Embedded Systems Programming - PA8001

http://bit.ly/15mmqf7

Lecture 2

Mohammad Mousavi

m.r.mousavi@hh.se

HALMSTAD UNIVERSITY

For the Development of Organisations, Products and Quality of Life

Center for Research on Embedded Systems

School of Information Science, Computer and Electrical Engineering

The preprocessor

Source manipulation before compilation

Macro expansion

Textually replace definitions.

File insertion

Include files as if you had written the code in your files.

Instructions to the compiler

For example not to compile certain parts of the program.

The preprocessor

Source manipulation before compilation

Macro expansion

Textually replace definitions.

File insertion

Include files as if you had written the code in your files.

Instructions to the compiler

For example not to compile certain parts of the program.

The preprocessor

Source manipulation before compilation

Macro expansion

Textually replace definitions.

File insertion

Include files as if you had written the code in your files.

Instructions to the compiler

For example not to compile certain parts of the program.

The preprocessor

Source manipulation before compilation

Macro expansion

Textually replace definitions.

File insertion

Include files as if you had written the code in your files.

Instructions to the compiler

For example not to compile certain parts of the program.

Preprocessing: macros

The program . . .

```
#define SIZE 5
#define init(v) x=v;y=v;z=v
main(){
  int x,y,z;
  init(SIZE);
}
```

becomes

```
main(){
  int x,y,z;
  x=5;y=5;z=5;
}
```

before compiling.

Preprocessing: macros

The program . . .

```
#define SIZE 5
#define init(v) x=v;y=v;z=v
main(){
  int x,y,z;
  init(SIZE);
}
```

becomes

```
main(){
  int x,y,z;
  x=5;y=5;z=5;
}
```

before compiling.

Preprocessing: including files

Larger programs organized in files
Separate interfaces and implementations in header and impl.
Preprocessor instructions to include header files.

```
typedef struct {int x;int y;} Pt;
#define initPoint(a,b)  { a, b }
double distance0 (Pt *p1);
```

point.h

```
#include "point.h"
#include <math.h>
double distance0 (Pt *p1){
  return sqrt(p1->x*p1->x + p1->y*p1->y);
}
```

point.c

Preprocessing: including files

Larger programs organized in files
Separate interfaces and implementations in header and impl.
Preprocessor instructions to include header files.

point.h

```
typedef struct {int x;int y;} Pt;
#define initPoint(a,b)  { a, b }
double distance0 (Pt *p1);
```

point.c

```
#include "point.h"
#include <math.h>
double distance0 (Pt *p1){
  return sqrt(p1->x*p1->x + p1->y*p1->y);
}
```

Preprocessing: including files

- Larger programs organized in files
- Separate interfaces and implementations in header and impl.
- Preprocessor instructions to include header files.

point.h

```
typedef struct {int x;int y;} Pt;
#define initPoint(a,b)  { a, b }
double distanceO (Pt *p1);
```

point.c

```
#include "point.h"
#include <math.h>
double distanceO (Pt *p1){
  return sqrt(p1->x*p1->x + p1->y*p1->y);
}
```

Preprocessing: including files

- Larger programs organized in files
- Separate interfaces and implementations in header and impl.
- Preprocessor instructions to include header files.

point.h

```c
typedef struct {int x;int y;} Pt;
#define initPoint(a,b)  { a, b }
double distance0 (Pt *p1);
```

point.c

```c
#include "point.h"
#include <math.h>
double distance0 (Pt *p1){
  return sqrt(p1->x*p1->x + p1->y*p1->y);
}
```

Preprocessing: including files

Programs can now use points as follows

The program ...

```
#include "point.h"
#include <stdio.h>
main(){
  Pt p = initPoint(3,4);
  printf("%f\n",
         distanceO(&p));
}
```

becomes

```
typedef struct {int x;int y;} Pt;
double distanceO (Pt *p1);
main(){
  Pt p = { 3, 4 };
  printf("%f\n",distanceO(&p));
}
```

after preprocessor (I do not show the expansion of stdio.h!)

Preprocessing: including files

Programs can now use points as follows

The program . . .

```
#include "point.h"
#include <stdio.h>
main(){
  Pt p = initPoint(3,4);
  printf("%f\n",
         distance0(&p));
}
```

becomes

```
typedef struct {int x;int y;} Pt;
double distance0 (Pt *p1);
main(){
  Pt p = { 3, 4 };
  printf("%f\n",distance0(&p));
}
```

after preprocessor (I do not show the expansion of `stdio.h`!)

Preprocessing: including files

Programs can now use points as follows

The program ...

```
#include "point.h"
#include <stdio.h>
main(){
  Pt p = initPoint(3,4);
  printf("%f\n",
            distance0(&p));
}
```

becomes

```
typedef struct {int x;int y;} Pt;
double distance0 (Pt *p1);
main(){
  Pt p = { 3, 4 };
  printf("%f\n",distance0(&p));
}
```

after preprocessor (I do not show the expansion of `stdio.h`!)

Compiling

Separate compilation

Even without a main, an object file can be generated

```
gcc -c point.c
```

will generate `point.o`, to be linked to form an executable.

Compilation

When compiling main program, provide the object files:

```
gcc usepoints.c point.o
```

Compiling

Separate compilation

Even without a main, an object file can be generated

```
gcc -c point.c
```

will generate `point.o`, to be linked to form an executable.

Compilation

When compiling main program, provide the object files:

```
gcc usepoints.c point.o
```

Preprocessing: instructions to the compiler

Compiling different versions of programs (for different platforms or including debugging printouts)

The program ...

```
#include "debug.h"
#include <stdio.h>
main(){
  #ifdef DEBUG
  printf("in debug mode");
  #endif
  printf("what has to
         be done ...");
}
```

Two programs, depending on the content of debug.h

Preprocessing: instructions to the compiler

Compiling different versions of programs (for different platforms or including debugging printouts)

The program ...

```
#include "debug.h"
#include <stdio.h>
main(){
   #ifdef DEBUG
  printf("in debug mode");
   #endif
  printf("what has to
          be done ...");
}
```

Two programs, depending on the content of debug.h

Preprocessing: instructions to the compiler

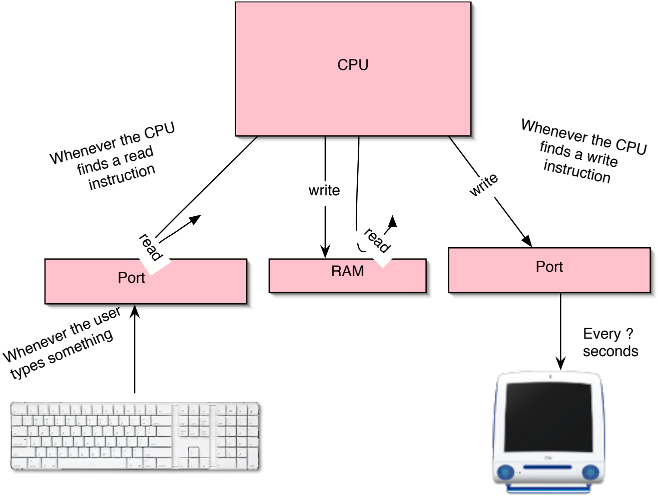
Compiling different versions of programs (for different platforms or including debugging printouts)

The program ...

```
#include "debug.h"
#include <stdio.h>
main(){
   #ifdef DEBUG
   printf("in debug mode");
   #endif
   printf("what has to
            be done ...");
}
```

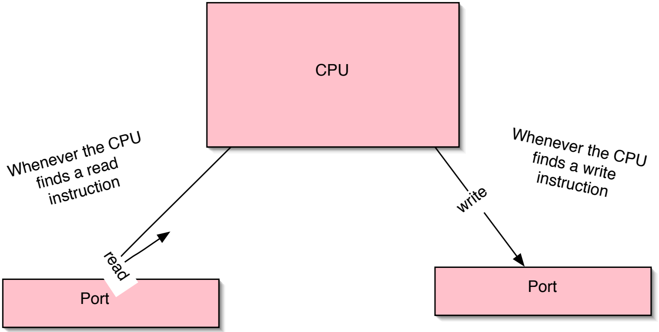
Two programs, depending on the content of `debug.h`

The naked computer



The naked computer

How to read from and write to IO ports (synchronization to be discussed later on)



IO hardware

Access via a set of registers, both to control the device operation and for data transfer; 2 general architecture:

Memory mapped

Some addresses reserved for device registers; typically with names defined in a platform-specific header file.

Separate bus

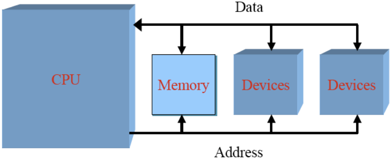
Different assembler instructions for memory access and for device registers

IO hardware

Access via a set of registers, both to control the device operation and for data transfer; 2 general architecture:

Memory mapped

Some addresses reserved for device registers; typically with names defined in a platform-specific header file.



Separate bus

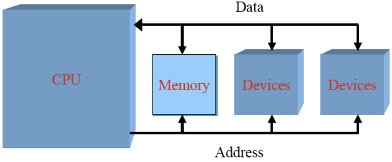
Different assembler instructions for memory access and for device registers

IO hardware

Access via a set of registers, both to control the device operation and for data transfer; 2 general architecture:

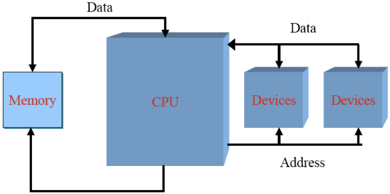
Memory mapped

Some addresses reserved for device registers; typically with names defined in a platform-specific header file.



Separate bus

Different assembler instructions for memory access and for device registers



Memory mapped – things to think about

The documentation of a microprocessor provides the addresses corresponding to ports. Addresses can be used as pointers. The type of the pointers depends on the size of the port.

```
char * port1; // 8 bits
int  * port2; // 16 bits
```

Use unsigned to avoid confusions with signed values!

Reading and writing is done as with ordinary variables

```
*port1 // read
*port1 = value; // write
```

Would you do this in a program?

```
*port = x; x = *port;
```

Yes if it is IO! The compiler should not optimize this away:

```
int * port;
```

Memory mapped – things to think about

The documentation of a microprocessor provides the addresses corresponding to ports. Addresses can be used as pointers. The type of the pointers depends on the size of the port.

```
char * port1; // 8 bits
int  * port2; // 16 bits
```

Use unsigned to avoid confusions with signed values!

Reading and writing is done as with ordinary variables

```
*port1 // read
*port1 = value; // write
```

Would you do this in a program?

```
*port = x; x = *port;
```

Yes if it is IO! The compiler should not optimize this away:

```
volatile int * port;
```

Memory mapped – things to think about

The documentation of a microprocessor provides the addresses corresponding to ports. Addresses can be used as pointers. The type of the pointers depends on the size of the port.

```
char * port1; // 8 bits
int  * port2; // 16 bits
```

Use unsigned to avoid confusions with signed values!

Reading and writing is done as with ordinary variables

```
*port1 // read
*port1 = value; // write
```

Would you do this in a program?

```
*port = x; x = *port;
```

Yes if it is IO! The compiler should not optimize this away:

```
volatile int * port;
```

Memory mapped – things to think about

The documentation of a microprocessor provides the addresses corresponding to ports. Addresses can be used as pointers. The type of the pointers depends on the size of the port.

```
char * port1; // 8 bits
int  * port2; // 16 bits
```

Use unsigned to avoid confusions with signed values!

Reading and writing is done as with ordinary variables

```
*port1 // read
*port1 = value; // write
```

Would you do this in a program?

```
*port = x; x = *port;
```

Yes if it is IO! The compiler should not optimize this away:

```
volatile int * port;
```

Memory mapped – things to think about

The documentation of a microprocessor provides the addresses corresponding to ports. Addresses can be used as pointers. The type of the pointers depends on the size of the port.

```
char * port1; // 8 bits
int  * port2; // 16 bits
```

Use unsigned to avoid confusions with signed values!

Reading and writing is done as with ordinary variables

```
*port1 // read
*port1 = value; // write
```

Would you do this in a program?

```
*port = x; x = *port;
```

Yes if it is IO! The compiler should not optimize this away:

```
volatile int * port;
```

Memory Mapped – more things to think about!

Addresses and ports

Two registers might be mapped to the same address: one supposed to be read from (like checking device status) and another to write to (like giving commands to a device).

example

```
#define IS_READY   (1 << 5)
#define CONVERT    (1 << 5)
#define STATUS_REG  *((char*)0x34c)
#define CMD_REG     *((char*)0x34c)

if (STATUS_REG & IS_READY) {CMD_REG = CONVERT;}
```

Potential problem

```
CMD_REG = CMD_REG | CONVERT;
```

Memory Mapped – more things to think about!

Addresses and ports

Two registers might be mapped to the same address: one supposed to be read from (like checking device status) and another to write to (like giving commands to a device).

example

```
#define IS_READY   (1 << 5)
#define CONVERT    (1 << 5)
#define STATUS_REG  *((char*)0x34c)
#define CMD_REG     *((char*)0x34c)

if (STATUS_REG & IS_READY) {CMD_REG = CONVERT;}
```

Potential problem

```
CMD_REG = CMD_REG | CONVERT;
```

Memory Mapped – more things to think about!

Addresses and ports

Two registers might be mapped to the same address: one supposed to be read from (like checking device status) and another to write to (like giving commands to a device).

example

```
#define IS_READY   (1 << 5)
#define CONVERT    (1 << 5)
#define STATUS_REG  *((char*)0x34c)
#define CMD_REG     *((char*)0x34c)

if (STATUS_REG & IS_READY) {CMD_REG = CONVERT;}
```

Potential problem

```
CMD_REG = CMD_REG | CONVERT;
```

Shadowing

These registers are better used via a shadow variable (another address! instead of just a def!)

example

```
#define CONVERT (1<<5)
#define CMD_REG *((char *)0x34c)
char cmd_shadow;
...
cmd_shadow = cmd_shadow | CONVERT;
CMD_REG = cmd_shadow;
```

Notice

All changes to CMD_REG should be reflected in cmd_shadow!

Fortunately all ports in AVR are read/write! (No shadowing!).

Shadowing

These registers are better used via a shadow variable (another address! instead of just a def!)

example

```
#define CONVERT (1<<5)
#define CMD_REG *((char *)0x34c)
char cmd_shadow;
...
cmd_shadow = cmd_shadow | CONVERT;
CMD_REG = cmd_shadow;
```

Notice

All changes to CMD_REG should be reflected in cmd_shadow!

Fortunately all ports in AVR are read/write! (No shadowing!).

Shadowing

These registers are better used via a shadow variable (another address! instead of just a def!)

example

```
#define CONVERT (1<<5)
#define CMD_REG *((char *)0x34c)
char cmd_shadow;
...
cmd_shadow = cmd_shadow | CONVERT;
CMD_REG = cmd_shadow;
```

Notice

All changes to `CMD_REG` should be reflected in `cmd_shadow`!

Fortunately all ports in AVR are read/write! (No shadowing!).

Shadowing

These registers are better used via a shadow variable (another address! instead of just a def!)

example

```
#define CONVERT (1<<5)
#define CMD_REG *((char *)0x34c)
char cmd_shadow;
...
cmd_shadow = cmd_shadow | CONVERT;
CMD_REG = cmd_shadow;
```

Notice

All changes to `CMD_REG` should be reflected in `cmd_shadow`!

Fortunately all ports in AVR are read/write! (No shadowing!).

Misc

Single write

It is not always needed to read the value of the port when doing a modification. In some cases you know exactly what value should be written to the port.

```
#define CTRL (1<<3)
#define SIZE1 (1<<4)
#define SIZE2 (2<<4)
#define FLAG (1<<6)
CMD_REG = FLAG | SIZE2 | CTRL;
```

Separate I/O Bus

The port registers are accessed via special assembler instructions, usually made available to a C program as preprocessor macros.

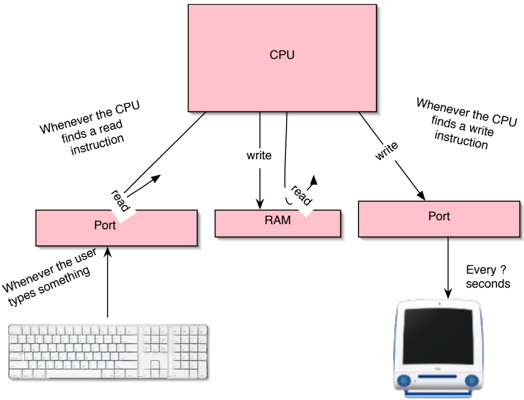
QNX real-time OS

Macros like `in8, out8, in16, out16` that are used as in

```
unsigned char val = in8(0x30d);
out32(0xf4,expr);
```

As you see, they cannot be used as ordinary variables!

I/O Synchronisation



How does the software become aware of changes in the key status?

2 models

- interrupt driven (more on this later in the course)
- status driven (today and lab1)

Busy Waiting

In the status driven model the CPU polls the status registers until a change occurs

Example

```
int old = KEY_STATUS_REG;
int val = old;
while(old==val){
  val = KEY_STATUS_REG;
}
```

On leaving the loop the status has changed!

The CPU is busy but is doing nothing useful!

The CPU has no control over when to exit the loop! What if KEY_STATUS_REG were an ordinary variable?

Busy Waiting

In the status driven model the CPU polls the status registers until a change occurs

Example

```
int old = KEY_STATUS_REG;
int val = old;
while(old==val){
  val = KEY_STATUS_REG;
}
```

On leaving the loop the status has changed!

The CPU is busy but is doing nothing useful!

The CPU has no control over when to exit the loop! What if KEY_STATUS_REG were an ordinary variable?

Busy Waiting

In the status driven model the CPU polls the status registers until a change occurs

Example

```
int old = KEY_STATUS_REG;
int val = old;
while(old==val){
  val = KEY_STATUS_REG;
}
```

On leaving the loop the status has changed!

The CPU is busy but is doing nothing useful!

The CPU has no control over when to exit the loop! What if KEY_STATUS_REG were an ordinary variable?

Busy Waiting

In the status driven model the CPU polls the status registers until a change occurs

Example

```
int old = KEY_STATUS_REG;
int val = old;
while(old==val){
  val = KEY_STATUS_REG;
}
```

On leaving the loop the status has changed!

The CPU is busy but is doing nothing useful!

The CPU has no control over when to exit the loop! What if KEY_STATUS_REG were an ordinary variable?

Busy waiting

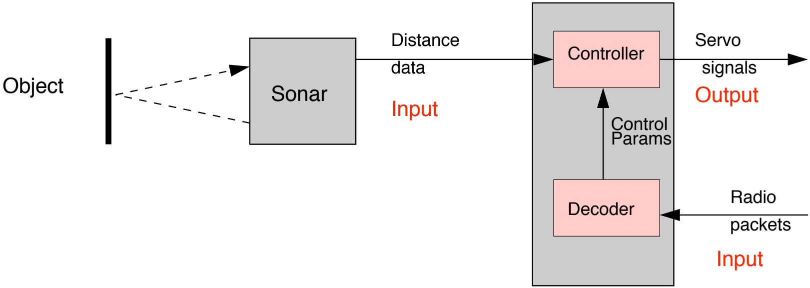
Why is it so appealing?

It can be used to define functions that make input look like reading variables (reading from memory!)

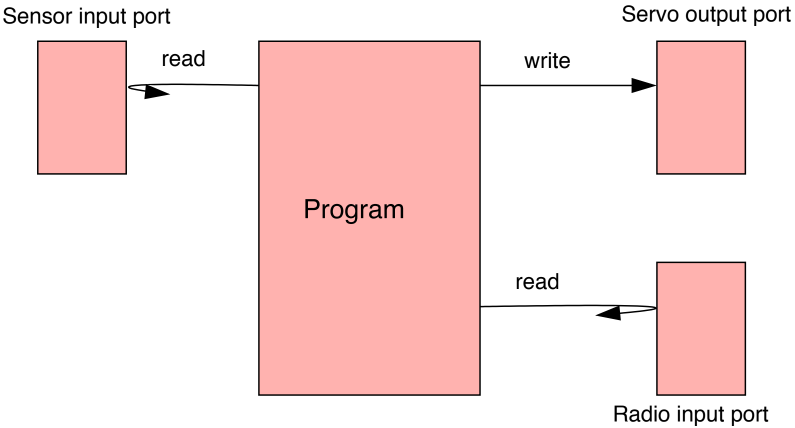
```
char getchar(){
  while(KEY_STATUS_REG & PRESSED);
  while(!(KEY_STATUS_REG & PRESSED));
  return KEY_VALUE_REG;
}
```

A simple embedded system

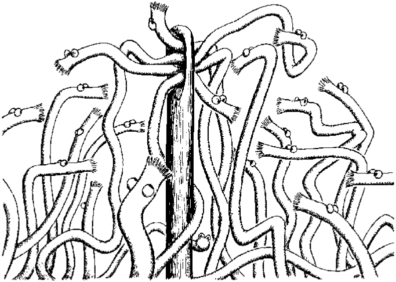
Follow (track) an object using sonar echoes. Control parameters are sent over wireless. The servo controls wheels.



The view from the processor



The program



We will go through a series of attempts to organize the program leading to the need for threads.

Next lecture

We discuss new problems that arise because of programming with threads.

Next lectures

Implementing threads.

The program: busy waiting input

```
int sonar_read(){
   while(SONAR_STATUS & READY == 0);
   return SONAR_DATA;
}
```

We can define *functions*. that create an *illusion* to the rest of the program!

```
void radio_read(struct Packet *pkt){
  while(RADIO_STATUS & READY == 0);
  pkt->v1 = RADIO_DATA1;
  ...
  pkt->vn = RADIO_DATAn;
}
```

We have assumed input ports that automatically reset status when data is read.

The program: busy waiting input

```
int sonar_read(){
    while(SONAR_STATUS & READY == 0);
    return SONAR_DATA;
}
```

We can define *functions*. that create an *illusion* to the rest of the program!

```
void radio_read(struct Packet *pkt){
    while(RADIO_STATUS & READY == 0);
    pkt->v1 = RADIO_DATA1;
    ...
    pkt->vn = RADIO_DATAn;
}
```

We have assumed input ports that automatically reset status when data is read.

The program: busy waiting input

```
int sonar_read(){
   while(SONAR_STATUS & READY == 0);
   return SONAR_DATA;
}
```

We can define *functions*. that create an *illusion* to the rest of the program!

```
void radio_read(struct Packet *pkt){
  while(RADIO_STATUS & READY == 0);
  pkt->v1 = RADIO_DATA1;
  ...
  pkt->vn = RADIO_DATAn;
}
```

We have assumed input ports that automatically reset status when data is read.

The program: busy waiting input

```
int sonar_read(){
   while(SONAR_STATUS & READY == 0);
   return SONAR_DATA;
}
```

We can define *functions*. that create an *illusion* to the rest of the program!

```
void radio_read(struct Packet *pkt){
  while(RADIO_STATUS & READY == 0);
  pkt->v1 = RADIO_DATA1;
  ...
  pkt->vn = RADIO_DATAn;
}
```

We have assumed input ports that automatically reset status when data is read.

The program: output

```
void servo_write(int sig){
  SERVO_DATA = sig;
}
```

The program: algorithms

Control

```
void control(int dist, int *sig, struct Params *p);
```

Calculates the servo signal.

Decode

```
void decode(struct Packet *pkt, struct Params *p)
```

Decodes a packet and calculates new control parameters

The program: algorithms

Control

```
void control(int dist, int *sig, struct Params *p);
```

Calculates the servo signal.

Decode

```
void decode(struct Packet *pkt, struct Params *p)
```

Decodes a packet and calculates new control parameters

The program: algorithms

Control

```
void control(int dist, int *sig, struct Params *p);
```

Calculates the servo signal.

Decode

```
void decode(struct Packet *pkt, struct Params *p)
```

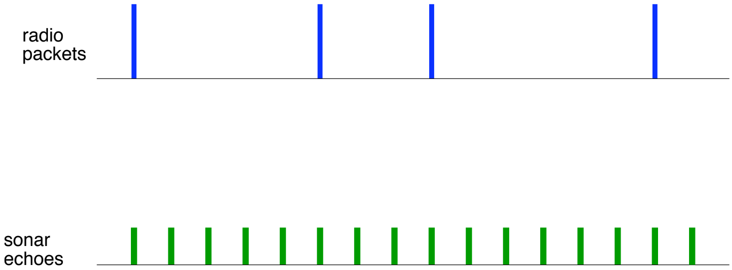
Decodes a packet and calculates new control parameters

The program: a first attempt

```
main(){
    struct Params params;
    struct Packet packet;
    int dist, signal;
    while(1){
      dist = sonar_read();
      control(dist, &signal, &params);
      servo_write(signal);

      radio_read(&packet);
      decode(&packet,&params);
    }
}
```

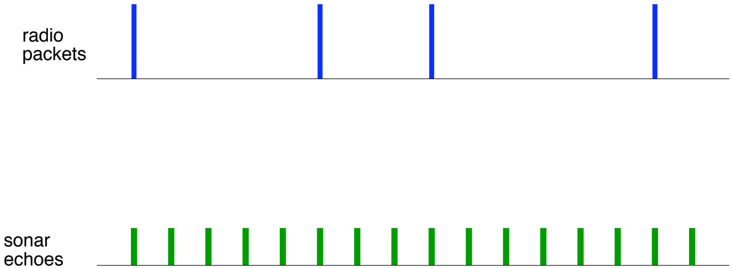
Problems?



We do not know what port will have new data next! The sonar and the radio generate events that are unrelated to each other!

Our program will ignore all events of one kind that happen while busy waiting for the other event!

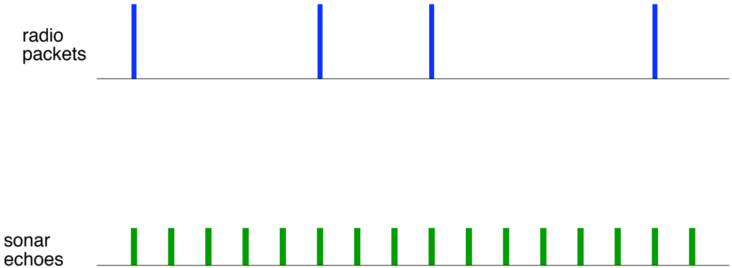
Problems?



We do not know what port will have new data next! The sonar and the radio generate events that are unrelated to each other!

Our program will ignore all events of one kind that happen while busy waiting for the other event!

Problems?



We do not know what port will have new data next! The sonar and the radio generate events that are unrelated to each other!

Our program will ignore all events of one kind that happen while busy waiting for the other event!

The problem explained

RAM and files vs. external input

- Data is already in place (. . . radio packets are not!)
- Even if there might be reasons for waiting, like for the disk head moving to point to the right sector, contents does not have to be created!
- They *produce* data only because they are asked to (. . . remote transmitters act on their own!)

The *illusion* that input is like reading from memory while blocking waiting for data requires that we choose the source of input before blocking!

The problem explained

RAM and files vs. external input

- Data is already in place (. . . radio packets are not!)
- Even if there might be reasons for waiting, like for the disk head moving to point to the right sector, contents does not have to be created!
- They *produce* data only because they are asked to (. . . remote transmitters act on their own!)

The *illusion* that input is like reading from memory while blocking waiting for data requires that we choose the source of input before blocking!

The problem explained

RAM and files vs. external input

- Data is already in place (...radio packets are not!)
- Even if there might be reasons for waiting, like for the disk head moving to point to the right sector, contents does not have to be created!
- They *produce* data only because they are asked to (...remote transmitters act on their own!)

The *illusion* that input is like reading from memory while blocking waiting for data requires that we choose the source of input before blocking!

The problem explained

RAM and files vs. external input

- Data is already in place (. . . radio packets are not!)
- Even if there might be reasons for waiting, like for the disk head moving to point to the right sector, contents does not have to be created!
- They *produce* data only because they are asked to (. . . remote transmitters act on their own!)

The *illusion* that input is like reading from memory while blocking waiting for data requires that we choose the source of input before blocking!

The problem explained

RAM and files vs. external input

- Data is already in place (. . . radio packets are not!)
- Even if there might be reasons for waiting, like for the disk head moving to point to the right sector, contents does not have to be created!
- They *produce* data only because they are asked to (. . . remote transmitters act on their own!)

The *illusion* that input is like reading from memory while blocking waiting for data requires that we choose the source of input before blocking!

The program: a second attempt

```
while(1){
 if(SONAR_STATUS & READY){
   dist = SONAR_DATA;
   control(dist,&signal,&params);
   servo_write(signal);
 }
 if(RADIO_STATUS & READY){
   packet->v1 = RADIO_DATA1;
    ...;
   packet->v2 = RADIO_DATAn;
   decode(&packet,&params);
 }
}
```

Destroy the functions for reading and have *only* *one* busy waiting loop!

Centralized busy waiting

- The new implementation checks both status registers in one big busy-waiting loop. This avoids waiting for the wrong input.
- We destroyed the simple read operations! VERY not modular!

100% CPU usage, no matter how frequent input data arrives.

Try to make the main loop run less often!

Centralized busy waiting

- The new implementation checks both status registers in one big busy-waiting loop. This avoids waiting for the wrong input.
- We destroyed the simple read operations! VERY not modular!

100% CPU usage, no matter how frequent input data arrives.

Try to make the main loop run less often!

Centralized busy waiting

- The new implementation checks both status registers in one big busy-waiting loop. This avoids waiting for the wrong input.
- We destroyed the simple read operations! VERY not modular!

100% CPU usage, no matter how frequent input data arrives.

Try to make the main loop run less often!

Centralized busy waiting

- The new implementation checks both status registers in one big busy-waiting loop. This avoids waiting for the wrong input.
- We destroyed the simple read operations! VERY not modular!

100% CPU usage, no matter how frequent input data arrives.

Try to make the main loop run less often!

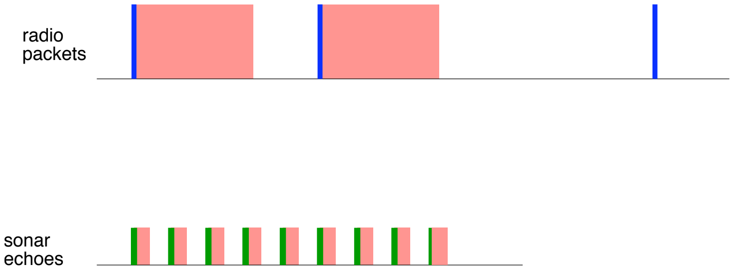
The program: a third attempt

The cyclic executive

```
while(1){
  sleep_until_next_timer_interrupt();
  if(SONAR_STATUS & READY){
     dist = SONAR_DATA;
     control(dist,&signal,&params);
     servo_write(signal);
  }
  if(RADIO_STATUS & READY){
     packet->v1 = RADIO_DATA1;
     ...;
     packet->v2 = RADIO_DATAn;
     decode(&packet,&params);
  }
}
```

The CPU runns at a fixed rate! The timer period must be set to trade power consumption against task response!

Problems?



If processing time for the infrequent radio packets is much longer than for the frequent sonar echoes . . .

Concurrent execution

- We could solve (in a rather ad-hoc way) how to wait concurrently.
- Now we need to express concurrent execution ...

Imagine ...

...that we could interrupt execution of packet decoding when a sonar echo arrives so that the control algorithm can be run. Then decoding could resume! The two tasks fragments are interleaved.

Concurrent execution

- We could solve (in a rather ad-hoc way) how to wait concurrently.
- Now we need to express concurrent execution ...

Imagine ...

... that we could interrupt execution of packet decoding when a sonar echo arrives so that the control algorithm can be run. Then decoding could resume! The two tasks fragments are interleaved.

Interleaving by hand

```
void decode(struct Packet *pkt, struct Params p){
    phase1(pkt,p);
    try_sonar_task();
    phase2(pkt,p);
    try_sonar_task();
    phase3(pkt,p);
}
```

```
void try_sonar_task(){
  if(SONAR_STATUS & READY){
    dist = SONAR_DATA;
    control(dist,&signal,&params);
    servo_write(signal);
  }
}
```

Again we break tha logical organization of the program in an ad-hoc way! How many phases of decode will we need to run the sonar often enough?

Interleaving by hand

```
void decode(struct Packet *pkt, struct Params p){
   phase1(pkt,p);
   try_sonar_task();
   phase2(pkt,p);
   try_sonar_task();
   phase3(pkt,p);
}
```

```
void try_sonar_task(){
  if(SONAR_STATUS & READY){
   dist = SONAR_DATA;
   control(dist,&signal,&params);
   servo_write(signal);
  }
}
```

Again we break tha logical organization of the program in an ad-hoc way! How many phases of decode will we need to run the sonar often enough?

Interleaving by hand

```
void decode(struct Packet *pkt, struct Params p){
    phase1(pkt,p);
    try_sonar_task();
    phase2(pkt,p);
    try_sonar_task();
    phase3(pkt,p);
}
```

```
void try_sonar_task(){
  if(SONAR_STATUS & READY){
   dist = SONAR_DATA;
   control(dist,&signal,&params);
   servo_write(signal);
  }
}
```

Again we break tha logical organization of the program in an ad-hoc way! How many phases of decode will we need to run the sonar often enough?

Interleaving by hand

```
void decode(struct Packet *pkt, struct Params p){
    phase1(pkt,p);
    try_sonar_task();
    phase2(pkt,p);
    try_sonar_task();
    phase3(pkt,p);
}
```

```
void try_sonar_task(){
  if(SONAR_STATUS & READY){
   dist = SONAR_DATA;
   control(dist,&signal,&params);
   servo_write(signal);
  }
}
```

Again we break tha logical organization of the program in an ad-hoc way! How many phases of decode will we need to run the sonar often enough?

Interleaving by hand

More fine breaking up might be needed . . .

```
void phase2(struct Packet *pkt, struct Params *p){
   while(expr){
     try_sonar_task();
     phase21(pkt,p);
   }
}
```

Interleaving by hand

More fine breaking up might be needed ...

```
void phase2(struct Packet *pkt, struct Params *p){
   int i = 0;
   while(expr){
     if(i%800==0)try_sonar_task();
     i++;
     phase21(pkt,p);
   }
}
```

Code can become very unstructured and complicated very soon.

And then someone might come up with a new, better decoding algorithm ...

Interleaving by hand

More fine breaking up might be needed ...

```
void phase2(struct Packet *pkt, struct Params *p){
   int i = 0;
   while(expr){
     if(i%800==0)try_sonar_task();
     i++;
     phase21(pkt,p);
   }
}
```

Code can become very unstructured and complicated very soon.

And then someone might come up with a new, better decoding algorithm ...

Interleaving by hand

More fine breaking up might be needed . . .

```
void phase2(struct Packet *pkt, struct Params *p){
   int i = 0;
   while(expr){
     if(i%800==0)try_sonar_task();
     i++;
     phase21(pkt,p);
   }
}
```

Code can become very unstructured and complicated very soon.

And then someone might come up with a new, better decoding algorithm . . .

Automatic interleaving?

There are 2 tasks, driven by independent input sources.

Handle sonar echoes running the control algorithm and updating the servo.

Handle radio packets by running the decoder.

Had we had access to 2 CPUs we could place one task in each. We can imagine some contruct that allows us to express this in our program.

Automatic interleaving?

There are 2 tasks, driven by independent input sources.

Handle sonar echoes running the control algorithm and updating the servo.

Handle radio packets by running the decoder.

Had we had access to 2 CPUs we could place one task in each. We can imagine some contruct that allows us to express this in our program.

Automatic interleaving?

There are 2 tasks, driven by independent input sources.

Handle sonar echoes running the control algorithm and updating the servo.

Handle radio packets by running the decoder.

Had we had access to 2 CPUs we could place one task in each. We can imagine some contruct that allows us to express this in our program.

Automatic interleaving?

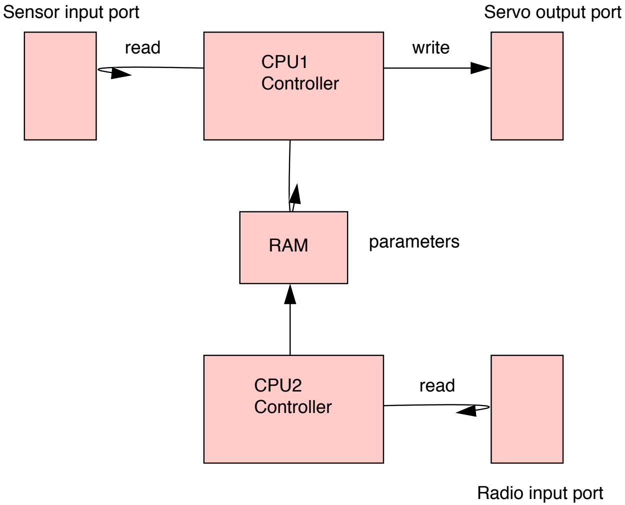
There are 2 tasks, driven by independent input sources.

Handle sonar echoes running the control algorithm and updating the servo.

Handle radio packets by running the decoder.

Had we had access to 2 CPUs we could place one task in each. We can imagine some contruct that allows us to express this in our program.

Two CPUs


Sensor input port
read
CPU1
Controller
write
Servo output port
RAM
parameters
CPU2
Controller
read
Radio input port

Two CPU's program

```
struct Params params;
```

```
void controller_main(){
  int dist, signal;
  while(1){
    dist = sonar_read();
    control(dist,
            &signal,
            &params);
    servo_write(signal);
  }
}
```

```
void decoder_main(){
  struct Packet packet;
  while(1){
    radio_read(&packet);
    decode(&packet,&params);
  }
}
```

We need some way of making one program of this! We will deal with it next lecture!

Concurrent Programming

Concurrent programming is the name given to programming notation and techniques for expressing potential parallelism and solving the resulting synchronization and communication problems.

A system supporting seemingly concurrent execution is called multi-threaded.

A thread is a unique execution of a sequence of machine instructions, that can be interleaved with other threads executing on the same machine.

Concurrent Programming

Concurrent programming is the name given to programming notation and techniques for expressing potential parallelism and solving the resulting synchronization and communication problems.

A system supporting seemingly concurrent execution is called multi-threaded.

A thread is a unique execution of a sequence of machine instructions, that can be interleaved with other threads executing on the same machine.

Concurrent Programming

Concurrent programming is the name given to programming notation and techniques for expressing potential parallelism and solving the resulting synchronization and communication problems.

A system supporting seemingly concurrent execution is called multi-threaded.

A thread is a unique execution of a sequence of machine instructions, that can be interleaved with other threads executing on the same machine.

Where should threads belong?

A programming language?

As in Java or Ada. Programs are well organized and are independent of the OS.

Libs and OS?

Like C with POSIX threads? Good for multilanguage composition given that OS standards are followed.

This course

For pedagogical purposes we choose to work with C and a small kernel.

Where should threads belong?

A programming language?

As in Java or Ada. Programs are well organized and are independent of the OS.

Libs and OS?

Like C with POSIX threads? Good for multilanguage composition given that OS standards are followed.

This course

For pedagogical purposes we choose to work with C and a small kernel.

Where should threads belong?

A programming language?

As in Java or Ada. Programs are well organized and are independent of the OS.

Libs and OS?

Like C with POSIX threads? Good for multilanguage composition given that OS standards are followed.

This course

For pedagogical purposes we choose to work with C and a small kernel.

Where should threads belong?

A programming language?

As in Java or Ada. Programs are well organized and are independent of the OS.

Libs and OS?

Like C with POSIX threads? Good for multilanguage composition given that OS standards are followed.

This course

For pedagogical purposes we choose to work with C and a small kernel.

Our first multithreaded program

```
struct Params params;

void controller_main(){
  int dist, signal;
  while(1){
    dist = sonar_read();
    control(dist,
            &signal,
            &params);
    servo_write(signal);
  }
}

void decoder_main(){
    struct Packet packet;
    while(1){
        radio_read(&packet);
        decode(&packet,&params);
    }
}

main(){
  spawn(decoder_main);
  controller_main();
}
```