Robust and Energy-Efficient Real-Time Systems

Lecture 1: *Time- vs. event-triggered systems*
First things first...

http://www.sm.luth.se/csee/courses/smd/D7020E/
The classical program

Provide input data

Ignore further input

Collect output data

Defer early output

Start

Ignore further input

Stop
In practice

Provide input data

Input kept constant

Time

Collect output data

Output not looked at

start

stop
Modern programs

Sensitive to evolving input

Time

Must produce evolving output
Why?

- Because modern computers are components among other evolving components like
  - Keyboards, mice and displays
  - Human users behind these components
  - Network interfaces
  - Other computers behind these components
  - Sensors and actuators
  - Real physical objects behind these components
- Because a modern computer program is very rarely in superior control of its full environment
The computer as a component

- **Control computer**
- **Collision avoidance computer**
- **Physical car**
- **Human driver**
- **Mobile phone**
- **Other cars & drivers**
Dealing with evolving input

- Approach 1: new input is **read** at the initiative of the program
  - (As often as "possible"...)
  - (Or in an ad hoc fashion...)
  - Or at **well-defined times**!

- Approach 2: new input is **written** into the program at the initiative of the **environment**
  - (Just to be stored somewhere...)
  - Or guaranteed to trigger an associated **reaction**!
Approach 1: Time-triggered systems

- Idea: read input at pre-defined times, chosen to match the expected variations in input
- Obvious special case: read input every $T$ time units (the periodic process)
- What happens between the computations? Nothing – the CPU can just shut down!
- How choose $T$? Use Nyqvist's sampling theorem!
- What if there are multiple inputs?
  - Let the highest frequency input determine $T$...
  - Or run multiple periodic processes in parallel!
Periodic time triggered systems

\[ T_1, T_2, T_3 \]
Non-deterministic read/write

Read sequence

Write sequence

Not generally considered by the term *time-triggered* (although certainly triggered by time!)

Still the most common process structure in practical real-time systems.

Requires

- Timer support to achieve stable $T$ (typical OS service)

Note correspondence to the realistic classic program model!
Deterministic read/write


The programming language Giotto – Henzinger et al. 2001

Requires
- Atomic r/w buffers
- Timer support to implement both $T$ and $C$

Corresponds to the ideal classic program model!
Approach 2: Event-triggered systems

- Idea: let the environment decide when input has changed enough to require some program action; i.e., when an event has occurred
- Well-known concept on the computer hardware level: the external interrupt!
- What happens between the event processing phases? Nothing – the CPU can just shut down!
- What if there are events with overlapping reactions?
  - Buffer up the events...
  - Or run multiple event-handlers in parallel!
Event-triggered systems

Event A

parallel

buffered

Event B

Event C
The event is just a signal to start a computation – the handler reads its input just like an ordinary program

This is the typical case for hardware interrupt handlers (the code starts by reading some external registers)
The event is a more complex signal that also carries some data (not unlike the program arguments passed over at the startup of a classical program)

This is the typical case for event-driven frameworks and OS services like `getNextEvent`
The event handler just writes its output like an ordinary program – no new computations are triggered by these steps.
Events and output

The event handler *generates new events* as its output, perhaps carrying data values with them.
Deterministic event-handling

Requires
• Atomic write buffers
• Timer support to implement $C$

Also corresponds to the *ideal* classic program model!
Secondary events – direct addressing

Time

parallel

buffered
Secondary events – via brokers
Event correlation (example)

subscribe to $A \& B$ within $T$
Time- vs. event-triggered systems

Time-triggered systems observe the environment and take action on basis of the changes they see.

Event-triggered systems are controlled by the environment, and take action when the environment so decides.

Suitable when input may be constantly changing and all value are equally interesting, like in control systems

Suitable when interesting input values are highly irregular, or when it is already discrete, like in communication systems

Both kinds are naturally non-deterministic, but can be made fully deterministic with some effort
Time- vs. event-triggered systems

- Both kinds of systems can be implemented on top of common OS services for timers and message queues
- Many middleware-based architectures exist
- Hybrid systems – with a mixture of event- and time-triggered processes – can certainly also be constructed
- The division of processes into either kind appears fundamental...
- We may however note two more things...
(1) If we allow offsets in events...

A time-triggered system

An event-triggered system with offsets
(2) If we allow recursive event-triggering...

A periodic time-triggered system

A self-referencing event-triggered system with offsets
(3) If we become explicit about startup...

Must have started at some point!

Explicit startup event!

A time-triggered system

A self-referencing event-triggered system with offsets

Ad infinitum...
Then:

Time-triggered behavior emerges as a special case of an event-triggered system!
Time-triggering as a special case

- A time-triggered behavior is just a chain of event reactions separated by well-defined time offsets.
- A periodic process is such a chain-reaction that oscillates (produces as many new events as those it reacts to).
- Many hybrid variants exist between the extremes of one single reaction and the oscillating periodic behavior.
- Allows us to seamlessly study trade-offs between the basic approaches.
- Note: not the commonly taught real-time systems view!
- It is however the view we find in TinyTimber!
TinyTimber

- A run-time kernel + a design style for programming embedded real-time systems in C
- Also a cut-down variant of the programming language Timber (timber-lang.org). Will appear later...

Basic ideas:
- Events can be triggered with time offsets
- Events = asynchronous method calls
- Methods belong to objects
- Objects = protected sets of state variables
- Also: synchronous method calls (mimic read/write)
TinyTimber

- Is fundamentally event-based, and therefore **non-deterministic** on the system level (but individual methods are of course deterministic)
- Uses **no event brokers** – all method calls are directed to a particular object (although brokers can be implemented as ordinary objects)
- Offers **no form of event correlation** – any such behavior must be explicitly coded
- Is inherently as **unsafe** as C (on the other hand, full Timber is a *very* safe language)
A TinyTimber run-time scenario

- Asynchronous calls
- Buffered objects
- Blocked synchronous call
- ... with offset

Time
In concrete C

typedef struct {
    Object super;
    int value;
    int enabled;
} Counter;

#define initCounter(en) { initObject(), 0, en }

int inc( Counter *self, int arg ) {
    if (self->enabled)
        self->value = self->value + arg;
    return self->value;
}

int enable( Counter *self, int arg ) {
    self->enable = arg;
    return 0;
}

Counter cA = initCounter(1);
Counter cB = initCounter(0);
Calling methods

Asynchronous call

Synchronous call

Asynchronous call with offset

Top-level application setup

Counter cA = initCounter(1);
Counter cB = initCounter(0);
MyApp obj = initMyApp();

main() {
    INSTALL( &cA, inc, IRQ1 );
    INSTALL( &obj, handler, IRQ2 );
    return TINYTIMBER( &obj, startup, 0 );
}
Executing periodically

int tick( MyApp *self, int arg ) {
    ...
    AFTER( SEC(2), self, tick, arg );
    ...
}

N.B.: Oscillation must be triggered by an initial event...

... either at application setup:

MyApp obj = initMyApp();
main() {
    return TINYTIMBER( &obj, tick, 0 );
}

... or with an explicit call:

int some( Some *self, int arg ) {
    ....
    ASYNC( &obj, tick, 0 );
    ....
}
Measuring time

```
Timer ti = initTimer();

int methA( MyApp *self, int arg ) {
    ...
    T_RESET( &ti );
    ...
}

...

int methB ( Some *self, int arg ) {
    ...
    Time measure = T_SAMPLE( &t7 );
    ...
}
```

Obtain the time between ...

... the baseline of last call to methA

... and the baseline of a subsequent call to methB
Measuring time

**Measured time**

- **T_RESET()**
- **T_SAMPLE()**

Event A

Event B

**Not measured**

*(not very interesting)*
If necessary...

N.B.: Time offset from baseline is usually small, and can be constrained by a deadline instead of being measured.
However...

In the degenerate case of **busy-waiting**:

Here `CURRENT_OFFSET()` becomes the *only* means of measuring time, since all execution takes place within one single method call.
Lab 1

- Exploring the effects time-triggered and event-triggered principles have on the energy consumption of an embedded system
- For more info on the TinyTimber kernel, see report Programming with the TinyTimber kernel (available on the course homepage)
- For help with the lab environment see the Lab PM (also on the homepage)
- Or see your lab instructor!
- Lab 1 deadline is ...