SWE30001 Real-Time Programming

17/03/2015

Lecture 3:

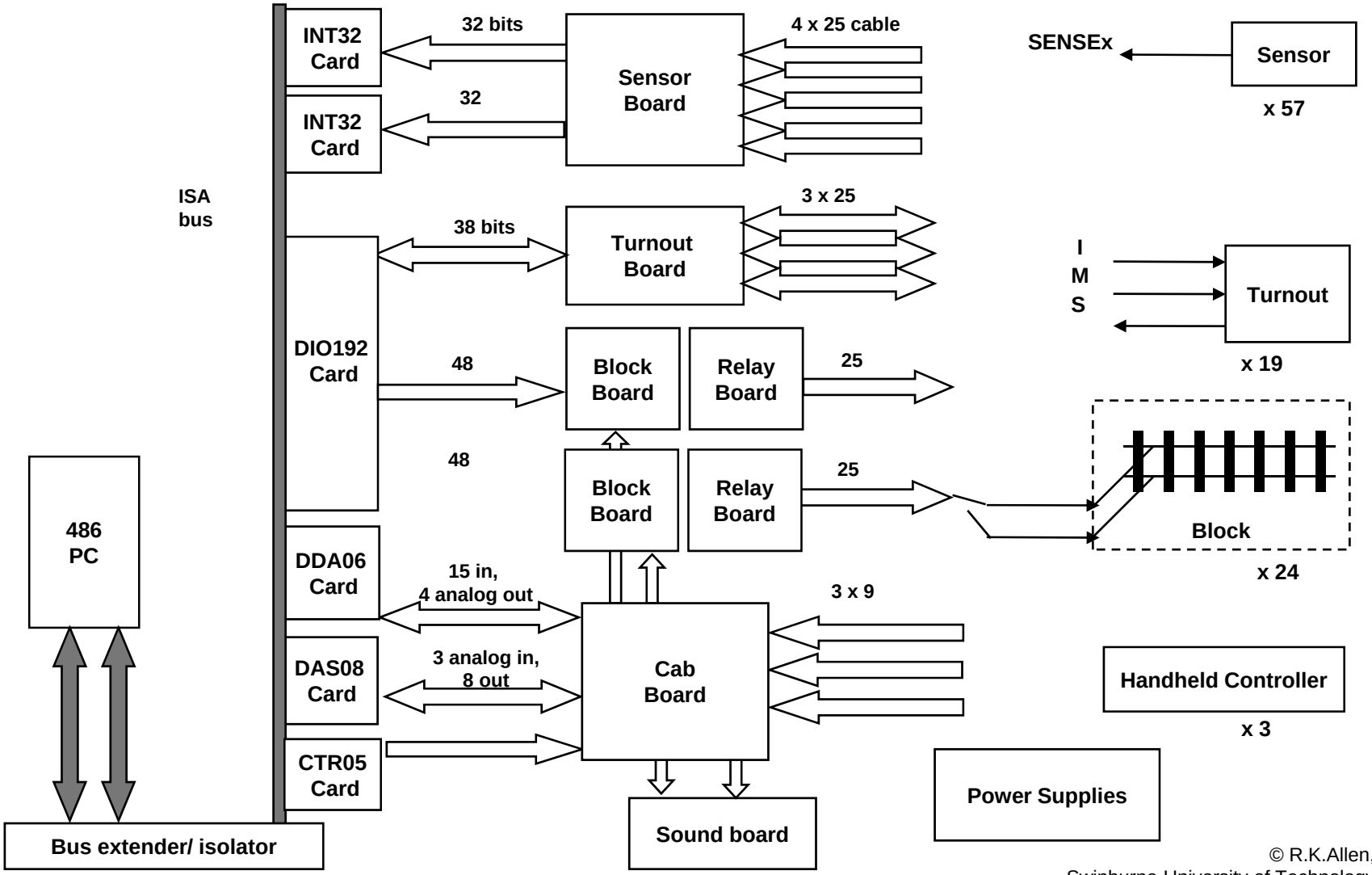
Train Interface and Representation in Ada95

Architecture,
representation clauses,
Dio192defs

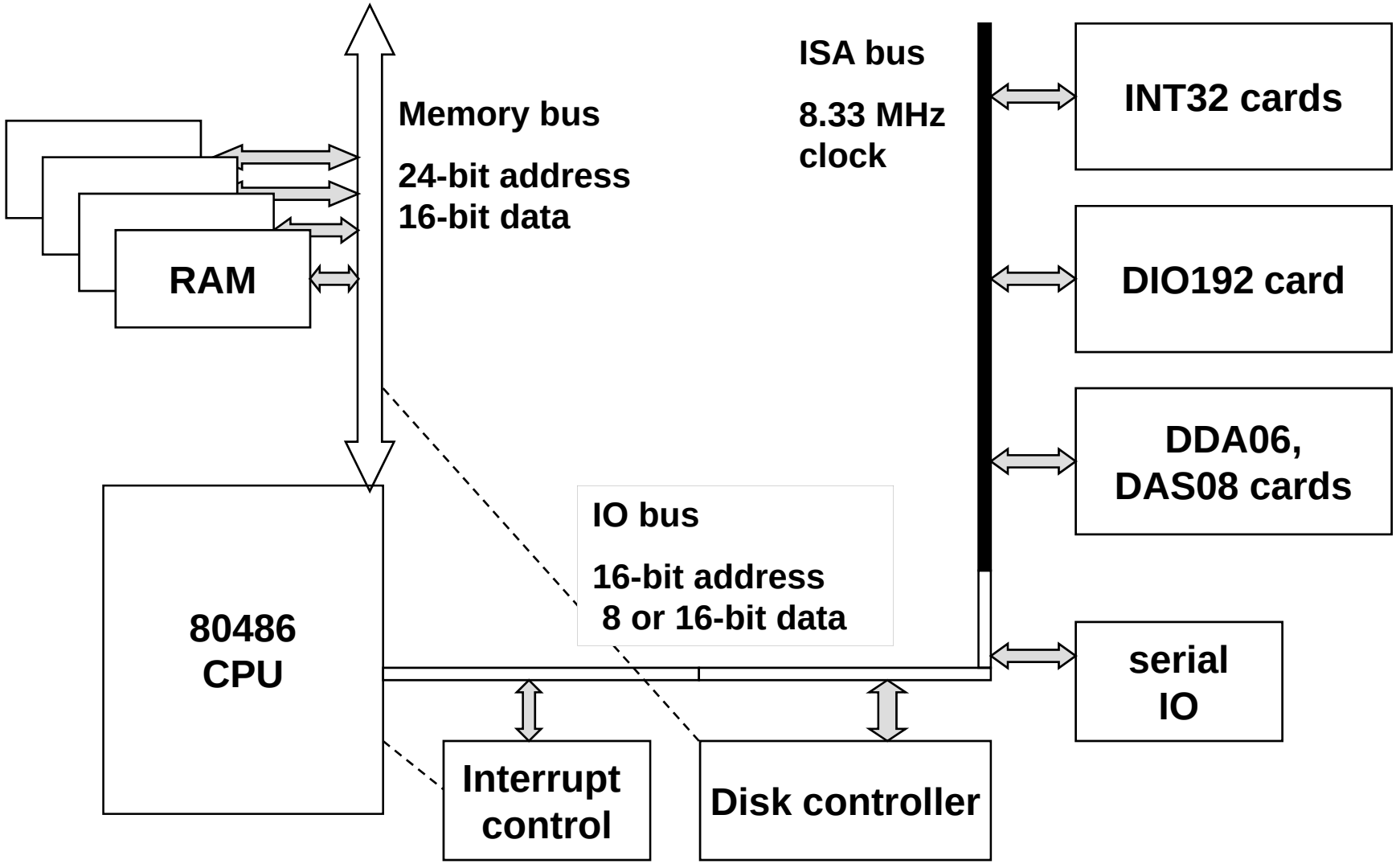
SWINBURNE UNIVERSITY OF TECHNOLOGY

Electronics Overview

17/03/2015

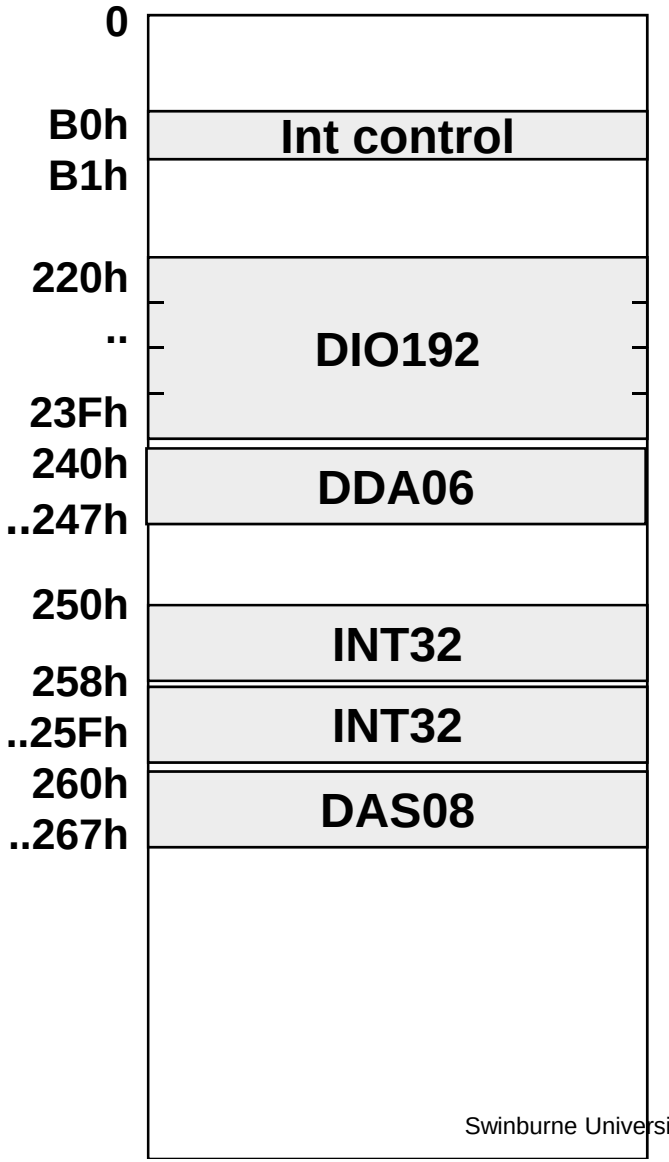


Devices on IO Bus

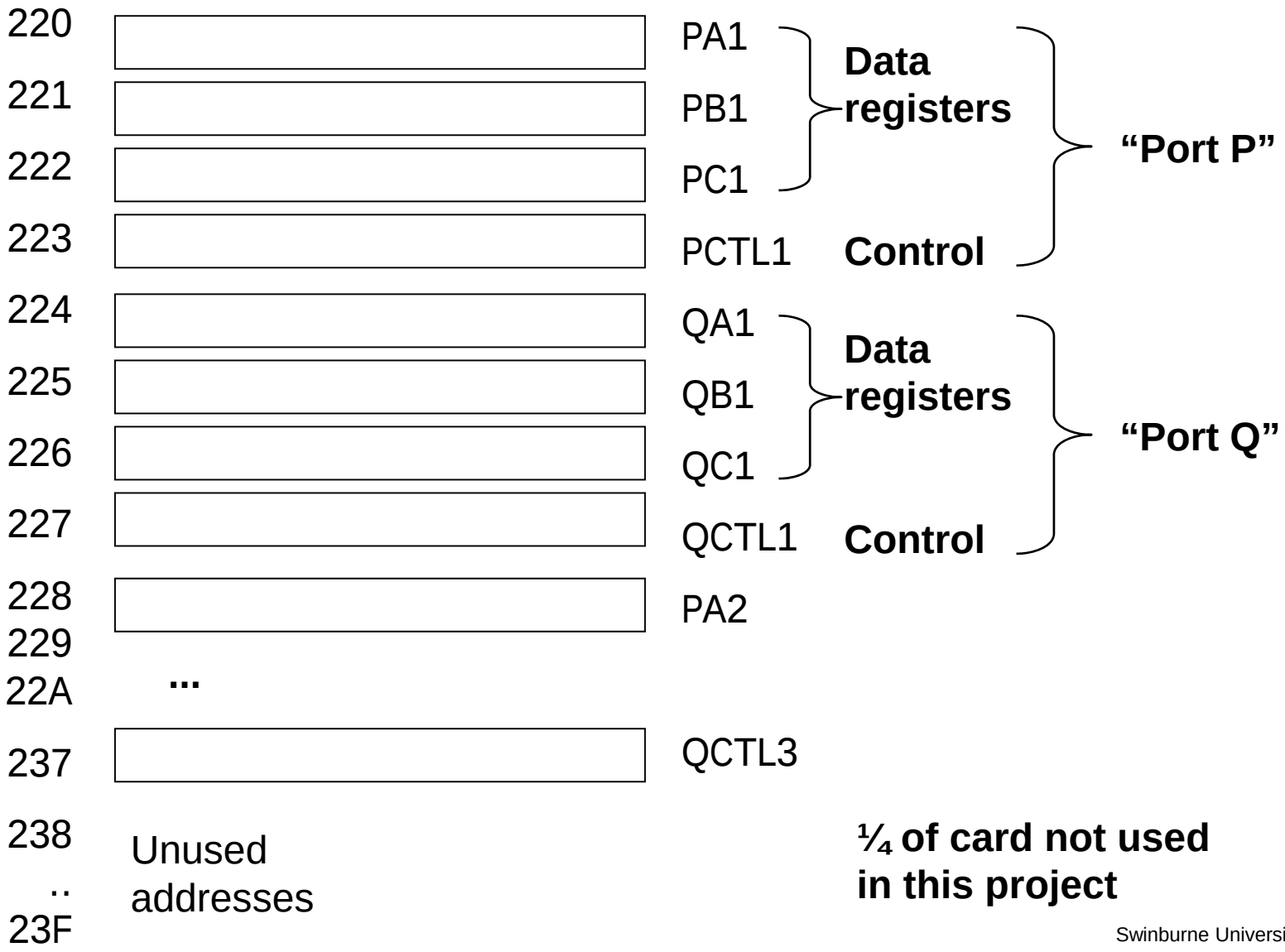


IO Address Space

- Each of these digital IO cards takes a multiple of 8 bytes of address space, but not all used
- Other devices not shown.
- Register addresses are specified in the separate "defs" files.



Dio192 Card Registers

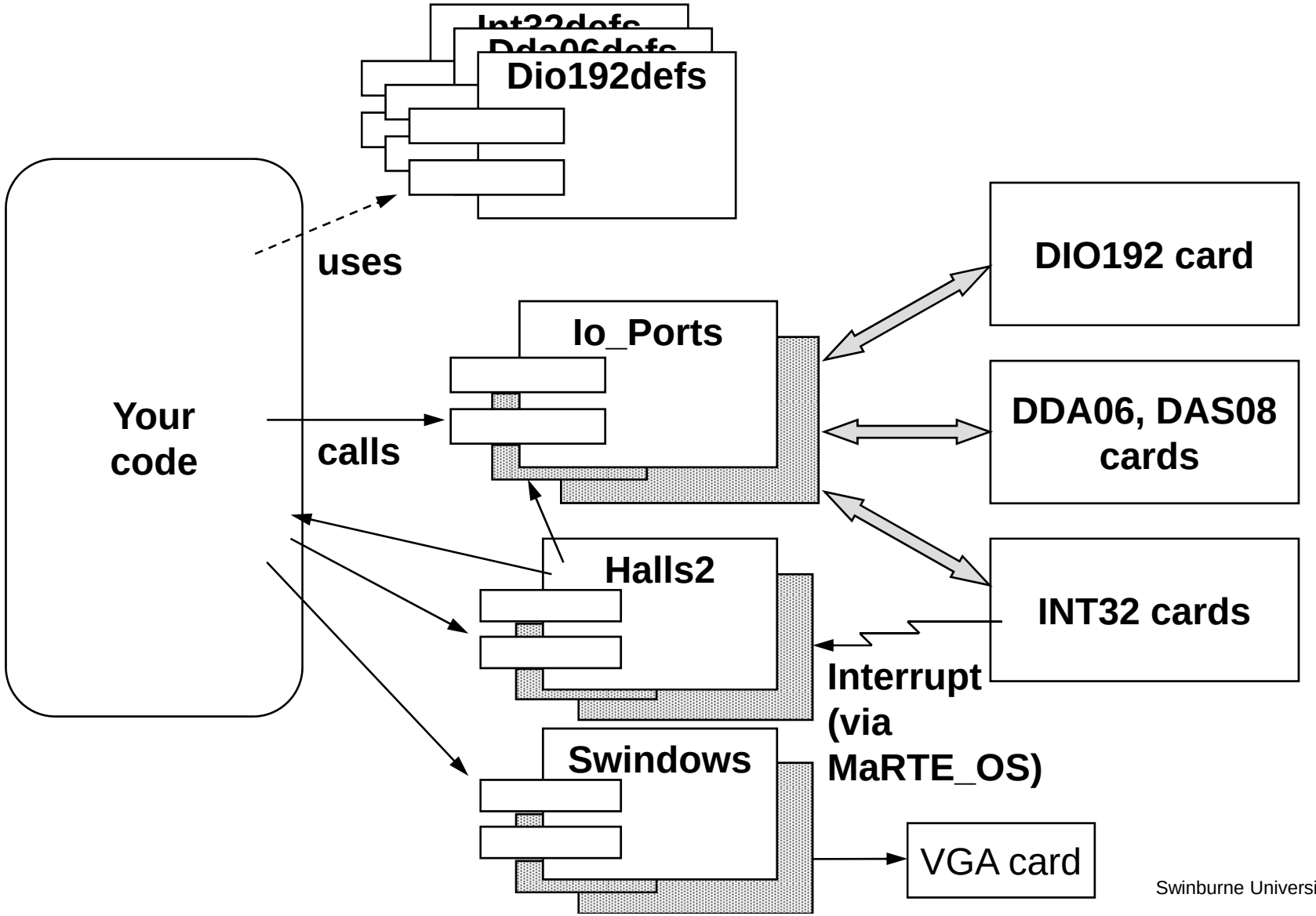


Accessing the IO Cards

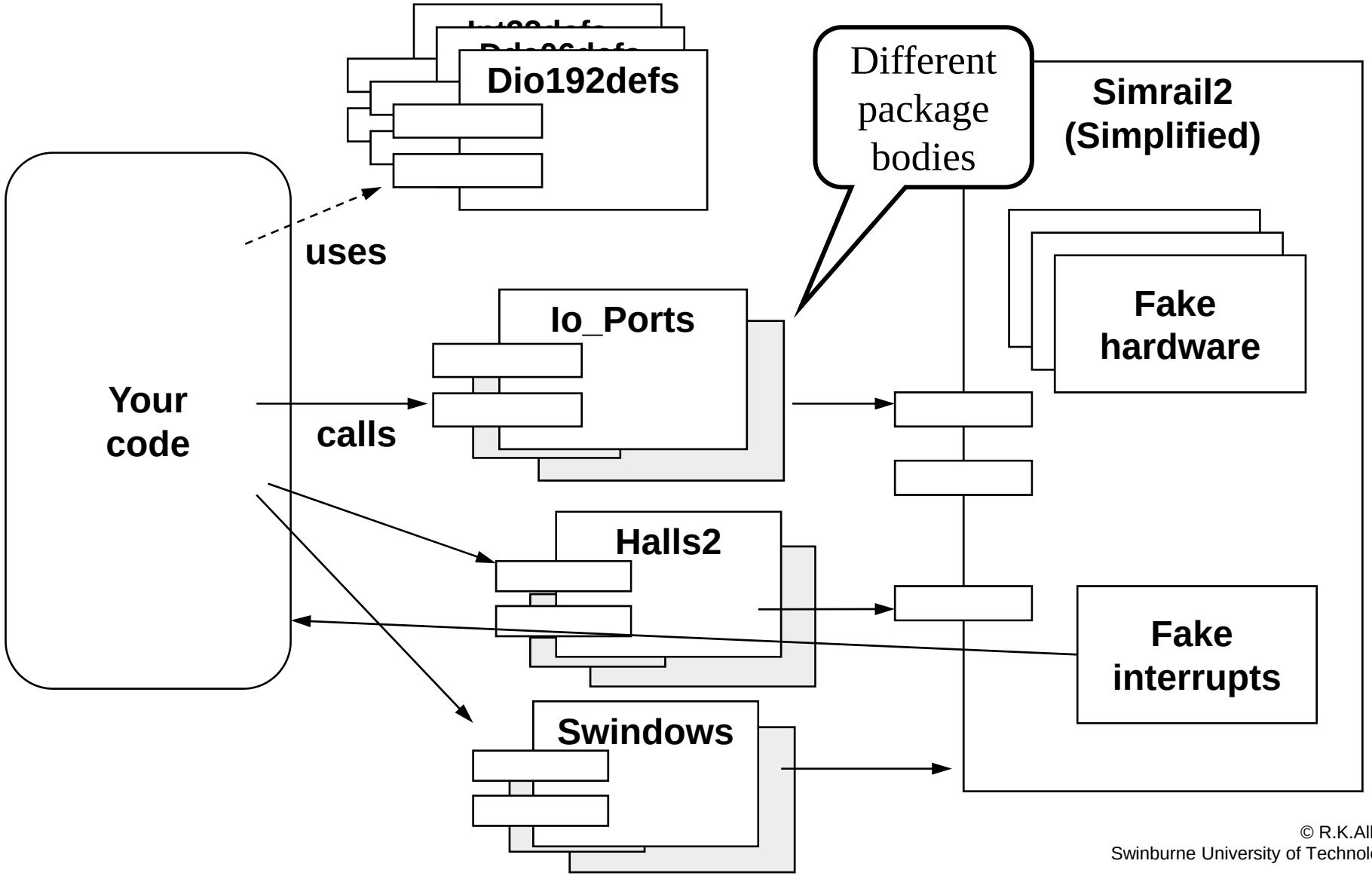
- 86-series CPUs have two instructions: IN and OUT
  - Available in PC assemblers
- We use a 3rd-party Ada package, Io_Ports, which links through to assembled code
  - textbook HMS has a similar one
- This is our only bridge to the train electronics
- Hence to make development easier almost all access to the simulator is via a “fake” version of Io_Ports
  - The package spec does not change
  - The package body does.

Software Architecture with Train HW

15/03/2012



Software Architecture with Simulator



Io_Ports


```
with Unsigned_Types; use Unsigned_Types;

package IO_Ports is

   procedure Write_IO_Port (Address : in Unsigned_16;
                            Value   : in Unsigned_8);

   procedure Read_IO_Port (Address : in Unsigned_16;
                           Value   : out Unsigned_8);

   -- Interrupt control
   procedure Enable_Interrupts;
   procedure Disable_Interrupts;

end Io_Ports;
```

Using Io_Ports

- eg Io_Ports.Write_Io_Port(16#220#, 215);
- or Io_Ports.Write_Io_Port(Pa1_Addr, Pa_Value);

Unsigned_Types.Unsigned_16

Unsigned_Types.Unsigned_8

- Problems:
  - Individual bits within bytes need to be read & written individually
    - "bit-fiddling“ is clumsy, eg: C: `v = v & ~(1<<b) | (x<<b);`
    - Ada (mod types):

      ```
      Reg := Reg and 16#7F# or Val*128;
      ```
  - Including “magic numbers” is error-prone, hard to read & maintain

Definition Packages

- Answer:
  - Set up package(s) that declare constants, types, *representation clauses* and type conversions
  - Hence the railroad code needs no "magic numbers" or "bit-fiddling"
- Dio192defs.ads – definitions for DIO192
- Int32defs.ads – definitions for INT32
- Das08defs – you write
- Raildefs – other “global” definitions
  - Used everywhere including in other "defs"
- Unsigned_Types – Solves a portability problem between the two platforms (MaRTE vs Win32 simulator)

Representation Clauses

- Have various forms that instruct the compiler about
  - absolute memory location or use;
    - Not IBM PC. Ada supports "memory-mapped" devices, that is, devices on the same bus as memory, in the same address space.
  - what binary values to use for different values;
  - size of type (in bits);
  - size of elements within an array type;
  - exact bit placement of fields within a record type .
- They allow us to avoid "bit-fiddling"
  - Declare Ada types, add suitable rep clauses,
  - (in other packages) declare variables of these types, then use ordinary Ada statements.
- All start with the keyword **for**
  - Examples soon…

Dio192 Card Bit Use

9/03/2008

register	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*	*Base address* = 16#220#
PA1	**p**	**X**	**X**	**X**	p	X	X	X	Blocks **1, 2** (p = polarity, 0=normal,
									xxx = 3-bit numbers for CAB selection)
PB1	**p**	**X**	**X**	**X**	p	X	X	X	Blocks **3, 4**
PC1	**p**	**X**	**X**	**X**	p	X	X	X	Blocks **5, 6**
PCTL1	**X**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	Mode 0 and all bits output. (bit 7 is Mode Set Bit, 1=set, subsequent 0 enables tristate, ie connects to outside)
QA1	**p**	**X**	**X**	**X**	p	X	X	X	Blocks **7, 8**
QB1	**p**	**X**	**X**	**X**	p	X	X	X	etc

From Spec2012.doc

Specifying I/O Addresses

Defines Unsigned_8, _16

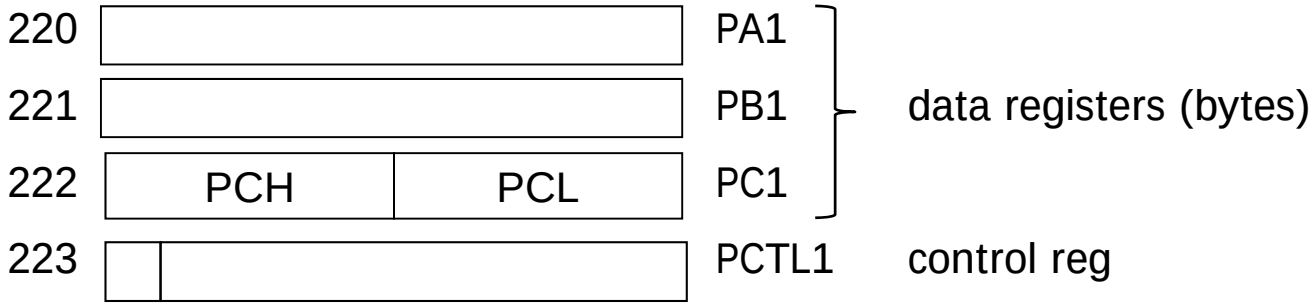
```
with Unsigned_Types, ...

package Dio192defs is
   Base_Address1 : constant := 16#220#;
   Pa1_Addr   : constant := Base_Address1 + 0;
   Pb1_Addr   : constant := Base_Address1 + 1;
   Pc1_Addr   : . . .

   -- card initialisation constants:

   Output_Init1 : constant Unsigned_8 := 2#10000000#;

   Output_Init2 : constant Unsigned_8 := 2#00000000#;
```

Address on IO bus

All 24 bits output*

(*Note: Turnout section initialisation constants differ)

Dio192 Card Initialisation



- At power-up the cards are configured for all bits output but disconnected (tri-state open).
- The bytes PA and PB and the two nibbles PCL (bits 0-3) and PCH (bits 4-7) can be separately configured for input or output.
  - For block control we want all output (all zeroes):

```
Io_Ports.Write_Io_Port(16#223#, 16#80#);          -- inelegant!
   … write initial values to PA1, PB1, PC1 here
Io_Ports.Write_Io_Port(16#223#, 0);
```

  - changing the top (most significant) bit from 1 to 0 completes the initialisation of this port (port P). Port Q similar.

Initialising the Card

Init code (somewhere):

```
with Raildefs, Unsigned_Types, Dio192defs;
. . .
use Raildefs, Unsigned_Types, Dio192defs;
. . .
Io_Ports.Write_Io_Port(Pctl1_Addr, Output_Init1);
. . .  -- write initial values
. . .  -- to PA1, PB1, PC1
Io_Ports.Write_Io_Port(Pctl1_Addr, Output_Init2);

. . .  -- similarly for QA1, QB1, ...Qctl2
```

Initialising the Card : Turnouts

Init code (somewhere):

```
. . .
Io_Ports.Write_Io_Port(Pctl3_Addr, Pctl3_Init1);
. . .  -- write initial value to PB3
Io_Ports.Write_Io_Port(Pctl3_Addr, Pctl3_Init2);

Io_Ports.Write_Io_Port(Qctl3_Addr, Qctl3_Init1);
. . .  -- write initial value to QA3
. . .  -- write initial value to QC3
Io_Ports.Write_Io_Port(Qctl3_Addr, Qctl3_Init2);
```

Describing Single Bits

- In Raildefs:

Bit	7	6	5	4	3	2	1	0
	X	X	X	X	X	X	X	X
index	7	6	5	4	3	2	1	0

```
Num_Turnouts : constant := 19;
type Turnout_Idx is range 0 .. 24;
subtype Turnout_Id is Turnout_Idx range 1..Num_Turnouts;
No_Turnout : constant Turnout_Idx := 0;
type Turnout_Drive_Bit is (Pull_St, Pull_Tu);
```

New type, not Boolean, order important

- In Dio192defs:

```
type Turnout_Drive_Register is array (Turnout_Idx range 0..7)
  of Turnout_Drive_Bit ;

for Turnout_Drive_Register'Component_Size use 1;
for Turnout_Drive_Register'Size use 8;
```

Rep. clauses

Describing Single Bits - 2

- Similarly
  - in Raildefs:

    type Turnout_Status_Bit is (Busy, In_Position);

    ...

    type Sensor_Bit is (On, Off);

    New types, not Boolean, order important
  - in Dio192defs:

    type Turnout_Status_Register is array ...
  - in Int32defs:

    type Sensor_State_Register is array ...

Bit	7	6	5	4	3	2	1	0
	X	X	X	X	X	X	X	X
index	7	6	5	4	3	2	1	0

Defining Constants

- In Dio192defs (contd)

```
Turnout_Drive_Init : constant Turnout_Drive_Register :=
   (others=>Pull_St);
All_In_Position : constant Turnout_Status_Register :=
   (others=>In_Position);
```

An aggregate, all elements the same

- (basic student code):

```
Pb3 : Turnout_Drive_Register := Turnout_Drive_Init;
. . .
Pb3(Turn_Bit) := Desired_Position;
. . .
Io_Ports.Write_Io_Port(Pb3_Addr, Unsigned(Pb3));
```

Ordinary array access

Note … , Pb3); won't compile

Dio192defs Type-Conversion Functions

A generic function, stand-alone in library

```
with Unsigned_Types, Unchecked_Conversion;
use Unsigned_Types;

package Dio192defs is
   . . .
   function Unsigned is new Unchecked_Conversion (
         Source=> Turnout_Drive_Register,  Target=>Unsigned_8 );
```

A generic instantiation

- Effectively:

```
function Unsigned(Value : Turnout_Drive_Register) return
      Unsigned_8 is
begin
      return <exactly the same 8 bits>;
end Unsigned;
-- optimised, ie the compiler generates no code for this function!
```

FYI

- Unchecked_Conversion is defined (see LRM):

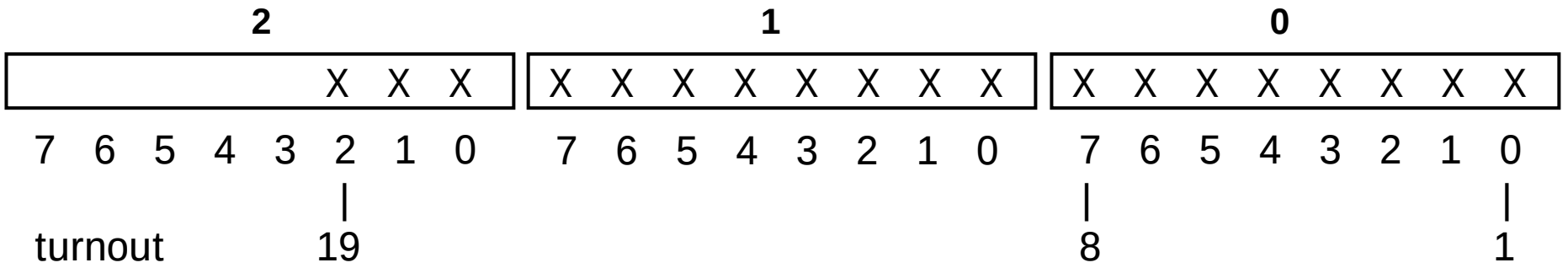
```
generic
  type Source is private;
  type Target is private;
function Unchecked_Conversion (
   Value : Source ) return Target;
-----------
function Unchecked_Conversion (
   Value : Source ) return Target is
begin
          return <exactly the same bits>;  -- not actually Ada
end Unchecked_Conversion;
```

Better Turnout Code

15/03/2012

You must declare this

Drive_Bytes




```
Tid      :  Turnout_Id;
Turn_Byte : Turnout_Idx;
Turn_Bit  : Turnout_Idx;

. . .
   Drive_Bytes(Turn_Byte)(Turn_Bit) := Desired_Position;
   Addr := Turnout_Drive_Addr(Turn_Byte);

   Io_Ports.Write_Io_Port(Addr, Unsigned(Drive_Bytes(Turn_Byte)));
```

array in Dio192defs

Describing 3-Bit Fields

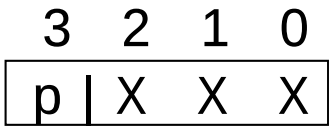
- In Raildefs:

```
type Cab_Type is mod 8;                    -- or      is range 0..7;
subtype Dac_Id is Cab_Type range 1..Max_Trains;  -- 1..4
type Polarity_Type is (Normal_Pol, Reverse_Pol);
```

- In Dio192defs:

```
type Block_Nibble is record
   Blk_Cab : Cab_Type;
   Blk_Pol  : Polarity_Type;
end record;
for Block_Nibble use record
   Blk_Cab at 0 range 0..2;
   Blk_Pol at 0 range 3..3;
end record;
for Block_Nibble'Size use 4;
```

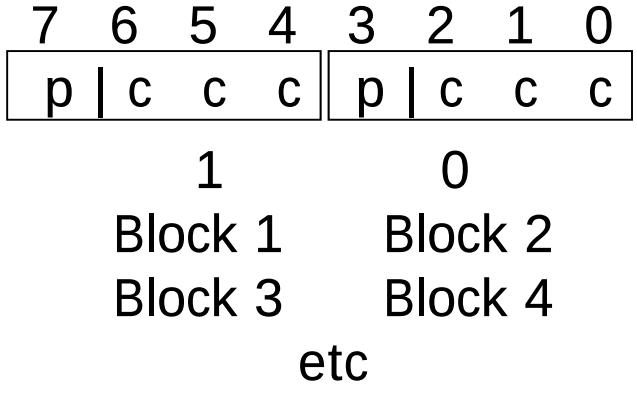
More representation clauses



Packing 4-Bit Components

- In Dio192defs:

```
type Block_Register is array (Block_Idx range 0..1)
of Block_Nibble;
for Block_Register'Component_Size use 4;
for Block_Register'Size use 8;
```



- Note annoying mismatch in nibble order, easily handled, eg:

```
Byte_No := (Blok-1) / 2;
Nibble_No := Blok mod 2;
   -- Blok is type Block_Id, other vars here type Block_Idx
```

Setting Block Cab & Polarity

- In Dio192defs:

```
Zero_Normal : constant Block_Register :=
    ( (Off,Normal_Pol), (Off,Normal_Pol) );
```

Aggregate of aggregates

```
function Unsigned is new Unchecked_Conversion
(Source=> Block_Register, Target=>Unsigned_8 );.
```

Unsigned is over-loaded name, different Source

- (basic student code):

```
A_Byte : Block_Register := Zero_Normal;
. . .
A_Byte(Nibble_No).Blk_Cab := Cab_Type(Current_Train_Id);
```

array access

record field access

ordinary type conversion

```
Io_Ports.Write_Io_Port( Addr,
    Unsigned(A_Byte) );
```

Setting Block Cab & Polarity

- (improved code)

```
type Block_Byte_Array is array (Block_Idx range 0..11) of
Block_Register;
. . .
Block_Bytes : Block_Byte_Array  := (others => Zero_Normal);
. . .
Block_Bytes(Byte_No)(Nibble_No).Blk_Cab :=

    Cab_Type(Current_Train_Id);


Addr := Block_Addr(Byte_No);

Io_Ports.Write_Io_Port( Addr,
    Unsigned(Block_Bytes(Byte_No)) );
```

aggregate

Select byte, nibble, field

ordinary type conversion

array in Dio192defs

NB: This extra complexity is justified...

[draw diagram]

In-Memory Copies

- For all the IO cards
  - Individual bits within bytes need to be read & written individually
  - Whole bytes are read and written
  - You can't read back registers configured for writing
  - So you must keep variables that remember the last values written.
  - There must be one variable for each register
  - There must be only one copy of each
  - It is easier if sets of similar variables are in arrays
  - They must be kept continuously -- package vars
  - They may need to be protected against concurrent update (next lecture)