Low level programming in Ada95

- Useful type declarations
- Bit manipulations and conversions
- Declaring Input/Output memory locations and volatile entities
- Ada95 and the hardware – UART example

Useful type declarations

C-type	Description	Range	Type definitions in Ada
unsigned char	8-bit integer no sign	0..255	**type** UINT8 **is** integer **range** 0..255; **for** UINT8'**size use** 8;
signed char	8 bit integer 2's complement	-128..+127	**type** SINT8 **is** integer **range** -128..127; **for** SINT8'**size use** 8;
unsigned short	16-bit integer no sign	0..65535	**type** UINT16 **is** integer **range** 0..65535; **for** UINT16'**size use** 16;
signed short	16 bit integer 2's complement	-32768.. 32767	**type** SINT16 **is** integer **range** -32768..32767; **for** SINT16'**size use** 16;

Useful type declarations cont'd

Bitfields

```
type BIT_TYPE is range 0..1; -- Named type and min..max values
for BIT_TYPE'SIZE use 1;     -- Object type  needs a bit
type BITFIELD8 is
record
       b0:    BIT_TYPE;
       b1:    BIT_TYPE;
       b2:    BIT_TYPE;
       b3:    BIT_TYPE;
       b4:    BIT_TYPE;
       b5:    BIT_TYPE;
       b6:    BIT_TYPE;
       b7:    BIT_TYPE;
end record;
```

Note: This declaration doesn't specify neither the size of `BITFIELD8`, nor the representation (bit order)

Useful type declarations cont'd

Representation clause

```
for BITFIELD8'SIZE use 8;     -- Object type needs 8 bits

for BITFIELD8 use
record
       b0 at 0 range 0..0;
       b1 at 0 range 1..1;
       b2 at 0 range 2..2;
       b3 at 0 range 3..3;
       b4 at 0 range 4..4;
       b5 at 0 range 5..5;
       b6 at 0 range 6..6;
       b7 at 0 range 7..7;
end record;
```

Byte position within actual post

Bit position within actual byte

Little Endian bit numbering means that b0 refers to the least significant bit (the rightmost) in the mathematical binary number.

Note: Called Low_Order_First representation in Ada 95

Useful type declarations cont'd

Representation clause

```
for BITFIELD8'SIZE use 8;      -- Object type needs 8 bits

for BITFIELD8 use
record
       b7 at 0 range 0..0;
       b6 at 0 range 1..1;
       b5 at 0 range 2..2;
       b4 at 0 range 3..3;
       b3 at 0 range 4..4;
       b2 at 0 range 5..5;
       b1 at 0 range 6..6;
       b0 at 0 range 7..7;
end record;
```

The interpretation of bit numbers can be set through the BIT_ORDER attribute of a data type

Gnu Ada 95 can only use (and has therefore as default) Big Endian representation

Big Endian bit numbering means that b0 refers to the most significant bit (the leftmost) in the mathematical binary number.

Note: Called High_Order_First representation in Ada 95

Bit manipulations

A scary example:

In C (or Java) what is the difference between the following statements?

```
if( a&1 )...;

if( a&&1 )...;
```

Bitwise AND

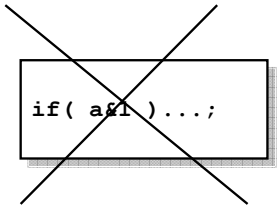
Logical AND

Suppose you made a "typo" error, estimate the time spent for tracking down and correcting this error.

And even worse, what if your tests never exposed this error?

Bit manipulation cont'd

There are no BIT-operators in the Ada language. Instead the types should have been declared to accomodate bit operations.

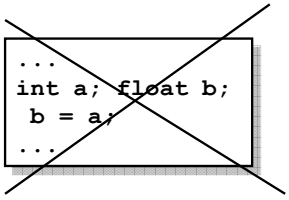



```
...
a       : BITFIELD8;

...
        if ( a.b7 == 1 ) then ...
...
```


Data type conversions

There are no "implicit" conversions in Ada




```
...
int a; float b;
 a = (int) b;
...
```

```
...
a       : integer;
b       : float;
...
        a = integer ( b );
        b = float ( a );
...
```

Easy (unchecked) conversions

The strong data typing in Ada can be overridden.

A single data copy between different types (of the same sizes) is allowed *presumed that you have already stated that it is legal*. You do so by creating new instances from the package ”unchecked_conversion”..

```
...
function TO_UINT8 is new
unchecked_conversion( BITFIELD8, UINT8 );
...
a       : BITFIELD8;
b       : UINT8;

...
       b:= TO_UINT8 ( a );
...
```


Easy (unchecked) conversions cont’d

Overloading is supported, we can use the same identifier for different conversions...

```
with Unchecked_Conversion
...
function TO_UINT8 is new
unchecked_conversion( BITFIELD8, UINT8 );

function TO_UINT8 is new
unchecked_conversion( SINT8, UINT8 );

function TO_UINT8 is new
unchecked_conversion( your_decided_name, UINT8 );
```

... as long as the sizes match ...

Declaring a ”variable” for an IO register

1. Create a type definition that represents the register bits.
2. Declare an object (”variable”) of this type
3. Use an *address clause*, to tell the compiler where this object resides

```
...
IO_port :     BITFIELD8;

-- address clause for this object:
for IO_port'address use constant System.address :=
       System.Storage_elements.to_address( memory address );
...
```


Volatile entities

The volatile pragma tells the compiler that an object can be changed independently of program control.

The generic example is IO interface registers (in the hardware).

```
...
IO_port :     BITFIELD8;
pragma Volatile( IO_Port );
-- address clause for this object:
for IO_port'address use constant System.address :=
       System.Storage_elements.to_address( memory address );
...
```

Why is ”Volatile” important?

Consider the following example, a decent compiler should reduce the loop into a single statement (test only once, or perhaps even remove it) unless the test value couldn't change between the loop iterations.

```
-- wait for device ready ...
while (IO_Port.b7 /= 0 ) loop
       NULL;
end loop;
```

The

```
pragma Volatile( IO_Port );
```

tells the compiler to do NO such optimizations here.

Ada 95 and the hardware – UART example

- The train simulator target computer MC68340.
- The serial interface – UART (Universal Asynchronous Receiver/Transmitter)
- Serial communication principles
- Programming the device – interpretation of the data sheet.

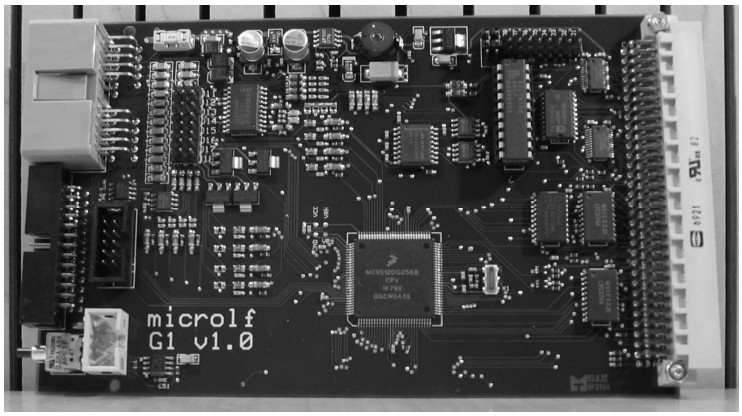
The train simulator control computer ”MC68”

512 kByte Read Only Memory

512 kByte Read/Write Memory

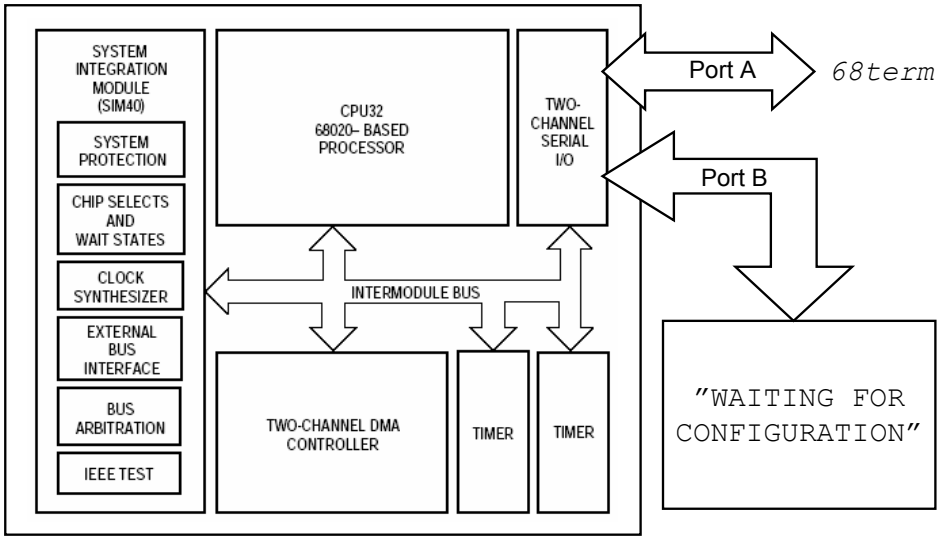
The train simulator target computer ”G1”

256 kByte Read Only Memory



14 kByte Read/Write Memory

Motorola MC68340



Serial communication module

Address	FC	Register Read (R/W = 1)	Register Write (R/W = 0)
700	S[1]	MCR (HIGH BYTE)	MCR (HIGH BYTE)
701	S	MCR (LOW BYTE)	MCR (LOW BYTE)
702	S	DO NOT ACCESS[3]	DO NOT ACCESS[3]
703	S	DO NOT ACCESS[3]	DO NOT ACCESS[3]
704	S	INTERRUPT LEVEL (ILR)	NTERRUPT LEVEL (ILR)
705	S	INTERRUPT VECTOR (IVR)	INTERRUPT VECTOR (IVR)
710	S/U[2]	MODE REGISTER 1A (MR1A)	MODE REGISTER 1A (MR1A)
711	S/U	STATUS REGISTER A (SRA)	CLOCK-SELECT REGISTER A (CSRA)
712	S/U	DO NOT ACCESS[3]	COMMAND REGISTER A (CRA)
713	S/U	RECEIVER BUFFER A (RBA)	TRANSMITTER BUFFER A (TBA)
714	S/U	INPUT PORT CHANGE REGISTER (IPCR)	AUXILIARY CONTROL REGISTER (ACR)
715	S/U	INTERRUPT STATUS REGISTER (ISR)	INTERRUPT ENABLE REGISTER (IER)
716	S/U	DO NOT ACCESS[3]	DO NOT ACCESS[3]
717	S/U	DO NOT ACCESS[3]	DO NOT ACCESS[3]
718	S/U	MODE REGISTER 1B (MR1B)	MODE REGISTER 1B (MR1B)
719	S/U	STATUS REGISTER B (SRB)	CLOCK-SELECT REGISTER B (CSRB)
71A	S/U	DO NOT ACCESS[3]	COMMAND REGISTER B (CRB)
71B	S/U	RECEIVER BUFFER B (RBB)	TRANSMITTER BUFFER B (TBB)
71C	S/U	DO NOT ACCESS[3]	DO NOT ACCESS[3]
71D	S/U	INPUT PORT REGISTER (IP)	OUTPUT PORT CONTROL REGISTER (OPCR)
71E	S/U	DO NOT ACCESS[3]	OUTPUT PORT (OP)[4] BIT SET
71F	S/U	DO NOT ACCESS[3]	OUTPUT PORT (OP)[4] BIT RESET
720	S/U	MODE REGISTER 2A (MR2A)	MODE REGISTER 2A (MR2A)
721	S/U	MODE REGISTER 2B (MR2B)	MODE REGISTER 2B (MR2B)

Control registers

Status registers

Data registers

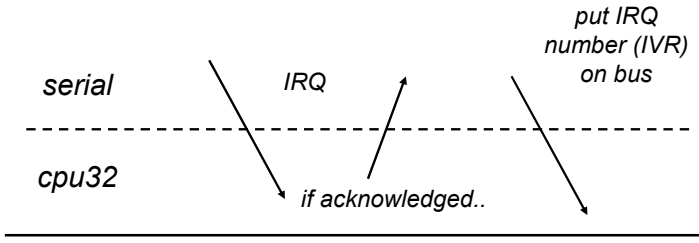
Interrupt vector

IVR $705

7	6	5	4	3	2	1	0
IVR7	IVR6	IVR5	IVR4	IVR3	IVR2	IVR1	IVR0
RESET:							
0	0	0	0	1	1	1	1

Read /Write — Supervisor Only

Interrupts and Ada will be thoroughly elaborated in E3



cpu32 calculates vector

(4 * IVR)

and then fetch address to the interrupt routine

Interrupt level

ILR $704

7	6	5	4	3	2	1	0
0	0	0	0	0	IL2	IL1	IL0
RESET:							
0	0	0	0	0	0	0	0

Read/Write — Supervisor Only

BASE for serial module is programmable in MC68340. In this system we use **16#FFFFF000#**

Useful definitions...

```
ILR_address : constant System.address := to_address( 16#FFFFF704#);
D_ILR: bits;
for D_ILR'address use ILR_address;
```

EXAMPLE:

```
D_ILR := 16#5#;      -- IRQ Level 5 for serial module
```

Use of interrupts

COS
- Change of State
DBA/DBB
- Delta Break

IER $715

7	6	5	4	3	2	1	0
COS	DBB	RxRDYB	TxRDYB	0	DBA	RxRDYA	TxRDYA
RESET:							
0	0	0	0	0	0	0	0

Write Only — Supervisor/User

RxRDYB—Channel B Receiver Ready or FIFO full
1 = Enable interrupt
0 = Disable interrupt

TxRDYB—Channel B Transmitter Ready
1 = Enable interrupt
0 = Disable interrupt

Use IER to enable interrupt sources from the serial module

Mode register 1, (one for each channel A and B)

MR1A, MR1B $710, $718

7	6	5	4	3	2	1	0
RxRTS	R/F	ERR	PM1	PM0	PT	B/C1	B/C0
RESET:							
0	0	0	0	0	0	0	0

Read/Write — Supervisor/User

B/C1	B/C0	Bits/Character
0	0	Five Bits
0	1	Six Bits
1	0	Seven Bits
1	1	Eight Bits

PM1	PM0	Parity Mode	PT	Parity Type
0	0	With Parity	0	Even Parity
0	0	With Parity	1	Odd Parity
0	1	Force Parity	0	Low Parity
0	1	Force Parity	1	High Parity
1	0	No Parity	X	No Parity
1	1	Multidrop Mode	0	Data Character
1	1	Multidrop Mode	1	Address Character

Clock Select Register A and B

CSRA, CSRB $711, $719

7	6	5	4	3	2	1	0
RCS3	RCS2	RCS1	RCS0	TCS3	TCS2	TCS1	TCS0
RESET:							
0	0	0	0	0	0	0	0

Write Only Supervisor/User

RCS3	RCS2	RCS1	RCS0	Set 1	Set 2
0	0	0	0	50	75
0	0	0	1	110	110
0	0	1	0	134.5	134.5
0	0	1	1	200	150
0	1	0	0	300	300
0	1	0	1	600	600
0	1	1	0	1200	1200
0	1	1	1	1050	2000
1	0	0	0	2400	2400
1	0	0	1	4800	4800
1	0	1	0	7200	1800
1	0	1	1	9600	9600
1	1	0	0	38.4k	19.2k
1	1	0	1	76.8k	38.4k
1	1	1	0	SCLK/16	SCLK/16
1	1	1	1	SCLK/1	SCLK/1

TCS3	TCS2	TCS1	TCS0	Set 1	Set 2
0	0	0	0	50	75
0	0	0	1	110	110
0	0	1	0	134.5	134.5
0	0	1	1	200	150
0	1	0	0	300	300
0	1	0	1	600	600
0	1	1	0	1200	1200
0	1	1	1	1050	2000
1	0	0	0	2400	2400
1	0	0	1	4800	4800
1	0	1	0	7200	1800
1	0	1	1	9600	9600
1	1	0	0	38.4k	19.2k
1	1	0	1	76.8k	38.4k
1	1	1	0	SCLK/16	SCLK/16
1	1	1	1	SCLK/1	SCLK/1

Communicating devices must use the same *baudrate*

Mode register 2, (one for each channel A and B)

MR2A, MR2B $720, $721

7	6	5	4	3	2	1	0
CM1	CM0	TxRTS	TxCTS	SB3	SB2	SB1	SB0
RESET:							
0	0	0	0	0	0	0	0

Read/Write Supervisor/User

RTS/CTS used for ”hand shaking”. I.e. hardware synchcronises receiver/transmitter.

CM1	CM0	Mode
0	0	Normal
0	1	Automatic Echo
1	0	Local Loopback
1	1	Remote Loopback

TxRTS—Transmitter Ready-to-Send
This bit controls the negation of the RTSA or RTSB signals.

TxCTS—Transmitter Clear-to-Send
1 = Enables clear-to-send operation.

SB3–SB0—Stop-Bit Length Control
These bits select the length of the stop bit appended to the transmitted character

Status register, (one for each channel A and B)

SRA, SRB — $711, $719

7	6	5	4	3	2	1	0
RB	FE	PE	OE	TxEMP	TxRDY	FFULL	RxRDY
RESET:							
0	0	0	0	0	0	0	0

Read Only — Supervisor/User

TxRDY—Transmitter Ready

This bit is duplicated in the ISR; bit 0 for channel A and bit 4 for channel B.

1 = The transmitter holding register is empty and ready to be loaded with a character. This bit is set when the character is transferred to the transmitter shift register. This bit is also set when the transmitter is first enabled. Characters loaded into the transmitter holding register while the transmitter is disabled are not transmitted and are lost.

0 = The transmitter holding register was loaded by the CPU32, or the transmitter is disabled.

RxRDY—Receiver Ready

1 = A character has been received and is waiting in the FIFO to be read by the CPU32. This bit is set when a character is transferred from the receiver shift register to the FIFO.

0 = The CPU32 has read the receiver buffer, and no characters remain in the FIFO after this read.

EXAMPLE: C or Java...

SRA, SRB — $711, $719

7	6	5	4	3	2	1	0
RB	FE	PE	OE	TxEMP	TxRDY	FFULL	RxRDY
RESET:							
0	0	0	0	0	0	0	0

Read Only — Supervisor/User

```
...
while( ! (SRB & TxRDY) )
      ;  // spin
// ok to transmit ...
...
```

EXAMPLE:
Ada

SRA, SRB $711, $719

7	6	5	4	3	2	1	0
RB	FE	PE	OE	TxEMP	TxRDY	FFULL	RxRDY
RESET:							
0	0	0	0	0	0	0	0

Read Only Supervisor/User

```
-- wait for device ready ...
while (SRB.TxRDY == 0 ) loop
       NULL; -- spin
end loop;
```

Assuming proper declaration
of IO register ”SRB”

Transmit
register

TBA, TBB $713, $71B

7	6	5	4	3	2	1	0
TB7	TB6	TB5	TB4	TB3	TB2	TB1	TB0
RESET:							
0	0	0	0	0	0	0	0

Write Only Supervisor/User

EXAMPLE:
C or Java...

```
...
while( ! (SRB & TxRDY) )
      ;  // spin
// ok to transmit ...
TBB = c;
```

Transmit register

TBA, TBB $713, $71B

7	6	5	4	3	2	1	0
TB7	TB6	TB5	TB4	TB3	TB2	TB1	TB0
RESET:							
0	0	0	0	0	0	0	0

Write Only Supervisor/User

EXAMPLE: Ada...

```
-- wait for device ready ...
while (SRB.TxRDY == 0 ) loop
       NULL; -- spin
end loop;
TBB := to_type( c );
```

Assuming proper type declaration and address clauses for IO register ”SRB”, ”TBB”

and appropriate unchecked conversion *to_type*

Transmit/ Receive registers shares address

cpu32 bus signal ”Read/Write” is used as a discriminating address bit

TBA, TBB $713, $71B

7	6	5	4	3	2	1	0
TB7	TB6	TB5	TB4	TB3	TB2	TB1	TB0
RESET:							
0	0	0	0	0	0	0	0

Write Only Supervisor/User

RBA, RBB $713, $71B

7	6	5	4	3	2	1	0
RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0
RESET:							
0	0	0	0	0	0	0	0

Read Only Supervisor/User

Control register

CRA, CRB — $712, $71A

7	6	5	4	3	2	1	0
MISC3	MISC2	MISC1	MISC0	TC1	TC0	RC1	RC0
RESET:							
0	0	0	0	0	0	0	0

Write Only — Supervisor/User

MISC3	MISC2	MISC1	MISC0	Command
0	0	0	0	No Command
0	0	0	1	No Command
0	0	1	0	Reset Receiver
0	0	1	1	Reset Transmitter
0	1	0	0	Reset Error Status
0	1	0	1	Reset Break-Change Interrupt
0	1	1	0	Start Break
0	1	1	1	Stop Break
1	0	0	0	Assert RTS
1	0	0	1	Negate RTS
1	0	1	0	No Command
1	0	1	1	No Command
1	1	0	0	No Command
1	1	0	1	No Command
1	1	1	0	No Command
1	1	1	1	No Command

TC1	TC0	Command
0	0	No Action Taken
0	1	Enable Transmitter
1	0	Disable Transmitter
1	1	Do Not Use

RC1	RC0	Command
0	0	No Action Taken
0	1	Enable Receiver
1	0	Disable Receiver
1	1	Do Not Use

Properties of a serial module

In a real-time system, the serial module is a shared resource.

EXAMPLES:

Different tasks will (perhaps simultanously) print a text string to the console
or

different tasks will (perhaps simultanously) send a command to the simulator
...

A serial module device driver, in a parallell computing environment, has to be implemented as a protected object (which insures mutual exclusion).

A serial port initialization

```
procedure init_port_B is
begin
        D_CRB := cmd_reset_receiver;
        D_CRB := cmd_reset_transmitter;
        D_CRB := cmd_reset_errorstatus;
        D_CRB := cmd_reset_break;
        D_MR1B := MR1B_init;
        D_MR2B := MR2B_init;
        D_CSRB := CSRB_init;
        D_ILR := ILEVEL;
        D_IVR := VECTOR;
        D_IER := rec_irq_enable;
        D_CRB := cmd_enable_receiver;
        D_CRB := cmd_enable_transmitter;
end init_port_B;
```

```
ivector : constant := Ada.Interrupts.Names.PORTBINT;
cmd_reset_receiver : bits := bits(16#20#);
cmd_reset_transmitter : bits := bits(16#30#);
cmd_reset_errorstatus : bits := bits(16#40#);
cmd_reset_break : bits := bits(16#50#);

rec_irq_enable: bits := bits(16#24#);
cmd_enable_receiver : bits := bits(16#01#);
cmd_enable_transmitter : bits := bits(16#04#);

MR1B_init : constant bits := bits(16#13#);
-- 8 bits, no parity
MR2B_init : constant bits := bits(16#07#);
-- normal, 1 stop bit

CSRB_init : constant bits := bits(16#BB#);
-- 9600 baud,  Rx and Tx
ILEVEL : bits := bits(16#04#);
-- Interrupt level 4, port A and portB !!
VECTOR : bits := bits(ivector);
-- Interrupt vector port A and port B !!
```


Serial port device driver

```
with appropriate packages...
package body Serial is
...
      protected body SerialB is
            procedure init_port_B is
            begin
                        -- initialize the port
            end;

            procedure handler is
            begin
                        -- handle interrupt
            end;

            entry write ( parameter ) when guard
            begin
                        -- send a character
            end;

            entry read ( parameter ) when guard
            begin
                        -- get a character
            end;
      end SerialB;
            -- Visible ...
      procedure InitSerialB is....
      procedure WriteSerialB ( parameter ) is....
      function ReadSerialB is ...

begin
            attach_handler(Serial.handler'access, ivector);
end Serial;
```

Summary

- Useful type declarations
  - We have seen register mappings and howto specify bit positions(locations)
- Bit manipulations and conversions
  - Ada type checking might seem frustrating, but following some simple rules get it right.
- Declaring Input/Output memory locations and volatile entities
  - Ada let's you specify hardware register addresses in a way that is easy to understand and at the same time indisputable.
- Ada95 and the hardware – UART example
  - A full blown device driver has been sketched.