mbps Token Ring (New)  HWaddr 00:04:AC:20:8E:48
          inet addr:9.164.185.132  Bcast:9.164.191.255  Mask:255.255.224.0
          UP BROADCAST RUNNING MULTICAST  MTU:2000  Metric:1
          RX packets:246134 errors:0 dropped:0 overruns:0 frame:0
          TX packets:5 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
```

If the device doesn't say "`UP`" then try:

```
/etc/rc.d/init.d/network start
```

This starts the network stack and (hopefully) invokes "`ifconfig tr0 up`". `ifconfig` looks at the output of `/proc/net/dev` and presents it in a more presentable form. Now ping the device from a machine in the same subnet. If the RX packets count and TX packets counts don't increment you probably have problems. Next,

```
cat /proc/net/arp
```

Do you see any hardware addresses in the cache? If not you may have problems.

Next try:

`ping -c 5 <broadcast_addr>` [That is, the Bcast field above in the output of `ifconfig`.]

Do you see any replies from machines other than the local machine? If not you may have problems. Also if the TX packets count in `ifconfig` hasn't incremented either you have serious problems in your driver (for example, the txbusy field of the network device being stuck on) or you may have multiple network devices connected.

CHANDEV

There is a new device layer for channel devices; some drivers (for example LCS) are registered with this layer. If the device uses the channel device layer you'll be able to find what interrupts it uses and the current state of the device. See the manpage `chandev.8` and type `cat /proc/chandev` for more info.

DEBUGGING DRIVERS

Some of the drivers now support a debug logging feature in `/proc/s390dbf`. See "ProcFS Interface" on page 63 for more information. For example, to switch on LCS debugging:

```
echo 5 > /proc/s390dbf/lcs/level
```

Then after the error occurred.

```
cat /proc/s390dbf/lcs/sprintf >/logfile
```

`logfile` now contains some information that may help tech support resolve a problem in the field.

If you have VM look also look at the chapter Debugging IO on S390 under VM.

Miscellaneous Techniques

The next sections describe various other debugging techniques.

Starting points for debugging scripting languages etc.

There are several different techniques depending on the environment.

BASH/SH

Use the -x option to trace the script: `bash -x <scriptname>.` For example:

```
bash -x /usr/bin/bashbug
+ MACHINE=i586
+ OS=linux-gnu
+ CC=gcc
+ CFLAGS= -DPROGRAM='bash' -DHOSTTYPE='i586' -DOSTYPE='linux-gnu' -DMACHTYPE='i586-pc-
linux-gnu' -DSHELL -DHAVE_CONFIG_H -I. -I. -I./lib -O2 -pipe
+ RELEASE=2.01
+ PATCHLEVEL=1
+ RELSTATUS=release
+ MACHTYPE=i586-pc-linux-gnu
```

PERL

Use the `-d` option of `perl` to invoke the interactive debugger: `perl -d <scriptname>`

JAVA

Use: `jdb <filename>` to invoke another fully interactive `gdb` style debugger. Type "`?`" for help when the debugger is invoked.

SysRq

Linux now supports this feature for s/390 and z/Architecture. To enable it do: "compile the kernel with"

```
Kernel Hacking -> Magic SysRq Key Enabled
echo "1" > /proc/sys/kernel/sysrq.
```

On 390 all commands are prefixed with "`^-`". For example,

- `^-t` will show tasks.
- `^-?` or some unknown command will display help.

The `sysrq` key reading is very picky (I have to type the keys in an xterm session and paste them into the x3270 console) and it may be wise to predefine the keys as described in the VM hints above This is particularly useful for syncing disks unmounting and rebooting if the machine gets partially hung. Read `Documentation/sysrq.txt` for more information.

References

- Enterprise Systems Architecture Reference Summary
- Enterprise Systems Architecture Principles of Operation
- Hartmut Penner's 390 stack frame sheet.
- IBM Mainframe Channel Attachment a technology brief from a CISCO webpage
- Various bits of man and info pages of Linux.
- Linux and GDB source.
- Various info and man pages.
- CMS Help on tracing commands.
- Linux for s/390 Elf Application Binary Interface
- Linux for z/Series Elf Application Binary Interface (Both Highly Recommended)
- z/Architecture Principles of Operation SA22-7832-00
- Enterprise Systems Architecture/390 Reference Summary SA22-7209-01
- Enterprise Systems Architecture/390 Principles of Operation SA22-7201-0

S/390 Debugging Facility

by
Denis Joseph Barrow (djbarrow@de.ibm.com, barrow_dj@yahoo.com)
Copyright © 2000 IBM Deutschland Entwicklung GmbH, IBM Corporation

The goal of this feature is to provide a kernel debug logging API where log records can be stored efficiently in memory, where each component (for example, device drivers) can have one separate debug log.

One purpose of this is to inspect the debug logs after a production system crash in order to analyze the reason for the crash.

If the system still runs but only a subcomponent which uses dbf fails, it is possible to look at the debug logs on a live system via the Linux `/proc` file system.

The debug feature may also very useful for kernel and driver development.

Design

Kernel components (for example, device drivers) can register themselves at the debug feature with the function call `debug_register()`. This function initializes a debug log for the caller. For each debug log exists a number of debug areas where exactly one is active at one time. Each debug area consists of contiguous pages in memory. In the debug areas there are stored debug entries (log records) that are written by event- and exception-calls.

An event-call writes the specified debug entry to the active debug area and updates the log pointer for the active area. If the end of the active debug area is reached, a wrap around is done (ring buffer) and the next debug entry will be written at the beginning of the active debug area.

An exception-call writes the specified debug entry to the log and switches to the next debug area. This is done in order to be sure that the records that describe the origin of the exception are not overwritten when a wrap around for the current area occurs.

The debug areas itself is also ordered in form of a ring buffer. When an exception is thrown in the last debug area, the following debug entries are then written again in the very first area.

There are three versions for the event- and exception-calls: One for logging raw data, one for text and one for numbers.

Each debug entry contains the following data:

- Timestamp
- CPU-Number of calling task
- Level of debug entry (0...6)
- Return Address to caller
- Flag, if entry is an exception or not

The debug logs can be inspected in a live system through entries in the `/proc` file system. Under the path `/proc/s390dbf` there is a directory for each registered component, which is named like the corresponding component.

The content of the directories are files which represent different views to the debug log. Each component can decide which views should be used through registering them with the function debug_register_view(). Predefined views for hex/ascii, sprintf and raw binary data are provided. It is also possible to define other views. Simply by reading the corresponding proc file you can inspect the content of a view.

All debug logs have an actual debug level (range from 0 to 6). The default level is 3. Event and Exception functions have a "level" parameter. Only debug entries with a level that is lower or equal than the actual level are written to the log. This means that high priority log entries should have a low level value whereas low priority entries should have a high one. The actual debug level can be changed with the help of the `/proc` file system through writing a number string "x" to the '`level`' proc file which is provided for every debug log. Debugging can be switched off completely by using "-" on the '`level`' proc file.

Example

```
> echo "-" > /proc/s390dbf/dasd/level
```

Kernel Interfaces

debug_register

Allocates memory for a debug log. Must not be called within an interrupt handler

```
debug_info_t *debug_register(char *name, int pages_index, int nr_areas, int buf_size);
```

`name`	Name of debug log (e.g. used for proc entry)
`pages_index`	2^pages_index pages will be allocated per area
`nr_areas`	Number of debug areas
`buf_size`	Size of data area in each debug entry

Return Value: Handle for generated debug area, NULL if register failed

debug_unregister

Frees memory for a debug log. Must not be called within an interrupt handler

```
void debug_unregister (debug_info_t * id);
```

`id`	Handle for debug log

Return Value: none

debug_set_level

Sets new actual debug level if new_level is valid.

```
void debug_set_level (debug_info_t * id, int new_level);
```

`id`	Handle for debug log
`new_level`	New debug level

Return Value: none

debug_event

Writes debug entry to active debug area (if level <= actual entry debug level).

```
debug_entry_t* debug_event (debug_info_t* id, int level, void* data, int length);
```

`id`	Handle for debug log
`level`	Debug level
`data`	Pointer to data for debug entry
`length`	Length of data in bytes

Return Value: Address of written debug entry

debug_int_event

Writes debug entry to active debug area (if level <= actual debug level).

```
debug_entry_t* debug_int_event (debug_info_t * id, int level, unsigned int data);
debug_entry_t* debug_long_event(debug_info_t * id, int level, unsigned long data);
```

`id`	Handle for debug log
`level`	Debug level
`data`	Integer/Long value for debug entry

Return Value: Address of written debug entry

debug_text_event

Writes debug entry in ASCII format to active debug area (if level <= actual debug level).

```
debug_entry_t* debug_text_event (debug_info_t * id, int level, const char* data);
```

`id`	Handle for debug log

level	Debug level
data	String for debug entry

Return Value: Address of written debug entry

debug_sprintf_event

Writes debug entry with format string and varargs (longs) to active debug area (if level $<=$ actual debug level). floats and long long datatypes cannot be used as varargs.

```
debug_entry_t* debug_sprintf_event (debug_info_t * id, int level, char* string,...);
```

id	Handle for debug log
Level	Debug level
String	Format string for debug entry
...	`varargs` used as `in sprintf()`

Return Value: Address of written debug entry

debug_exception

Writes debug entry to active debug area (if level <= actual debug level) and switches to next debug area.

```
debug_entry_t* debug_exception (debug_info_t* id, int level, void* data,  int length);
```

id	Handle for debug log
level	Debug level
data	Pointer to data for debug entry
length	Length of data in bytes

Return Value: Address of written debug entry

debug_int/long_exception

Writes debug entry to active debug area (if level <= actual debug level) and switches to next debug area.

```
debug_entry_t* debug_int_exception (debug_info_t * id, int level, unsigned int data);
debug_entry_t* debug_long_exception(debug_info_t * id, int level, unsigned long data);
```

id	Handle for debug log
level	Debug level
data	Integer/Long value for debug entry

Return Value: Address of written debug entry

debug_text_exception

Writes debug entry in ascii format to active debug area (level <= actual debug level) and switches to next debug area.

```
debug_entry_t* debug_text_exception (debug_info_t * id, int level, const char* data);
```

id	Handle for debug log
level	Debug level
data	String for debug entry

Return Value: Address of written debug entry

debug_sprintf_exception

Writes debug entry with format string and varargs (longs) to active debug area (if level $<=$ actual debug level) and switches to next debug area. floats and long long datatypes cannot be used as varargs.

```
debug_entry_t* debug_sprintf_exception (debug_info_t * id, int level, char* string,...);
```

id	Handle for debug log
Level	Debug level
String	Format string for debug entry
...	`varargs` used as in `sprintf()`

Return Value: Address of written debug entry

debug_register_view

Registers new debug view and creates proc dir entry

```
int debug_register_view (debug_info_t * id, struct debug_view *view);
```

id	Handle for debug log
view	Pointer to `debug_view` structure

Return Value: 0: ok; < 0: Error

debug_unregister_view

Unregisters debug view and removes proc dir entry

```
int debug_unregister_view (debug_info_t * id, struct debug_view *view);
```

id	Handle for debug log
view	Pointer to `debug_view` structure

Return Value: 0: ok; < 0: Error

Predefined views

```
extern struct debug_view debug_hex_ascii_view;
extern struct debug_view debug_raw_view;
extern struct debug_view debug_sprintf_view;
```

Examples

hex_ascii + raw-view

```
/* hex_ascii- + raw-view Example */

#include <linux/module.h>
#include <asm/debug.h>

static debug_info_t* debug_info;

int init_module(void)
{
    /* register 4 debug areas with one page each and 4 byte data field */

    debug_info = debug_register ("test", 0, 4, 4 );
    debug_register_view(debug_info,&debug_hex_ascii_view);
    debug_register_view(debug_info,&debug_raw_view);

    debug_text_event(debug_info, 4 , "one ");
    debug_int_exception(debug_info, 4, 4711);
    debug_event(debug_info, 3, &debug_info, 4);

    return 0;
}

void cleanup_module(void)
{
    debug_unregister (debug_info);
}
```

sprintf-view

```
/* sprintf-view Example */

#include <linux/module.h>
#include <asm/debug.h>

static debug_info_t* debug_info;

int init_module(void)
{
    /* register 4 debug areas with one page each and data field for */
    /* format string pointer + 2 varargs (= 3 * sizeof(long))       */

    debug_info = debug_register ("test", 0, 4, sizeof(long) * 3);
    debug_register_view(debug_info,&debug_sprintf_view);

    debug_sprintf_event(debug_info, 2 , "first event in %s:%i\n",__FILE__,__LINE__);
    debug_sprintf_exception(debug_info, 1, "pointer to debug info: %p\n",&debug_info);

    return 0;
}

void cleanup_module(void)
{
    debug_unregister (debug_info);
}
```

ProcFS Interface

Views to the debug logs can be investigated through reading the corresponding proc-files:

Example – Viewing the Debug Log

```
> ls /proc/s390dbf/dasd
hex_ascii  level      raw
> cat /proc/s390dbf/dasd/hex_ascii | sort +1
00 00974733272:680099 2 - 02 0006ad7e  07 ea 4a 90 | ....
00 00974733272:682210 2 - 02 0006ade6  46 52 45 45 | FREE
00 00974733272:682213 2 - 02 0006adf6  07 ea 4a 90 | ....
00 00974733272:682281 1 * 02 0006ab08  41 4c 4c 43 | EXCP
01 00974733272:682284 2 - 02 0006ab16  45 43 4b 44 | ECKD
01 00974733272:682287 2 - 02 0006ab28  00 00 00 04 | ....
01 00974733272:682289 2 - 02 0006ab3e  00 00 00 20 | ...
01 00974733272:682297 2 - 02 0006ad7e  07 ea 4a 90 | ....
01 00974733272:684384 2 - 00 0006ade6  46 52 45 45 | FREE
01 00974733272:684388 2 - 00 0006adf6  07 ea 4a 90 | ....
```

See section about predefined views for explanation of the above output!

Example - Changing the debug level

```
> cat /proc/s390dbf/dasd/level
3
> echo "5" > /proc/s390dbf/dasd/level
> cat /proc/s390dbf/dasd/level
5
```

Flushing Debug Areas

Debug areas can be flushed by piping the number of the desired area (0…n) to the `/proc` file "`flush`". When using "-" all debug areas are flushed.

Examples:

1. Flush debug area 0:

   ```
   echo "0" > /proc/s390dbf/dasd/flush
   ```

2. Flush all debug areas:

   ```
   echo "-" > /proc/s390dbf/dasd/flush
   ```

lcrash Interface

It is planned that the dump analysis tool `lcrash` gets an additional command '`s390dbf`' to display all the debug logs. With this tool it will be possible to investigate the debug logs on a live system and with a memory dump after a system crash.

Investigating raw memory

One last possibility to investigate the debug logs at a live system and after a system crash is to look at the raw memory under VM or at the Service Element. It is possible to find the anchor of the debug-logs through the '`debug_area_first`' symbol in the `System.map`. Then one has to follow the correct pointers of the data-structures defined in `debug.h` and find the debug-areas in memory. Normally modules which use the debug feature will also have a global variable with the pointer to the debug-logs. Following this pointer it will also be possible to find the debug logs in memory.

For this method it is recommended to use '`16 * x + 4' byte (x = 0..n)`' for the length of the data field in `debug_register()` in order to see the debug entries well formatted.

Predefined Views

There are three predefined views: hex_ascii, raw and sprintf. The hex_ascii view shows the data field in hex and ascii representation (e.g. '`45 43 4b 44 | ECKD`'). The raw view returns a byte stream as the debug areas are stored in memory.

The sprintf view formats the debug entries in the same way as the `sprintf` function would do. The sprintf event/exception functions write to the debug entry a pointer to the format string (`size = sizeof(long)`) and for each `vararg` a long value. So, for example, for a debug entry with a format string plus two `varargs` one would need to allocate a (`3 * sizeof(long)`) byte data area in the `debug_register()` function.

NOTE: If using the sprintf view do NOT use other event/exception functions than the sprintf-event and -exception functions.

The format of the hex_ascii and sprintf view is as follows:

- Number of area
- Timestamp (formatted as seconds and microseconds since 00:00:00 Coordinated Universal Time (UTC), January 1, 1970)
- Level of debug entry
- Exception flag (* = Exception)
- CPU-Number of calling task
- Return Address to caller
- Data field

The format of the raw view is:

- Header as described in `debug.h`
- Datafield

A typical line of the hex_ascii view will look like the following (first line is only for explanation and will not be displayed when '`cat`ing' the view):

```
area  time           level exception cpu caller    data (hex + ascii)
--------------------------------------------------------------------
00    00964419409:440690 1 -         00 88023fe
```

Defining views

Views are specified with the '`debug_view`' structure. There are defined callback functions that are used for reading and writing the proc files:

```
struct debug_view {
        char name[DEBUG_MAX_PROCF_LEN];
        debug_prolog_proc_t* prolog_proc;
        debug_header_proc_t* header_proc;
        debug_format_proc_t* format_proc;
        debug_input_proc_t*  input_proc;
        void*                private_data;
};
```

Where:

```
typedef int (debug_header_proc_t) (debug_info_t* id,
                                   struct debug_view* view,
                                   int area,
                                   debug_entry_t* entry,
                                   char* out_buf);
typedef int (debug_format_proc_t) (debug_info_t* id,
                                   struct debug_view* view, char* out_buf,
                                   const char* in_buf);
typedef int (debug_prolog_proc_t) (debug_info_t* id,
                                   struct debug_view* view,
                                   char* out_buf);
typedef int (debug_input_proc_t) (debug_info_t* id,
                                  struct debug_view* view,
                                  struct file* file, const char* user_buf,
                                  size_t in_buf_size, loff_t* offset);
```

The "private_data" member can be used as pointer to view specific data. The debug feature itself does not use it.

The output when reading a debug-proc file is structured like this:

```
"prolog_proc output"
"header_proc output 1"  "format_proc output 1"
"header_proc output 2"  "format_proc output 2"
"header_proc output 3"  "format_proc output 3"
...
```

When a view is read from the proc file system, the Debug Feature calls the 'prolog_proc' once for writing the prolog. Then 'header_proc' and 'format_proc' are called for each existing debug entry.

The input_proc can be used to implement functionality when it is written to the view (for example, like with echo "0" > /proc/s390dbf/dasd/level).

For header_proc there can be used the default function debug_dflt_header_fn() which is defined in debug.h and which produces the same header output as the predefined views. For example,

```
00 00964419409:440761 2 - 00 88023ec
```

In order to see how to use the callback functions check the implementation of the default views!

Example

```
#include <asm/debug.h>

#define UNKNOWNSTR "data: %08x"

const char* messages[] =
{"This error...........\n",
 "That error...........\n",
 "Problem..............\n",
 "Something went wrong.\n",
 "Everything ok........\n",
 NULL
};

static int debug_test_format_fn(
   debug_info_t * id, struct debug_view *view,
   char *out_buf, const char *in_buf
)
{
  int i, rc = 0;

  if(id->buf_size >= 4) {
     int msg_nr = *((int*)in_buf);
     if(msg_nr < sizeof(messages)/sizeof(char*) - 1)
        rc += sprintf(out_buf, "%s", messages[msg_nr]);
     else
        rc += sprintf(out_buf, UNKNOWNSTR, msg_nr);
  }
 out:
   return rc;
}

struct debug_view debug_test_view = {
  "myview",                 /* name of view */
```

```
  NULL,                       /* no prolog */
  &debug_dflt_header_fn,      /* default header for each entry */
  &debug_test_format_fn,      /* our own format function */
  NULL,                       /* no input function */
  NULL                        /* no private data */
};
```

RESULTS

```
debug_info_t *debug_info;
...
debug_info = debug_register ("test", 0, 4, 4 ));
debug_register_view(debug_info, &debug_test_view);
for(i = 0; i < 10; i ++) debug_int_event(debug_info, 1, i);

> cat /proc/s390dbf/test/myview
00 00964419734:611402 1 - 00 88042ca   This error...........
00 00964419734:611405 1 - 00 88042ca   That error...........
00 00964419734:611408 1 - 00 88042ca   Problem..............
00 00964419734:611411 1 - 00 88042ca   Something went wrong.
00 00964419734:611414 1 - 00 88042ca   Everything ok........
00 00964419734:611417 1 - 00 88042ca   data: 00000005
00 00964419734:611419 1 - 00 88042ca   data: 00000006
00 00964419734:611422 1 - 00 88042ca   data: 00000007
00 00964419734:611425 1 - 00 88042ca   data: 00000008
00 00964419734:611428 1 - 00 88042ca   data: 00000009
```

Common I/O Layer

This section has been copied from the Documentation/s390 subdirectory of the kernel source tree.

Command line parameters

- `cio_msg = yes | no`

  Determines whether information on found devices and sensed device characteristics should be shown during startup, that is, messages of the types "`Detected device 4711 on subchannel 42`" and "`SenseID: device 4711 reports:...`".

  Default is off.

- `cio_notoper_msg = yes | no`

  Determines whether messages of the type "`Device 4711 became 'not operational'`" should be shown during startup; after startup, they will always be shown.

  Default is on.

- `cio_ignore = <range of device numbers>, <range of device numbers>, ...`

  The given device numbers will be ignored by the common I/O-layer; no detection and device sensing will be done on any of those devices. The subchannel to which the device in question is attached will be treated as if no device was attached.

  An ignored device can be un-ignored later; see "/proc entries" on page 69 for details.

  The device numbers must be given hexadecimal. For example:

  ```
  cio_ignore=0x23-0x42,0x4711
  ```

  This will ignore all devices with device numbers ranging from 23 to 42 and the device with device number 4711, if detected.

  By default, no devices are ignored.

- `cio_proc_devinfo = yes | no`

  Determines whether the entries under `/proc/deviceinfo/` (see next section) should be created. Since there are problems with systems with many devices attached, it was made configurable.

  Until the problems are dealt with, default is off.

/proc entries

The following sections described various interesting entries.

/proc/subchannels

Shows for each subchannel:

- Device number
- Subchannel number
- Device type/model (if applicable; if not, this is empty) and control unit type/model
- Whether the device is in use (that is, a device driver has requested ownership and registered an interrupt handler)
- Path installed mask (PIM), as reflected by last store subchannel (STSCH)
- Path available mask (PAM), as reflected by last store subchannel (STSCH)
- Path operational mask (POM) , as reflected by last store subchannel (STSCH)
- The channel path IDs (chpids)

All fields are separated by spaces. The chpids are in blocks of four.

/proc/deviceinfo/

Shows in subdirectories for each device some characteristics:

- `/proc/deviceinfo/<devno>/chpids`: the channel path IDs
- `/proc/deviceinfo/<devno>/in_use`: whether the device is in use
- `/proc/deviceinfo/<devno>/sensedata`: the device type/model and if applicable control unit type/model of the device

NOTE: Since the number of inodes, which can be dynamically allocated by procfs, is limited: device entries will only be created up to a magic number of devices. The kernel will utter a warning that not all entries can be created. In this case, you shouldn't use "`cio_proc_devinfo=yes`" (see previous section).

/proc/cio_ignore

Lists the ranges of device numbers that are ignored by common I/O.

You can un-ignore certain or all devices by piping to `/proc/cio_ignore`. "`free all`" will un-ignore all gnored devices, "`free <devnorange>, <devnorange>, ...`" will un-ignore the specified devices.

For example, if devices 23 to 42 and 4711 are ignored,

- `echo free 0x30-0x32 > /proc/cio_ignore` will un-ignore devices 30 to 32 and will leave devices 23 to 2F, 33 to 42 and 4711 ignored;
- `echo free 0x41 > /proc/cio_ignore` will furthermore un-ignore device 41;

- `echo free all > /proc/cio_ignore` will un-ignore all remaining ignored devices.

When a device is un-ignored, device recognition and sensing is performed and the device driver will be notified if possible, so the device will become available to the system.

You can also add ranges of devices to be ignored by piping to `/proc/cio_ignore`; "`add <devnorange>, <devnorange>, ...`" will ignore the specified devices.

Note: Already known devices cannot be ignored; this also applies to devices that are gone after a machine check.

For example, if device `abcd` is already known and all other devices `a000-afff` are not known, "`echo add 0xa000-0xaccc, 0xaf00-0xafff > /proc/cio_ignore`" will add `af00-afff` to the list of ignored devices and skip `a000-accc`.

/proc/s390dbf/cio_*/ (S/390 debug feature)

Some views generated by the debug feature to hold various debug outputs.

- `/proc/s390dbf/cio_crw/sprintf`: Messages from the processing of pending channel report words (machine check handling), which will also show when `CONFIG_DEBUG_CRW` is defined.
- `/proc/s390dbf/cio_msg/sprintf`: Various debug messages from the common I/O-layer; generally, messages which will also show when `CONFIG_DEBUG_IO` is defined.
- `/proc/s390dbf/cio_trace/hex_ascii`: Logs the calling of functions in the common I/O-layer and, if applicable, which subchannel they were called for.

The level of logging can be changed to be more or less verbose by piping to `/proc/s390dbf/cio_*/level` a number between 0 and 6. See "Debugging IO on S390 under VM" on page 43 for more information.

/proc/irq_count

This entry counts how many times `s390_process_IRQ` has been called for each CPU. This info is in `/proc/interrupts` on other architectures.

/proc/chpids

This entry will only show up if you specified `CONFIG_CHSC=y` during kernel configuration.

This entry serves a dual purpose:

1. Show which chpids are currently known to Linux and their status (online, logically offline)
2. Toggling known chpids logically online and offline

To toggle a known chpid logically offline:

```
echo off <chpid> > /proc/chpids
```

<chpid> is interpreted as hex, even if you omit the "0x". The chpid will be treated by Linux as if it were not online, which can mean some devices will become unavailable.

You can toggle a logically offline chpid online again by:

```
echo on <chpid> > /proc/chpids
```

Of devices had become unavailable by toggling the chpid logically offline, they will become available again after you toggle the chpid online again.

Channel Device Layer

by
Denis Joseph Barrow (djbarrow@de.ibm.com, barrow_dj@yahoo.com)
Copyright © 2000 IBM Deutschland Entwicklung GmbH, IBM Corporation

The channel device layer is a layer to provide a consistent interface for configuration and default machine check (devices appearing and disappearing) handling Linux for zSeries channel devices.

These include among others:

- LCS - The most common Ethernet/token ring/fddi standard on zSeries)
- CTC/ESCON - High speed like serial link standard on zSeries.
- CLAW - Used to talk to CISCO routers.
- QETH - Gigabit Ethernet.
- OSAD – Used by OSA/SF to configure OSA devices, for example, to share an OSA card between two or more VM guests. OSAD is just added to the channel device layer for completeness. There are no plans, at the current time, to write a driver to exploit this under Linux.

These devices use two channels one read and one write for configuration and or communication. The motivation behind producing this layer was that there is a lot of duplicate code among the drivers for configuration so the LCS and CTC drivers tended to fight over 3088/08's and 3088/1F's which could be either 2216/3172 LCS compatible devices or ESCONs/CTC's and to resolve this fight both device drivers had to be reconfigured rather than doing the configuration in a single place (the channel device layer).

This layer is not invasive and it is quite okay to use channel drivers that do not use the channel device layer in conjunction with drivers that do.

The current setup can be read from `/proc/chandev`. Arguments can be entered by:

- Piping to `/proc/chandev`. For example, `echo reprobe >/proc/chandev` will cause unitialized channel devices to be probed.
- Entering them into `/etc/chandev.conf` comments are prefixed #.

- Or from the boot command line using the ‘`chandev=`’ keyword For example,

  `chandev=noauto,0x0,0x480d;noauto,0x4810,0xffff`

  This will allow only device numbers 0x480e and 0x480f to bee autodetected.

Multiple options can be passed separated by semicolons, but no spaces are allowed between parameters on the kernel parameter line as it complicates parsing. Spaces are allowed in `/proc/chandev` and `chandev.conf`. New-line characters are only allowed in `chandev.conf`.

To be consistent with other hot-pluggable architectures, the script pointed to `/proc/sys/kernel/hotplug` (normally `/sbin/hotplug`) will be called automatically on startup or a machine check of a device, as follows:

```
/sbin/hotplug chandev <start starting_devnames> <machine_check (devname
last/pre_recovery_status) (current/post_recovery_status>
```

The chandev layer doe not open `stdin`, `stdout`, or `stderr`, so it is advisable that you add the following lines to the start of your script. Here is a sample script that starts devices as they become available:

The chandev layer does not open `stdin`, `stdout`, or `stderr` so it is advisable that you add the following lines to the start of your script.

```
#!/bin/bash
exec >/dev/console 2>&1 0>&1
# Uncomment line below for debugging
# echo $*
if [“$1” = “chandev”] && [“$2” = “start”]
then
   shift 2
   while [“$1” != “”] && [ “$1” != “machine_check”]
      isup=’ifconfig $1 2>/dev/null | grep UP’
      if [“$isup” = “”]
      then
         ifup $1
      fi
      shift
   done
fi
```

For example, if (nearly simultaneously) tr0 and ctc0 were starting up, eth0 and eth1 devices disappears, and eth2 got a revalidate machine check (which is normally fully recoverable), the parameters would be:

```
/sbin/hotpug chandev start tr0 ctc0 machine_check eth0 gone gone eth gone gone eth2
revalidate good
```

This can be used, for example, to call `/etc/rc.d/init.d/network` start when a device appears and make the `ipldelay` kernel boot parameter obsolete on native machines or recover from bad machine checks where the default machine check handling isn't adequate. The machine checks that can be presented as parameters are: `good`, `not_operational`, `no_path`, `revalidate`, and `device_gone`. Normally you would not want to do anything like stop networking when a device disappears, as this is (hopefully) temporary. I just added it to be complete. The chandev layer waits a few seconds for machine checks to settle before running `/sbin/hotplug` because several machine checks usually happen at once and the forked scripts would possibly race against each other to shutdown and start resources at the same time.

Chandev Arguments

Valid chandev arguments are (<> indicate optional parameters, |indicates a choice):

Glossary

`devno`: is a 16 bit unsigned number used to uniquely identify a subchannel to a device.

`force list`: is a term specific to channel device layer describing a range of devno's to be forced to configure in a particular manner as opposed to autodetect

Commonly Used Options

`(ctc | escon | lcs | osad | qeth) <devif_num>, read_devno,write_devno, <data_devno, memory_usage_in_k, port_no/protocol_no, check-sum_received_ip_pkts, use_hw_stats>`

`devif_num` of -1 indicates you don't care what device interface number is chosen, omitting it indicates this is a range of devices for which you want to force to be detected as a particular type, qeth devices can't be forced as a range as it makes no sense for them. The `data_devno` field is only valid for qeth devices, all parameters including and after `memory_usage_in_k` can be set optionally, if not set they go to default values. `memory_usage_in_k` (0 the default) means let the driver choose, `checksum_received_ip_pkts` and `use_hw_stats` are set to false. For example,

```
ctc0,0x7c00,0x7c01
```

Tells the channel layer to force ctc0 if detected to use cuu's 7c00 and 7c01 port, `port_no` is the relative adapter number on lcs, on ctc/escon this field is the ctc/escon protocol number (default 0), do not do checksumming on received ip packets as ctc doesn't have hardware statistics so it ignores this parameter. This can be used for instance to force a device if it presents bad sense data to the IO layer and autodetection will fail.

In the following example:

```
lcs,0x7c00,0x7d00,4096,-1
```

All devices between 0x7c00 and0x7d00 should be detected as lcs; let the driver use 4096k for each instance, don't care what port relative adapter number is chosen; do not checksum received ip packets; and use hardware statistics.

For the next example,

```
qeth1,0x7c00,0x7c01,0x7c02
```

Interface 1 will use 0x7c00 as the read device, 0x7c01 for write and 0x7c02 for control; do not checksum received ip packets; and use hardware statistics.

`claw devif_num, read_devno, write_devno <,memory_usage_in_k, checksum_received_ip_pkts, use_hw_stats,> host_name, adapater_name, api_type`

Currently, CLAW is not autodetected as the `host_name`, `adapter_name` and `api_type` need to be set up. Perhaps some convention for automatically setting these may be contrived in the future and auto-detection may be done. The names `host_name`, `adapter_name`, and `api_type` may be up to 8 characters in length. `host_name` is the name of this host, `adapter_name` is the name of the adjacent host, and `api_type` is typically set to "API" or "TCPIP" .

A typical setup may be

```
claw0,0xe00,0xe01,linuxa,rs6k,TCPIP
```

add_parms ,chan_type,<lo_devno,hi_devno,>string

For this option chan_type is bitfield that has one of the following values:

- ctc=0x1,
- escon=0x2,
- lcs=0x4,
- osad=0x8,
- qeth=0x10,
- claw=0x20.

The string parameter is for device driver specific options passed as a string to the driver not dealt with by the channel device layer. It cannot contain spaces.

The low_devno and hi_devno parameters are optional used to specify a range. The channel device layer does not concatenate strings if device ranges overlap, before passing to a device driver.

del_parms <,chan_type,exact_match,lo_devno>

This option is used to delete some or all device driver specific options. Not specifying chan_type causes it to delete all the strings. Specify exact_match=1 to remove driver parameters where chan_type is exactly equal. Specify exact_match=0 to remove parameters where any bit matches chan_type. lo_devno is an optional parameter the delete to only happen if lo_devno matches a lo_devno in one of the ranges.

noauto <,lo_devno,hi_devno>

Do not probe a range of device numbers for channel devices.

use_devno_names

Tells the channel layer to assign device names based on the read channel cuu number. For example, a token ring read channel 0x7c00 would have an interface called tr0x7c00 this avoids name collisions on devices.

Power User Options

del_noauto ,<devno>

Delete a range or all noauto ranges when devno is within a range.

del_force ,read_devno

Delete a forced channel device from force list.

dont_use_devno_names

Opposite to use_devno_names described above.

add_model ,chan_type, cu_type, cu_model, dev_type, dev_model, max_port_no, automatic_machine_check_handling

Tells the channel layer to probe for the device described. The value -1 for any of the parameters other than `chan_type` and `automatic_machine_check_handling` is a wildcard. Set `max_port_no` to 0 for non lcs devices. The parameter `auto_machine_check_handling` is a bit-field that may take the following values:

- not_operational=0x1,
- no_path=0x2,
- revalidate=0x4,
- gone=0x8

The `chan_type` parameter is a bit-field that may take the following values:

- ctc=0x1,
- escon=0x2,
- lcs=0x4,
- osad=0x8,
- qeth=0x10,
- claw=0x20

`del_model ,cu_type,cu_model,dev_type,dev_model`

Note, -1 for any parameter is a wildcard.

`del_all_models`

Delete all modesl.

`non_cautious_auto_detect`

Tells the channel device layer to attempt to auto detect devices even if their type/model pairs don't unambigously identify the device. For example, 3088/1F's can either be escon CTC's or channel attached 3172 lcs compatible devices. If the wrong device driver attempts to probe these channels there may be big delays on startup or even a kernel lockup, use this option with caution.

`cautious_auto_detect`

See `non_cautious_auto_detect` this is the default.

`auto_msck <,lo_devno>,<hi_devno>,auto_msck_recovery`

This is used to specify the kind of machine check recovery that occurs over a device range.

`del_auto_msck <,devno>`

Delete a range or all machine check recovery ranges when `devno` is within a range.

`reset_clean`

Resets all model info, forced devices and noauto lists to null.

`reset_conf`

Resets all model info, forced devices and noauto lists back to default settings.

`reset_conf_clean`

Resets all model info, forced devices and noauto lists to empty.

`shutdown <device name|read devno>`

Shuts down a particular device by device name or read device number, deregisters it, and releases its interrupts or shuts down all devices if no parameter is used.

`reprobe`

Calls probe method for channels whose interrupts are not owned.

`unregister_probe <probefunc_addr>`

Unregisters a single probe function or all of them.

`unregister_probe_by_chan_type`

Unregisters all probe functions which match the `chan_type` bitfield exactly, useful if you want a configuration to service a kernel upgrade.

`read_conf`

Read instructions from `/etc/chandev.conf`.

`dont_read_conf`

Do not automatically read `/etc/chandev.conf` on boot.

`persist ,chan_type`

Force drivers modules to stay loaded even if no device is found, this is useful for debugging and one wishes to examine debug entries in `/proc/s390dbf/` to find out why a module failed to load. For example,

- `persist,-1` forces all devices to persist.
- `persist,0` forces all channel devices to be non persistent.

The following sequence of commands should be roughly equivalent to rebooting for channel devices:

```
shutdown
reset_conf
read_conf
reprobe
```

Common Device Support

The following section was copied from the `Documentation/390` directory of the Linux distribution. It was written by Aldo Lung and is copyright IBM 1999-2002, under the GNU Public License.

This chapter describes the common device support routines for Linux/390. Different than other hardware architectures, ESA/390 has defined a unified I/O access method. This gives relief to the device drivers as they don't have to deal with different bus types, polling versus interrupt processing, shared versus non-shared interrupt processing, DMA versus port I/O (PIO), and other hardware features more. However, this implies that either every single device driver needs to implement the hardware I/O attachment functionality itself, or the operating system provides for a unified method to access the hardware, providing all the functionality that every single device driver would have to provide itself.

The document does not intend to explain the ESA/390 hardware architecture in every detail. This information can be obtained from the ESA/390 Principles of Operation manual (IBM Form. No. SA22-7201).

In order to build common device support for ESA/390 I/O interfaces, a functional layer was introduced that provides generic I/O access methods to the hardware. The following figure shows the usage of the common device support of Linux/390 using a TCP/IP driven device access an example. Similar figures could be drawn for other access methods (for example, file system access to disk devices).

The common device support layer shown above comprises the I/O support routines defined below. Some of them implement common Linux device driver interfaces, while some of them are ESA/390 platform specific.

Function	Description
get_dev_info_by_IRQ()/ get_dev_info_by_devno()	Allow a device driver to determine the devices attached (visible) to the system and their current status.
get_IRQ_by_devno()/ get_devno_by_IRQ()	Get IRQ (subchannel) from device number and vice versa.
read_dev_chars()	Read device characteristics
request_IRQ()	Obtain ownership for a specific device.
free_IRQ()	Release ownership for a specific device.
disable_IRQ()	Disable a device from presenting interrupts.
enable_IRQ()	Enable a device, allowing for I/O interrupts.

do_IO()	Initiate an I/O request.
resume_IO()	Resume channel program execution.
halt_IO()	Terminate the current I/O request processed on the device.
do_IRQ()	Generic interrupt routine. This function is called by the interrupt entry routine whenever an I/O interrupt is presented to the system. The do_IRQ() routine determines the interrupt status and calls the device specific interrupt handler according to the rules (flags) defined during I/O request initiation with do_IO().

The next sections describe the functions, other than `do_IRQ()` in more details. The `do_IRQ()` interface is not described, as it is called from the Linux/390 first level interrupt handler only and does not comprise a device driver callable interface. Instead, the functional description of `do_IO()` also describes the input to the device specific interrupt handler.

Note: All explanations also apply to the 64-bit architecture (s390x).

General Information

The following sections describe the I/O related interface routines the Linux/390 common device support (CDS) provides to allow for device specific driver implementations on the IBM ESA/390 hardware platform. Those interfaces intend to provide the functionality required by every device driver implementation to allow driving a specific hardware device on the ESA/390 platform. Some of the interface routines are specific to Linux/390 and some of them can be found on other Linux platforms' implementations too.

Miscellaneous function prototypes, data declarations, and macro definitions can be found in the architecture specific "C header file" `linux/arch/s390/kernel/IRQ.h`.

Overview of CDS interface concepts

Different to other hardware platforms, the ESA/390 architecture does not define interrupt lines managed by a specific interrupt controller and bus systems that may or may not allow for shared interrupts, DMA processing, etceteras. Instead, the ESA/390 architecture has implemented a so-called channel subsystem, which provides a unified view of the devices physically attached to the systems. Though the ESA/390 hardware platform knows about a huge variety of different peripheral attachments like disk devices (also known as DASD), tapes, communication controllers, they can all by accessed by a well defined access method and they are presenting I/O completion a unified way: I/O interruptions. Every single device is uniquely identified to the system by a so-called subchannel, where the ESA/390 architecture allows for 64k devices to be attached.

Linux, however was first built on the Intel PC architecture, with its two cascaded 8259 programmable interrupt controllers (PICs), that allow for a maximum of 15 different interrupt lines. All devices attached to such a system share those 15 interrupt levels. Devices attached to the ISA bus system must not share interrupt levels (also known as IRQs), as the ISA bus bases on edge triggered interrupts. MCA, EISA, PCI and other bus systems base on level triggered interrupts, and thus allow for shared IRQs. However,

if multiple devices present their hardware status by the same (shared) IRQ, the operating system has to call every single device driver registered on this IRQ in order to determine the device driver owning the device that raised the interrupt.

In order not to introduce a new I/O concept to the common Linux code, Linux/390 preserves the IRQ concept and semantically maps the ESA/390 subchannels to Linux as IRQs. This allows Linux/390 to support up to 64k different IRQs, uniquely representig a single device each.

During its startup the Linux/390 system checks for peripheral devices. A so-called "subchannel" uniquely defines each of those devices by the ESA/390 channel subsystem. While the subchannel numbers are system generated, each subchannel also takes a user-defined attribute, the so-called "device number". Both, subchannel number and device number cannot exceed 65535. The `init_IRQ()` routine gathers the information about control unit type and device types that imply specific I/O commands (channel command words or CCWs) in order to operate the device. Device drivers can retrieve this set of hardware information during their initialization step to recognize the devices they support using `get_dev_info_by_IRQ()` or `get_dev_info_by_devno()` respectively.

These methods imply that Linux/390 does not require to probe for free (not armed) interrupt request lines (IRQs) to drive its devices with. Where applicable, the device drivers can use the `read_dev_chars()` to retrieve device characteristics. This can be done without having to request device ownership previously.

When a device driver has recognized a device it wants to claim ownership for, it calls `request_IRQ()` with the device's subchannel id serving as pseudo IRQ line. One of the required parameters it has to specify is dev_id, defining a device status block, the CDS layer will use to notify the device driver's interrupt handler about interrupt information observed. It depends on the device driver to properly handle those interrupts.

In order to allow for easy I/O initiation the CDS layer provides a `do_IO()` interface that takes a device specific channel program (one or more CCWs) as input sets up the required architecture specific control blocks and initiates an I/O request on behalf of the device driver. The `do_IO()` routine allows for different I/O methods, synchronous and asynchronous, and allows to specify whether it expects the CDS layer to notify the device driver for every interrupt it observes, or with final status only. It also provides a scheme to allow for overlapped I/O processing. See "do_IO() - Initiate I/O Request" on page 92 for more details. A device driver must never issue ESA/390 I/O commands itself, but must use the Linux/390 CDS interfaces instead.

For long running I/O request to be canceled, the CDS layer provides the `halt_IO()` function. Some devices require to initially issue a HALT SUBCHANNEL (HSCH) command without having pending I/O requests. This function is also covered by `halt_IO()`.

When done with a device, the device driver calls `free_IRQ()` to release its ownership for the device. During `free_IRQ()` processing the CDS layer also disables the device from presenting further interrupts: the device driver does not need to assure it. The device will be re-enabled for interrupts with the next call to `request_IRQ()`.

GET_IRQ_FIRST()/GET_IRQ_NEXT() – RETRIEVE INFORMATION ABOUT AVAILABLE IRQS

A device driver can use those interface routines to retrieve information for those IRQs only that have valid device information available. As Linux for S/390 supports a maximum of 65535 subchannels (devices), it might be a waste of CPU to scan for the maximum number of devices while a fraction is available/usable only. `get_irq_first()` will retrieve the first usable IRQ. Using this as input, `get_irq_next()` will retrieve the next IRQ available.

```
int get_irq_first(void);
int get_irq_next(int irq);
```

IRQ	Defines the subchannel to start scanning with. This must be a valid subchannel or an error is returned.

The `get_irq_first()` / `get_irq_next()` functions return:

0	Sucessful completion
-ENODEV	IRQ or devno don't specify a known subchannel or device number.

For example,

```
irq = get_irq_first();
while ( irq != -ENODEV)
{
 get_dev_info_by_irq( irq, &dinfo);
 if (    dinfo.devno == devno_to_look_for
      || dinfo.sid_data.cu_type == cu_type_to_look_for )
 {
  do_some_action( irq, &dinfo );
 } /* endif */
  irq = get_irq_next(irq);
}
```

GET_DEV_INFO_BY_() - RETRIEVE DEVICE INFORMATION

During system startup - `init_IRQ()` processing - the generic I/O device support checks for the devices available. For all devices found it collects the Sense-ID information. For those devices supporting the command it also obtains extended Sense-ID information.

```
int get_dev_info_by_IRQ( int IRQ, s390_dev_info_t *devinfo);
int get_dev_info_by_devno( __u16 devno, s390_dev_info_t *devinfo);
```

IRQ	Defines the subchannel, status information is to be returned for.
devno	device number.
devinfo	Pointer to a user buffer of type `s390_dev_info_t` that should be filled with device

	specific information.

```
typedef struct {
       int    irq;          /* irq, aka subchannel */
       __u16  devno;        /* device number */
       unsigned int status; /* device status */
       senseid_t sid_data;  /* senseID data */
} s390_dev_info_t;
```

devno	Device number as configured in the IOCDS
status	Device status
sid_data	Data obtained by a SenseID call

Possible status values are:

DEVSTAT_NOT_OPER - Device was found not operational. In this case the caller should disregard the sid_data buffer content.

DEVSTAT_UNFRIENDLY_DEV – Device is locked by someone else. The sid_data buffer does not contain valid data.

DEVSTAT_UNKNOWN_DEV – The device is unknown, and the sid_data buffer does not contain valid data.

DEVSTAT_DEVICE_OWNED – An interrupt handler is registered.

```
//
// SenseID response buffer layout
//
typedef struct {
       /* common part */
       __u8 reserved;       /* always 0x'FF' */
       __u16 cu_type;       /* control unit type */
       __u8 cu_model;       /* control unit model */
       __u16 dev_type;      /* device type */
       __u8 dev_model;      /* device model */
       __u8 unused;         /* padding byte */
       /* extended part */
       ciw_t ciw[MAX_CIWS];        /* variable # of CIWs */
} __attribute__ ((packed(4))) s390_senseid_t;
```

MAX_CIWS is currently defined as 8.

The ESA/390 I/O architecture defines certain device specific I/O functions. The device returns the device specific command code together with the Sense-ID data in so called Command Information Words (CIW):

```
typedef struct _ciw {
   __u32        et       :  2; // entry type
```

```
   __u32         reserved :  2; // reserved
   __u32         ct       :  4; // command type
   __u32         cmd      :  8; // command
   __u32         count    : 16; // count
} __attribute__ ((packed)) ciw_t;
```

Possible CIW entry types are:

```
#define CIW_TYPE_RDC 0x0; // read configuration data
#define CIW_TYPE_SII 0x1; // set interface identifier
#define CIW_TYPE_RNI 0x2; // read node identifier
```

The device driver may use these commands as appropriate.

The `get_dev_info_by_IRQ()` / `get_dev_info_by_devno()` functions return:

0	Sucessful completion
-ENODEV	IRQ or devno don't specify a known subchannel or device number.
-EINVAL	Invalid devinfo value.
-EUSERS	Device is locked by someone else.

Usage Notes

In order to scan for known devices a device driver should scan all IRQs by calling `get_dev_info()` until it returns -ENODEV as there are not any more available devices.

If a device driver wants to request ownership for a specific device it must call `request_IRQ()` prior to be able to issue any I/O request for it, including above mentioned device dependent commands.

Please see the "ESA/390 Common I/O-Commands and Self Description" manual, with IBM form number SA22-7204 for more details on how to read the Sense-ID output, CIWs and device independent commands.

GET_IRQ_BY_DEVNO() - CONVERT DEVICE IDENTIFIERS

While some device drivers act on the IRQ (subchannel) only, others take user defined device configurations on device number base, according to the device numbers configured in the IOCDS. The following routines serve the purpose to convert IRQ values into device numbers and vice versa.

```
int get_IRQ_by_devno( unsigned int devno );
unsigned int get_devno_by_IRQ( int IRQ );
```

The functions return:

- The requested IRQ/devno values

- -1 if the requested conversion could not be accomplished. This could either be caused by IRQ/devno be outside the valid range (value > 0xffff or value < 0) or not identifying a known device.

READ_DEV_CHARS() - READ DEVICE CHARACTERISTICS

This routine returns the characteristics for the device specified.

The function is meant to be called without an IRQ handler being in place. However, the IRQ for the requested device must not be locked or this will cause a deadlock situation. Further, the driver must assure that nobody else has claimed ownership for the requested IRQ yet or the owning device driver's internal accounting may be affected.

In case of a registered interrupt handler, the interrupt handler must be able to properly react on interrupts related to the `read_dev_chars()` I/O commands. While the request is processed synchronously, the device interrupt handler is called for final ending status. In case of error situations the interrupt handler may recover appropriately. The device IRQ handler can recognize the corresponding interrupts by the interruption parameter being 0x00524443. If using the function with an existing device interrupt handler in place, the IRQ must be locked prior to call `read_dev_chars()`.

The function may be called enabled or disabled.

```
int read_dev_chars( int IRQ, void **buffer, int length );
```

`IRQ`	specifies the subchannel the device characteristic retrieval is requested for
buffer	pointer to a buffer pointer. The buffer pointer itself may be NULL to have the function allocate a buffer or must contain a valid buffer area.
length	length of the buffer provided or to be allocated.

The read_dev_chars() function returns :

0	Successful completion
-ENODEV	IRQ does not specify a valid subchannel number
-EINVAL	An invalid parameter was detected
-EBUSY	An irrecoverable I/O error occurred or the device is not operational

Usage Notes

The function can be used in two ways:

1. If the caller does not provide a data buffer, `read_dev_chars()` allocates a data

buffer and provides the device characteristics together. It is the caller's responsibility to release the kernel memory if not longer needed. This behavior is triggered by specifying a NULL buffer area (*buffer == NULL).

2. Alternatively, if the user specifies a buffer are, nothing is allocated.

In either case the caller must provide the data area length: for the buffer specified or the buffer wanted allocated.

READ_CONF_DATA() – READ CONFIGURATION DATA

Retrieve the device dependent configuration data. Please have a look at your device dependent I/O commands for the device specific layout of the node descriptor elements.

The function is meant to be called without an IRQ handler being in place. However, the IRQ for the requested device must not be locked or this will cause a deadlock situation!

The function may be called enabled or disabled.

```
int read_conf_data( int irq, void **buffer, int *length, __u8 lpm);
```

irq	Specifies the subchannel the configuration data is to be retrieved for.
buffer	Pointer to a buffer pointer. The read_conf_data() routine will allocate a buffer and initialize the buffer pointer accordingly. It's the device driver's responsibility to release the kernel memory if no longer needed.
length	Length of the buffer allocated and retrieved.
lpm	Logical path mask to be used for retrieving the data. If zero the data is retrieved on the next path available.

The read_conf_data() function returns :

0	Successful completion
-ENODEV	irq doesn't specify a valid subchannel number
-EINVAL	An invalid parameter was detected
-EIO	An irrecoverable I/O error occured or the device is not operational.
-ENOMEM	The read_conf_data() routine couldn't obtain storage
-EOPNOTSUPP	The device doesn't support the read configuration data command.

As previously discussed a device driver will scan for the devices its supports by calling `get_dev_info()`. Once it has found a device it will call `request_IRQ()` to request ownership for it. This call causes the subchannel to be enabled for interrupts if it was found operational.

Note: This function is obsolete and provided for compatibility purposes only. Device drivers should use `s390_request_irq_special()` instead.

```
int request_IRQ( unsigned int IRQ, int (*handler)( int, void *, struct pt_regs *),
unsigned long irqflags, const char *devname, void *dev_id);
```

`IRQ`	Specifies the subchannel the ownership is requested for
`handler`	Specifies the device driver's interrupt handler to be called for interrupt processing
`irqflags`	IRQ flags, must be 0 (zero) or SA_SAMPLE_RANDOM
`devname`	Device name
`dev_id`	Required pointer to a device specific buffer of type devstat_t

```
typedef struct {
     __u16         devno;    /* device number, aka. "cuu" from irb */
     unsigned long intparm;  /* interrupt parameter */
     __u8          cstat;    /* channel status - accumulated */
     __u8          dstat;    /* device status - accumulated */
     __u8          lpum;     /* last path used mask from irb */
     __u8          unused;   /* not used - reserved */
     unsigned int  flag;     /* flag : see below */
     __u32         cpa;      /* CCW address from irb at primary status */
     __u32         rescnt;   /* res. count from irb at primary status */
     __u32         scnt;     /* sense count, if DEVSTAT_FLAG_SENSE_AVAIL */
     union {
        irb_t   irb;         /* interruption response block */
        sense_t sense;       /* sense information */
     } ii;                   /* interrupt information */
} devstat_t;
```

During `request_IRQ()` processing, the devstat_t layout does not matter as it will not be used during `request_IRQ()` processing. See "do_IO() - Initiate I/O Request" on page 92 for a functional description of its usage.

The `request_IRQ()` function returns :

0	Successful completion

-EINVAL	An invalid parameter was detected
-EBUSY	Device (subchannel) already owned
-ENODEV	The device is not operational
-ENOMEM	Not enough kernel memory to process request

Usage Notes

While Linux for Intel defines dev_id as a unique identifier for shared interrupt lines it has a totally different purpose on Linux/390. Here it serves as a shared interrupt status area between the generic device support layer, and the device specific driver. The value passed to `request_IRQ()` must therefore point to a valid devstat_t type buffer area the device driver must preserve for later usage. That is, it must not be released prior to a call to `free_IRQ()`.

Currently, irqflags are ignored by the CDS layer. The Linux/390 kernel does not know about "fast" interrupt handlers, or does it allow for interrupt sharing. Remember, the term interrupt level (IRQ), device, and subchannel are used interchangeably in Linux/390.

If `request_IRQ()` was called in enabled state, or if multiple CPUs are present, the device may present an interrupt to the specified handler prior to `request_IRQ()` return to the caller already. This includes the possibility of unsolicited interrupts or a pending interrupt status from an earlier solicited I/O request. The device driver must be able to handle this situation properly or the device may become non-operational.

Although the interrupt handler is defined to be called with a pointer to a `struct pt_regs` buffer area, the Linux/390 generic I/O device driver support layer does not implement this. The device driver's interrupt handler must therefore not rely on this parameter on function entry.

S390_REQUEST_IRQ_SPECIAL() – REQUEST DEVICE OWNERSHIP

As previously discussed a device driver will scan for the devices its supports by calling `get_dev_info()`. Once it has found a device it will call `request_irq()` to request ownership.

Note: This function replaces `request_irq()` described previously.

```
int s390_request_irq_special(
                     int                        irq,
                     io_handler_func_t          io_handler,
                     not_oper_handler_func_t   not_oper_handler,
                     unsigned long              irqflags,
                     const char                *devname,
                     void                      *dev_id);
```

`irq`	Specifies the subchannel the ownership is requested for
`io_handler`	Specifies the device driver's interrupt handler to be called for interrupt processing
`not_oper_handler`	Specifies a device driver "not operational" handler
`irqflags`	IRQ flags, currently ignored
`devname`	Device name
`dev_id`	Required pointer to a device specific buffer of type `devstat_t`

```
typedef struct {
     __u16         devno;    /* device number, aka. "cuu" from irb */
     unsigned long intparm;  /* interrupt parameter */
     __u8          cstat;    /* channel status - accumulated */
     __u8          dstat;    /* device status - accumulated */
     __u8          lpum;     /* last path used mask from irb */
     __u8          unused;   /* not used - reserved */
     unsigned int  flag;     /* flag : see below */
     __u32         cpa;      /* CCW address from irb at primary status */
     __u32         rescnt;   /* res. count from irb at primary status */
     __u32         scnt;     /* sense count, if DEVSTAT_FLAG_SENSE_AVAIL */
     union {
        irb_t   irb;         /* interruption response block */
        sense_t sense;       /* sense information */
     } ii;                   /* interrupt information */
} devstat_t;
```

During `request_irq()` processing, the devstat_t layout does not matter as it won't be used during `request_irq()` processing. See "do_IO() - Initiate I/O Request" on page 92 for a functional description of its usage.

```
typedef  void (* io_handler_func_t) ( int             irq,
                                      void           *devstat,
                                      struct pt_regs *rgs);
```

`irq`	IRQ the interrupt handler is called for
`devstat`	Device status block
`rgs`	Obsolete

```
typedef (void)(* not_oper_handler_func_t)( int irq,
                                           int status );
```

irq	IRQ the not operational status has been encountered for
status	Device status

Status may contain the following:

DEVSTAT_NOT_OPER   - device is not operational

DEVSTAT_REVALIDATE - revalidate device number

DEVSTAT_DEVICE_GONE - no such device (irq)

Note: Revalidate indicates that running under VM the device number has been modified by means of a `DEFINE xxxx [as] yyyy` command. Therewith device number xxxx was altered to yyyy. It's the device driver's responsibility to decide whether device ownership can be retained.

Gone indicates that the device was detached under VM, or the device number became invalid (native, LPAR). In order to prevent further I/O the IRQ was implicitly freed on behalf of the device driver. The driver must not call free_irq itself.

Not Oper indicates the device became not operational. It's the device driver's responsibility whether it wants to maintain ownership for the IRQ, or not.

The s390_request_irq_special() function returns :

0	Successful completion
-EINVAL	An invalid parameter was detected
-EBUSY	Device (subchannel) already owned
-ENODEV	The device is not operational
-ENOMEM	Not enough kernel memory to process request

Usage Notes

While Linux for Intel defines dev_id as a unique identifier for shared interrupt lines, it has a totally different purpose on Linux for S/390. Hereit serves as a shared interrupt status area between the generic device support layer and the device specific driver. The value passed to `request_irq()` must therefore point to a valid devstat_t type buffer area the device driver must preserve for later usage. That is, it must not be released prior to a call to `free_irq()`.

Currently, the value of irqflags is ignored. The Linux for S/390 kernel does neither know about "fast" interrupt handlers, nor does it allow for interrupt sharing. Remember, the term interrupt level (IRQ), device, and subchannel are used interchangeably in Linux for S/390.

Other than `request_irq()`, this function does allow for a not operational handler to be defined. This handler is called when a device either became not operational, the last path to a device became not operational, or the device was detached from the system. A detach could be a "detach" under VM or that the device became unassigned by the Support Element (SE) or Hardware Management Console (HMC).

If `s390_request_irq_special()` was called in enabled state, or if multiple CPUs are present, the device may present an interrupt to the specified handler prior to `request_irq()` return to the caller already! This includes the possibility of unsolicited interrupts or a pending interrupt status from an earlier solicited I/O request. The device driver must be able to handle this situation properly or the device may become unoperational!

Although the interrupt handler is defined to be called with a pointer to a `struct pt_regs` buffer area, this is not implemented by the Linux for S/390 platform specific common I/O support layer. The device driver's interrupt handler must therefore not rely on this parameter on function entry.

FREE_IRQ() - RELEASE DEVICE OWNERSHIP

A device driver may call `free_IRQ()` to release ownership of a previously acquired device.

```
void free_IRQ( unsigned int IRQ, void *dev_id);
```

IRQ	Specifies the subchannel the ownership is requested for
dev_id	Required pointer to a device specific buffer of type devstat_t. This must be the same as the one specified during a previous call to request_IRQ().

Usage Notes

Unfortunately `free_IRQ()` is defined not to return error codes. That is, if called with wrong parameters a device may still be operational although there is no device driver available to handle its interrupts. Further, during `free_IRQ()` processing we may possibly find pending interrupt conditions. As those need to be processed, we have to delay `free_IRQ()` returning until a clean device status is found by synchronously handling them.

The call to `free_IRQ()` will also cause the device (subchannel) be disabled for interrupts. The device driver must not release any data areas required for interrupt processing prior to `free_IRQ()` return to the caller as interrupts can occur prior to `free_IRQ()` returning. This is also true when called in disabled state if either multiple CPUs are presents or a pending interrupt status was found during `free_IRQ()` processing.

DISABLE_IRQ() - DISABLE INTERRUPTS FOR A GIVEN DEVICE

This function may be called at any time to disable interrupt processing for the specified IRQ. However, as Linux/390 maps IRQs to the device (subchannel) one-to-one, this may require more extensive I/O processing than anticipated, especially if an interrupt status is found pending on the subchannel that requires synchronous error processing.

```
int disable_IRQ( unsigned int IRQ );
```

`IRQ`	Specifies the subchannel to be disabled

The `disable-IRQ()` routine may return:

0	Successful completion
-EBUSY	Device (subchannel) already owned
-ENODEV	The device is not operational or the IRQ does not specify a valid subchannel

Usage Notes

Unlike the Intel based hardware architecture the ESA/390 architecture does not have a programmable interrupt controller (PIC) where a specific interrupt line can be disabled. Instead the subchannel logically representing the device in the channel subsystem must be disabled for interrupts. However, if there are still interrupt conditions pending they must be processed first in order to allow for proper processing after re-enabling the device at a later time. This may lead to delayed disable processing.

As described previously the disable processing may require extensive processing. Therefore disabling and re-enabling the device using `disable_IRQ()` or `enable_IRQ()` should be avoided and is not suitable for high frequency operations.

Linux for Intel defines this function

```
void disable_IRQ( int IRQ);
```

This is suitable for the Intel PC architecture as this only causes to mask the requested IRQ line in the PIC, which is not applicable for the ESA/390 architecture. Therefore we allow for returning error codes.

ENABLE_IRQ() - ENABLE INTERRUPTS FOR A GIVEN DEVICE

This function is used to enable a previously disabled device (subchannel). See "disable_IRQ() - Disable Interrupts for a given Device" on page 90 for more details.

```
int enable_IRQ( unsigned int IRQ );
```

`IRQ`	Specifies the subchannel to be enabled

The `enable_IRQ()` routine may return:

0	Successful completion
-EBUSY	Device (subchannel) busy, which implies the device is already enabled

-ENODEV	The device is not operational or the IRQ does not specify a valid subchannel

DO_IO() - INITIATE I/O REQUEST

The `do_IO()` routines is the I/O request front-end processor. All device driver I/O requests must be issued using this routine. A device driver must not issue ESA/390 I/O commands itself. Instead the `do_IO()` routine provides all interfaces required to drive arbitrary devices.

This description also covers the status information passed to the device driver's interrupt handler as this is related to the rules (flags) defined with the associated I/O request when calling `do_IO()`.

```
int do_IO( int IRQ, ccw1_t *cpa, unsigned long intparm, unsigned int lpm, unsigned long
flag);
```

`IRQ`	IRQ (subchannel) the I/O request is destined for
`cpa`	Logical start address of channel program
`intparm`	User-specific interrupt information; will be presented back to the device driver's interrupt handler. Allows a device driver to associate the interrupt with a particular I/O request.
`lpm`	Defines the channel path to be used for a specific I/O request. Valid with flag value of DOIO_VALID_LPM only.
`flag`	Defines the action to be performed for I/O processing

Possible flag values are:

DOIO_EARLY_NOTIFICATION	Allow for early interrupt notification
DOIO_VALID_LPM	LPM input parameter is valid (see usage notes for details)
DOIO_WAIT_FOR_INTERRUPT	Wait synchronously for final status
DOIO_REPORT_ALL	Report all interrupt conditions

The cpa parameter points to the first format 1 CCW of a channel program:

```
typedef struct {
      __u8  cmd_code;/* command code */
```

```
      __u8  flags;   /* flags, like IDA adressing, etc. */
      __u16 count;   /* byte count */
      __u32 cda;     /* data address */
} __attribute__ ((packed,aligned(8))) ccw1_t;
```

The following CCW flags values are defined:

CCW_FLAG_DC	Data chaining
CCW_FLAG_CC	Command chaining
CCW_FLAG_SLI	Suppress incorrect length
CCW_FLAG_SKIP	Skip
CCW_FLAG_PCI	PCI
CCW_FLAG_IDA	Indirect addressing
CCW_FLAG_SUSPEND	Suspend

The `do_IO()` function returns:

0	Successful completion or request successfully initiated
-EBUSY	The `do_io()` function was called out of sequence. The device is currently processing a previous I/O request
-ENODEV	IRQ does not specify a valid subchannel, the device is not operational (check dev_id.flags) or the IRQ is not owned.
-EINVAL	Both `DOIO_EARLY_NOTIFICATION` and `DOIO_REORT_ALL` flags have been specified. The usage of those flags is mutual exclusive.

When the I/O request completes, the CDS first level interrupt handler will setup the dev_id buffer of type `devstat_t` defined during `request_irq()` processing. See "request_IRQ() - Request Device Ownership" on page 86 for the `devstat_t` data layout. The `dev_id->intparm` field in the device status area will contain the value the device driver has associated with a particular I/O request. If a pending device status was recognized `dev_id->intparm` will be set to 0 (zero). This may happen during I/O initiation or delayed by an alert status notification.

In any case this status is not related to the current (last) I/O request. In case of a delayed status notification no special interrupt will be presented to indicate I/O completion as the I/O request was never started, even though `do_IO()` returned with successful completion.

Possible `dev_id->flag` values are:

DEVSTAT_FLAG_SENSE_AVAIL	Sense data is available
DEVSTAT_NOT_OPER	Device is not operational
DEVSTAT_START_FUNCTION	Interrupt is presented as a result of a call to `do_IO()`
DEVSTAT_HALT_FUNCTION	Interrupt is presented as a result of a call to `halt_IO()`
DEVSTAT_STATUS_PENDING	A pending status was found. The I/O request (if any) was not initiated. This status might have been presented delayed, after `do_IO()` or `halt_IO()` have successfully be started previously.
DEVSTAT_FINAL_STATUS	This is a final interrupt status for the I/O request identified by intparm.
DEVSTAT_PCI	A Program Controlled Interrupt was received.
DEVSTAT_SUSPENDED	A "suspended" intermediate status was received.

If device status `DEVSTAT_FLAG_SENSE_AVAIL` is indicated in field `dev_id->flag`, `field dev_id->scnt` describes the number of device specific sense bytes available in the sense area `dev_id->ii.sense`. No device sensing by the device driver itself is required.

```
typedef struct {
      __u8          res[32];   /* reserved   */
      __u8          data[32];  /* sense data */
   } __attribute__ ((packed)) sense_t;
```

The device interrupt handler can use the following definitions to investigate the primary unit check source coded in sense byte 0:

SNS0_CMD_REJECT	0x80
SNS0_INTERVENTION_REQ	0x40
SNS0_BUS_OUT_CHECK	0x20

SNS0_EQUIPMENT_CHECK	0x10
SNS0_DATA_CHECK	0x08
SNS0_OVERRUN	0x04

Depending on the device status, multiple of those values may be set together. Please refer to the device specific documentation for details.

The `devi_id->cstat` field provides the (accumulated) subchannel status:

SCHN_STAT_PCI	Program controlled interrupt
SCHN_STAT_INCORR_LEN	Incorrect length
SCHN_STAT_PROG_CHECK	Program check
SCHN_STAT_PROT_CHECK	Protection check
SCHN_STAT_CHN_DATA_CHK	Channel data check
SCHN_STAT_CHN_CTRL_CHK	Channel control check
SCHN_STAT_INTF_CTRL_CHK	Interface control check
SCHN_STAT_CHAIN_CHECK	Chaining check

The `dev_id->dstat` field provides the (accumulated) device status:

DEV_STAT_ATTENTION	Attention
DEV_STAT_STAT_MOD	Status modifier
DEV_STAT_CU_END	Control unit end
DEV_STAT_BUSY	Busy
DEV_STAT_CHN_END	Channel end

`DEV_STAT_DEV_END`	Device end
`DEV_STAT_UNIT_CHECK`	Unit check
`DEV_STAT_UNIT_EXCEP`	Unit exception

Please see the ESA/390 Principles of Operation manual for details on the individual flag meanings.

In rare error situations the device driver may require access to the original hardware interrupt data beyond the scope of previously mentioned information. For those situations the Linux/390 common device support provides the interrupt response block (IRB) as part of the device status block in `dev_id->ii.irb`.

Usage Notes

Prior to call `do_IO()` the device driver must assure disabled state, that is, the I/O mask value in the PSW must be disabled. This can be accomplished by calling `__save_flags(flags)`. The current PSW flags are preserved and can be restored by `__restore_flags(flags)` at a later time.

If the device driver violates this rule while running in a uni-processor environment an interrupt might be presented prior to the `do_IO()` routine returning to the device driver main path. In this case we will end in a deadlock situation, as the interrupt handler will try to obtain the IRQ lock the device driver still owns.

The driver must assure to hold the device specific lock. This can be accomplished by

1. `s390irq_spin_lock(IRQ)`, or
2. `s390irq_spin_lock_irqsave(IRQ, flags)`

Option (1) should be used if the calling routine is running disabled for I/O interrupts already. Option (2) obtains the device gate and puts the CPU into I/O disabled state by preserving the current PSW flags.

See the descriptions of `s390irq_spin_lock()` or `s390irq_spin_lock_irqsave()` for more details.

The device driver is allowed to issue the next `do_IO()` call from within its interrupt handler already. It is not required to schedule a bottom-half, unless an non deterministically long running error recovery procedure or similar needs to be scheduled. During I/O processing the Linux/390 generic I/O device driver support has already obtained the IRQ lock, that is, the handler must not try to obtain it again when calling `do_IO()` or we end in a deadlock situation. Anyway, the device driver's interrupt handler must only call `do_IO()` if the handler itself can be entered recursively if `do_IO()`, for example, it finds a status pending and needs to all the interrupt handler itself.

Device drivers should not rely on `DOIO_WAIT_FOR_INTERRUPT` synchronous I/O request processing too heavily. All I/O devices, but the console device are driven using a single shared interrupt subclass (ISC).

For synchronous processing the device is temporarily mapped to a special ISC while the calling CPU waits for I/O completion. As this special ISC is gated, all synchronous requests in an SMP environment are serialized which may cause other CPUs to spin. Primarily, this service is meant to be used during device driver initialization for ease of device setup.

If the device driver is using the `DOIO_TIMEOUT` parameter, it is a good idea also to specify `DOIO_CANCEL_ON_TIMEOUT`. Otherwise, the failing channel program may prevent the execution of any other channel program at the subchannel.

The lpm input parameter might be used for multi-path devices shared among multiple systems as the Linux/390 CDS is not grouping channel paths. Therefore, its use might be required if multiple access paths to a device are available and the device was reserved by means of a reserve device command (for devices supporting this technique). When issuing this command the device driver needs to extract the `dev_id->lpum` value and restrict all subsequent channel programs to this channel path until the device is released by a device release command. Otherwise a deadlock may occur.

If a device driver relies on an I/O request to be completed prior to start the next it can reduce I/O processing overhead by chaining a no-op I/O command `CCW_CMD_NOOP` to the end of the submitted CCW chain. This will force Channel-End and Device-End status to be presented together, with a single interrupt.

However, this should be used with care as it implies the channel will remain busy, not being able to process I/O requests for other devices on the same channel. Therefore, for example, read commands should never use this technique, as the result will be presented by a single interrupt anyway.

In order to minimize I/O overhead, a device driver should use the `DOIO_REPORT_ALL` only if the device can report intermediate interrupt information prior to device-end the device driver urgently relies on. In this case all I/O interruptions are presented to the device driver until final status is recognized.

If a device is able to recover from asynchronously presented I/O errors, it can perform overlapping I/O using the `DOIO_EARLY_NOTIFICATION` flag. While some devices always report channel-end and device-end together, with a single interrupt, others present primary status (channel-end) when the channel is ready for the next I/O request and secondary status (device-end) when the data transmission has been completed at the device.

The previously mentioned flag allows exploitation of this feature, for example, for communication devices that can handle lost data on the network to allow for enhanced I/O processing.

Unless the channel subsystem at any time presents a secondary status interrupt, exploiting this feature will cause only primary status interrupts to be presented to the device driver while overlapping I/O is performed. When a secondary status without error (alert status) is presented, this indicates successful completion for all overlapping `do_IO()` requests that have been issued since the last secondary (final) status.

During interrupt processing the device specific interrupt handler should avoid basing its processing decisions on the interruption response block (IRB) that is part of the dev_id buffer area. The IRB area

represents the interruption parameters from the last interrupt received. Unless the device driver has specified `DOIO_REPORT_ALL` or is called with a pending status (`DEVSTAT_STATUS_PENDING`), the IRB information may or may not show the complete interruption status, but the last interrupt only. Therefore the device driver should usually base its processing decisions on the values of `dev_id->cstat` and `dev_id->dstat` that represent the accumulated subchannel and device status information gathered since `do_IO()` request initiation.

Channel programs that intend to set the suspend flag on a channel command word (CCW) must start the I/O operation with the `DOIO_ALLOW_SUSPEND` option or the suspend flag will cause a channel program check. At the time the channel program becomes suspended an intermediate interrupt will be generated by the channel subsystem.

RESUME_IO – RESUME CHANNEL PROGRAM EXECUTION

If a device driver chooses to suspend the current channel program execution by setting the CCW suspend flag on a particular CCW, the channel program execution is suspended. In order to resume channel program execution the CIO layer provides the resume_IO() routine.

```
int resume_IO( int irq);
```

`irq`	IRQ (subchannel) the halt operation is requested for

The `resume_IO()` function returns:

0	Suspended channel program is resumed
-EBUSY	Status pending
-ENODEV	Invalid or not operational subchannel
-EINVAL	Resume function not applicable
-ENOTCONN	There is no I/O request pending for completion

Usage Notes Please have a look at the `do_IO()` usage notes on page 96 for more details on suspended channel programs.

HALT_IO() - HALT I/O REQUEST PROCESSING

Sometimes a device driver might need a possibility to stop the processing of a long-running channel program or the device might initially require issuing a halt subchannel (HSCH) I/O command. For those purposes the `halt_IO()` command is provided.

```
int halt_IO( int IRQ,                  /* subchannel number */
             unsigned long intparm,    /* dummy intparm */
             unsigned long flag);      /* operation mode */
```

`IRQ`	IRQ (subchannel) the halt operation is requested for

`intparm`	Interruption parameter; value is only used if no I/O is outstanding, otherwise the intparm associated with the I/O request is returned
`flag`	0 (zero) or `DOIO_WAIT_FOR_INTERRUPT`

The `halt_IO()` function returns:

`0`	Successful completion or request successfully initiated
-EBUSY	The device is currently performing a synchronous I/O operation: `do_IO()` with flag `DOIO_WAIT_FOR_INTERRUPT` or an error was encountered and the device is currently be sensed
-ENODEV	The IRQ specified does not specify a valid subchannel, the device is not operational (check dev_id.flags) or the IRQ is not owned.

Usage Notes

A device driver may write a never-ending channel program by writing a channel program that at its end loops back to its beginning by means of a transfer in channel (TIC) command (CCW_CMD_TIC). Usually network device drivers perform this by setting the PCI CCW flag (CCW_FLAG_PCI). Once this CCW is executed a program controlled interrupt (PCI) is generated. The device driver can then perform an appropriate action. Prior to interrupt of an outstanding read to a network device (with or without PCI flag) a `halt_IO()` is required to end the pending operation.

We do not allow the stopping of synchronous I/O requests by means of a `halt_IO()` call. The function will return -EBUSY instead.

Miscellaneous Support Routines

This section describes various routines to be used in a Linux/390 device driver programming environment.

S390IRQ_SPIN_LOCK() / S390IRQ_SPIN_UNLOCK()

These two macro definitions are required to obtain the device specific IRQ lock. The lock needs to be obtained if the device driver intends to call `do_IO()` or `halt_IO()` from anywhere but the device interrupt handler (where the lock is already owned). Those routines must only be used if running disabled for interrupts already. Otherwise use `s390irq_spin_lock_irqsave()` and the corresponding unlock routine instead.

```
s390irq_spin_lock(int IRQ);
s390irq_spin_unlock(int IRQ);
```

S390IRQ_SPIN_LOCK_IRQSAVE() / S390_IRQ_SPIN_UNLOCK_IRQRESTORE()

These two macro definitions are required to obtain the device specific IRQ lock. The lock needs to be obtained if the device driver intends to call `do_IO()` or `halt_IO()` from anywhere but the device interrupt handler (where the lock is already owned). Those routines should only be used if running enabled for interrupts. If running disabled already, the driver should use `s390irq_spin_lock()` and the corresponding unlock routine instead.

```
s390irq_spin_lock_irqsave( int IRQ, unsigned long flags);
s390irq_spin_unlock_irqrestore( int IRQ, unsigned long flags);
```

Special Console Interface Routines

This section describes the special interface routines required for system console processing. Though they are an extension to the Linux/390 device driver interface concept, they base on the same principles. It was necessary to build those extensions to assure a deterministic behavior in critical situations, for example, `printk()` messages by other device drivers running disabled for interrupts during I/O interrupt handling or in case of a `panic()` message being raised.

SET_CONS_DEV() - SET CONSOLE DEVICE

This routine allows specification of the system console device. This is necessary as the console is not driven by the same ESA/390 interrupt subclass as are other devices, but it is assigned its own interrupt subclass. Only one device can act as system console. See "wait_cons_dev() - Synchronously Wait for Console Processing" on page 101 for details.

```
int set_cons_dev( int IRQ);
```

`IRQ`	Subchannel identifying the system console device

The `set_cons_dev()` function returns

`0`	Successful completion
-EIO	An unhandled interrupt condition is pending for the specified subchannel (IRQ) - status pending
-ENODEV	IRQ does not specify a valid subchannel or the device is not operational
-EBUSY	The console device is already defined

RESET_CONS_DEV() - RESET CONSOLE DEVICE

This routine allows for resetting the console device specification. See "set_cons_dev() - Set Console Device" on page 100 for details.

```
int reset_cons_dev( int IRQ);
```

`IRQ`	Subchannel identifying the system console device

The `reset_cons_dev()` function returns

`0`	Successful completion
-EIO	An unhandled interrupt condition is pending for the specified subchannel (IRQ) - status pending
-ENODEV	IRQ does not specify a valid subchannel or the device is not operational

WAIT_CONS_DEV() - SYNCHRONOUSLY WAIT FOR CONSOLE PROCESSING

The `wait_cons_dev()` routine is used by the console device driver when its buffer pool for intermediate request queuing is exhausted and a new output request is received. In this case the console driver uses the `wait_cons_dev()` routine to synchronously wait until enough buffer space is gained to enqueue the current request. Any pending interrupt condition for the console device found during `wait_cons_dev()` processing causes its interrupt handler to be called.

```
int wait_cons_dev( int IRQ);
```

`IRQ`	Subchannel identifying the system console device

The `wait_cons_dev()` function returns :

`0`	Successful completion
-EINVAL	The IRQ specified does not match the IRQ configured for the console device by `set_cons_dev()`

Usage Notes

The function should be used carefully. Especially in a SMP environment the `wait_cons_dev()` processing requires that all but the special console ISC are disabled. In a SMP system this requires the other CPUs to be signaled to disable/enable those ISCs.

DASD Device Driver

The following section was copied from the `Documentation/390` directory of the Linux distribution. It was written by Aldo Lung and is copyright IBM 1999, under the GNU Public License.

Linux manages S/390_s disk devices (DASD) via the DASD device driver. It is valid for all types of DASDs and represents them to Linux as block devices, namely "DASD". Currently the DASD driver uses a single major number (94) and 4 minor numbers per volume (1 for the physical volume and 3 for partitions). With respect to partitions see the following discussion. Thus you may have up to 64 DASD devices in your system.

The kernel parameter 'dasd=from-to,...' may be issued arbitrary times in the kernel's parameter line or not at all. The 'from' and 'to' parameters are to be given in hexadecimal notation without a leading 0x.

If you supply kernel parameters the different instances are processed in order of appearance and a minor number is reserved for any device covered by the supplied range up to 64 volumes. Additional DASDs are ignored. If you do not supply the 'dasd=' kernel parameter at all, the DASD driver registers all supported DASDs of your system to a minor number in ascending order of the subchannel number.

The driver currently supports ECKD-devices and there are stubs for support of the FBA and CKD architectures. For the FBA architecture only some smart data structures are missing to make the support complete.

We performed our testing on 3380 and 3390 type disks of different sizes, under VM and on the bare hardware (LPAR), using internal disks of the Multiprise as well as a RAMAC virtual array. Disks exported by an Enterprise Storage Server (Seascape) should work fine as well.

We currently implement one partition per volume, which is the whole volume, skipping the first blocks up to the volume label. These are reserved for IPL records and IBM's volume label to assure accessibility of the DASD from other operating systems. In a later stage we will provide support of partitions, maybe VTOC oriented or using a kind of partition table in the label record.

Usage

Low-level format

For using an ECKD-DASD as a Linux hard disk you have to low-level format the tracks by issuing the `BLKDASDFORMAT-ioctl` on that device. This will erase any data on that volume including IBM volume labels, VTOCs etceteras. The ioctl may take a 'struct format_data *' or 'NULL' as an argument.

```
typedef struct {
       int start_unit;
       int stop_unit;
```

```
        int blksize;
} format_data_t;
```

When a NULL argument is passed to the `BLKDASDFORMAT ioctl` the whole disk is formatted to a blocksize of 1024 bytes. Otherwise start_unit and stop_unit are the first and last track to be formatted. If stop_unit is -1 it implies that the DASD is formatted from start_unit up to the last track. blksize can be any power of two between 512 and 4096. We recommend no blksize lower than 1024 because the ext2fs uses 1kB blocks anyway and you gain approximately 50% of capacity increasing your blksize from 512 byte to 1kB.

Make a filesystem

Then you can mk??fs the filesystem of your choice on that volume or partition. For reasons of sanity you should build your filesystem on the partition `/dev/dd?1` instead of the whole volume. You only lose 3kB but may be sure that you can reuse your data after introduction of a real partition table.

Bugs

- Performance sometimes is rather low because we do not fully exploit clustering

TODO-List

- Add IBM'S Disk layout to genhd
- Enhance driver to use more than one major number
- Enable usage as a module
- Support Cache fast write and DASD fast write (ECKD)

Tape Support

The LINUX for zSeries tape device driver manages channel-attached tape drives which are compatible to IBM 3480 or IBM 3490 magnetic tape subsystems. This includes various models of these devices (for example the 3490E).

Tape driver features

The device driver supports a maximum of 128 tape devices. No official LINUX device major number is assigned to the zSeries tape device driver. It allocates major numbers dynamically and reports them on system startup.

Typically it will get major number 254 for both the character device front-end and the block device front-end.

The tape device driver needs no kernel parameters. All supported devices present are detected on driver initialization at system startup or module load. The devices detected are ordered by their subchannel numbers. The device with the lowest subchannel number becomes device 0, the next one will be device 1 and so on.

Tape character device front-end

The usual way to read or write to the tape device is through the character device front-end. The zSeries tape device driver provides two character devices for each physical device—the first of these will rewind automatically when it is closed, the second will not rewind automatically.

The character device nodes are named `/dev/rtibm0` (rewinding) and `/dev/ntibm0` (non-rewinding) for the first device, `/dev/rtibm1` and `/dev/ntibm1` for the second, and so on.

The character device front-end can be used as any other LINUX tape device. You can write to it and read from it using LINUX facilities such as GNU tar. The tool `mt` can be used to perform control operations, such as rewinding the tape or skipping a file.

Most LINUX tape software should work with either tape character device.

Tape block device front-end

The tape device may also be accessed as a block device in read-only mode. This could be used for software installation in the same way as it is used with other operation systems on the zSeries platform (and most LINUX distributions are shipped on compact disk using ISO9660 filesystems).

One block device node is provided for each physical device. These are named `/dev/btibm0` for the first device, `/dev/btibm1` for the second and so on.

You should only use the ISO9660 filesystem on LINUX for zSeries tapes because the physical tape devices cannot perform fast seeks and the ISO9660 system is optimized for this situation.

Tape block device example

In this example a tape with an ISO9660 filesystem is created using the first tape device. ISO9660 filesystem support must be built into your system kernel for this.

The mt command is used to issue tape commands and the mkisofs command to create an ISO9660 filesystem:

- Create a LINUX directory (somedir) with the contents of the filesystem

  ```
  mkdir somedir
  cp contents somedir
  ```

- Insert a tape

- Ensure the tape is at the beginning

  ```
  mt -f /dev/ntibm0 rewind
  ```

- Set the blocksize of the character driver. The blocksizes 512, 1024 and 2048 bytes are supported by ISO9660. 1024 is the default, which will be used here. `mt -f /dev/ntibm0 setblk 1024`

- write the filesystem to the character device driver

  ```
  mkisofs -o /dev/ntibm0 somedir
  ```

- rewind the tape again

  ```
  mt -f /dev/ntibm0 rewind
  ```

- Now you can mount your new filesystem as a block device:

  ```
  mount -t iso9660 -o ro,block=1024 /dev/btibm0 /mnt
  ```

TODO List

- The backend code has to be enhanced to support error-recovery actions.

- The seeking algorithm of the block device has to be improved to speed things up

BUGS

There are lots of weaknesses still in the code. This is why it is EXPERIMENTAL.

If an error occurs which cannot be handled by the code you will get a sense-data dump. In that case please do the following:

1. Set the tape driver debug level to maximum:

   ```
   echo 6 >/proc/s390dbf/tape/level
   ```

2. Re-perform the actions that produced the bug. (Hopefully the bug will reappear.)

3. Get a snapshot from the debug-feature:

   ```
   cat /proc/s390dbf/tape/hex_ascii >somefile
   ```

4. Now put the snapshot together with a detailed description of the situation that led to the bug:

   - Which tool did you use?
   - Which hardware do you have?
   - Was your tape unit online?
   - Is it a shared tape unit?

5. Send an email with your bug report to: mailto:Linux390@de.ibm.com

3270 Display System Support

This file describes the driver that supports local channel attachment of IBM 3270 devices written by Dick Hitt <rbh00@utsglobal.com>. It consists of three sections:

- Introduction
- Installation
- Operation

INTRODUCTION

This paper describes installing and operating 3270 devices under Linux/390. A 3270 device is a block-mode rows-and-columns terminal of which I'm sure hundreds of millions were sold by IBM and clone makers twenty and thirty years ago.

You may have 3270s in-house and not know it. If you're using the VM-ESA operating system, define a 3270 to your virtual machine by using the command "DEF GRAF <hex-address>". This paper presumes you will be defining four 3270s with the CP/CMS commands

```
DEF GRAF 620
DEF GRAF 621
DEF GRAF 622
DEF GRAF 623
```

Your network connection from VM-ESA allows you to use x3270, tn3270, or another 3270 emulator, started from an xterm window on your PC or workstation. With the DEF GRAF command, an application such as xterm, and this Linux-390 3270 driver, you have another way of talking to your Linux box.

OPERATION

The driver defines three areas on the 3270 screen: the log area, the input area, and the status area.

The log area takes up all but the bottom two lines of the screen. The driver writes terminal output to it, starting at the top line and going down. When it fills, the status area changes from “`Linux Running`” to “`Linux More...`”. After a scrolling timeout of (default) 5 sec, the screen clears and more output is written, from the top down.

The input area extends from the beginning of the second-to-last screen line to the start of the status area. You type commands in this area and hit ENTER to execute them.

The status area initializes to "`Linux Running`" to give you a warm fuzzy feeling. When the log area fills up and output waits, it changes to "`Linux More...`". At this time you can do several things or nothing. If you do nothing, the screen will clear in (default) 5 sec and more output will appear. You may hit ENTER with nothing typed in the input area to toggle between "`Linux More...`" and "`Linux Holding`", which indicates no scrolling will occur. (If you hit ENTER with "`Linux Running`" and nothing typed, the application receives a newline.)

You may change the scrolling timeout value. For example, the following command line changes the scrolling timeout value to 60 seconds:

```
echo scrolltime=60 > /proc/tty/driver/tty3270
```

Set scrolltime to 0 if you wish to prevent scrolling entirely.

Other things you may do when the log area fills up are: hit `PA2` to clear the log area and write more output to it, or hit `CLEAR` to clear the log area and the input area and write more output to the log area.

Some of the Program Function (PF) and Program Attention (PA) keys are preassigned special functions. The ones that are not yield an alarm when pressed.

`<PA1>` causes a `SIGINT` to the currently running application. You may do the same thing from the input area, by typing "`^C`" and hitting `<ENTER>`.

`<PA2>` causes the log area to be cleared. If output awaits, it is then written to the log area.

`<PF3>` causes an EOF to be received as input by the application. You may cause an EOF also by typing "`^D`" and hitting `<ENTER>`.

No PF key is preassigned to cause a job suspension, but you may cause a job suspension by typing "`^Z`" and hitting `<ENTER>`. You may wish to assign this function to a PF key. To make `<PF7>` cause job suspension, execute the command:

```
echo pf7=^z > /proc/tty/driver/tty3270
```

If the input you type does not end with the two characters "`^n`", the driver appends a newline character and sends it to the tty driver; otherwise the driver strips the "`^n`" and does not append a newline. The IBM 3215 driver behaves similarly.

`<PF10>` causes the most recent command to be retrieved from the tube's command stack (default depth 20) and displayed in the input area. You may hit `<PF10>` again for the next-most-recent command, and so on. A command is entered into the stack only when the input area is not made invisible (such as for password entry) and it is not identical to the current top entry. `<PF10>` rotates backward through the command stack; `<PF11>` rotates forward. You may assign the backward function to any PF key (or PA key, for that matter), say, `<PA3>`, with the command:

```
echo -e pa3=\\033k > /proc/tty/driver/tty3270
```

This assigns the string ESC-k to <PA3>. Similarly, the string ESC-j performs the forward function. (Rationale: In bash with vi-mode line editing, ESC-k and ESC-j retrieve backward and forward history. Suggestions welcome.)

Is a stack size of twenty commands not to your liking? Change it on the fly. To change to saving the last 100 commands, execute the command:

```
echo recallsize=100 > /proc/tty/driver/tty3270
```

Have a command you issue frequently? Assign it to a PF or PA key! Use the command to execute the commands mkdir foobar and cd foobar immediately when you hit <PF24>:

```
echo pf24="mkdir foobar; cd foobar" > /proc/tty/driver/tty3270
```

Want to see the command line first, before you execute it? Use the -n option of the echo command:

```
echo -n pf24="mkdir foo; cd foo" > /proc/tty/driver/tty3270
```

Happy testing! I welcome any and all comments about this document, the driver, etc etc.

XPRAM

The S/390 architecture supports more RAM than can be accessed as main memory. The LINUX for S/390 main memory is limited to 2 GB. However, additional memory can be declared as expanded storage. The S/390 architecture allows applications to access up to 16 TB of expanded storage (although the current hardware can only be equipped with up to 32 GB memory). Memory in the expanded storage range can be copied in 4 KB blocks to, or from, the main memory.

An interesting feature of expanded storage is that is persistent with respect to IPLs (booting) but volatile with respect to IMLs (power off/on).

The XPRAM device driver is a block device driver that supports LINUX for S/390 allowing it to access the expanded storage. Thus XPRAM can be used as a basis for fast swap devices and/or fast file systems.

Features

XPRAM automatically detects whether expanded storage is available on the system. The expanded storage can be subdivided into up to 32 partitions, the default being a single partition. The XPRAM device driver has major number 35.

The partitions have minor numbers 0 through 31. The hard sector size of XPRAM is set to 4096 bytes.

Limitations

If expanded storage is not available, XPRAM cannot be used. Its initialization fails gracefully with a log message reporting the lack of expanded storage.

Configuration option

`CONFIG_XPRAM`

Module name

XPRAM can be used as module. Its module name is `xpram.o`.

Kernel parameter syntax

The kernel parameter is optional. The default defines the whole expanded storage to be one partition.

```
xpram_parts=<number_of_partition>[,<partition_size>[,...]]
```

Where `<number_of_partitions>` defines how many partitions the expanded storage is split into. The *i*-th `<partition_size>` defines the size of the *i*-th partition.

The syntax for sizes is:

```
[0x]<non-negative_integer>[k|K|m|M|g|G]
```

If the `0x` prefix is used the subsequent number is interpreted as a hexadecimal value, otherwise it is interpreted as a decimal value (default). The `<non-negative_integer>` value may be followed by a magnitude:

- `k` or `K` for kilo (1024) is the default
- `m` or `M` for Mega (1024*1024)
- `g` or `G` for Giga (1024*1024*1024)

The `<non-negative_integer>` value multiplied by its magnitude defines the partition's size in bytes. The default size is 0.

Any partition defined with a non-zero size is allocated the amount of memory specified by its `<non-negative_integer>` parameter.

You can automatically allocate the remaining memory between a set of partitions by specifying zero for the size of each partition in the set. The following formula is used to calculate the size for each of these partitions:

```
               (available exp. storage - sum of all non-zero sizes specified)
computed size = --------------------------------------------------------------
                            number of partitions with zero sizes
```

This formula is only a good approximation of the actual size allocated to each partition. Because of the requirement to assign blocks in multiples of 4K, partitions can be larger or smaller than the estimate produced by the calculation. In addition, there might be an amount of memory left as a "guard +space" between two partitions.

Example

```
xpram_parts=4,0x800M,0,0,0x1000M
```

This allocates the extended storage into four partitions. Partition 1 has 2 GB, partition 4 has x 4 GB, and partitions 2 and 3 use equal parts of the remaining storage. If the total amount of extended storage was 16 GB, then partitions 3 and 4 would each have approximately 5 GB.

Module parameter syntax

XPRAM may be used as module. The syntax of the module parameters passed to `insmod` differs from the kernel parameter syntax:

```
[devs=<number_of_devices> [sizes=<size>[,<size>,...]]]
```

Where:

- `<number_of_devices>` is used to define the number of partitions.
- `<size>` is a non-negative integer that defines the partition's size.

Only decimal values are allowed and no magnitudes are accepted. The size will be interpreted in kilobytes.

Example

```
devs=4 sizes=2097152,8388608,4194304,2097152
```

This allocates a total of 16 GB of extended storage into four partitions, of (respectively) size 2 GB, 8 GB, 4 GB, and 2 GB.

Usage

XPRAM is a block device driver with major 35. Using the standard naming scheme the partitions of XPRAM can be accessed through `/dev/slram0`, ... , `/dev/slram31`.

XPRAM does not require any formatting. Partitioning is only possible during device initialization by kernel or module parameters. Note that if both the expanded storage and the partitioning parameters are left unchanged between two device initializations (even if LINUX was IPLed in the meantime) then XPRAM behaves like a persistent storage. This is not true if the system is IMLed.

You can make a files system on a XPRAM partition (for example, `mke2fs`) with a block size that is a multiple of 4096 bytes and mount this file system.

Alternatively, an XPRAM partition can be used as a swap device (`mkswap`, `swapon`).

CISCO CLAW SUPPORT

The c7000 module provides support for a channel attached Cisco 7xxx family router on Linux/390. The parameters for the module are as follows:

Parameter	Description
`base0=0xYYYY`	This parameter defines the base unit address of the channel-attached router.
`lhost0=s1`	This parameter defines the local host name and must match the claw directive "host-name" field (first string). The default value is "UTS".
`uhost0=s2`	This parameter defines the unit's name and must match the claw directive "device-name" field (second string). The default value is "C7011".
`lappl0=s3`	This parameter defines the local application name and must match the claw directive "host-app" field (third string). The default value is "TCPIP".
`uappl0=s4`	This parameter defines the unit application name and must match the claw directive "device-app" field (fourth string). The default value is "TCPIP".
`dbg=x`	This parameter defines the message level. Higher numbers will result in additional diagnostic messages. The default value is 0.
`noauto=z`	This parameter controls the automatic detection of the unit base address (base0). When set to a non-zero value, automatic detection of unit base addresses is not done. The default value is 0.

Note that the values coded in strings s1 - s4 are case sensitive.

For example, assume that the following claw directive has been coded in the Cisco router:

```
claw 0100 6C 129.212.61.101 UTS C7011 TCPIP TCPIP
```

The module can be loaded using the following command:

```
insmod c7000 base0=0x336c lhost0="UTS" uhost0="C7011" lappl0="TCPIP" \
uappl0="TCPIP" dbg=0 noauto=1
```

Additional interfaces can be defined via parameters base1 - base3, lhost1 - lhost3, lappl1 - lappl3, uhost1 - uhost3, uappl1 - uappl3.

The interfaces are named "ci0" – "ci3". After loading the module, the ifconfig command is used to configure the interface. For example:

```
ifconfig ci0 129.212.61.101
ifconfig ci0 netmask 255.255.255.0 broadcast 129.212.61.0
```

```
ifconfig ci0
```

The route command is used to specify the router as the default route:

```
route add default gw 129.212.61.200
```

The interface can be automatically activated at boot time by following this procedure:

1. Add the following two lines to file `/etc/conf.modules`:

   ```
   alias ci0 c7000
   options c7000 base0=0xYYYY lhost0=s1 uhost0=s2 lappl0=s3 uappl0=s4
   ```

2. Edit file `/etc/sysconfig/network` as follows:

   ```
   NETWORKING=yes
   FORWARD_IPV4=no
   HOSTNAME=your-hostname
   GATEWAYDEV=ci0
   GATEWAY=your-gateway-ip-address
   ```

   Substitute your own host name and gateway IP address.

3. Create a file in directory `/etc/sysconfig/network-scripts` called `ifcfg-ci0`. The contents are as follows:

   ```
   DEVICE=ci0
   USERCTL=no
   ONBOOT=yes
   BOOTPROTO=none
   BROADCAST=your-broadcast-ip-address
   NETWORK=your-network-address
   NETMASK=your-netmask
   IPADDR=your-ip-address
   ```

   Substitute your IP address, broadcast IP address, network address and network mask.

4. Next issue: `chmod +x ifcfg-ci0`

IUCV

To explore any of the IUCV functions, one must first register their program using iucv_register_program(). Once your program has successfully completed a register, it can exploit the other functions.

For furthur reference on all IUCV functionality, refer to the CP Programming Services book, also available on the web thru www.ibm.com/s390/vm/pubs, manual # SC24-5760.

Definition of Return Codes:

- All positive return codes including zero are reflected back from CP except for iucv_register_program. The definition of each return code can be found in CP Programming Services book. Also available on the web thru www.ibm.com/s390/vm/pubs, manual # SC24-5760
- Return Code of:
  - (-EINVAL) Invalid value
  - (-ENOMEM) storage allocation failed

  pgmask defined in iucv_register_program will be set depending on input paramters.

iucv_accept

```
int iucv_accept (u16 pathid,
                 u16 msglim_reqstd,
                 uchar user_data[16],
                 int flags1,
                 iucv_handle_t handle,
                 void *pgm_data, int *flags1_out, u16 * msglim);
```

This function is issued after the user receives a Connection Pending external interrupt and now wishes to complete the IUCV communication path.

Parameters

pathid	Path identification number
msglim_reqstd	The number of outstanding messages requested.
user_data	Data specified by the iucv_connect function.
flags1	Contains options for this path:

	• IPPRTY (0x20) - Specifies if you want to send priority message. • IPRMDATA (0x80) - Specifies whether your program can handle a message in the parameter list. • IPQUSCE (0x40) - Specifies whether you want to quiesce the path being established.
handle	Address of handler.
pgm_data	Application data passed to interrupt handlers.
flags1_out	Pointer to an int. If not NULL, on return the options for the path are stored at the given location. • IPPRTY (0x20) - Indicates you may send a priority message.
msglim	Pointer to a __u16. If not NULL, on return the maximum number of outstanding messages is stored at the given location.

Returns

Return code from CP.

iucv_connect

```
int iucv_connect (u16 * pathid,
                  u16 msglim_reqstd,
                  uchar user_data[16],
                  uchar userid[8],
                  uchar system_name[8],
                  int flags1,
                  int *flags1_out,
                  u16 * msglim, iucv_handle_t handle, void *pgm_data);
```

This function establishes an IUCV path. Although the connect may complete successfully, you are not able to use the path until you receive an IUCV Connection Complete external interrupt.

Parameters

pathid	Path identification number
msglim_reqstd	Number of outstanding messages requested
user_data	16-byte user data
userid	8-byte of user identification

system_name	8-byte identifying the system name
flags1	Specifies options for this path: ▪ IPPRTY (0x20) - Specifies if you want to send priority message. ▪ IPRMDATA (0x80) - Specifies whether your program can handle a message in the parameter list. ▪ IPQUSCE (0x40) -Specifies whether you want to quiesce the path being established. ▪ IPLOCAL (0x01) - Allows an application to force the partner to be on the local system. If local is specified then target class cannot be specified.
flags1_out	Pointer to an int. If not NULL, on return the options for the path are stored at the given location. ▪ IPPRTY (0x20) - Indicates you may send a priority message.
msglim	Pointer to a __u16. If not NULL, on return the maximum number of outstanding messages is stored at the given location.
handle	Address of handler.
pgm_data	Application data to be passed to interrupt handlers.

Returns

The functions returns either the return code from CP (>=0), or one of the following:

-ENOMEM	Return code from iucv_declare_buffer
-EINVAL	Invalid handle passed by application or pathid entry being used by another application
-EINVAL	Pathid address is NULL
-ENOMEM	Pathid table storage allocation failed

iucv_purge

```
int iucv_purge (u16 pathid, u32 msgid, u32 srccls, __u32 *audit);
```

A call to this API cancels a message you have sent.

Parameters

pathid	Path identification number
msgid	Message ID of message to purge.
srccls	Message class of the message to purge.
audit	Pointer to a __u32. If not NULL, on return, information about asynchronous errors that may have affected the normal completion of this message is stored at the given location.

Returns

Return code from CP.

iucv_query_maxconn

```
ulong iucv_query_maxconn (void);
```

Determines the maximum number of connections thay may be established.

Parameters

None.

Returns

Maximum number of connections that can be established.

iucv_query_bufsize

```
ulong iucv_query_bufsize (void);
```

This function determines how large an external interrupt buffer IUCV requires to store information.

Parameters

None.

Returns

Maximum number of connection the virtual machine may establish.

iucv_quiesce

```
int iucv_quiesce (u16 pathid, uchar user_data[16]);
```

This function temporarily suspends incoming messages on an IUCV path. You can later reactivate the path by invoking the iucv_resume function.

Parameters

pathid	Path identification number

user_data	16-bytes of user data

Returns

Return code from CP IUCV call.

iucv_receive

```
int iucv_receive (u16 pathid,
                  u32 msgid,
                  u32 trgcls,
                  void *buffer,
                  ulong buflen,
                  int *flags1_out,
                  ulong * residual_buffer, ulong * residual_length);
```

This function receives messages that are being sent to you over established paths. Data will be returned in buffer for length of buflen.

Parameters

pathid	Path identification number.
msgid	Specifies the message ID.
trgcls	Specifies target class.
buffer	Address of buffer to receive.
buflen	Length of buffer to receive.
flags1_out	Contains information about this path. • IPNORPY - 0x10 Specifies this is a one-way message and no reply is expected. • IPPRTY - 0x20 Specifies if you want to send priority message. • IPRMDATA - 0x80 specifies the data is contained in the parameter list
residual_buffer	Address of buffer updated by the number of bytes you have received.
residual_length	Contains one of the following values, if the receive buffer is: • The same length as the message, this field is zero. • Longer than the message, this field contains the number of bytes remaining in the buffer.

	• Shorter than the message, this field contains the residual count (that is, the number of bytes remaining in the message that does not fit into the buffer. In this case b2f0_result = 5.

Returns

Return code from CP IUCV call or -EINVAL (buffer address is pointing to NULL).

iucv_receive_array

```
int iucv_receive_array (u16 pathid,
                        u32 msgid,
                        u32 trgcls,
                        iucv_array_t * buffer,
                        ulong buflen,
                        int *flags1_out,
                        ulong * residual_buffer, ulong * residual_length);
```

This function receives messages that are being sent to you over established paths. Data will be returned in first buffer for length of first buffer.

Parameters

pathid	Path identification number.
msgid	Specifies the message ID.
trgcls	Specifies target class.
buffer	Address of an array of buffers to receive data.
buflen	Total length of buffers.
flags1_out	Contains information about this path. • IPNORPY - 0x10 Specifies this is a one-way message and no reply is expected. • IPPRTY - 0x20 Specifies if you want to send priority message. • IPRMDATA - 0x80 specifies the data is contained in the parameter list
residual_buffer	Address points to the current list entry IUCV is working on.
residual_length	Contains one of the following values, if the receive buffer is: • The same length as the message, this field is zero.

	• Longer than the message, this field contains the number of bytes remaining in the buffer. • Shorter than the message, this field contains the residual count (that is, the number of bytes remaining in the message that does not fit into the buffer. In this case b2f0_result = 5.

Returns

Return code from CP IUCV call or -EINVAL (buffer address is pointing to NULL).

iucv_reject

```
int iucv_reject (u16 pathid, u32 msgid, u32 trgcls);
```

The reject function refuses a specified message. Between the time you are notified of a message and the time that you complete the message, the message may be rejected.

iucv_reply

```
int iucv_reply (u16 pathid,
                u32 msgid,
                u32 trgcls,
                int flags1,
                void *buffer, ulong buflen, ulong * residual_buffer,
                ulong * residual_length);
```

This function responds to the two-way messages that you receive. You must identify completely the message to which you wish to reply. That is, pathid, msgid, and trgcls.

Parameters

pathid	Path identification number.
msgid	Specifies the message ID.
trgcls	Specifies target class.
buffer	Address of reply buffer.
buflen	Length of reply buffer.
flags1_out	Contains information about this path. • IPPRTY - 0x20 Specifies if you want to send priority message.
residual_buffer	Address of buffer updated by the number of bytes you have sent.

residual_length	Contains one of the following values: • If the answer buffer is the same length as the reply, this field contains zero. • If the answer buffer is longer than the reply, this field contains the number of bytes remaining in the buffer. • If the answer buffer is shorter than the reply, this field contains a residual count (that is, the number of bytes remaining in the reply that does not fit into the buffer. In this case b2f0_result = 5.

Returns

Return code from CP IUCV call or -EINVAL (buffer address is pointing to NULL).

iucv_reply_array

```
int iucv_reply_array (u16 pathid,
                      u32 msgid,
                      u32 trgcls,
                      int flags1,
                      iucv_array_t * buffer,
                      ulong buflen, ulong * residual_address,
                      ulong * residual_length);
```

This function responds to the two-way messages that you receive. You must identify completely the message to which you wish to reply. That is, pathid, msgid, and trgcls. The array identifies a list of addresses and lengths of discontiguous buffers that contains the reply data.

Parameters

pathid	Path identification number.
msgid	Specifies the message ID.
trgcls	Specifies target class.
buffer	Address of an array of buffers containing reply data.
buflen	Total length of buffers.
flags1_out	Contains information about this path. • IPPRTY - 0x20 Specifies if you want to send priority message.
residual_buffer	Address points to the current list entry IUCV is working on.
residual_length	Contains one of the following values:

	• If the answer buffer is the same length as the reply, this field contains zero. • If the answer buffer is longer than the reply, this field contains the number of bytes remaining in the buffer. • If the answer buffer is shorter than the reply, this field contains a residual count (that is, the number of bytes remaining in the reply that does not fit into the buffer. In this case b2f0_result = 5.

Returns

Return code from CP IUCV call or -EINVAL (buffer address is pointing to NULL).

iucv_reply_prmmsg

```
int iucv_reply_prmmsg (u16 pathid, u32 msgid,
                       u32 trgcls, int flags1, uchar prmmsg[8]);
```

This function responds to the two-way messages that you receive. You must identify completely the message to which you wish to reply. That is, pathid, msgid, and trgcls. Prmmsg signifies the data is moved into the parameter list.

Parameters

msgid	Specifies the message ID.
trgcls	Specifies target class.
flags1	Option for path. • IPPRTY - 0x20 Specifies if you want to send priority message.
prmmsg	8 bytes of data to be placed into the parameter. list.

Returns

Return code from CP IUCV call.

iucv_resume

```
int iucv_resume (u16 pathid, uchar user_data[16]);
```

This function restores communications over a quiesced path.

Parameters

pathid	Path identification number.
user_data	16 bytes of user data.

Returns

Return code from CP IUCV call.

iucv_send

```
int iucv_send (u16 pathid,
               u32 * msgid,
               u32 trgcls,
               u32 srccls, u32 msgtag, int flags1, void *buffer, ulong buflen);
```

This function transmits data to another application. Data to be transmitted is in a buffer and this is a one-way message and the receiver will not reply to the message.

Parameters

pathid	Path identification number.
msgid	Specifies the message ID.
trgcls	Specifies target class.
srccls	Specifies the source message class.
msgtag	Specifies a tag to be associated with the message.
flags1	Option for path. • IPPRTY - 0x20 Specifies if you want to send priority message.
buffer	Address of send buffer.
buflen	Length of send buffer.

Returns

Return code from CP IUCV call or -EINVAL (buffer address is NULL).

iucv_send2way

```
int iucv_send2way (u16 pathid,
                   u32 * msgid,
                   u32 trgcls,
                   u32 srccls,
                   u32 msgtag,
                   int flags1,
                   void *buffer, ulong buflen, void *ansbuf, ulong anslen);
```

This function transmits data to another application. Data to be transmitted is in a buffer. The receiver of the send is expected to reply to the message anda buffer is provided into which IUCV moves the reply to this message.

Parameters

pathid	Path identification number.
msgid	Specifies the message ID.
trgcls	Specifies target class.
srccls	Specifies the source message class.
msgtag	Specifies a tag to be associated with the message.
flags1	Option for path. • IPPRTY - 0x20 Specifies if you want to send priority message.
buffer	Address of send buffer.
buflen	Length of send buffer.
ansbuf	Address of buffer into which IUCV moves the reply of this message.
anslen	Length reply buffer.

Returns

Return code from CP IUCV call or -EINVAL (buffer address is NULL).

iucv_send2way_array

```
int iucv_send2way_array (u16 pathid,
                         u32 * msgid,
                         u32 trgcls,
                         u32 srccls,
                         u32 msgtag,
                         int flags1,
                         iucv_array_t * buffer,
                         ulong buflen, iucv_array_t * ansbuf, ulong anslen);
```

This function transmits data to another application. The contents of buffer is the address of the array of addresses and lengths of discontiguous buffers that hold the message text. The receiver of the send is expected to reply to the message and a buffer is provided into which IUCV moves the reply to this message.

Returns

pathid	Path identification number.
msgid	Specifies the message ID.

trgcls	Specifies target class.
srccls	Specifies the source message class.
msgtag	Specifies a tag to be associated with the message.
flags1	Option for path. • IPPRTY - 0x20 Specifies if you want to send priority message.
buffer	Address of array of send buffers.
buflen	Total length of send buffers.
ansbuf	Address of array of buffer into which IUCV moves the reply of this message.
anslen	Length reply buffers.

Returns

Return code from CP IUCV call or -EINVAL (buffer address is NULL).

iucv_send_array

```
int iucv_send_array (u16 pathid,
                     u32 * msgid,
                     u32 trgcls,
                     u32 srccls,
                     u32 msgtag,
                     int flags1, iucv_array_t * buffer, ulong buflen);
```

This function transmits data to another application. The contents of buffer is the address of the array of addresses and lengths of discontiguous buffers that hold the message text. This is a one-way message and the receiver will not reply to the message.

Parameters

pathid	Path identification number.
msgid	Specifies the message ID.
trgcls	Specifies target class.
srccls	Specifies the source message class.
msgtag	Specifies a tag to be associated with the message.
flags1	Option for path.

	• IPPRTY - 0x20 Specifies if you want to send priority message.
buffer	Address of send buffers.
buflen	Length of send buffer.

Returns

Return code from CP IUCV call or -EINVAL (buffer address is NULL).

iucv_send2way_prmmsg

```
int iucv_send2way_prmmsg (u16 pathid,
                          u32 * msgid,
                          u32 trgcls,
                          u32 srccls,
                          u32 msgtag,
                          ulong flags1,
                          uchar prmmsg[8], void *ansbuf, ulong anslen);
```

This function transmits data to another application. Prmmsg specifies that the 8-bytes of data are to be moved into the parameter list. This is a two-way message and the receiver of the message is expected to reply. A buffer is provided into which IUCV moves the reply to this message.

Parameters

pathid	Path identification number.
msgid	Specifies the message ID.
trgcls	Specifies target class.
srccls	Specifies the source message class.
msgtag	Specifies a tag to be associated with the message.
flags1	Option for path. • IPPRTY - 0x20 Specifies if you want to send priority message.
prmmsg	8 bytes of data to be placed in parameter list.
ansbuf	Address of buffer into which IUCV moves the reply of this message.
anslen	Length of buffer.

Returns

Return code from CP IUCV call or -EINVAL (buffer address is NULL).

iucv_send2way_prmmsg_array

```
int iucv_send2way_prmmsg_array (u16 pathid,
                                u32 * msgid,
                                u32 trgcls,
                                u32 srccls,
                                u32 msgtag,
                                int flags1,
                                uchar prmmsg[8],
                                iucv_array_t * ansbuf, ulong anslen);
```

This function transmits data to another application. Prmmsg specifies that the 8-bytes of data are to be moved into the parameter list. This is a two-way message and the receiver of the message is expected to reply. A buffer is provided into which IUCV moves the reply to this message. The contents of ansbuf is the address of the array of addresses and lengths of discontiguous buffers that contain the reply.

Parameters

pathid	Path identification number.
msgid	Specifies the message ID.
trgcls	Specifies target class.
srccls	Specifies the source message class.
msgtag	Specifies a tag to be associated with the message.
flags1	Option for path. • IPPRTY - 0x20 Specifies if you want to send priority message.
prmmsg	8 bytes of data to be placed in parameter list.
ansbuf	Address of array of buffer into which IUCV moves the reply of this message.
anslen	Lengths of buffer.

Returns

Return code from CP IUCV call or -EINVAL (buffer address is NULL).

iucv_send_prmmsg

```
int iucv_send_prmmsg (u16 pathid,
                      u32 * msgid,
                      u32 trgcls,
                      u32 srccls, u32 msgtag, int flags1, uchar prmmsg[8]);
```

This function transmits data to another application. Prmmsg specifies that the 8-bytes of data are to be moved into the parameter list. This is a one-way message and the receiver will not reply to the message.

Parameters

pathid	Path identification number.
msgid	Specifies the message ID.
trgcls	Specifies target class.
srccls	Specifies the source message class.
msgtag	Specifies a tag to be associated with the message.
flags1	Option for path. • IPPRTY - 0x20 Specifies if you want to send priority message.
prmmsg	8 bytes of data to be placed in parameter list.

Returns

Return code from CP IUCV call.

iucv_setmask

```
int iucv_setmask (int SetMaskFlag);
```

This function enables or disables the following IUCV external interruptions: Nonpriority and priority message interrupts, nonpriority and priority reply interrupts.

Parameters

SetMaskFlag - options for interrupts:

- 0x80 - Nonpriority_MessagePendingInterruptsFlag
- 0x40 - Priority_MessagePendingInterruptsFlag
- 0x20 - Nonpriority_MessageCompletionInterruptsFlag
- 0x10 - Priority_MessageCompletionInterruptsFlag
- 0x08 - IUCVControlInterruptsFlag

iucv_sever

```
int iucv_sever (u16 pathid, uchar user_data[16]);
```

This function terminates an IUCV path.

Parameters

pathid	Path identification number.
user_data	16 bytes of user data.

Returns

Return code from CP IUCV call or -EINVAL (interal error, wild pointer).

Iucv_register_program

```
iucv_handle_t iucv_register_program (uchar pgmname[16],
                                     uchar userid[8],
                                     uchar pgmmask[24],
                                     iucv_interrupt_ops_t * ops,
                                     void *pgm_data);
```

To explore any of the IUCV functions, you must first register your program using `iucv_register_program()` API. Once your program has successfully completed a register, it can exploit the other functions.

Parameters

pgmname	User identification
userid	Machine identification
pgmmask	Indicates which bits in the pgmname and userid combined will be used to determine who is given control.
ops	Address of interrupt handler table.
pgm_data	Application data to be passed to interrupt handlers.

Returns

The address of handler, or NULL on failure.

Notes

For pgmmask:

- If pgmname, userid and pgmmask are provided, pgmmask is entered into the handler as is.
- If pgmmask is NULL, the internal mask is set to all 0xff's
- When userid is NULL, the first 8 bytes of the internal mask are forced to 0x00.
- If pgmmask and userid are NULL, the first 8 bytes of the internal mask are forced to 0x00 and the last 16 bytes to 0xff.

ops is a vector of functions that handle IUCV interrupts:

```
typedef struct {
        void (*ConnectionPending) (iucv_ConnectionPending * eib,
                                   void *pgm_data);
        void (*ConnectionComplete) (iucv_ConnectionComplete * eib,
                                    void *pgm_data);
        void (*ConnectionSevered) (iucv_ConnectionSevered * eib,
                                   void *pgm_data);
        void (*ConnectionQuiesced) (iucv_ConnectionQuiesced * eib,
                                    void *pgm_data);
        void (*ConnectionResumed) (iucv_ConnectionResumed * eib,
                                   void *pgm_data);
        void (*MessagePending) (iucv_MessagePending * eib, void *pgm_data);
        void (*MessageComplete) (iucv_MessageComplete * eib, void *pgm_data);
} iucv_interrupt_ops_t;
```

The parameter list for these functions is defined as follows:

eib	A pointer to a 40-byte area described with one of the structures above.
pgm_data	This data is strictly for the interrupt handler that is passed by the application. This may be an address or token.

iucv_unregister_program

```
int iucv_unregister_program (iucv_handle_t handle);
```

Use this API to unregister your application with IUCV.

Parameters

handle	Address of handler

Returns

0	Normal return
-EINVAL	Internal error, wild pointer