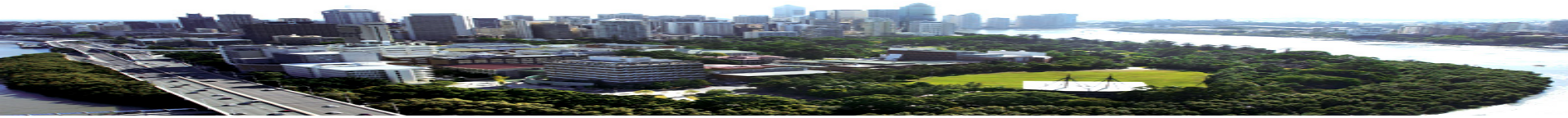
ENB 350 Real-time Computer based systems

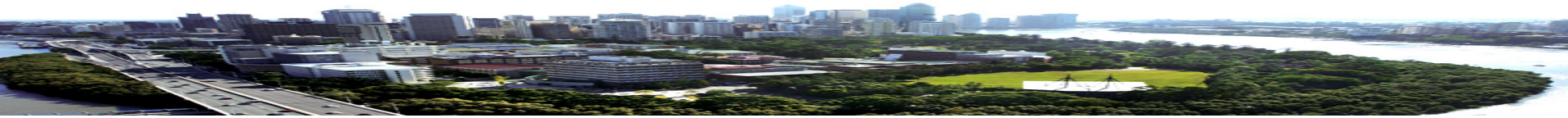
Lecture 7 – COOPERATIVE MULTITASKING

V. Chandran



Contents

- Real time problem, busy wait, state machines
- Tasks and multitasking
- Cooperative and Pre-emptive
- Polling, Foreground/Background, Multitasking
- Costates and Cofunctions
- Slice
- Timing considerations

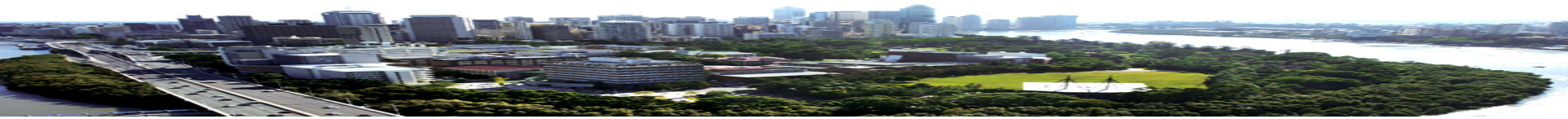


A real time problem example

1. Wait for a pushbutton to be pressed.
2. Turn on the first device.
3. Wait 60 seconds.
4. Turn on the second device.
5. Wait 60 seconds.
6. Turn off both devices.
7. Go back to the start.

Q. How would you write C code for this?

Make an attempt.



A state machine solution

```
task1state = 1;                    // initialization:
while(1){
   switch(task1state){
      case 1:
         if( buttonpushed() ){
            task1state=2;    turnondevice1();
            timer1 = time;          // time incremented every second
         }
         break;
      case 2:
         if( (time-timer1) >= 60L){
            task1state=3;    turnondevice2();
            timer2=time;
         }
         break;
      case 3:
         if( (time-timer2) >= 60L){
            task1state=1;    turnoffdevice1();
            turnoffdevice2();
         }
         break;
   }
   /* other tasks or state machines */
}
```

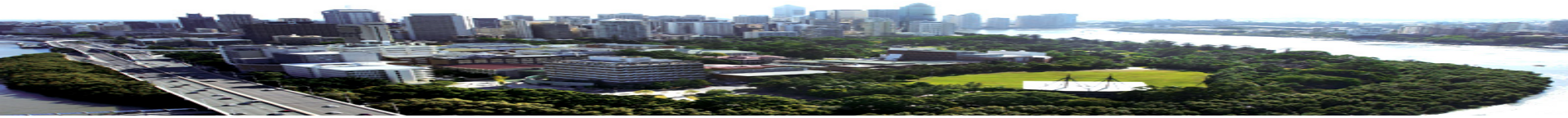
From the Dynamic C manual.

Q. Where does a break statement take execution to?

Q. Will other state machines be in different while loops?

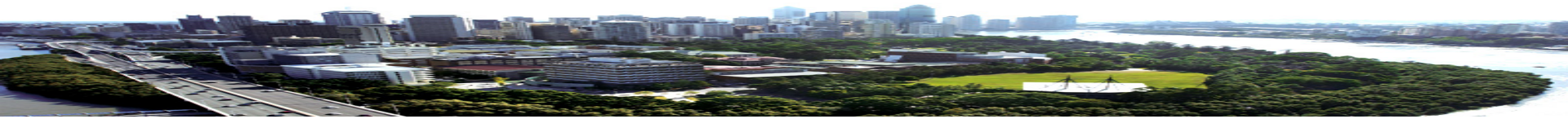
Q. Is busy wait idling of the CPU avoided?

Q. Are instructions executed unnecessarily?



Task

- Sequence of instructions performing a connected set of desired actions
- This may be one “program” on a single CPU system
- With multitasking several tasks can run on a single CPU system

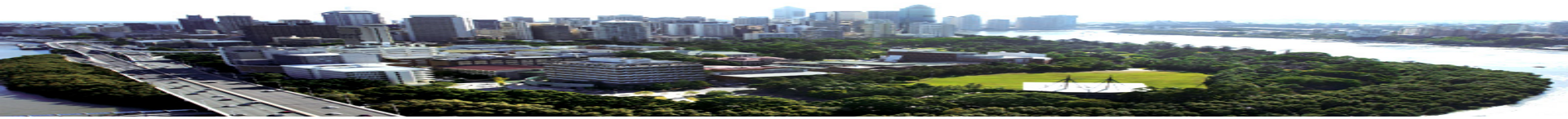


Multitasking

A way to perform several different tasks at ‘virtually’ the same time

Appears to run several “programs” on the same CPU at the same time

Takes advantage of task delays waiting for events



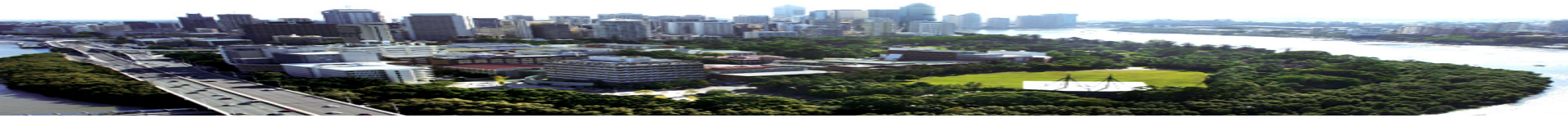
Demonstration - Example

```
main() {
int secs; // seconds counter
secs = 0; // initialize counter
while (1) { // endless loop
                                    // First task will print the seconds elapsed.
 costate {
 secs++;                            // increment counter
 waitfor( DelayMs(1000) );          // wait one second
  printf("%d seconds\n", secs); // print elapsed seconds
 }
                                   // Second task will check if any keys have been pressed.
costate {
 if ( !kbhit() ) abort; // key been pressed?
 printf(" key pressed = %c\n", getchar() );
 }

} // end of while loop
} // end of main
```

Each costatement is like a little program with its own statement pointer

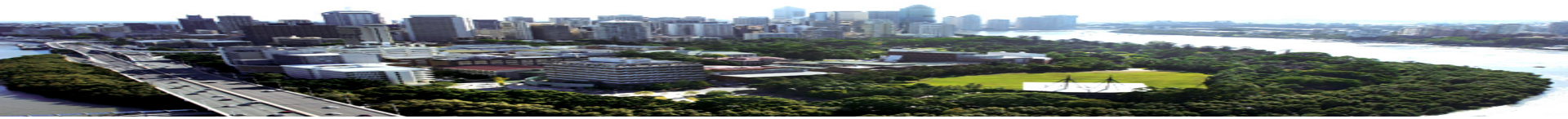
Note the waitfor. What happens when a waitfor condition is encountered?

Q. What is a costate?



Program? Task? Thread?

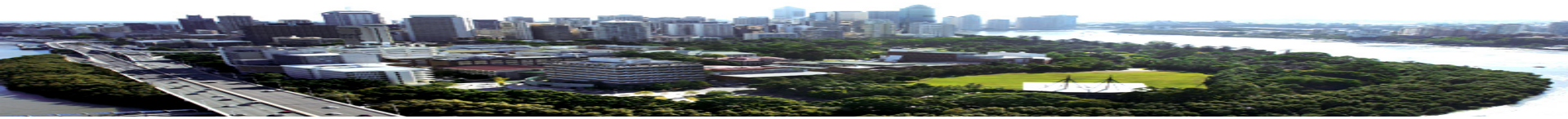
Q. What are the differences?



Why multitasking?

- Uses the CPU more efficiently – utilizes CPU during i/o waits
- Highly responsive to external events
- Provides a meaningful structure to the program
- Allows different tasks for different devices / kernel components and easier system development

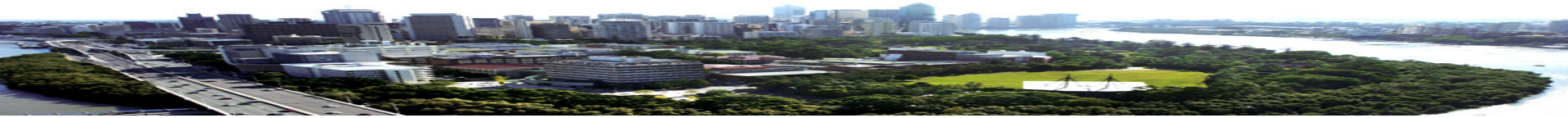
Q. What are some events that a task will need to wait for?



Why multitasking?

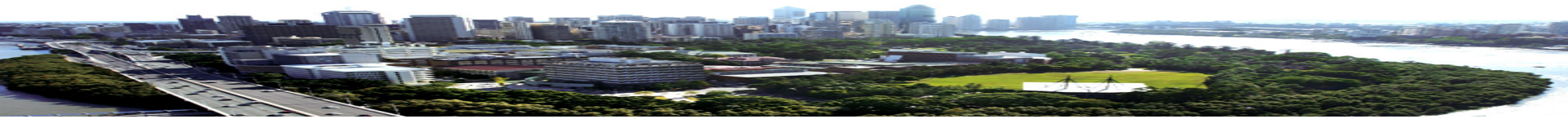
- Uses the CPU more efficiently – utilizes CPU during i/o waits
- Highly responsive to external events
- Provides a meaningful structure to the program
- Allows different tasks for different devices / kernel components and easier system development

Q. What are some events that a task will need to wait for?
input or output to be available, right conditions for state changes,

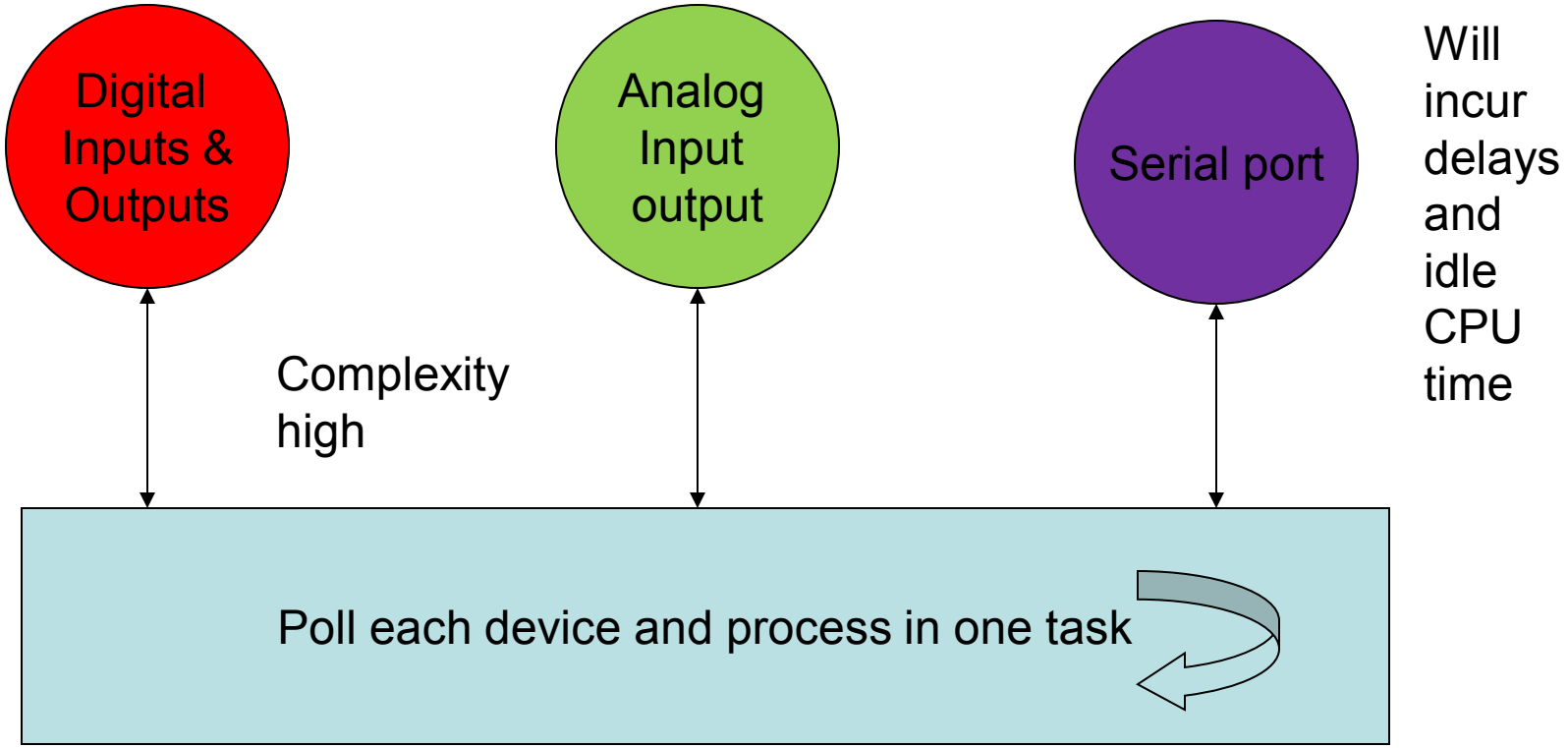


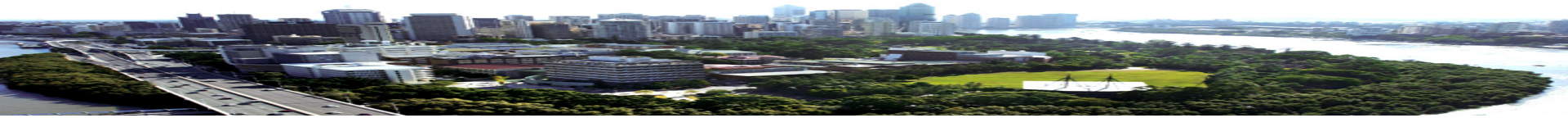
Types of multitasking

- In cooperative multitasking each task voluntarily surrenders the CPU at programmer determined points
- In pre-emptive multitasking the CPU is forcibly taken away from a task via an interrupt

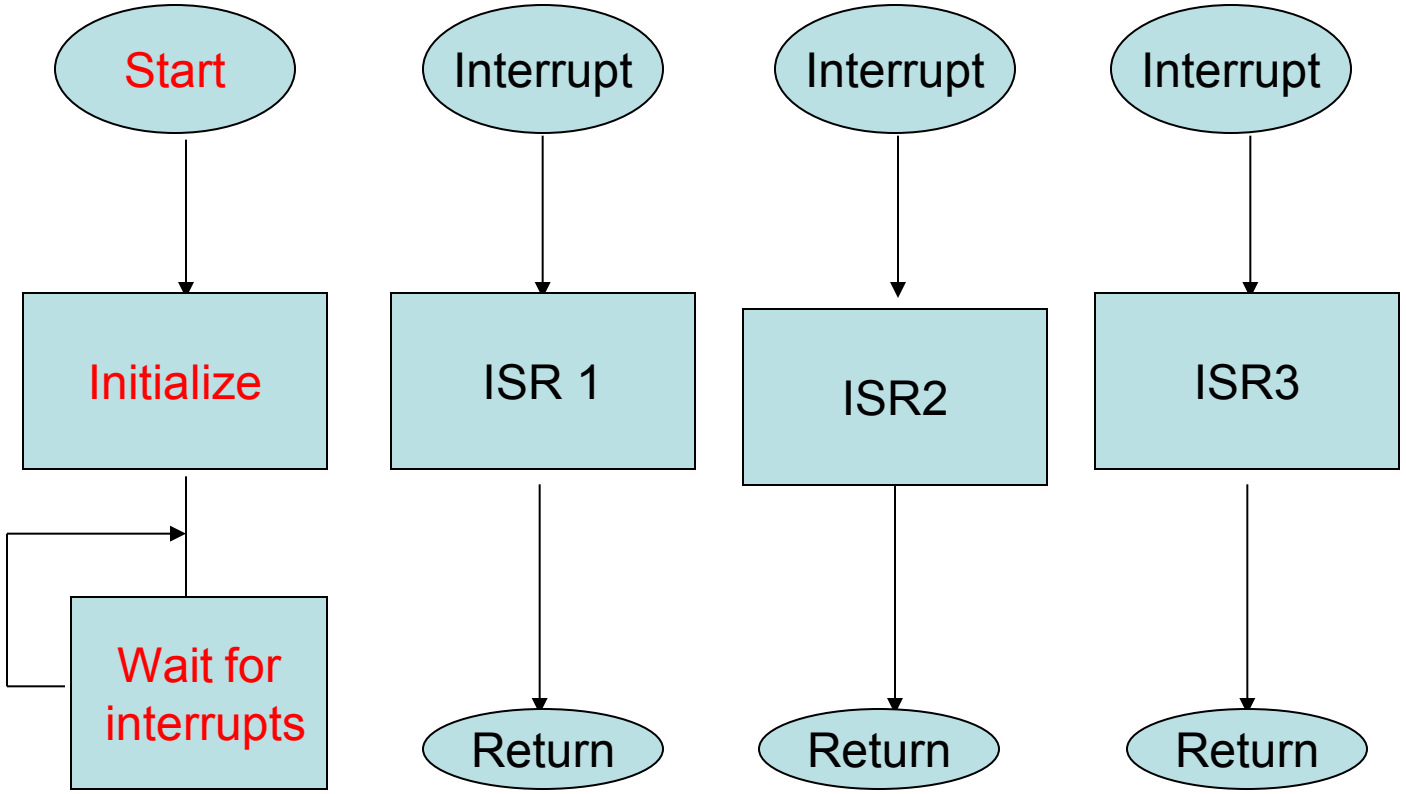


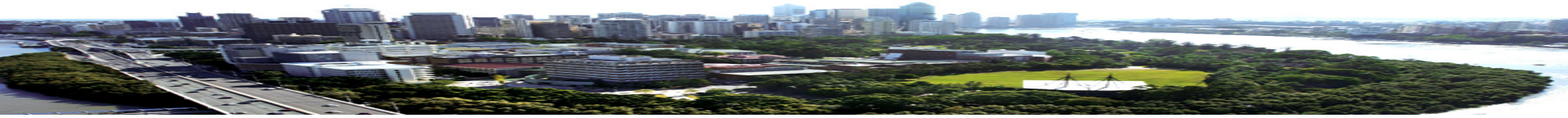
No multitasking and no interrupts





Foreground / Background

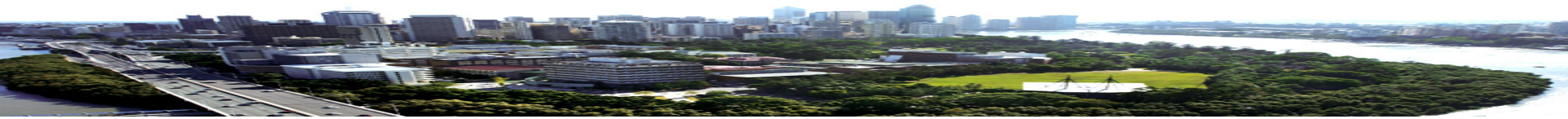




Foreground / Background

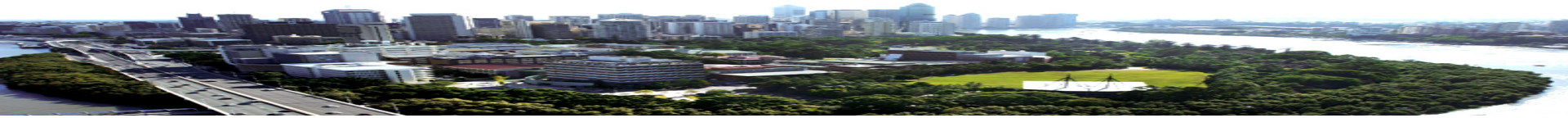
- Main program performs initialization and enters a loop waiting for interrupts
- Most work is done by the ISRs

- Response to external events with predictable latency



If the ISR takes too long?

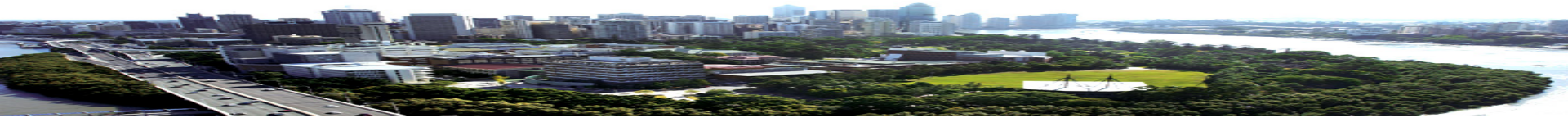
- there can be multiple interrupts from the same device before the ISR is exited.
- If interrupts are masked, lower priority interrupts may have to wait too long
- If an output device must be polled within an ISR, there is a loop and the ISR can take very long.



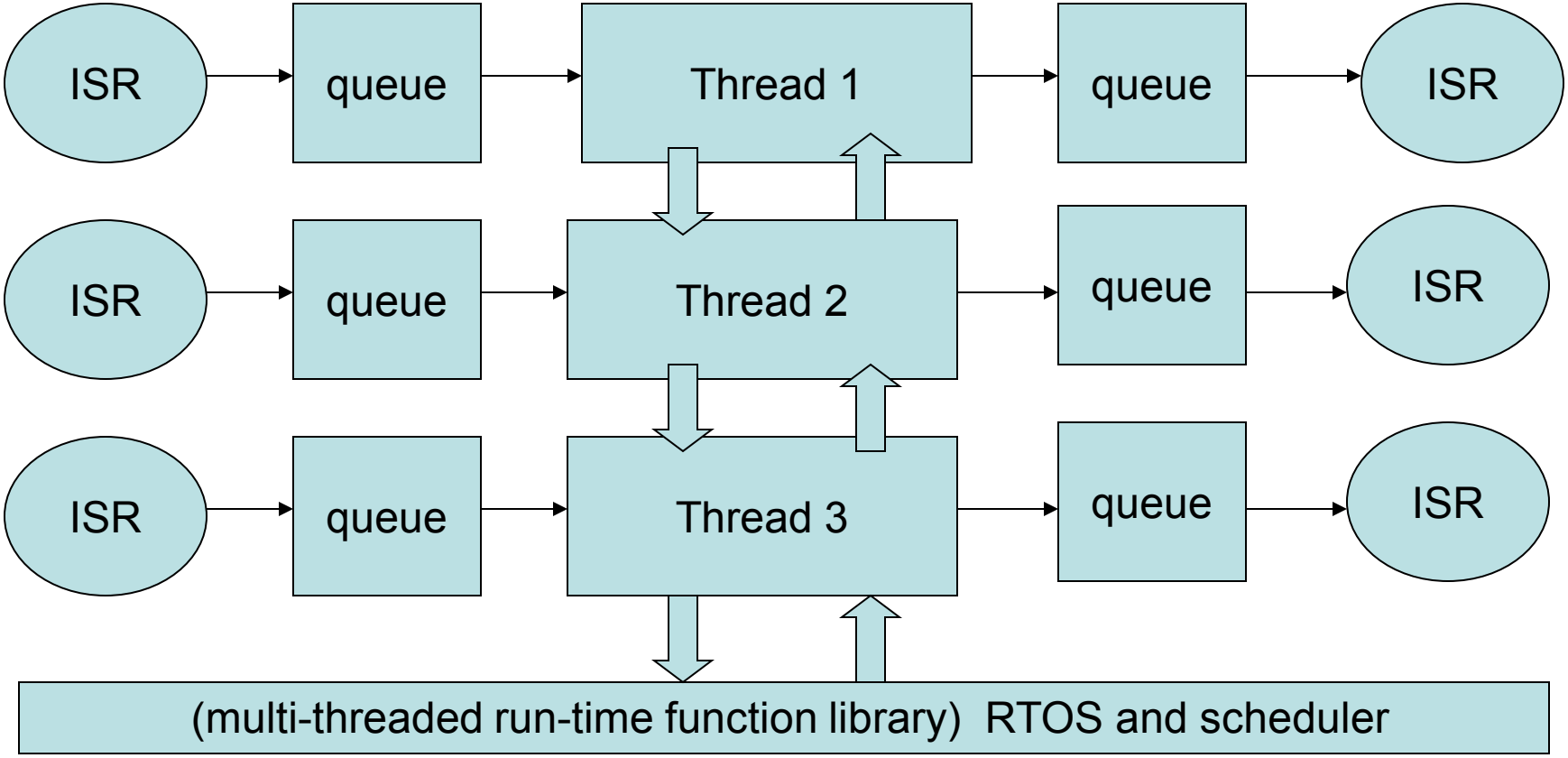
Queueing/Buffering data

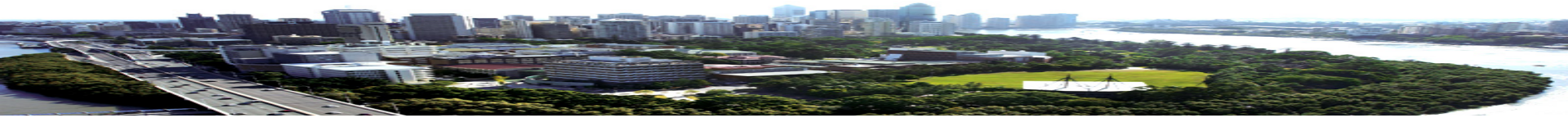
- A solution is to queue the input data and return
- Output can be made part of the background task; background task checks whether the output device is ready.
- Short constant latency for lower priority interrupts but overall response may still be slow.

- Why not queue output data also?



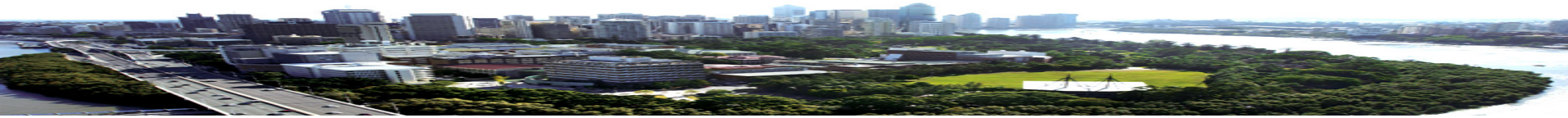
Multitasking program





Benefits

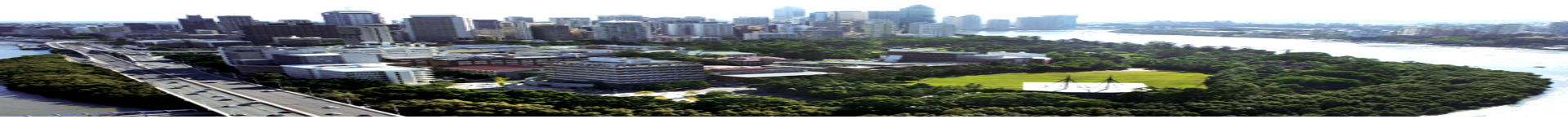
- ISRs can now be ‘straight line’ code
- Optimizes latency of individual ISRs
- Background is now a collection of queues and routines – a collection of threads and the kernel or operating system.



Threads

- A Thread is usually written as an initialization followed by a wait for an event, or a condition or data. It then processes the data and returns to the wait in an infinite loop.

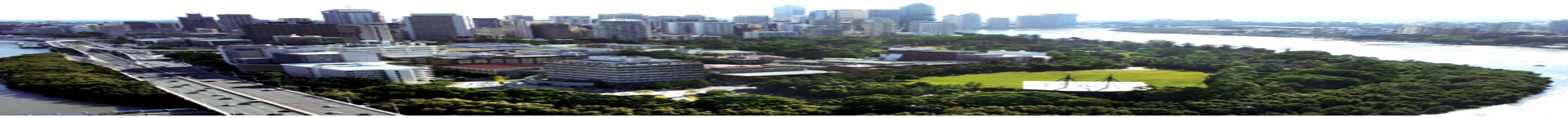
- Threads can be designed and coded pretty much independently. Total program complexity is reduced



Context

- CPU registers including a stack pointer and the program counter and other information stored in a special region of memory. This is known as the ‘context’ of the task.

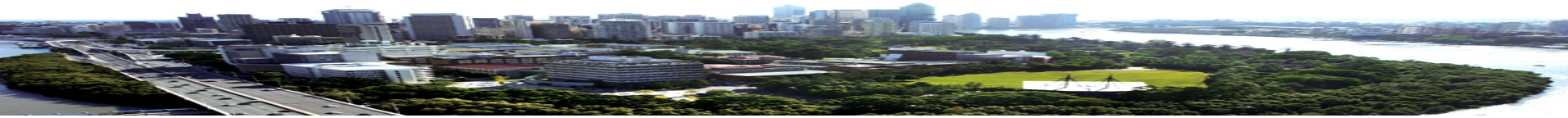
- A task may be ‘light weight’ and may not need to save all such information.
- Preemptive multitasking requires tasks to have their own stack and save full context.
- Cooperative tasks need not have their own stack.



Context switch

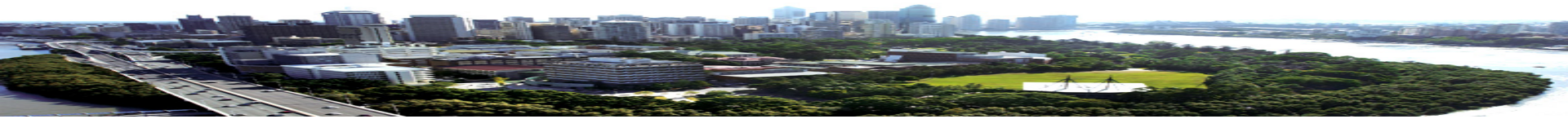
- A context switch from thread “A” to thread “B”
- Saves context of A and restores that of B

- Including PC (and SP for threads with their own stack)



Mutlitasking with Dynamic C

- costate and cofunction (cooperative)
- Slice (preemptive)
- Ucos-II included as a library (preemptive)

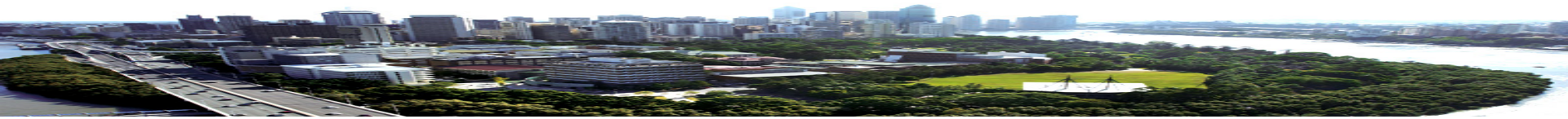


Costate solution

```
while(1){
   costate{ ... }                          // task 1

   costate{                                // task 2
      waitfor( buttonpushed() );
      turnondevice1();
      waitfor( DelaySec(60L) );
      turnondevice2();
      waitfor( DelaySec(60L) );
      turnoffdevice1();
      turnoffdevice2();
   }
   costate{ ... }                          // task n
}
```

Q. What happens at the waitfor ?

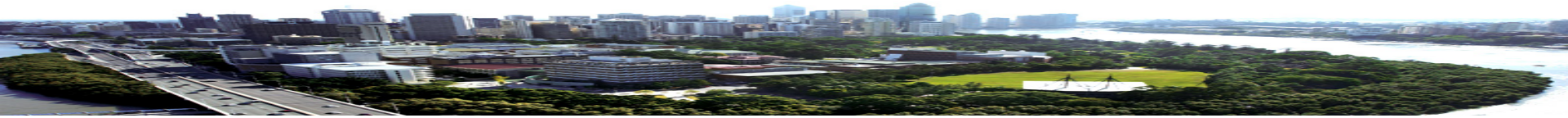
Q. Is this better than the state machine solution? How?



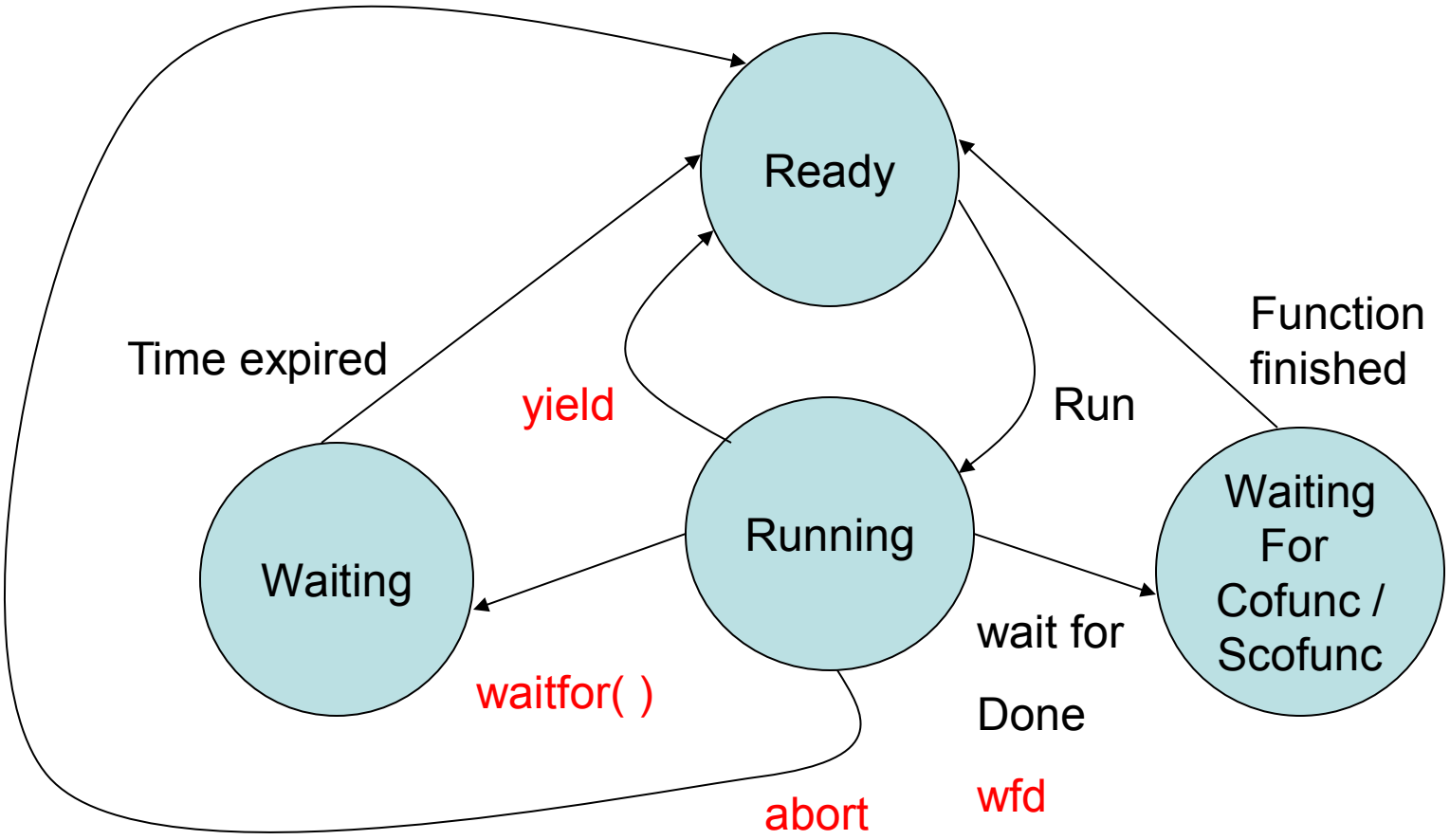
Costate

```
costate [ name [state] ] { [ statement | yield; | abort; |
   waitfor( expression ); ] . . .}
```

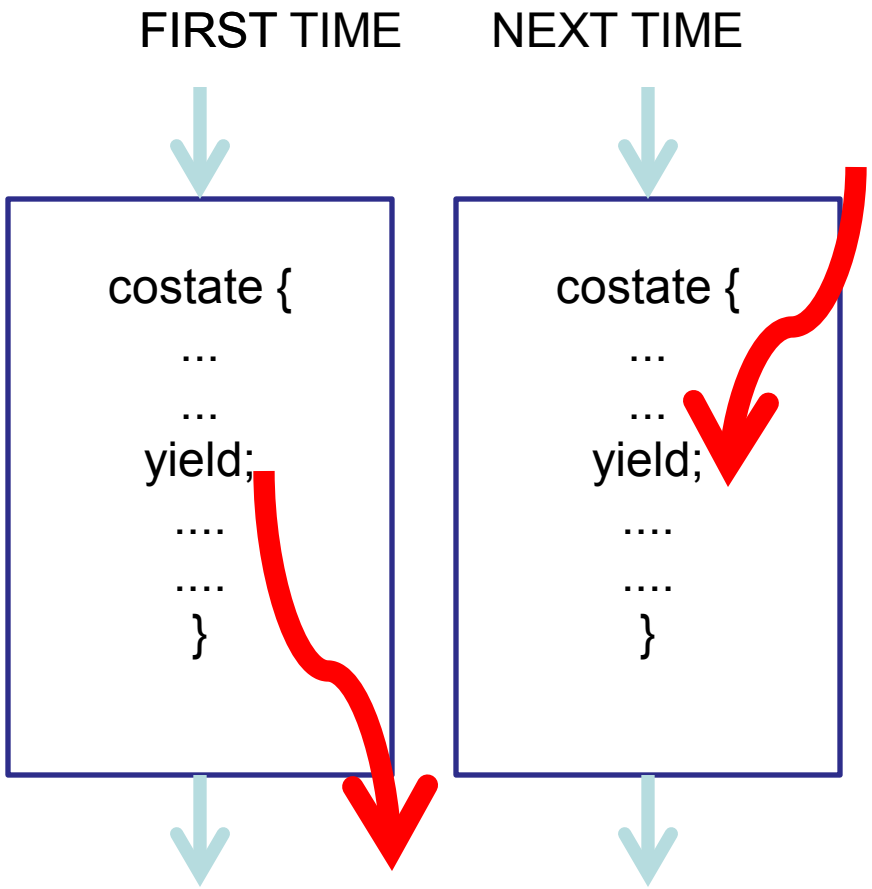
- statements enclosed within { } are identified as a costatement
- costate can be named or unnamed
- name can be
    - any valid C name or
    - the name of a previously defined CoData structure
    - or a pointer to a previously defined CoData structure
- state can be ‘always_on’ or ‘init_on’
- ‘always_on’ executes each time encountered
- ‘init_on’ executes the first time it is encountered and is then inactive
- default for unnamed costatements is ‘always_on’
- default for named costatements is ‘init_on’ and paused

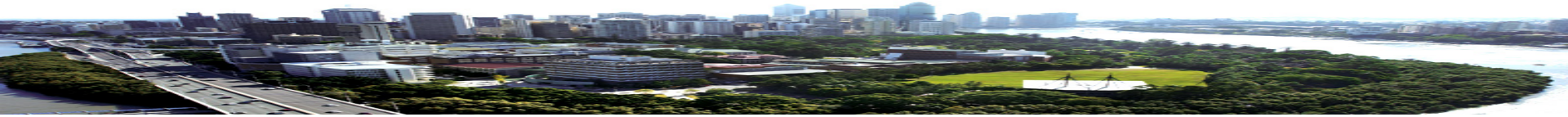


Task states in co-operative multitasking

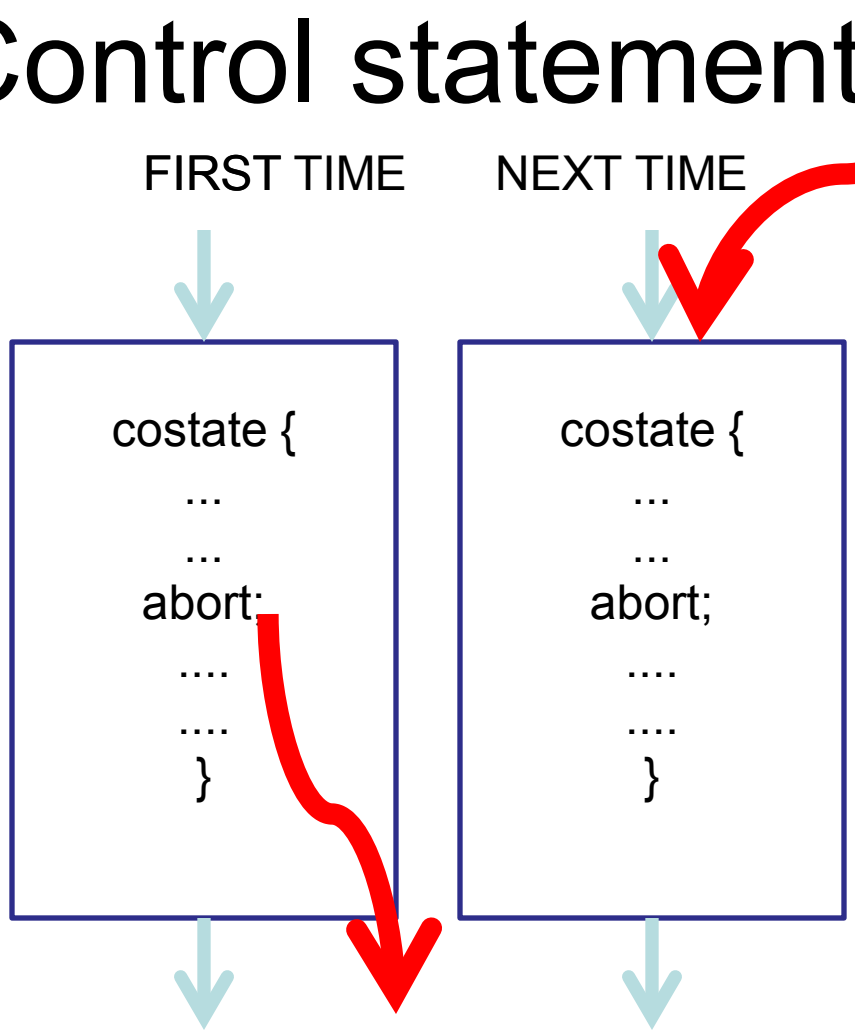


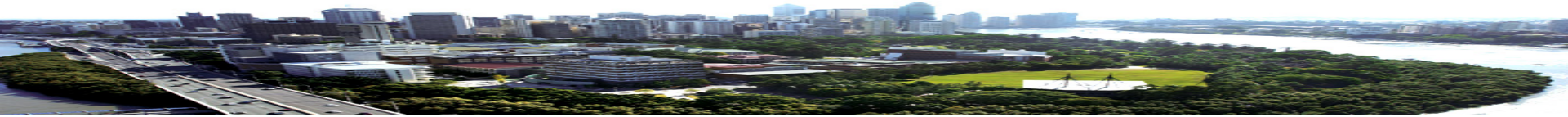
Control statement - yield



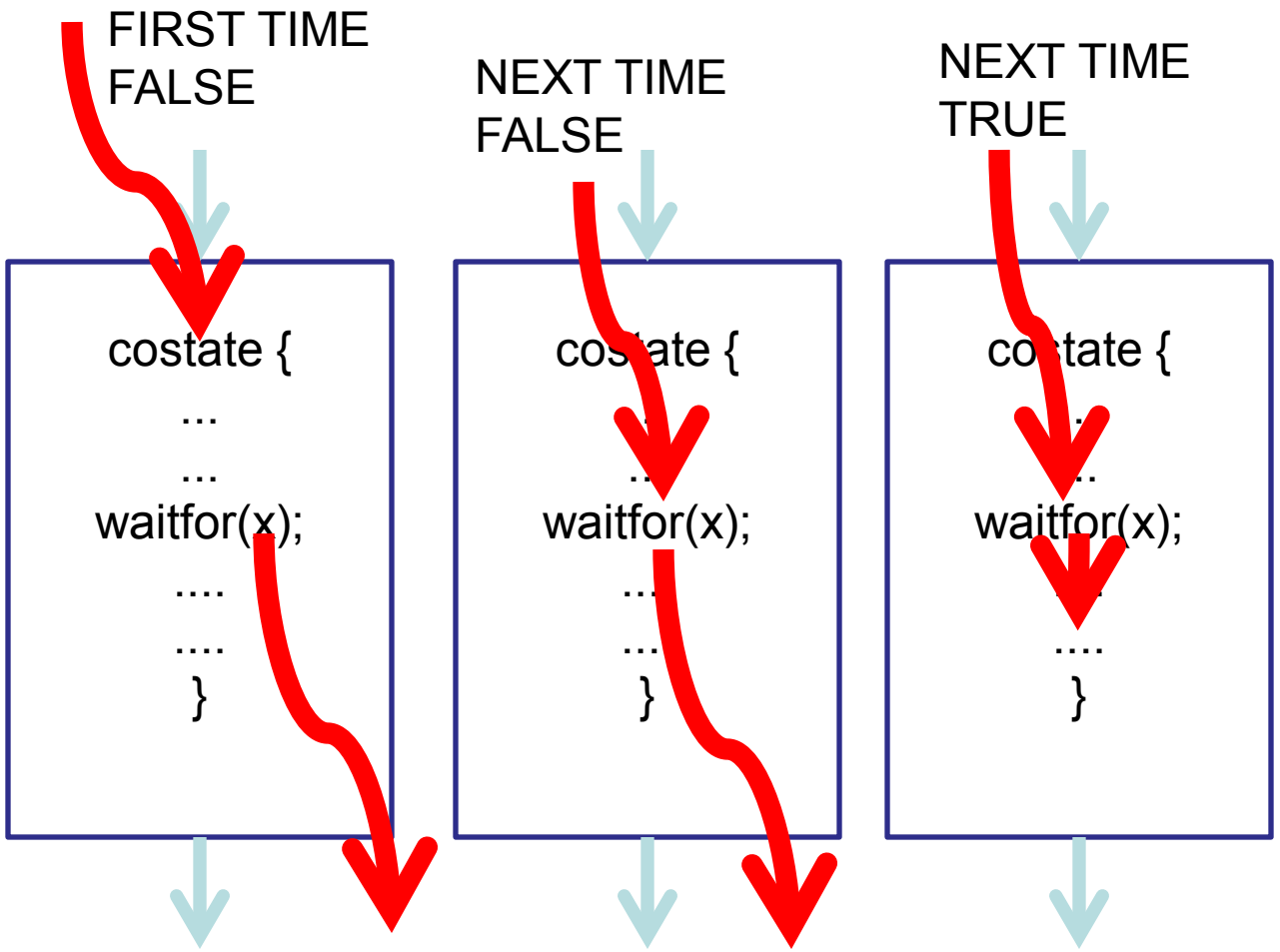


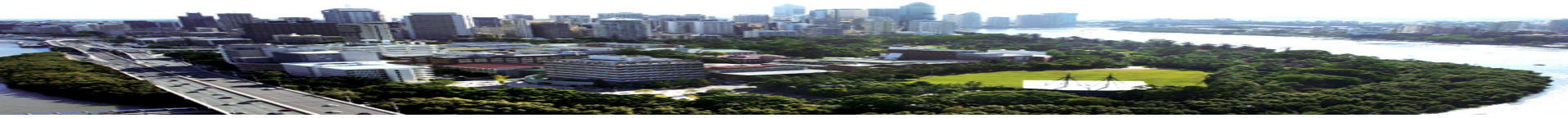
Control statement - abort





Control statement - waitfor





Delay functions

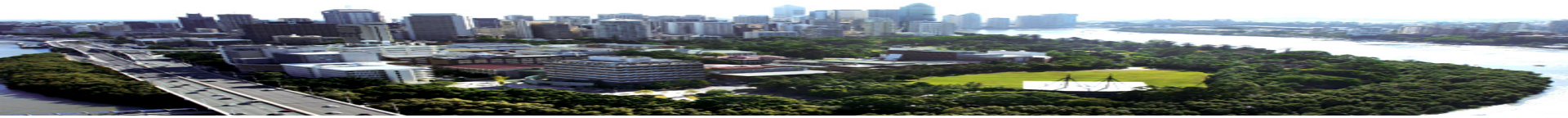
DelayMs(), DelaySec(), DelayTicks(), IntervalMs(), IntervalSec(), IntervalTick()

These functions should be called inside a waitfor statement.

They return 0 to indicate that the desired time has not yet elapsed.

They use BIOS variables like SEC_TIMER, MS_TIMER.

Q. How is it possible for the same function to be called from more than one costate?



Delay functions

DelayMs(), DelaySec(), DelayTicks(), IntervalMs(), IntervalSec(), IntervalTick()

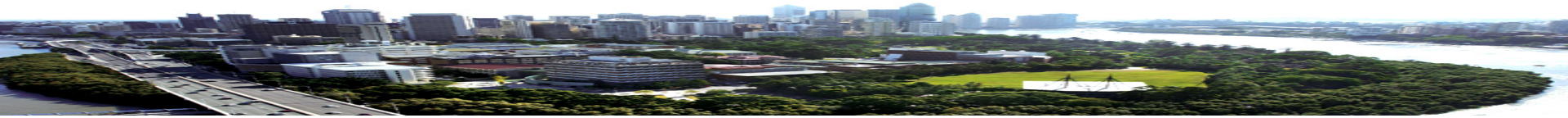
These functions should be called inside a waitfor statement.

They return 0 to indicate that the desired time has not yet elapsed.

They use BIOS variables like SEC_TIMER, MS_TIMER.

Q. How is it possible for the same function to be called from more than one costate?

They have as an implicit parameter a pointer to the CoData structure of the costatement that calls them when called for the first time.



CoData structure

```
typedef struct {
        char CSState;
        unsigned int lastlocADDR;
        char lastlocCBR;
        char ChkSum;
        char firsttime;
        union{
                unsigned long ul;
                struct {
                        unsigned int u1;
                        unsigned int u2;
                } us;
        } content;
        char ChkSum2;
} CoData;
```

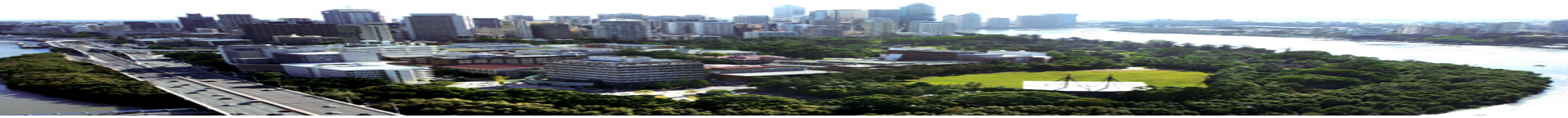
Each costatement has a structure of type CoData.

It contains state and timing information and the address of the statement that will execute when the program thread NEXT reaches the costatement.

What is this used for?

What is this used for?

What is this used for?



CoData structure

```
typedef struct {
        char CSState;
        unsigned int lastlocADDR;
        char lastlocCBR;
        char ChkSum;
        char firsttime;
        union{
                unsigned long ul;
                struct {
                        unsigned int u1;
                        unsigned int u2;
                } us;
        } content;
        char ChkSum2;
} CoData;
```

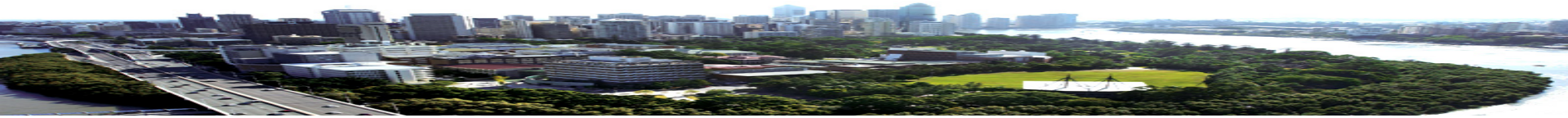
Each costatement has a structure of type CoData.

It contains state and timing information and the address of the statement that will execute when the program thread NEXT reaches the costatement.

24 bit address of last location

Flag used with waitfor and wfd. Set to 1 before evaluated for the first time

Used by delay routines to store a delay count



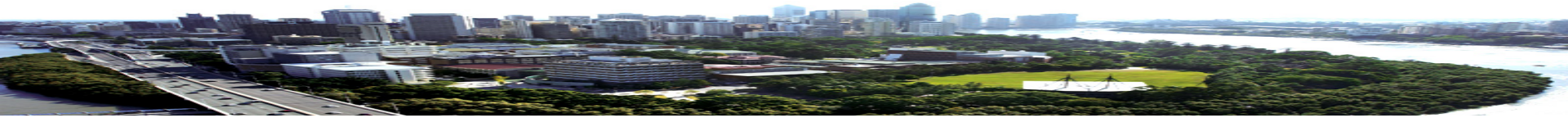
Demonstration - CoData

```
#class auto
#define N       10          // number of machines

CoData Machine[N];          // array of CoData blocks
CoData *pMachine;           // pointer to current machine

main()
{
      int   i;

      for (i=0; i<N; i++) {
            CoBegin(&Machine[i]);   // enable machines
      }

      for (;;) {                    // endless loop
       for (i=0; i<N; i++) {

      pMachine = &Machine[i];     // pointer to current machine

      costate pMachine always_on {

      // identical code for all machines goes here
      printf("Machine %d executing...\n", i);
                  }
            }
      }
}
```

Avoids repetition of code in different costates

Q. What can you do if the code is not identical?

Q. How can you selectively pause and resume?



Cofunctions

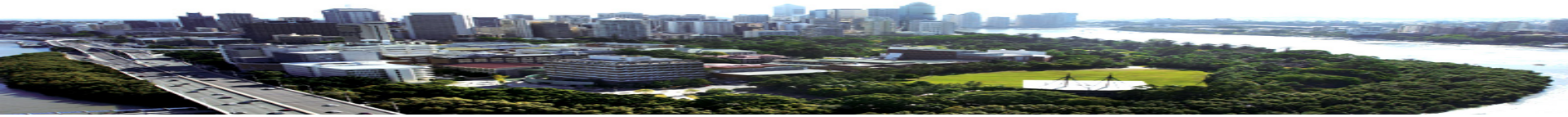
- Are like costatements but arguments can be passed to them

- Cofunctions must be called from inside a wait for done
- wait for done must be inside a costatement or cofunction
- Do not use costates within cofunctions

- Have the storage class 'Instance' and an instance variable is like a static variable and the value persists between function calls

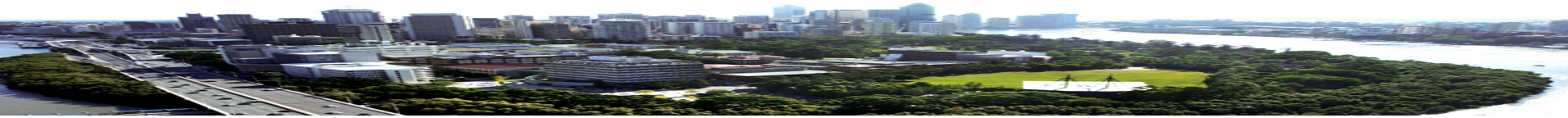
SYNTAX: **cofunc|scofunc type [name][[dim]]([type arg1, ..., type argN]) {**
[*statement* | *yield;* | *abort;* | *waitfor(expression);]... }

USAGE: **int j; j = waitfordone { x = cofunc1(…); }**



Demonstration - Cofunction

```
//
#define CINBUFSIZE 31
#define COUTBUFSIZE 31

main() {
   int c;
   serCopen(19200);
   loopinit();     // initializes internal data structures
   while (1) {
      loophead();  // for proper scof abandonment handling
   costate {
      wfd c = cof_serCgetc();
      wfd cof_serCputc(c);
      }
   }
   serCclose();
}
```

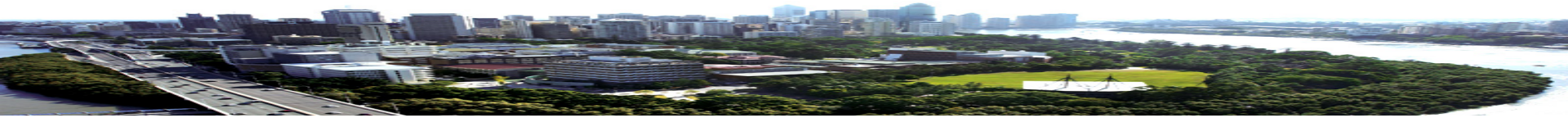


Single user cofunctions

- Will allow only one caller at a time and block others
- They can be called from several places within the same big loop.

wfd {cofuncA(); cofuncA();} Is this okay?

wfd {scofuncA(); scofuncA();} Is this okay?

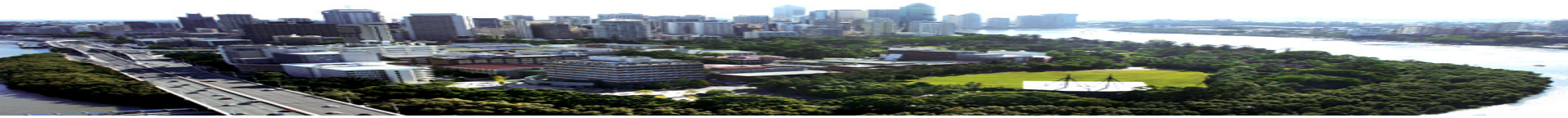


Single user cofunctions

- Will allow only one caller at a time and block others
- They can be called from several places within the same big loop.

wfd {cofuncA(); cofuncA();} ×

wfd {scofuncA(); scofuncA();} ✓

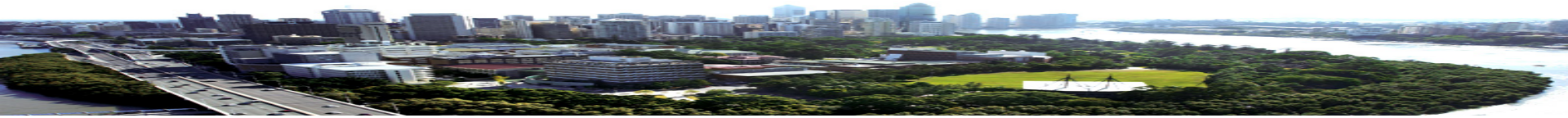


Dynamic C support for pre-emptive multitasking

- slice statement allows running a task for a specified period of time (in clock ticks)

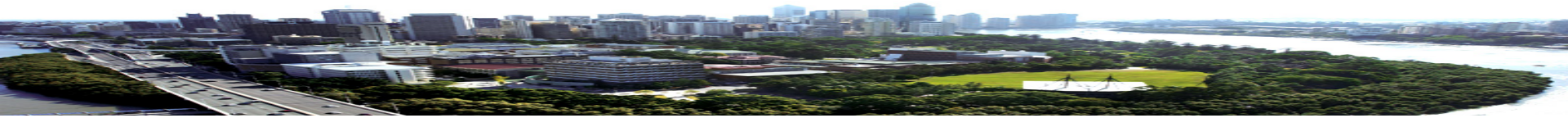
  Example: Slice(500,20) { ….. } // Buffer size, ticks

- time_slice is the amount of time in ticks for a task to run
- The body of a slice may contain
  - Regular C statements
  - yield
  - abort
  - waitfor



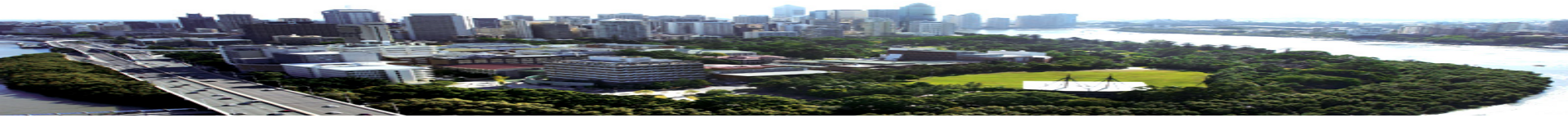
Slice

```
main () {
    int x, y, z;
    for (;;) {
        costate a {          // will run until yield or abort or completion
        ...
        }
        slice_A(500, 20) {    // slice_A  runs 20 ticks and is suspended
                              // buffer size is 500 bytes
        ...
        }
        slice_B(500, 40) {    // How long does slice_B run for?
        ...
        }
    }
```



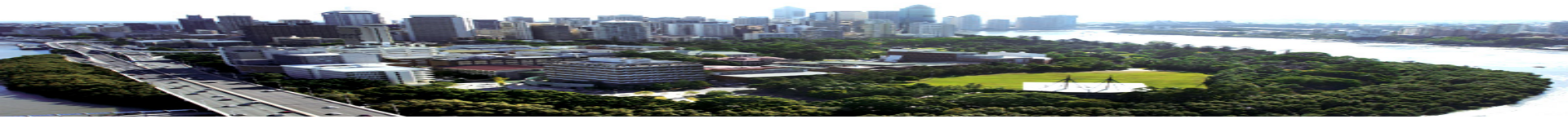
Slice

```
main () {
    int x, y, z;
    for (;;) {
        costate a {          // will run until yield or abort or completion
        ...
        }
         slice_A(500, 20) {     // slice_A runs 20 ticks and is suspended
                                // buffer size is 500 bytes
        ...
        }
         slice_B(500, 40) {    // slice_B runs 40 ticks and is suspended
        ...
        }
    }
```

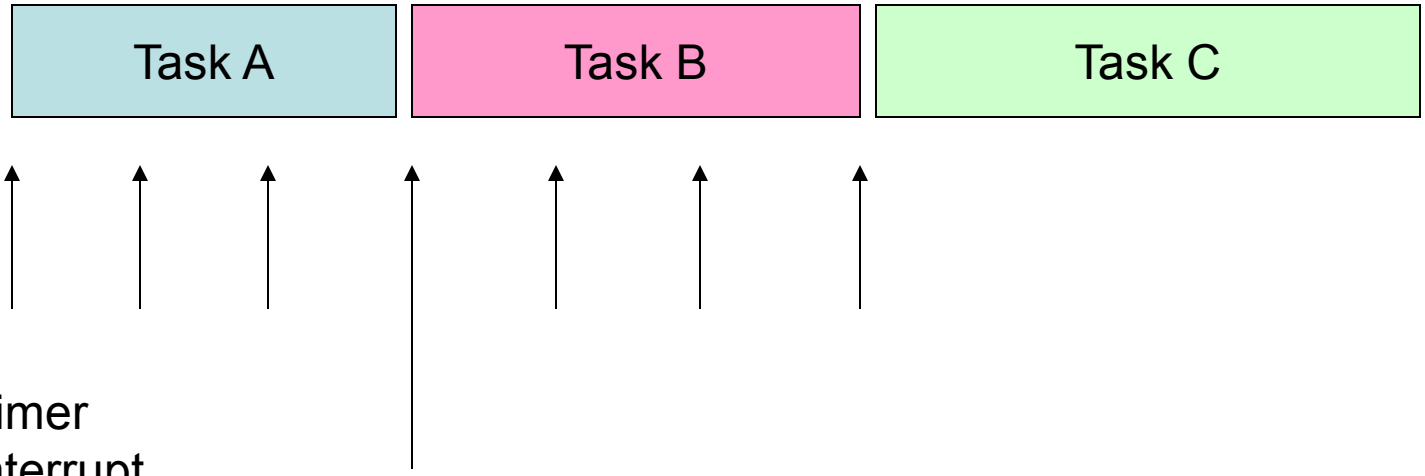


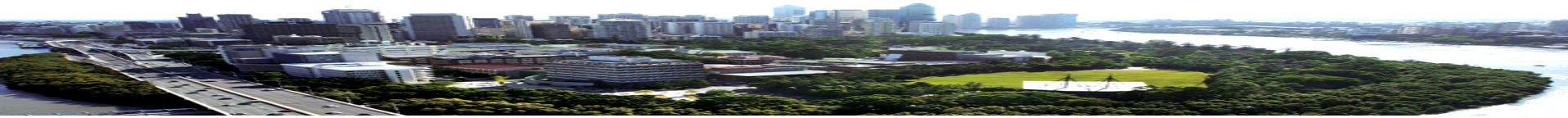
Dynamic C Slice limitations

- A slice has its own stack. Local variables and parameters cannot be accessed while in the context of a slice (it has its own)
- Only one slice can be active at any time. Therefore they cannot be nested.
- They cannot be used with ucos-II or with TCP/IP
- Exiting code via goto, break, continue, return is not supported.



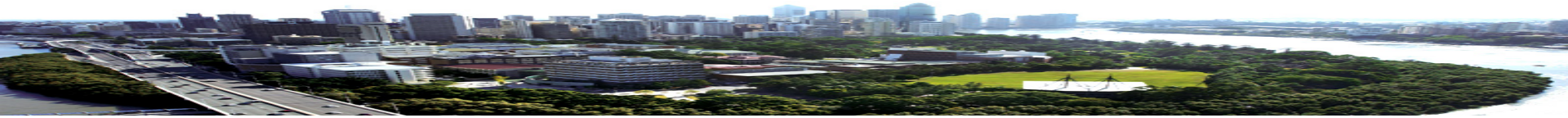
Preempting at Time Slices





Slice data structure

```
struct SliceData {
    int time_out;
    void* my_sp;
    void* caller_sp;
    CoData codata;
}
struct SliceBuffer {              // buffer size specified at creation
    SliceData slice_data;         // data
    char stack[];                 // stack takes rest of the slice buffer
};          // How large should the buffer size be?
```

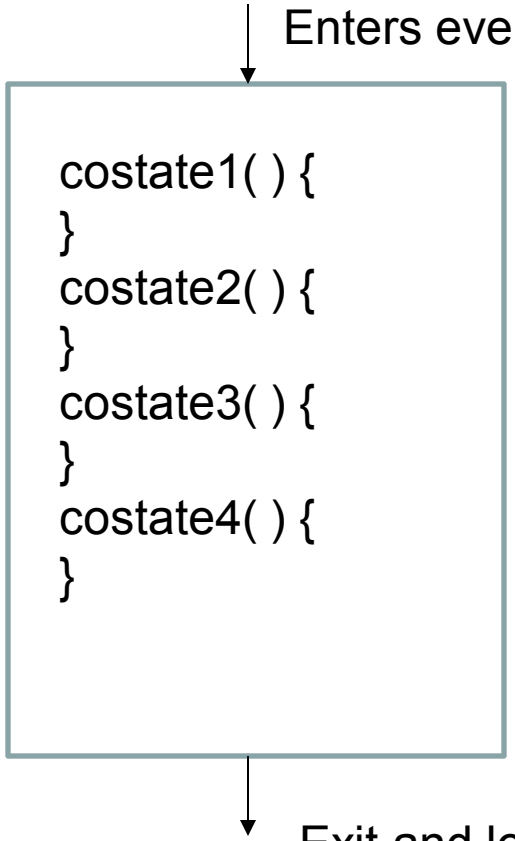


Timing considerations

If worst case sum of waits is less than 45 ms, overall period will be satisfied

Individual waits may vary

Individual deadlines and periods cannot be fully satisfied with cooperative multitasking



Execution time for all four without waits = 5 ms

Overall laxity = 45 ms

Q. What if one costate must eject a rejected part with no more than 5 ms delay?