Hier kommt eine Figure 9.1

9.1 Dispatching Interrupts

Vectored Interrupts (1)

- The 11/2 processor uses *vectored interrupts* to handle exceptions and interrupts from external devices.
- Whenever an external device needs to communicate with the processor
  - it places a signal on the interrupt bus line
  - if the processor is running with interrupts enabled:
    its checks this line after every time an instruction has been executed to see whether an interrupt needs processing
  - to handle the interrupt:
    the processor sends an acknowledgement over the interrupt bus line
  - the first external device with an pending request receives the acknowledgement and return its interrupt vector address v along the bus
  - when the processor receive v:
    * the processor pushes the current PC and PS on the stack
    * loads the PC and PS from tow words in memory
    * starts at location v- see figure 9.1
    * continues executing instructions at the new location.

HIER KOMMT EINE FIGURE

Vectored Interrupts (2)

- Each external device is assigned an unique interrupt vector address v before it is inserted into the bus, thus enabling the system to distinguish among them
- J.e., every device is identified by its unique interrupt vector address$v$
- The interrupt vector locations contain a valid PC and PS.
  - This PC points to an interrupt handling routine (interrupt handler) for the interrupting device
- The processor execute the code of the interrupt routine, returning to the place where the user's program has interrupted
- To make the interrupt transparent (i.e., invisible) to the running, program
  - the interrupt routine saves and restore the stack of the machine - the values of R0-R6 (since HW saves R7(PC) and PS)

Vectored Interrupts (3)

- In practise SP(=R6) needs no saving provided the interrupt routine pops off the stack whenever it pushed the stack before returning (i.e., the stack is restored to its original position)
- To prevent the interrupt routine from being interrupted itself by another devices,
  - the PS stored in the interrupt vector specifying a priority level of 7.
  - Thus interrupt processing begins with interrupts disabled
- Interrupt processing ends by executing a return from interrupt instruction:
  - `rti` or `rtt`,
  - restoring ,in a single step, the old PC and PS from the stack
  - and returning the stack pointer SP to its original value

Handling exceptions

- Exceptions are handled similarly, except that they can never be turned off (i.e., disabled)
- Examples of exceptions occur
  - when nonexistent memory is addressed
  - upon an array-bound error
  - execution of invalid code
  - power failure

9.2 Input And Output Interrupt Dispatchers

- Devices connect to computer system through hardware mechanisms called **controllers**
- **Controllers** reside on boards plugged into the system bus; their hardware converts digital data to wave forms and signals controlling peripheral devices s.a. terminals and disks
- Xinu allows both input and output operations to the **console** device
  - with output to console displayed on screen
  - input from console read from keyboard
- input interrupts for characters typed on a terminal's keyboard trap t a different vector from output interrupts (for chars. sent to the display)
- See figure 9.1

HIER KOMMT EINE FUGURE (9.7)

When interrupts occurs

- old `PC` and `PS` stacked
- new `PC` and `PS` loaded from interrupt vector
- new `PC` points to interrupt dispatch routine
- The interrupt dispatch routine :
  - Saves copy of `PS` (because condition codes change lower-order bits of register)
  - masks off low-order 4 bits to compute an index into a global interrupt dispatch table witch contains the addresses of the hight-level interrupt handlers for each connected device
  - See the code for *intmap* on later slide.
  - I/O interrupts routines receive an argument, identifying the interrupting device
  - I/O interrupt routines defined in *intmap* are *getchar(), putchar(ch) fgetchar(), fputchar(ch).*
  - I/O interrupt dispatchers contains in file *ioint.s*

HIER KOMMT EINE FUGRE (9.9)

After returning from interrupt routine

HIER KOMMT EINE FUGURE (9.10)

```
/* io.h - fgetc, fputc, getchar, isbaddev, putchar */

#define INTVECI inint  /* input interrupt dispatch routine   */
#define INTVECO outint /* output interrupt dispatch routine  */
extern  int     INTVECI();
extern  int     INTVECO();

struct  intmap {      /* device-to-interrupt routine mapping */
               /* address of input interrupt to input routine*/
        int     (*iin)();
        int     icode;   /* argument passed to input routine */
                     /* address of output interrupt routine */
        int     (*iout)();
        int     code;  /* arguments passed to output routine */
        };
#define NDEVS
extern  struct  intmap intmap[NDEVS];
#define isbaddev(f)    ( (f)<0 || (f)>=NDEVS )
#endif

/* In-lines I/O procedures */

#define getchar()      getc(CONSOLE)
#define putchar()      putc(CONSOLE, (ch))
#define fgetc(unit)    getc((unit))
```

```
#define fputc(unit,ch) putc((unit),(ch))

struct vector {
       char   *vproc;   /* address of interrupt procedure */
       int    vps;      /* saved process status word      */
       };
```

```
/* ioint.s - inint, outint */
/* I/P interrupts trap here. Original PC and PS are on top
/* of the stack upon entry. Low order 4 bits of the current
/* PS contain the device decriptor, Inerrupts are disabled */

        .globl    _inint,_outint,_intmap
_outint:                        / Output interrupt entry point
        mfps      -(sp)         / Save device descriptor from PS
        mov       r0,-(sp)      / Save r0 (csv does not)
                                / point r0 to output in intmap
mov       $_intmap+4,r0
        br        ioint         / Go do common part of code

_inint:                         / Input interrupt entry point
        mfps      -(sp)         / Save device code from PS
        mov       r0,-(sp)      / Save r0 (csv does not)
        mov       $_intmap,r0   / Point r0 to input in intmap

ioint:                          / Code common to input & output
        mov       r1,-(sp)      / Save r1 (csv does not)
        mov       4(sp),r1      / Get saved PS in r1
        bic       $177760,r1    / Mask off device descriptor
        ash       $3,r1         / pick correct entry in intmap
        add       r1,r0         / Form pointer to intmap entry
        mov       2(r0),-(sp)   / Push pointer to intmap entry
```

```
jsr     pc,*(r0)     / Call interrupt routine
mov     2(sp),r1     / Restore r1 and R0 from stack
mov     4(sp),r0
add     $8,sp        / Pop arg, saved r0, r1 and PS
rtt                  / Return from interrupt
```

9.3 The Rules for Interrupt Processing

1. Interrupt routines examine and modify global data structure s.a I/O buffers, so interference from over processes must be prevented.

   This is ess. done in tow ways :

   - disabling interrupts, with interrupts still disabled after interrupt routine returns; so only after *ioint* returns interrupts enabled again
   - disabling interrupts, but: high level interrupt routines may enable interrupts by calling *resched*.

     In case CPU switches to a process with interrupt enabled ! (*send* called, starting the receiver waiting for a message, or *signal* making first process on waiting queue ready to run)
     - interrupt routines should leave global data in a valid state before calling *resched()*
     - No process enables interrupts unless it previously disabled them

2. Interrupts should not be disabled too long.
   If thy do, devices will fail to perform correctly, e.g., when processor does not accept character from an I/O device before the next one arrives, data will be lost

   Interrupt routines must be designed to enable further interrupts as quickly as possible

3. Interrupt code is executed by whatever process is running when the interrupt occurs.
   This is especially the NULL process.

   Recall that *resched()* requires that there's at least one process ready to run, so the NULL process must always be current or ready

   Interrupt routines can only call procedures that leave the executing process in the current or ready states.