MÁSTER UNIVERSITARIO EN SISTEMAS ESPACIALES
POR LA UNIVERSIDAD POLITÉCNICA DE MADRID

2021.03.18

Data handling
3. Programming

Juan Antonio de la Puente <juan.de.la.puente@upm.es>

STRAST

Outline

- Programs and programming
  - machine code & high-level languages
- Compilation process
  - compilation toolchain
  - embedded software & cross-compilation toolchain
- Programming languages
  - instructions & data
  - program structure
  - examples in Ada
- Laboratory
  - compilation tools for embedded on-board software

Programs and programming

Computer programs

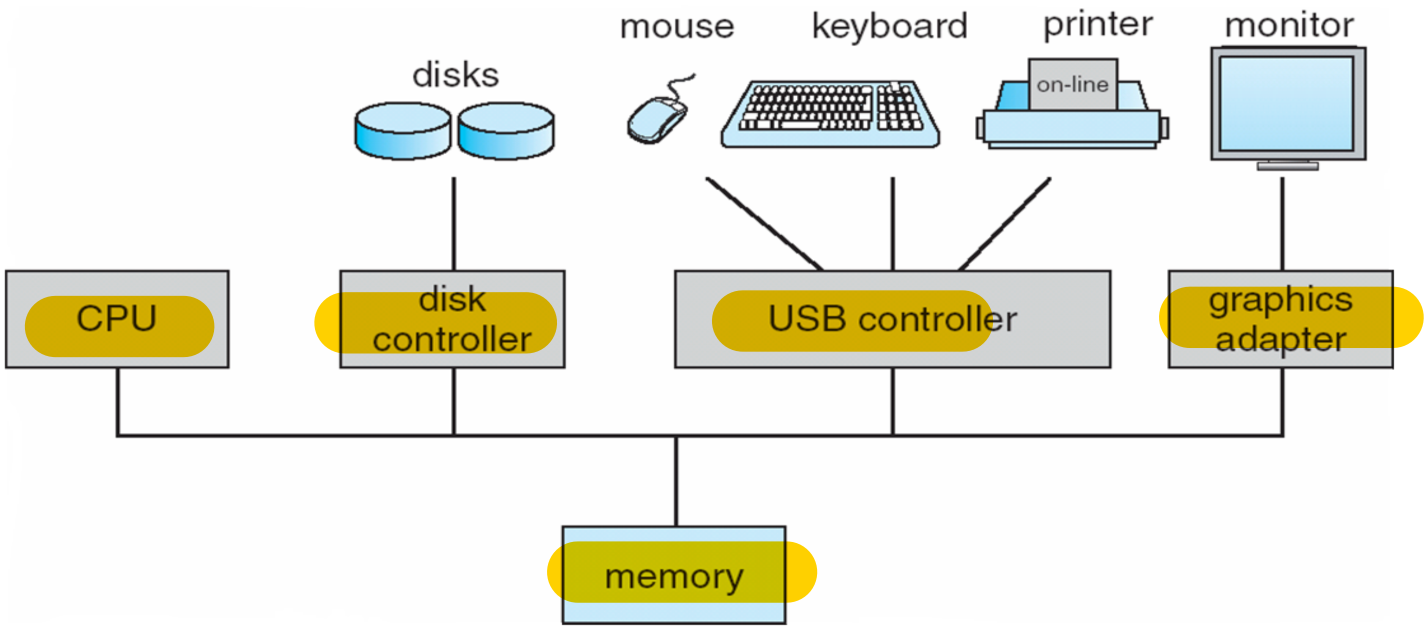
- Computers work by executing programs
  - instructions are executed in sequence
  - instructions and data are stored in the main memory
    - John von Neumann (1945)
  - machine language
    - binary codes for operations and data, e.g

```
1011 0110 0000 0100 0000 0000 0001 1010
```

  - assembly language
    - symbolic representation of machine language, e.g.

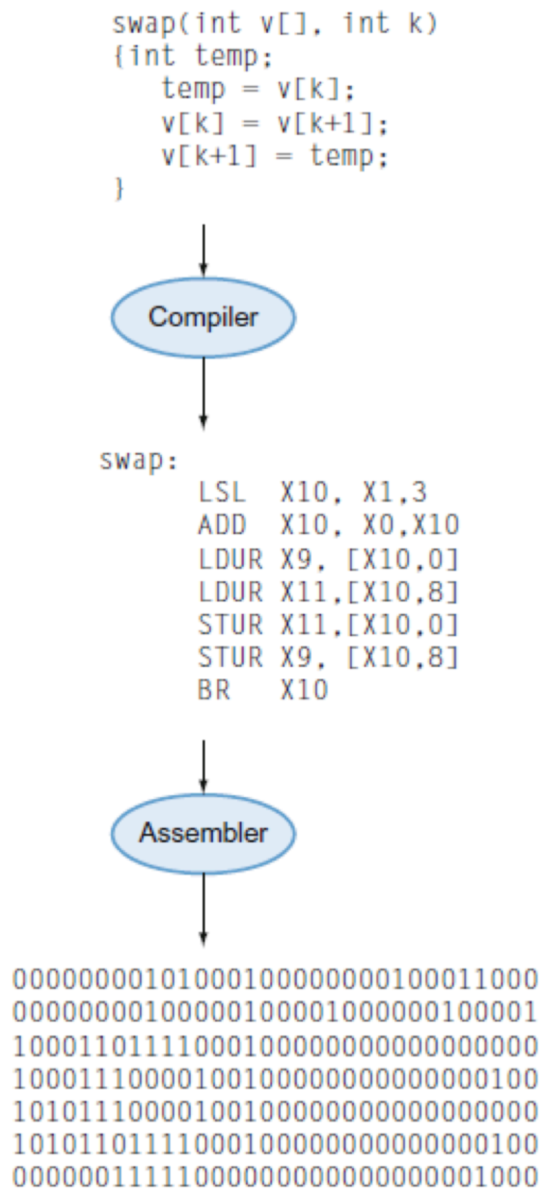
```
add  %r27, %r16, %r26
```

Von Neumann Computer Architecture



Programming languages

- High abstraction language (C, Ada)
  - Adequate for solving the problem
  - Allows productivity and portability
- Assembler language
  - Textual representation of the machine instructions. Relation 1 a 1
  - This code is for the ARMv8 processor
- Machine language
  - Binary digits (ARMv8)
  - Encoded data and instructions



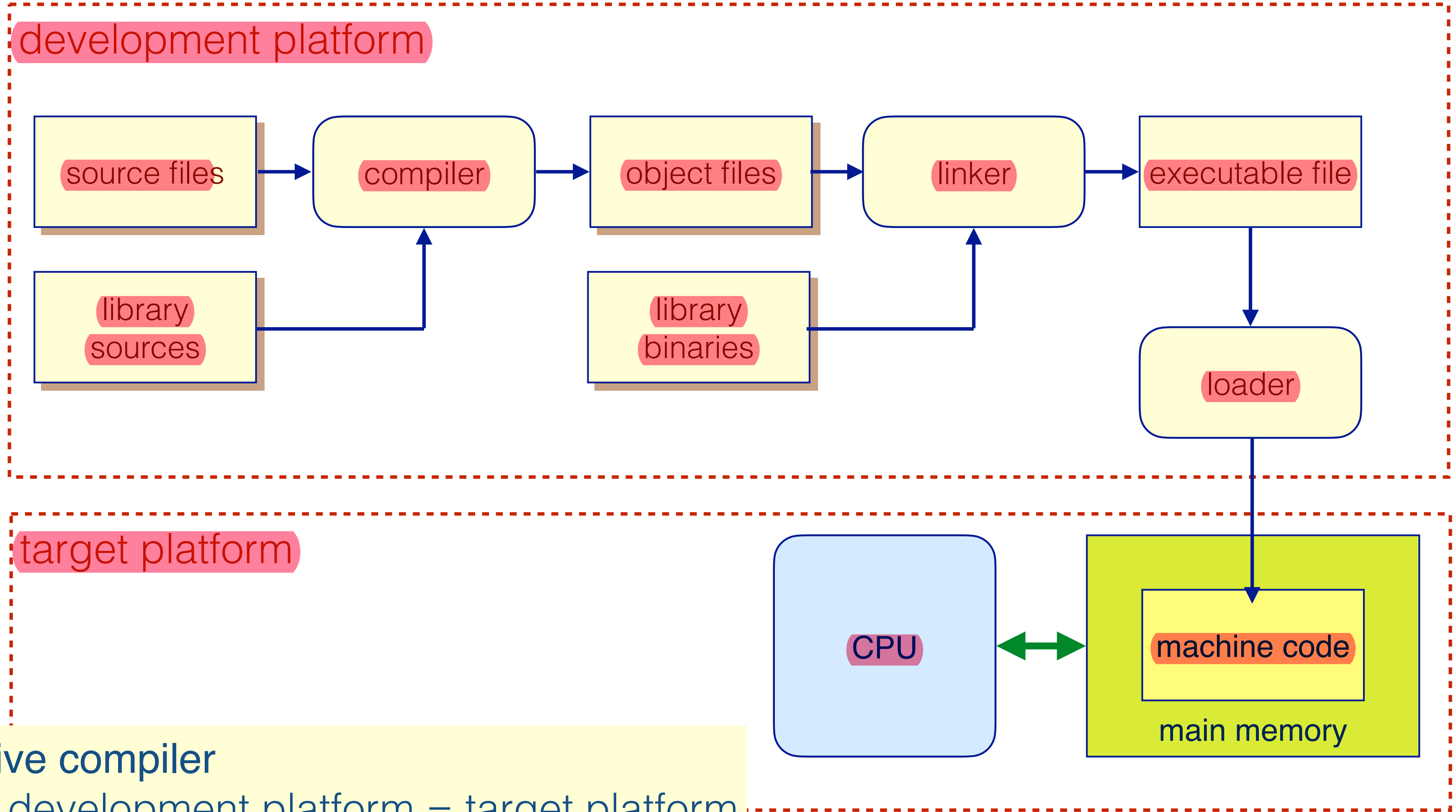
Programming languages

- Binary (or assembly) code is difficult to write and understand
- Programming languages have been developed to facilitate the building of programs
  - the first widespread programming language was Fortran
    - John Backus (IBM,1956)
  - high-level languages are formally defined so they can be automatically translated into machine language
    - the programmer writes the program in an abstract notation and does not have to deal with binary code
    - a single sentence can be translated into many machine instructions
- Commonly used programming languages in the space domain include C and Ada

Compilation process

- Most programs are written as a set of source files
  - “divide and conquer” strategy helps dealing with complexity
- Source files are compiled (translated) into a set of binary object files
- Object files are linked (combined) to produce a binary executable file
  - usually including pre-compiled library files
- Executable files are stored on disc and loaded into main memory when they are to be executed

Compilation chain



native compiler

- development platform = target platform

cross compiler

- development platform ≠ target platform

Programming languages

Instructions and data

- Instructions specify operations to be executed when running the program, e.g.

```
x := y + z;
y := cos (w*t);
```

- Data specifications are needed to define the data values (variables and constants) on which the operations are performed, e.g.

```
x, y, z : Float;
N : Integer := 10;
```

- Type specifications are needed to define the possible values and the operations that can be executed on a set of data items, e.g.

```
type Azimuth is digits 6 range 0.0..360.0;
type Counter is range 1..1000;
```

Program structure

- A program is usually composed of modules
  - source files encapsulating data definitions and operations
- Operations are usually defined as subprograms
  - sequences of instructions that are executed when the subprogram is called
- There is usually a main subprogram which is called by the operating system when the program is loaded
  - the main subprogram usually calls subprograms in other modules

Common programming languages

- C was designed for building operating systems
  - in particular the original Unix
    - Dennis Ritchie, AT&T Bell Labs (1969)
- Ada was designed for building real-time embedded systems
    - Jean Ichbiah et al. (CII Honeywell Bul), under contract to the US Department of Defense (1977–1983)

C features

- Simple, efficient, easy to learn
  - efficient compilation into machine code
  - but error-prone in some ways
- Compilers for many different architectures
  - part of the `gcc` toolchain
    - ✓ GNU Compiler Collection
- C standard library
  - common functions for mathematics, I/O, memory allocation, etc.
  - `glib` is the GNU implementation
  - `newlib` is a limited implementation for embedded systems
- Current standard is ISO/IEC 9899:2018

Example

main function

library definitions

```c
#include <stdio.h>

int main()
{
    printf("Hello, world\n");
}
```

library function

Example

library definitions

```
with Ada.Text_IO;
procedure Hello is
   use Ada.Text_IO;
begin
  Put_Line ("Hello, world!");
end Hello;
```

main procedure

access to library

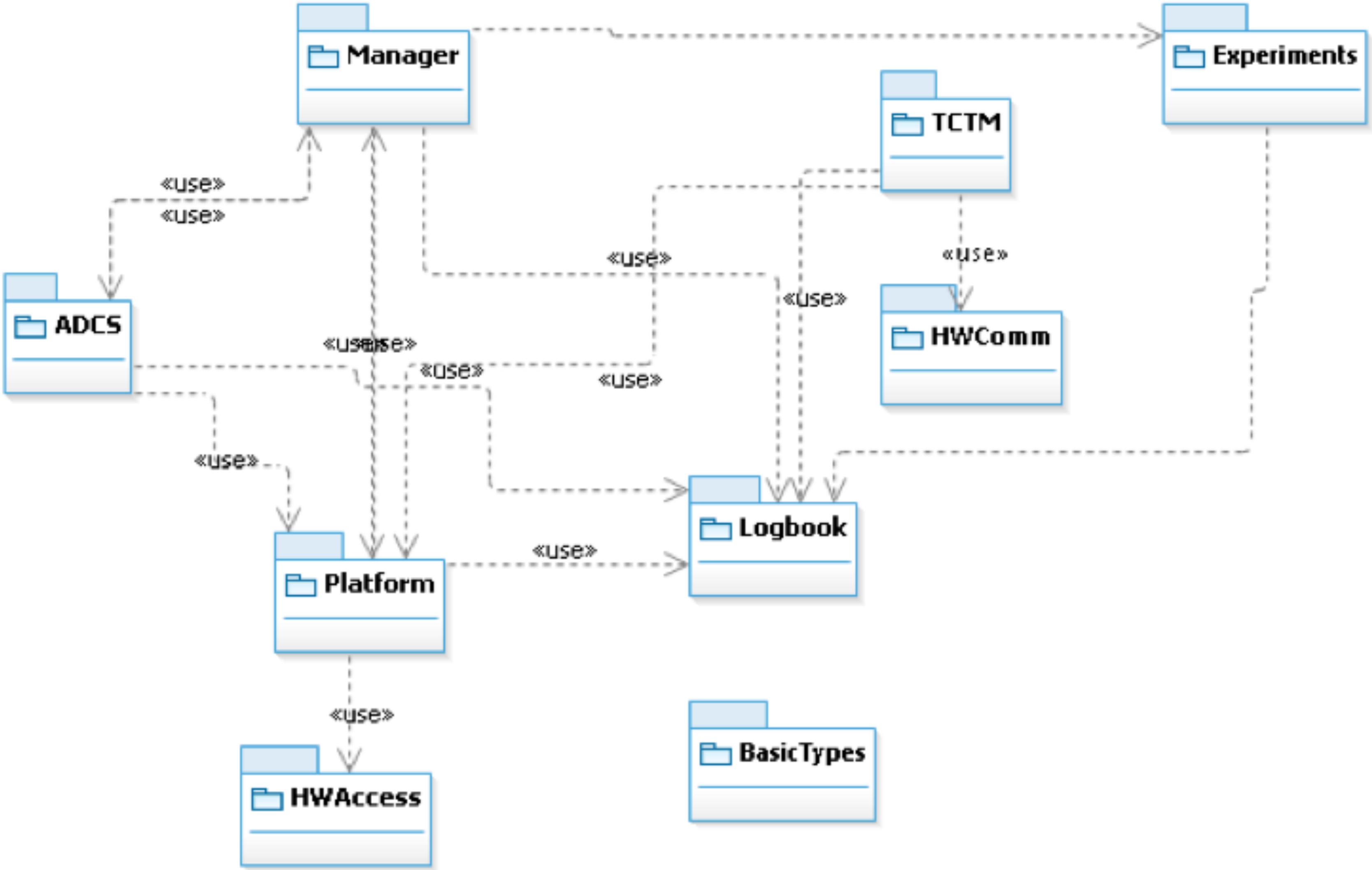
library procedure

Ada basics

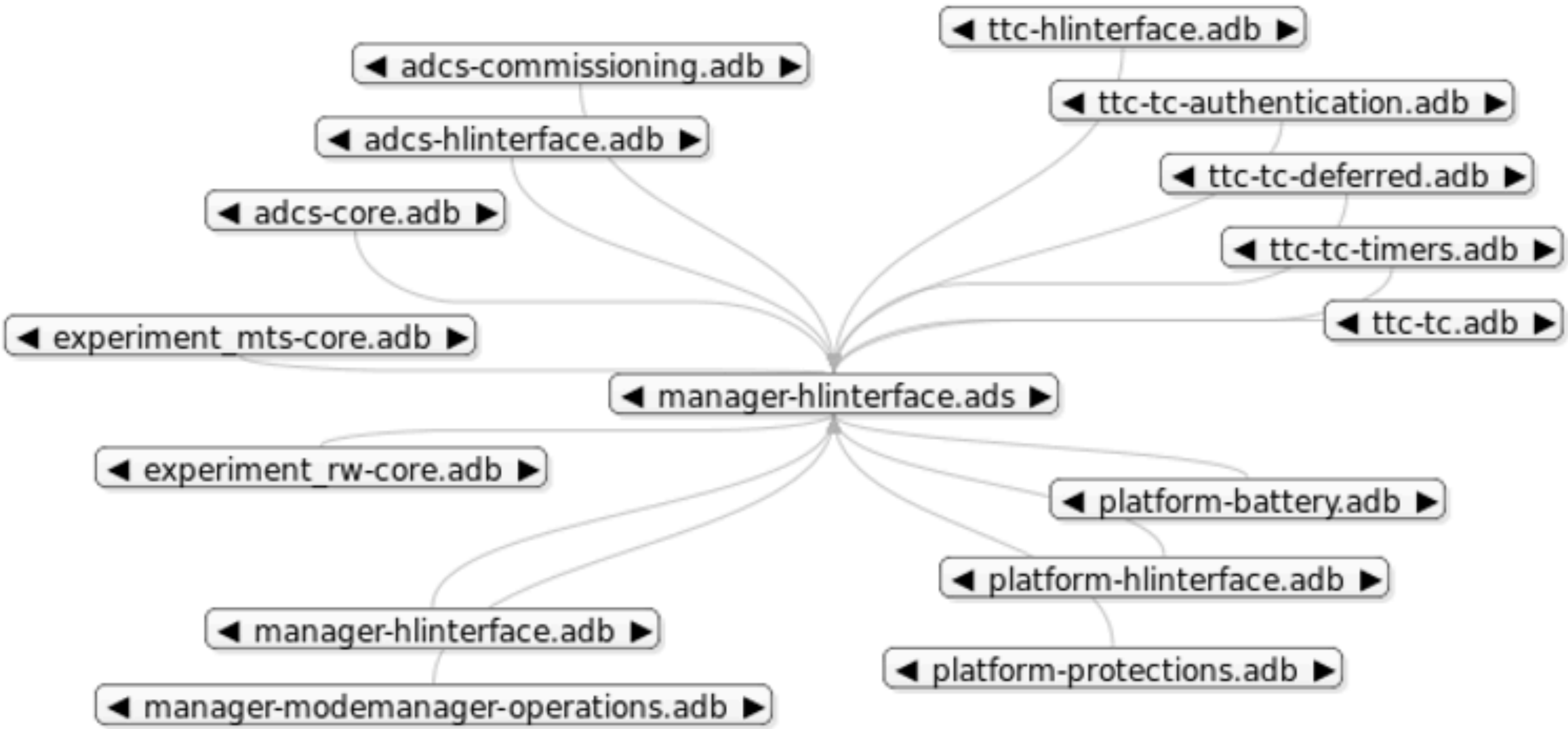
Ada program structure

- An Ada program consists of one or more program units
  - subprograms / operations
    - functions and procedures
  - packages
    - encapsulate data and operation definitions
- Subprograms and packages can also be compilation units
  - source files that are compiled separately
- There is a main procedure
    - may have any name
  - the rest of the program goes into packages
- There is a library with pre-defined packages

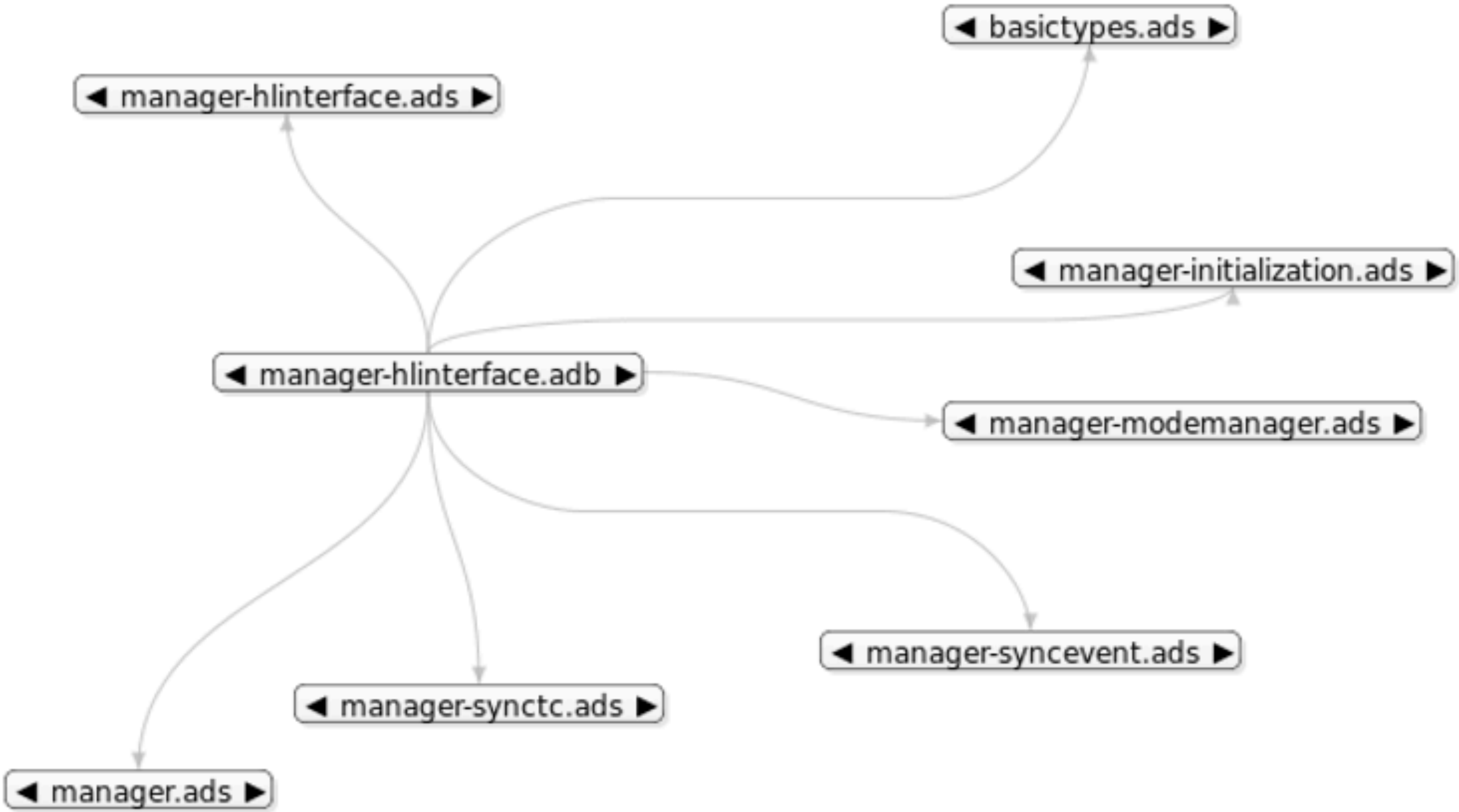
Architecture software



Manager.ads: external dependencies



Manager.adb: internal dependencies



Ada

- Embedded system programming language
  - first designed for building defence-related systems
  - named after Lady Ada Lovelace (1815-1852)
- Advanced features
  - object-oriented programming, templates, contracts, concurrency, real-time
  - support for high-integrity programming
  - support for mixed language programming and interaction with hardware
- Compilers for mainstream processor architectures
  - GNAT toolchain integrated with gcc
  - many other tools available
- Current standard is ISO/IEC 8652:2012

Specification and body

- The specification of a program unit contains only the minimum definitions that are needed by other parts of the program
  - what it does

```
procedure Write (S : in String);
```

- The body of a program unit contains the full details that are needed to make the unit execute
  - how it does it

```
procedure Write (S : in String) is
  c : Character;
begin
  for i in S'Range loop
    c := S(i);
  end loop;
end Write;
```

Procedures and functions

- A procedure is an abstraction of an action
  - named as a verb

```
procedure Write (S : in String) is
  c : Character;
begin
  for i in S'Range loop
    c := S(i);
  end loop;
end Write;
```

- A function is an abstraction of a value
  - named as a noun
  - returns a value
  - Boolean functions named after "Is..."

```
function Square (X : Float)
   return Float is
begin
  return X*X;
end Square;
```

```
function IsEven (N : Natural)
   return Boolean is
begin
  return N mod 2 = 0;
end Square;
```

- Subprograms are executed when called from other program units

Declarations and instructions

- Declarations define the names, types, and other properties of data and other program elements
- Instructions define the operations that the program executes

```
procedure P (…) is
   -- declarations go here
begin
   -- instructions go here
end P;
```

- When a program unit is required to execute
  - declarations are *elaborated*
    - e.g. create local variables
  - then the instructions are *executed* from the beginning

Subprogram arguments

- Procedures can have arguments in three modes
  - `in` mode
    - the value of the argument is not changed during the execution
    - this is the default
  - `out` mode
    - the value of the argument is computed by the subprogram and returned to the caller
  - `in out` mode
    - the caller provides a value that can be changed and returned to the caller

Packages

- A package is a program module where data and operations can be declared
- The package specification contains its visible interface
  - declarations of data, data types, and subprograms
  - no instructions or bodies are allowed
- The package body contains the implementation details
  - bodies of subprograms declared in specification
  - other data and subprograms not needed in other parts of the program
- Packages are Ada's mechanism for encapsulation and information hiding
  - David Parnas (1972)

Package specification and body

```
package P is
   -- declarations
   -- subprogram specs
end P;
```

```
package body P is
   -- declarations
   -- subprogram bodies
end P;
```

- Specifications and bodies usually go into separate source files
  - implementations can be changed while keeping the interfaces

Example

```
package Simple_IO is

  procedure Get (F : out Float);
  procedure Put (F : in  Float);
  procedure Put (S : in  String);
  procedure New_Line;

end Simple_IO;
```

```
with Ada.Text_IO, Ada.Float_text_IO; -- library packages
package body Simple_IO is

  procedure Get (F : out Float) is
  begin
     Ada.Float_Text_IO.Get(F);
  end Get;
  ...
end Simple_IO;
```

Example using Simple_IO

```
with Simple_IO;    -- compile with Simple_IO
procedure Compute_Squares is
   use Simple_IO; -- direct access to Simple_IO declarations
   X : Float      -- local variable
begin
   for I in 1..10 loop     -- repeat 10 times
      Get(X);              -- read a real number
      Put(X);
      Put("^2" = ");
      Put(X*X);            -- write its square
      New_Line;
   end loop;
end Compute_Squares;
```

Data types

Data types in Ada

- A data type is a set of values and a set of operations that can be applied to those values
  - e.g. the set of integer numbers and the integer arithmetic operations
- Ada is a strongly-typed language
  - an object can only have a type

```
type T is ...; -- most types are defined by a type declaration
X : T;         -- An object declaration defines its name and type
```

  - a value of a type cannot be used where another type is expected

```
X : Integer;
Y : Float;

Y := X + 1;    -- illegal
```

Enumeration types

- Programmer-defined
  - values: type Mode is (Off, Safe, Nominal);
- Boolean
  - values: `True, False`
  - operators: and, or, not, xor
- Character
  - values: 8-bit latin-1 (row 00 of ISO/IEC 10646:2003 BMP)
  - e.g. 'a' '0' '#' 'ñ'
  - also
    - Wide_Character: 16-bit ISO 10646 BMP
    - Wide_Wide_Character 32-bit ISO 10646

Integer types

- Predefined

  ```
  type Integer
  ```

  - values: `Integer'First..Integer'Last`
  - implementation-dependent, not recommended

- Programmer-defined

  ```
  type Index is range 1..10;
  ```

  - better for portability

- Modular

  ```
  type Byte is mod 2**8;
  ```

  - values 0..255
  - modular arithmetic: 255 + 1 = 0

- Integer operators

  ```
  + - abs * / rem mod **
  ```

Subtypes

- A subtype is a subset of values of a type

```
subtype Small_Index is Index range 1..5;
```

- An object of a subtype belongs to the base type
  - but values are checked for validity at run-time

```
X : Index          -- values from 1 to 10
Y : Small_Index     -- values from 1 to 5
-- Y is of the same type as X,
-- but is in a restricted subtype
X := 4;            -- valid
Y := X;            -- valid
Y := X + 2;        -- invalid
```

- Predefined integer subtypes

```
subtype Natural  is Integer range 0..Integer'Last;
subtype Positive is Integer range 1..Integer'Last;
```

Real types

- Predefined

  `type Float`

  - values: `Float'First..Float'Last`
  - `precision: Float'Digits`
  - implementation-dependent, not recommended

- Programmer-defined

  `type Voltage is digits 5 range 0.0..10.0;`

  - better for portability

- Floating-point operators

  `+ - abs * /`

  `** integer power`

- Fixed-point types

  `type Measurement is delta 0,125 range 0.0..10.0;`

Objects

- Variables

```
X : Natural := 0;
-- a value may be assigned
Y : Voltage;
```

- Constants

```
N      : constant Positive    := 10;
-- a value must be assigned
V_Max  : constant Voltage     := 9.8;
Pi     : constant             := 3.14159_26535_89793;
-- named literals are not objects
```

Arrays

- Arrays are collections of values of the same type
  - individual values are accessed by an index

```
type Voltages is array (Index) of Voltage;
type Values   is array (1..10) of Value;

V : Voltages;

V(1) := 2.5;
V(2) := V(1) + 0.5;
```

- Strings are arrays of characters

```
S : String (1..10);         -- the index of a String must be defined explicitly
T : String := "example";    -- ... or implicitly by assigning it a literal string
S(1..3)    := "abc";        -- a string slice
S(4)       := 'x' ;         -- a string element
```

Records

- Records are collections of values of different type
  - individual values are accessed by a component name

```
type State is record
   Operating_Mode  : Mode;
   Measurement     : Voltage;
end record;

S : State;
S.Operating_Mode := Safe;
X := S.Measurement;
```

Access types

- Access types are references to objects of some type

```
type State_Reference is access State;

R : State_Reference := null;
R := S'Access;
R.Operating_Mode := Nominal;
```

- Access types can be used to create objects dynamically

```
R : State_Reference := new State'(Off, 0.0);
-- not recommended in critical systems
```

Instructions

Instructions in Ada

- Simple instructions
  - assignment

    ```
    X := (Y + Z)/2;
    ```
  - procedure call

    ```
    Simple_IO.Get(X);
    ```
  - null instruction

    ```
    null;
    ```
- Compound instructions
  - block
  - selection
  - iteration

Blocks

- Local declarations and sequence of statements

```
swap:
    declare
        T : Integer;
    begin
        T := X; X := Y; Y := T
    end;
```

Selection

- If-then-else statement

```
if T <= 100.0 then
  P := Max_Power;
elsif T >= 200.0 then
  P := Min_Power;
else
  P := Control(R,t);
end if;
```

- Case statement

```
case Day is
  when Monday                 => Start_Week;
  when Tuesday .. Thursday => Continue_Work;
  when Friday | Saturday    => End_Week;
  when others                 => Relax;
end case;
```

  - all possible values have to be covered

Iteration

- While loop

```
while (X > 0.0) loop
   Get(X);
end loop;
```

- For loop

```
for I in 1..N loop
   Get(V(I));
end loop;
```

- General loop

```
loop
   Get(X);
   Y := Control(X);
   Put(T);
end loop;
```

Data abstraction

Abstract data types

- A type declared in a package together with a set of primitive operations
  - the structure of the type is usually hidden (private)

```
package TM_Messages is
  type TM_Message is private;
  procedure Build (Body    : in array() of Byte;
                   Message : out TM_Message);
  procedure Send (Message : TM_Message);
private
  type TM_Message is record
     Header : array(1..128)  of Byte;
     Body   : array(1..1024) of Byte;
     CRC    : array(1..4) of Byte;
  end TM_Message;
end TM_Messages;
```

Type extension

- An ADT can be extended

```
type Error_Message is new TM_Message with record
   Error : Error_Code;
end Error_Message
```

  - primitive operations are inherited
  - new operations can be added
  - old operations can be redefined (overriden)

- These mechanisms enable Object-Oriented Programming (OOP)

Other data abstraction mechanisms

- Tagged types
  - for class-wide programming and polymorphic operations
- Abstract types
- Interfaces

Generic units

- Templates for building program units using data types and other program elements as parameters

```
generic
   type Byte_Stream is array <> of Byte;
package TM_Messages is
   type TM_Message is private;
   ...
private
  type TM_Message is record
      ...
      Body : Byte_Stream;
      ...
  end record;
end TM_Messages;
```

```
package Data_Message is
   new TM_Messages (Data_Stream);

package Error_Message is
   new TM_Messages (Error_Stream);
```

Bibliography

- Alan Burns & Andy Wellings
  Analysable Real-Time Systems Programmed in Ada
  Addison-Wesley, 2016
- John Barnes
  Programming in Ada 2012
  Cambridge University Press, 2014