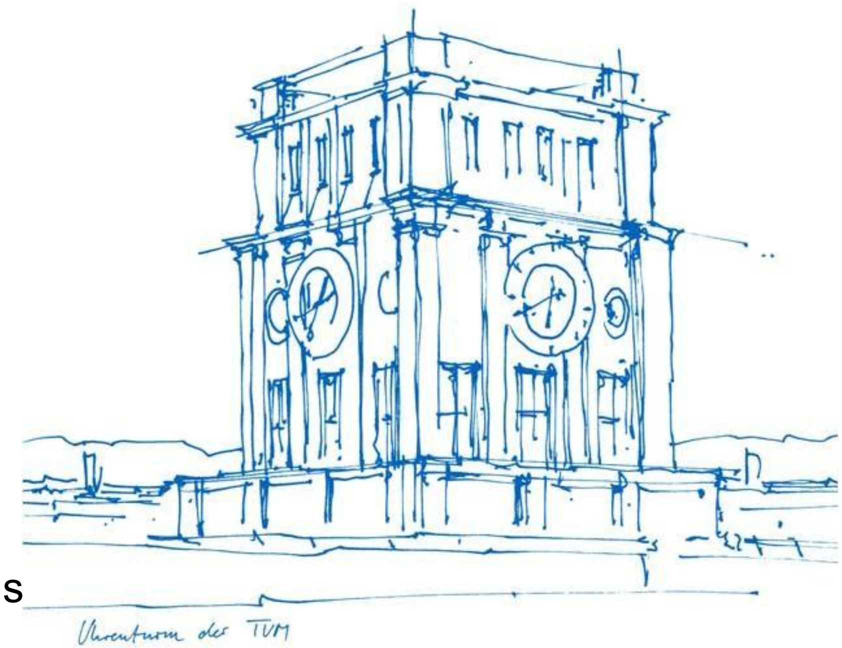
Real-Time and Embedded Systems @ SIT

Lecture 1: Introduction & Software Development overview

Alexander Hoffman

Technical University of Munich
Department of Electrical and Computer

Engineering Chair of Real-Time Computer Systems



Course Organization with me - This week

- Lecture every day from 9:30-12:30

- Tutorial from 13:30-15:30 in computer lab
  - C-programming lab

Topics:

- Programming and compiling C code
- FreeRTOS
- Scheduling
- WCET and WCRT Analysis

Please Ask Questions

- There is no textbook for this course (slides get uploaded)
- I don’t know your background
- If you have a question, you are very likely not alone with it
- Makes it easier to stay awake

Embedded Systems

Quiz:

What is an embedded system?

What is an Embedded System?

What is an Embedded System?

What is an Embedded System?

What is an Embedded System?

What is an Embedded System?

Embedded Systems

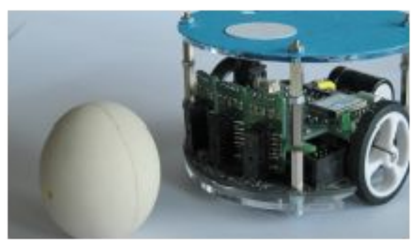
Quiz:

What is an embedded system?

A computer system that is embedded in (i.e. an integral part of) the actual product / device that the user interacts with.

Why are embedded systems so special?

Embedded Systems



Interacts with environment, mostly mobile, no keyboard, no monitor, safety-critical.

Personal Computer

Lot of memory, fast processor, direct I/O via monitor, keyboard.

Why are embedded systems so special?

Embedded

puter

Some systems are a combination of both

Interacts with environ memory, fast processor, direct
no keyboard, no monitor, critica ia monitor, keyboard.

Embedded Systems

Quiz:

Name five characteristics of typical embedded systems

- No (general purpose) human-machine-interface
- Real-time constraints
- Dedicated functionality
- Interaction with the physical environment
- Heterogeneous (different parts manufactured by different vendors, have different interfaces, scheduling policies, etc.)
- Constraints on power, size, manufacturing cost …
- Irregular design (hardware software partitioning)

Real-Time Systems

What is Real-Time?

- Physical time that passes while program is running
- Real-Time Programs interact with environment that CAN NOT WAIT.

Examples for Real-Time Systems:

- Driver/pilot assistant
- Virtually any computer that directly controls a machine
- Telecommunication
- Stock market trading clients
- (Some) Computer games

Real-Time Systems

Stockmarket price for some effect over one day

Real-Time Systems

Soft-Time vs Real-Time:

- Soft-Time is the (abstract) model the programmer has of the progression of physical time during program execution

Soft Real-Time vs Hard Real Time

- Soft: Deadlines **may** be missed (Result **less** useful after Deadline)
- Hard: Deadlines **must not** be missed (Result **not** useful after Deadline)
- Whether a deadline is hard or soft is determined by the specification of the system

Real-Time Systems

Programming Real-Time Systems is not (just) about being fast on average.

- It is about being predictable
- Especially for hard real-time systems, **knowing** and **optimizing** for the worst case is more important that optimizing the average case.

Some other important Concepts

Reactive Systems

Reactive Systems

- Programs that run continuously and react to external events (e.g. change in sensor readings).

- As opposed to simple transformational programs that are started, load an existing dataset from memory, transform that data and store it back to memory and terminate (e.g. file compression)

- It is important that a system can react to higher priority events even while processing a low priority event.

Some other important Concepts

Concurrency and Parallelism

“Nock Nock”
“Race Condition”
“Who’s there?”

Concurrency and Parallelism

What is the difference?

- Concurrency: Multiple different things can happen on the same system during overlapping time periods independently until interaction is required.
- Parallelism: A task is split up in separate chunks that are processed literally at the same time by different parts of the system (multiple cores, CPU + GPU, vector instructions)

Concurrency and Parallelism

What is the advantage?

- Concurrency:
  - Stay reactive
  - Better decoupling of different tasks, separation of concerns

- Parallelism:
  - Get task(s) completed faster.

Concurrency and Parallelism

What is the problem?

- Overhead (synchronization and communication requires additional logic)

- Race-Conditions: Timing does not only become important w.r.t. the physical world, but also w.r.t. other parts of the program.

Race condition: when two or more operations are performed at the same time which require a certain order to be done correctly.

Eg. writing to memory while it‘s still being read.

Some other important Concepts

Models

Models

Quiz:

What is a model?

A (usually formal and simplified) description of certain aspects of a system expressed in some modelling language.

Models

Examples

- Model of gravity
- Time series models
- Simulink models of mechanical systems.
- C - Programs

Purpose:

- Describe, analyze and understand an existing system
- Specify the desired behavior of a system.
  - Some models can be used to automatically generate a valid implementation of that specification.

Models

Models can be created in different ways:

- By observation of an existing system
- From other models
  - By composing them
  - By abstracting lower level models
  - By refining higher level models
- By designing one for later implementation

Important:

- A model describes the observable behavior of a system, not necessarily how the system actually works. eg. state machine

Models

Common trade offs:

- The more accurate a model is, the more complex it becomes and the harder it is to analyze.
- In theory, different aspects of a system can be captured in different models. In practice this separation of concerns comes with a loss of accuracy during analysis and/or efficiency during design.

Embedded Software Development

Or

“How do we write software (firmware)”

Embedded Software Development

Brainstorming:

How do you develop embedded software?

- Determine functional and non-functional
- requirements System design / task decomposition
- Design application logic (e.g. control
- algorithm) Write / extend software
- Compile & deploy
- Verify functional properties
- Verify non-functional properties
- Debugging

Programming Languages

Or

“How to tell the computer what to do”

Programming Languages

What is a programming language?

- Set of syntactic and semantic rules defining a valid program
- Specifies observable behavior of a program generated from a valid piece of code

What is the program code?

- Detailed, formal specification of desired observable behavior of the program.
- NOT a direct representation of actual instructions executed on CPU (except for assembler code)

Programming Paradigms

Paradigm

- Central abstractions around which code is organized

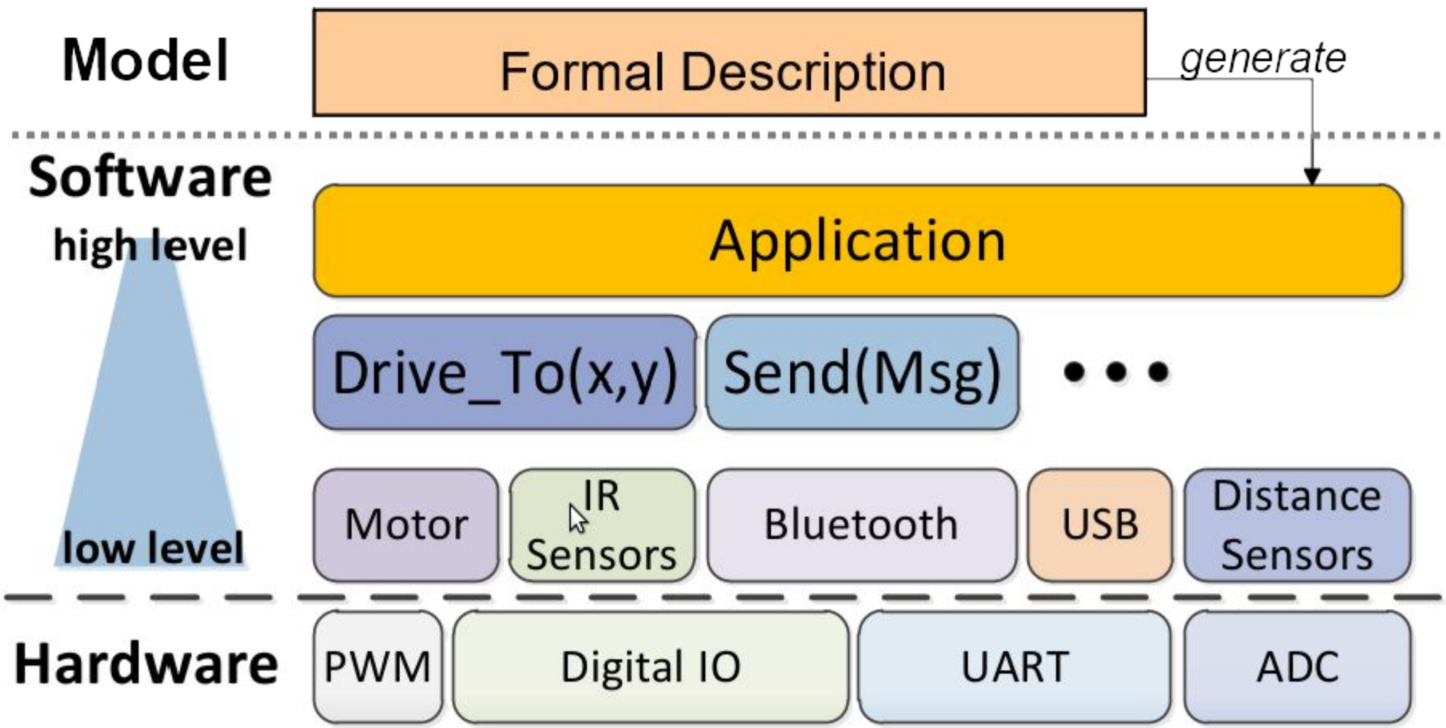
Each language can follow multiple paradigms:

- Imperative: you give commands
- Object-oriented: object with properties
- Declarative: you say „what“, but not „how“ it happens
- … (knock yourself out...Wikipedia knows them all)

Real-time programming paradigms:

- Synchronous
- Time-triggered
- Scheduled

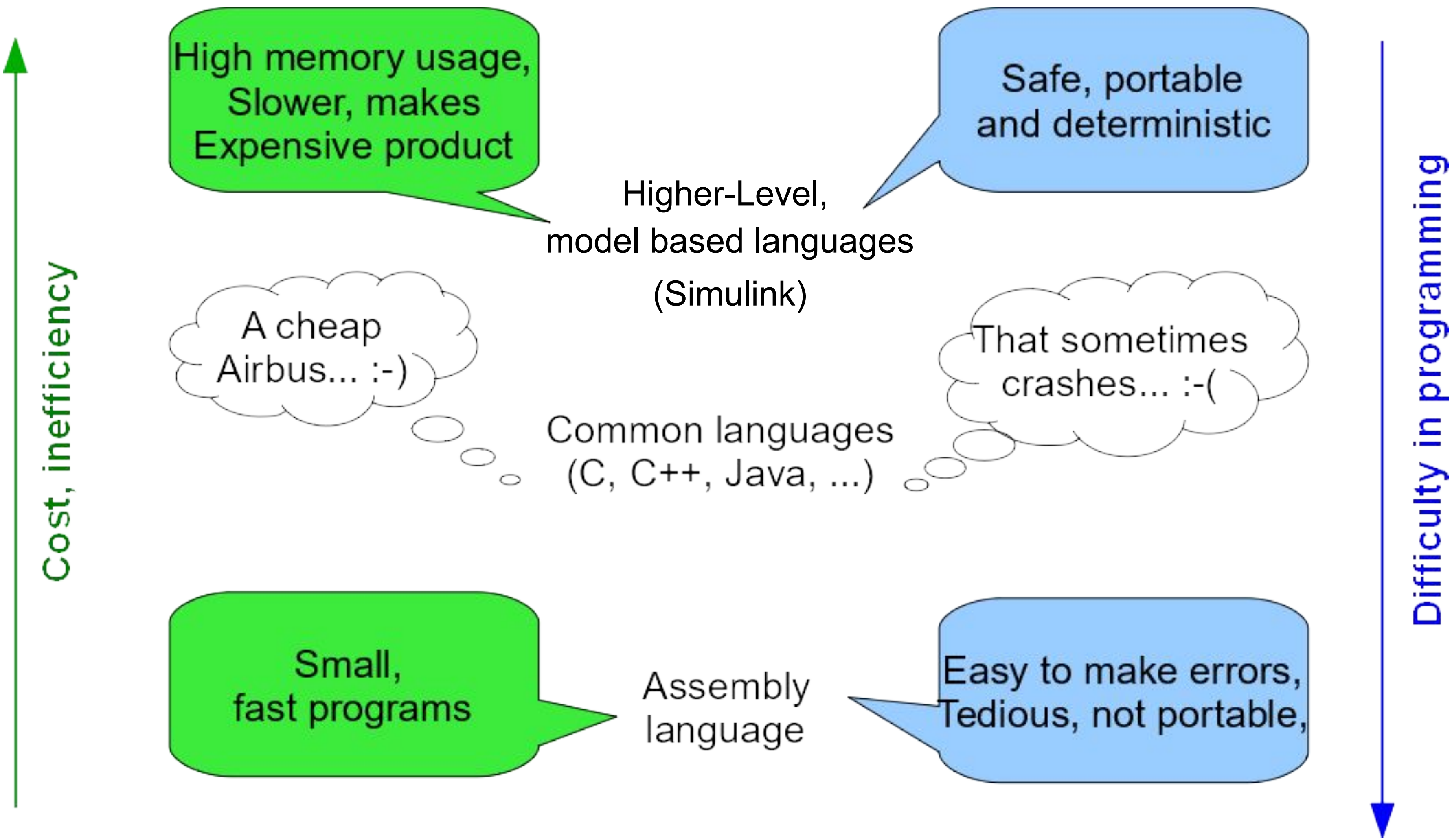
Programming Languages



Programming tasks:

- Low-level code for custom sensors or actuators („drivers") or speed-critical
- parts Abstract high-level code as much as possible, ideally coming from even higher- layer models

Programming Languages



Why not write everything in C?

Easy to make mistakes

- null pointer, logic errors, uninitialized variables
- Embedded system could behave incorrectly, freeze or crash

No intrinsic support for concurrency

- A lot of manual work
- Risk of deadlock, race conditions → freeze, crash etc.

Reasoning about „correct code“ is arbitrarily hard

- How to proof that the program never crashes?

Why not write everything in C?

Some constructs in C have undefined semantics (see C std)

C cannot express time, only function (no „temporal semantics“)

- For real-time systems, a specification of behavior w/o timing is useless!
- Lot of effort to get simple time specifications right
- apparent nondeterminism

Same holds for most other „mainstream“ languages.

Why is C used so much in Embedded Systems?

Next to no overhead

- No virtual machine
- Almost no startup code necessary
- Decades worth of compiler optimizations
- Does not have to track runtime information

Operates on raw memory

- Necessary for memory mapped I/O
- Allows control over data layout

Compatibility:

- Every system has a C compiler
- Lots of existing software

Compilation

(Cross-) Compilation: A Definition

Translates code into another - more low-level - code form:

- E.g. C code into „assembly code“, creating object files
- Output is processor- and operating-system-specific

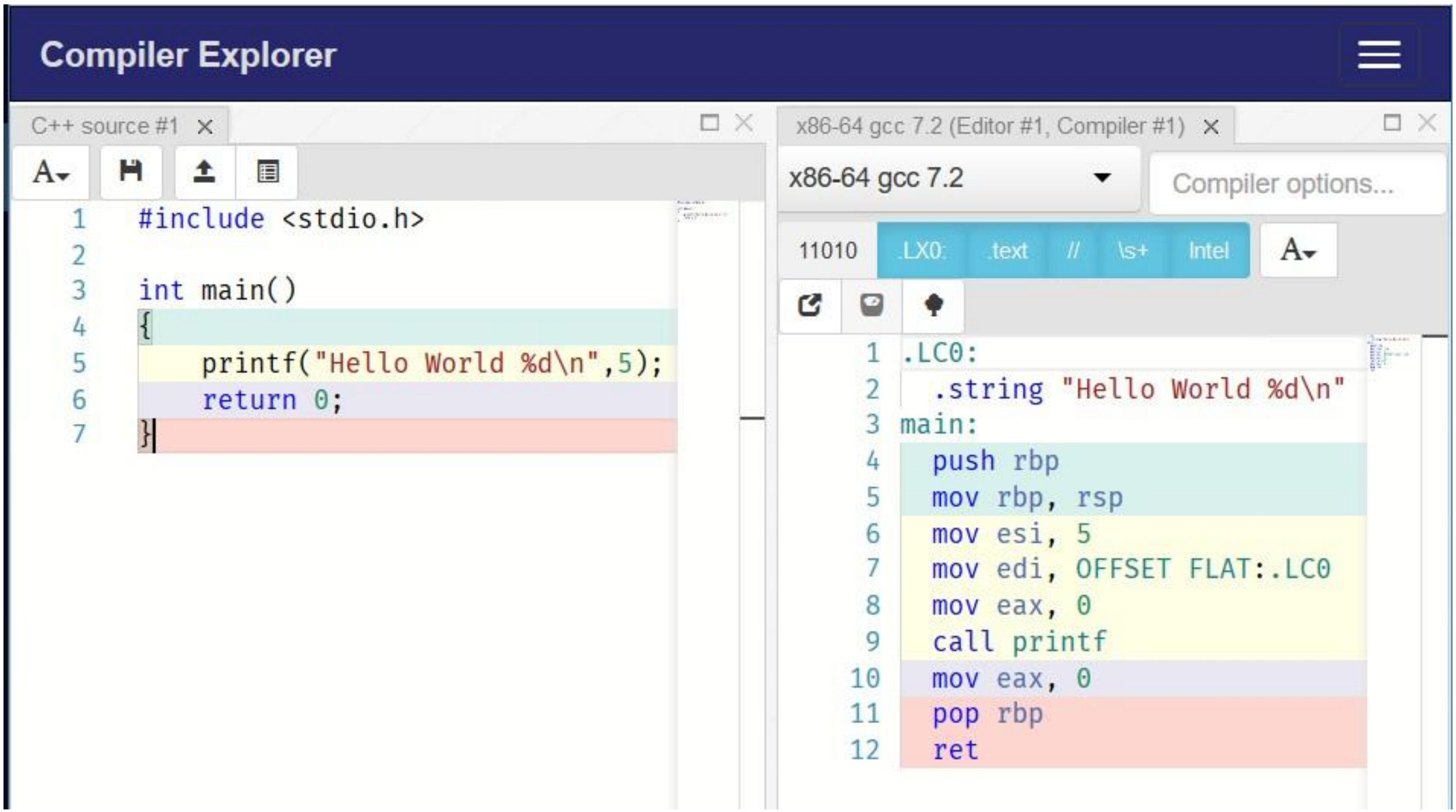
Steps during compilation:

- Analyze code (syntax, semantics) according to a grammar and spec
- Generate low-level code (assign registers, organize function calls, ...)
- Optimize (unreachable code, constant values, re-ordering, ...)

„Cross-Compilation“:

- Output format for a different architecture than the one the compiler runs on
- compile on host, but execute on target → most common for embedded systems

Compilation Example

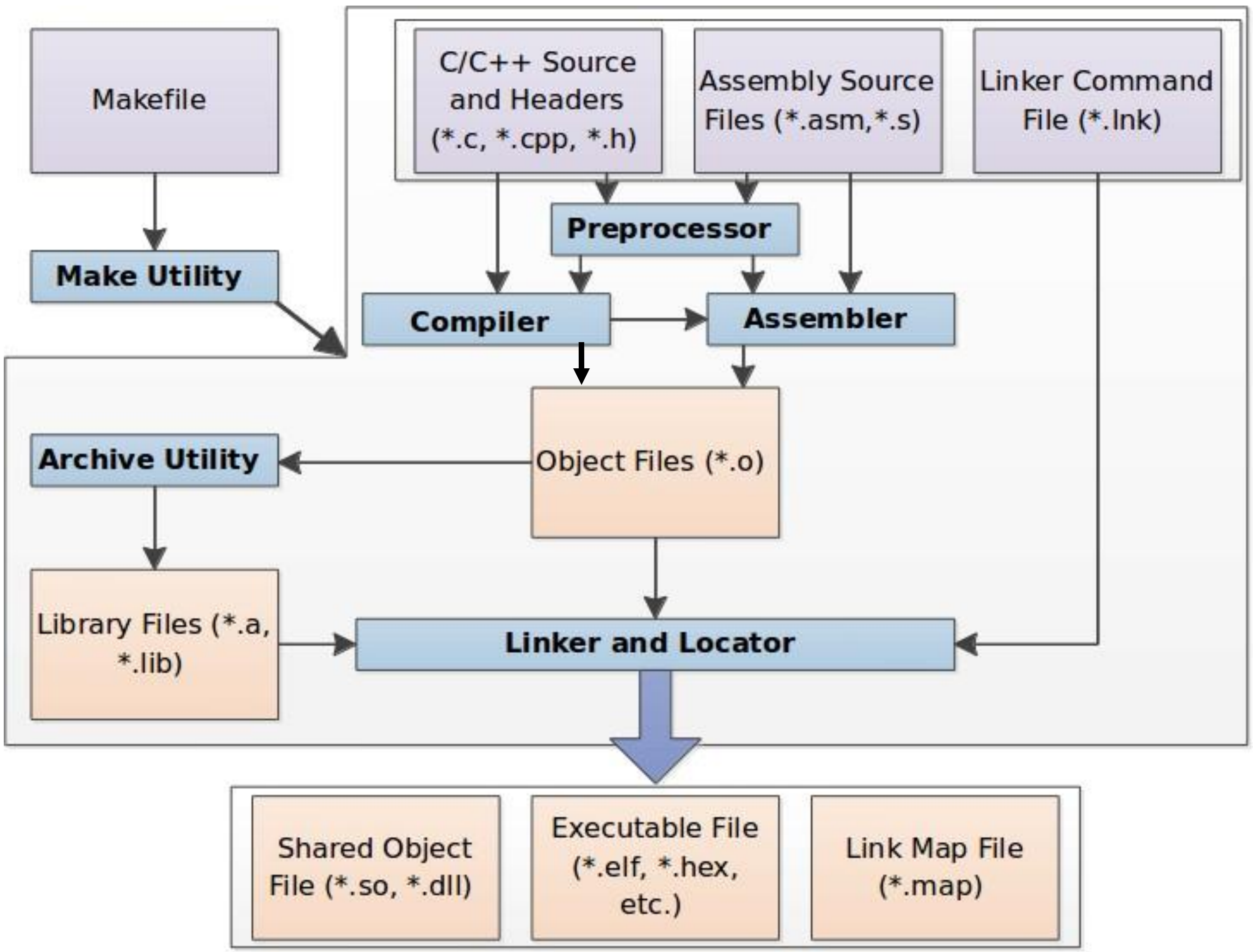


https://gcc.godbolt.org

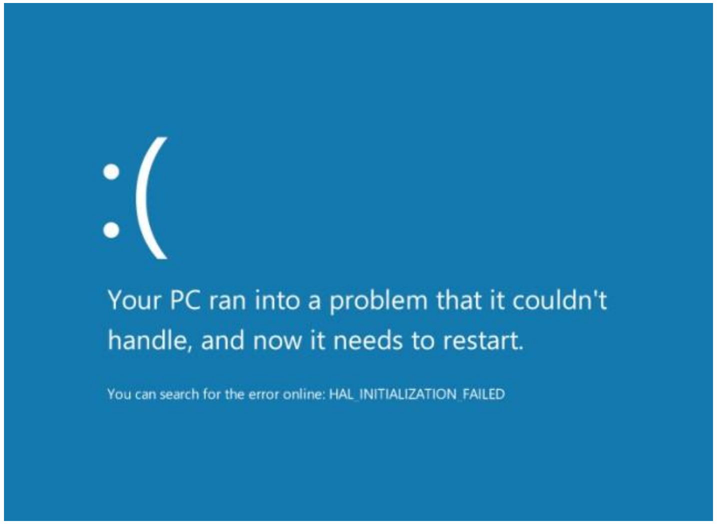
Linking

- Combine object files and libraries to a single executable program
- “dynamic”: application still is lacking some functions and will query the operating system for libraries when executed → smaller programs by re-use of libraries, but sometimes problems with the version of the library (expected vs. provided)
- “static”: self-contained application. Bigger programs, but no version conflicts.
- Often you “link against” libraries from the board vendor, contained in the “board support package (BSP)”
- Also defines memory addresses and mappings
- Resolves name conflicts
- Usually the output from the linker can be downloaded to the target and be executed (HEX/ELF file)

From Source Code to an Executable Program



Functional Verification



Functional Verification

Naïve approach (Testing) :

- Testing by feeding input data and see what happens
- In general impossible cover all cases, need statistics
- No formal proof („...the new Airbus is unlikely to crash...“)
  →would that be good enough for you?

Formal proof of program correctness („**Model Checking**“)

- e.g., proof that no combination of buttons can bring the airplane into *parking mode during flight*

Temporal


PLEASE WAIT...
km/h
1/min x 1000
6.8 km/h
300
Zeit
14:54
km
00044
044.4
A-Temp
+26.5 °C

Temporal Verification

- Timing of the embedded system is a crucial part of the correctness
- Need machine instructions to judge (i.e., assembler output)
- Need to make sure that
  - The computations don't take too long (think of Windows...can you tell how long it takes to start MS Word?) → meeting deadlines
  - The system can operate at a desired rate → processor is not overloaded

Debugging

Debugging

Assume: Program executing correctly until particular point

1) Identify bug‘s location roughly by logging or user output
    - Timing behavior
    - State changes
    - etc.
2) Introduce break conditions, e.g.
    - Execution of particular line of code (program counter value)
    - Memory R/W access to specific address
    - Variable‘s value
3) Then
    - Step-by-step execution
    - Watch variables, registers, memory
    - Analyze I/O (e.g., logic analyzer)

Debugging

Debugging

- Some bugs are easy to find
- Some bugs disappear when you start debugging
- Some bugs are too unlikely to show up
- Some bugs cannot be debugged, e.g., timing
- Some bugs will even never happen in the real system, but be caused by debugging itself

Debugging is not the answer for safety-critical real-time systems!