The Programming Language "*immediate C*" Programming Manual

John E. Wulff, B.E., M. Eng. Sc.

http://immediateC.net/
https://github.com/JohnWulff/immediateC/
http://raspberrypi.education/immediateC/

contact the author at
immediateC@gmail.com

`$Id: iC_prog.odt 1.3   2017/09/27`

Preface

immediate C (*iC*) is a new programming language and more specifically a new style of language, which programmers will not be so familiar with. It is *declarative*, which means that it declares the relationship between variables, which will be forced to be up to date by the run time system and not by following an *imperative* sequence of instructions. *iC* is not purely *declarative*, because it allows the execution of snippets of pure *C* code on certain conditions. Such languages are called *hybrid* – one example is *yacc*, which specifies a context free grammar declaratively, but includes code snippets from a host language, which is usually imperative (such as *C*). The structure of *yacc* code has been used as a model in the design if *iC*, which is also compiled into pure *C* code by a pre-compiler, just like *yacc*.

iC is related to Hardware Description Languages (HDL) and Programmable Logic Controller (PLC) languages, but it is aimed at generating efficient executables on any computer capable of running *C* and **not** for designing hardware or for running on specialised PLC hardware. *iC* is a simple language, which is based on the same concepts as logical and analog IC circuits and electromechanical relays. *iC* uses the syntax of *C* to give meaning to statements that have no semantic support in *C*. This should make *iC* very easy to learn. *iC* statements mostly express logical or analog relationships but it can run *C* code directly on specific events.

Since *immediate C* is an extension of *C*, in a similar way that *C++* is in extension of *C* – using the same declaration syntax and the same variables and operators, this manual does not cover any details which are the same as in *C*. This manual concentrates on explaining the differences – mainly how *immediate* variables carry forward event information with *immediate* expressions.

An *iC* program consists mostly of a series of logical and arithmetic *immediate* expressions, which are assigned to outputs or intermediate variables or are used in function block calls. Each such expression **declares** the relationship between some inputs and an output. *immediate* expressions are not executed in sequence as is the case for all instruction flow languages but only when an input to one of the expressions changes. The fundamental assumption for *iC* is, that **the output of an expression does not change if none of its inputs change** and therefore does not need to be executed until one of its inputs does change – but then it should be executed immediately (at least as soon as possible).

I have often been asked what can you do with *immediate C* ? The short answer is:

- Any programming task which involve logical or analog events, which are related to express actions, which are also events, to act on the environment or on other programs.

More specific uses are:

- Embedded control programs. Since the language was originally developed to be a faster PLC, with negligible CPU loading one of the primary uses of *iC* is for controlling home environments, machines and robots. With the advent of small but powerful micro computers like the Raspberry Pi it is possible to run embedded control programs written in *iC* using hardware GPIOs and other peripherals to provide physical input and output. I/O drivers for the Raspberry Pi come with the system.

- Logic support for GUIs. Wikepedia defines a GUI is a type of interface that allows users to interact with electronic devices through graphical icons and visual indicators. With *iC* only the output of individual icons need to be turned into events, which are transmitted to *iC* executables as standardised messages. The indicator actions are handled similarly by event messages received from *iC* executables. The GUI thus reduces to a graphics wrapper, with *iC* handling all the logic of the application.

- Gaming programs. Such programs are essentially GUIs, where event generating entities and display indicators are hidden in lifelike simulations. In particular the inputs from gaming consoles must be captured by a suitable driver and movements of figures and shifts and rotations of the display must be tied to logical or analog messages received from *iC*. But the program of the gaming graphics can be limited to such motions, with the internal logic of the game delegated to an *iC* program.

The main target for *iC* programs is for real I/O or for interacting with graphical wrapper programs. Nevertheless a simulated I/O program *iCbox* has been provided for testing *iC* programs in an environment without any real I/O. That way users will be able to run *iC* programs immediately to learn the language and test ideas. Also provided is *iClive*, an Integrated Development Environment (IDE) coupled with a live display debugger. *iClive* can be used to enter program text, build an executable and run that executable while showing the state of all displayed variables with different colours. Of course *iC* program sources can be generated with any editor. Syntax high lighting for *iC* has been provided for *vim* with the files ic.vim, filetype.vim and wulff.vim and for printing under Linux with ic.ssh for a2ps.

Following the example of K&R in "*The C Programming Language*" this manual is organized similarly:

Chapter 1 is a tutorial of the central part of *iC* to get the reader started as quickly as possible, since the best way to learn a new language is to write programs in it. The tutorial assumes a basic knowledge of *C*, although much useful *iC* code can be written without any knowledge of much of *C* except expression and assignment elements common to all programming languages.

Chapter 2 describes the I/O interface, which is the only unusual feature of the language. A rationale for the reasons this form was chosen is provided. It is covered first because it is so central to all *iC* programs.

Chapter 3 through 6 discuss various aspects of *iC* in more detail, and rather more formally than in the tutorial, although the emphasis is still on examples of complete programs rather than isolated fragments.

Chapter 3 deals with data types additional to *C*, and the way operators and expressions are handled with these new data types.

Chapter 4 discusses clocking, clocked built-in functions including clock generators and the generation of delays. This is another area, which may not be familiar to most programmers, but is very important in generating *iC* programs which are robust and free of timing races.

Chapter 5 treats conditional statements **if-else** and **switch**, which are not control flow statements like in *C*, but rather initiate the execution of *C* code from *iC* events.

Chapter 6 covers function blocks and program structure – external variables, scope rules, multiple source files and so on.

Chapter 7 discusses the *iC* pre-processor ***immac***, which handles macros like the *C* pre-processor, but whose main function is to generate blocks of *iC* code for arrays of *iC* variables. This allows the generation of different versions of similar *iC* programs from the same source, where the size of arrays is declared in the command line at compile time.

Chapter 8 discusses virtual and real I/O drivers and how these are integrated into a complete network with compiled *iC* applications via a common server called *iCserver*.

Chapter 9 deals with the development of a big control program for an elevator system built from Meccano parts, which has all the motors, buttons and indicators of a real elevator system. The program is developed in a number of steps to show the refinements necessary for a real controller.

I cannot do better than follow the lead of Brian W. Kernighan and Dennis M. Ritchie in their book "*The C Programming Language*" and use that book as a template for this manual with direct quotes where appropriate. Their influence has been very important in designing *iC* and is hereby gratefully acknowledged.

Another strong influence has been "*The UNIX Programming Environment*" again by Brian W. Kernighan with Rob Pike. That book taught me the UNIX way of developing programs and how to write compilers with yacc – building up such a hard topic in easy and exciting steps.

Larry Wall taught me a lot about the linguistic nature of programming languages – making sure they flow easily out of your thoughts. I used that influence in small ways, for example – allowing commas at the end of all comma separated lists, which makes writing long parameter lists vertically so much easier. I use *Perl* for all the auxiliary programs around *iC*, because *Perl* is flexible and makes robust programs. Sriram Srinivasan taught me the Foundations and Techniques for developing real Perl Applications in his book "*Advanced Perl Programming*". Nancy Walsh and later Steve Lidie opened the way to "*Mastering Perl/Tk*", developed originally by Nick Ing-Simmons.

I extend my thanks to all these authors and developers.

John E. Wulff

Bowen Mountain, Australia

4

Table of Contents

1   A Tutorial Introduction

As K&R say in "*The C Programming Language*" let us begin with a quick introduction to *iC*. The aim is to show essential elements of the language in real programs, but without getting bogged down in details, rules, and exceptions. At this point, I am not trying to be complete or even precise (save that the examples are meant to be correct). I want to get you as quickly as possible to the point where you can write useful programs, and to do that I have to concentrate on the basics: variables and constants, logic and arithmetic, conditionals and the rudiments of input and output.

1.1 Getting started

The only way to learn a new programming language is by writing programs in it. The first program to write is the same for all languages:

> *Print the words*

> **hello, world**

in *iC* this is a two line program:

```
%{ #include <stdio.h> %}
if (IX0.0) { printf("hello, world\n"); }
```

Create this program in a file ending in ".**ic**", such as **hello.ic**.

To build this program type the command

> **iCmake hello.ic**

which in turn calls the *immediate C* compiler **immcc** and the *C* compiler and linker *gcc* to produce the executable file **hello**. If you run the command

> **$ hello**

the *iC* run time system will generate (auto-vivify) a small simulated I/O box with a single button labelled **.0** in a column labelled **IX0**. Every time you turn the button **IX0.0** on (**hi**) the program will print

> **hello, world**

The same would happen if you had a real input **IX0.0**. Type **q** to quit the program.

Unlike in *C*, the *iC* code is not placed in *C* style functions, but is placed where one would normally have global variables. Each *iC* statement is executed when one of the *iC* variables making up the statement changes. In the program **hello.ic** a change of state of the external variable **IX0.0** in the **if** statement triggers the execution of the **printf** function call, which is pure *C* code. The *C* code must be enclosed in braces, which are mandatory for *iC* to define a block of *C* code. The block of C code immediately after the **if** condition in braces is executed every time the condition changes state from **lo** to **hi**.

The first line of the program   **hello.ic** is a block of *C* code enclosed in special braces **%{ ... %}**, which is called a Literal Block. These blocks are copied nearly verbatim, but without the special braces, to the generated *C* code ahead of any *C* code embedded in *iC* statements, like the **printf** call above. Literal Blocks are useful for declaring *C* variables, declaring or defining auxiliary *C* functions, defining *C* pre-processor macros with #define and including *C* header files with #include.

The Literal Block **%{ #include <stdio.h> %}** is required by *C in this case* to declare the function prototype of **printf** in the *C* standard I/O library.

If you also want to have an output when you turn the button **IX0.0** off (**lo**) extend the **if** statement with an **else** followed by another block of *C* code

```
if (IX0.0) { printf("hello, world\n"); }
      else { printf("good bye\n"); }
```

Excercise 1-1. Run the "hello, world" program on your system. Experiment with leaving out parts of the program, to see what error or warning messages you get**.**

Excercise 1-2. Extend the program with more external inputs **IX0.1** to **IX0.7** to print different messages.

1.2 immediate Logical Expressions

The next *iC* program **andOr.ic** explores the use of the logical operators AND and OR to act on external outputs **QX0.0** to **QX0.3**. These statements can be placed in any order in the *iC* program without changing its function.

```
QX0.0 = IX0.0 & IX0.1;                  // AND operator &
QX0.1 = IX0.2 | IX0.3;                  // OR operator  |
QX0.2 = IX0.4 & IX0.5 | IX0.6 & IX0.7;  // & has precedence
```

Another way to write the last statement is:

```
QX0.3 = IX0.4 & IX0.5 |                 // AND OR in the style
        IX0.6 & IX0.7;                  // of PLC Ladder Logic
```

Allowing and encouraging this Ladder Logic like coding is deliberate and makes reviewing this common AND/OR construct very obvious.
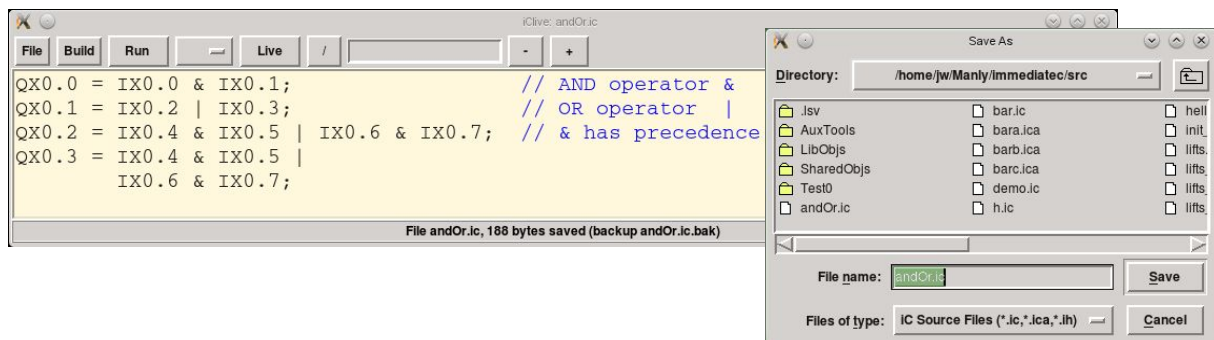
The IDE *iClive* is an easy way to type *iC* sources, build executables and run them. Execute *iClive* and press *File > New* if *iClive* was previously working on a different source. Type or copy the above statements into the Edit window and press *File > SavAs*, typing **andOr.ic** into the Filename: box and *Save*.

Now press *Build > Build executable*. Unless you made a typing mistake the bottom status line of *iClive* will display **'andOr' successfully built**. At this point you can press the *Run* button, which will run the executable **andOr** after auto-vivifying an *iCbox* for all the external I/O variables in **andOr**. You can now experiment, turning various inputs *on* and *off* to see the results in the **QX0** outputs. To activate the debugging mode of *iClive*, press the *Live* button. This will colour all active *iC* bit variables in the program green/black for **off/lo/0** or yellow/red for **on/hi/1.**
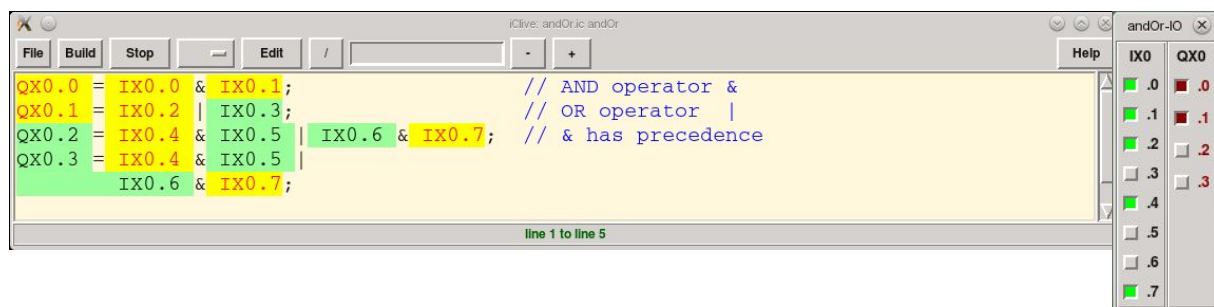
Shutting down *iClive* with *File > Quit* or the ⊗ button in the top right corner will stop **andOr** and close *iCbox*. If you want to leave *iClive* running, stop **andOr** with the *Stop* button and close *iCbox* manually with its ⊗ button. Always close *iCbox* before running a new or modified *iC* program, because it may not have the same external inputs and outputs.

Excercise 1-3. Change the statement order of **andOr.ic** to see if it makes any difference to the output. Tip: use copy (ctrl-C) and paste (ctrl-V) with *iClive* in *Edit* mode. Use [Help] for editor details.

Excercise 1-4. Extend the logical expressions with more inputs. Tip: the next lot of inputs are **IX1.0** to **IX1.7**. Similarly the next outputs are **QX1.0** to **QX1.7**.

Excercise 1-5. Add more complicated logical expressions using parentheses for OR expressions nested in AND expressions because of precedence – just like in *C*.

1.3 immediate Variables and Arithmetic Expressions

The next program uses the formula °F = ((°C*9)/5)+32 to convert an external analog input representing *°Celsius* to an analog output representing *°Fahrenheit*. Additionally an output **tooHigh** will be turned on if the temperature exceeds 25°C.

Just like in *C*, all variables in *iC* should be declared before they are used, except external I/O variables, which follow the IEC-1131 industry standard. External input names start with the letter **I**, external outputs with the letter **Q**. These are the only *immediate* variables we have used up to now. They will be explained in detail in Chapter 2. For all other *immediate* variables a *declaration* announces the properties of variables and reserves storage for them; in *iC* a *declaration* starts with the type modifier **imm**, a type name and a list of variables, such as

```
imm int celsius, fahr;
imm bit tooHigh;
```

The only value types available in *immediate C* are **imm bit** and **imm int.** Type **imm bit** declares variables capable of holding the values **0** and **1**. The word 'boolean' was avoided deliberately, because it has a different semantic bias in languages where it is used. (Truth of a test rather than a single bit object). **imm int** hold numerical or analog integers in the normal *C* way.

Assignment statements in which the right hand side is a single variable or a constant is an *alias* in *iC*. Such a statement produces no code. The *alias* name on the left hand side is simply an alternative name for the *immediate* variable on the right hand side. Aliases are particularly useful for giving meaningful names to external input and output IEC-1131 variables.

This is the full program **cf.ic**

```
/* Convert and display °C to °F and turn on a tooHigh indicator */

imm int celsius, fahr;
imm bit tooHigh;

/* Alias statements which define external inputs and outputs */
celsius = IB1;                   // celsius =
QB1     = fahr;                  // fahr    =
QX0.0   = tooHigh;

fahr    = ((celsius*9)/5)+32;
tooHigh = celsius > 25;
```

Build and run **cf**, which produces the following output with Live mode enabled
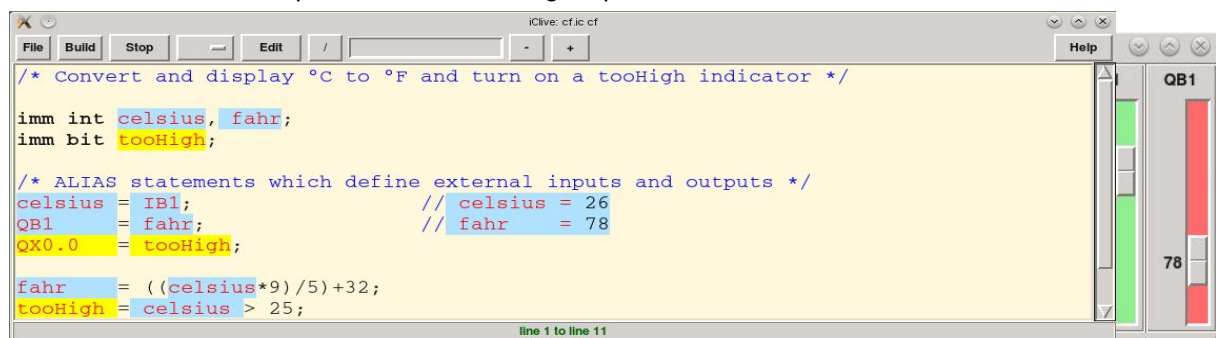
iC supports both **/* C style comments */** and **// C++ style comments**

Both styles have been used in the previous examples. A very special *C++* comment has been used in the following two lines

```
celsius = IB1;                   // celsius =
QB1     = fahr;                  // fahr    =
```

Inside a comment an **imm int** variable name followed by an equal sign **=** at the very end of the line will cause *iClive* to display the numerical value of the variable in *live* mode. Apart from this **imm int** variables are coloured light blue to distinguish them from **imm bit** variables. The lettering is black for a value of 0 and red otherwise. The current numerical value of all *immediate* variables can also be displayed in a balloon by hovering the mouse cursor over a live *immediate* variable.

Care must be taken with integer arithmetic that multiplications are done before division. Thus the following conversion statement will give misleading results

```
fahr    = ((celsius/5)*9)+32;
```

That expression will give the same result of 77 for all **celsius** values from 25 to 29. Floating point *immediate* variables have not been implemented in *iC*, although they are possible. *C* floating point variables can be used effectively in *C* code embedded in *iC* code.

The final statement

```
tooHigh = celsius > 25;
```

demonstrates that an arithmetic relation normally produces an **imm bit** result. Apart from that an arithmetic expression may be assigned to an **imm bit** variable and a logical expression may be assigned to an **imm int** variable. Sensible conversions are done both ways.

Excercise 1-6. Add another output **tooLow** which turns on when the temperature falls below 21°C.

Excercise 1-7. Take the comparison temperature for the indicators **tooHigh** and **tooLow** from another external input and call it **setTemp**. Use **setTemp ± 2** to compare for **tooHigh** and **tooLow**.

1.4 Logical inversion

The unary operator **~** is used in *C* for the bitwise complement of an integer variable. It is used in *iC* for the same purpose on **imm int** variables and for logical inversion of **imm bit** variables, although in practice it is much more commonly used in *iC* for logical inversion of **imm bit** variables.

The following program **waterheater.ic** uses comparisons between integer variables as we have seen in the last example, which return a bit value and logical AND expressions with normal and inverted bit variables. Many *immediate C* control programs follow this simple pattern.

```
/****************************************************************
 *  Control program for a simple urn to provide boiling water
 *
 *  Inputs are an on/off switch, water level and temperature sensor.
 *  Outputs are an electrically operated water tap to fill the urn,
 *  a heating element and a ready light.
 ****************************************************************/

use strict;

imm bit on          = IX0.0;  // on/off switch
imm int waterLevel  = IB1;    // water level sensor
imm int temperature = IB2;    // temperature sensor

imm bit waterLo     = waterLevel <= 90;
imm bit tempHi      = temperature >= 100;
imm bit fill        = on & waterLo; // fill until 90% full
imm bit heat        = on & ~waterLo & ~tempHi;
imm bit ready       = on & tempHi;  // heat till water boils

QX0.0 = fill;
QX0.1 = heat;
QX0.2 = ready;
```

The logic is straightforward, using aliases of input and output variables and intermediate variables to implement the logic. This version of the program uses the compiler directive **use strict**, which is now the default and can be left out. This forces programmers to declare every immediate variable. With the directive **no strict** all undeclared variables are assumed to be **imm bit**, which can lead to subtle errors. The following with **no strict** is allowed but strongly deprecated.
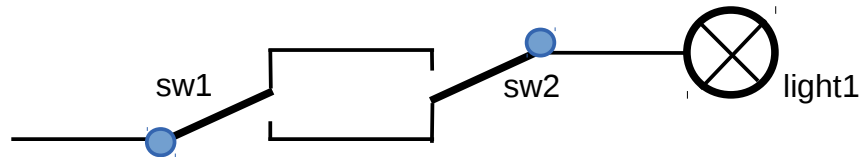
```
no strict;
waterLo = IB1 <= 90;
tempHi  = IB2 >= 100;
QX0.0   = IX0.0 & waterLo;    // fill until 90% full
QX0.1   = IX0.0 & ~waterLo & ~tempHi;
QX0.2   = IX0.0 & tempHi;     // heat till water boils
```

1.5 Logical Exclusive Or

As an example we want to switch a light on or off from two different places – a very common arrangement in most homes, which can be implemented with switches as follows:

This *iC* statement using logical inversions has the same functionality:

```
light1 = sw1 & ~sw2 | ~sw1 & sw2;
```

The expression above is equivalent to a logical *exclusive or*, which expresses the above functionality more simply as follows:

```
light2 = sw1 ^ sw2;
```

One advantage of *exclusive or* is that it can be cascaded – we can easily arrange for more than two switches to each turn on and off one light:

```
light3 = sw5 ^ sw6 ^ sw7 ^ sw8;
```

This can only be done with mechanical switches using so called **cross switches**:
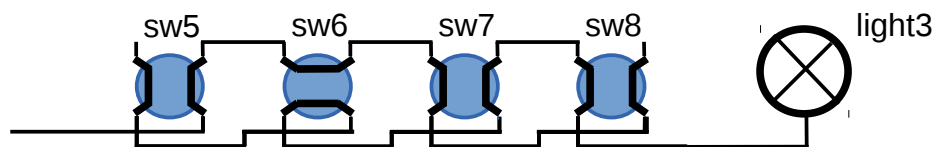
sw6 is up, the others are down and the light is off. Any switch changing will turn the light on.

1.6 Built-in Function Blocks

Function blocks in *iC* serve the same purpose as functions in *C*. A function block provides a convenient way to encapsulate some computation, which can then be used without worrying about its implementation. With properly designed function blocks, it is possible to ignore how a job is done; knowing what is done is sufficient. *iC* has a number of built-in function blocks, which are defined in the supporting run time package as pre-compiled function blocks.

In the following program we will use the built-in function block **LATCH**, with the following function block prototype:

```
imm bit LATCH(bit set, bit reset);
```

This function block is the simplest flip flop or memory element When **set** is **1** and **reset** is 0 the output of **LATCH** goes to **1**; when **reset** is **1** and **set** is **0** the output of **LATCH** goes to **0**. **LATCH** remembers its previous state when **set** and **reset** are both **0** or when they are both **1**.

The following program **aircon.ic** controls an air conditioner, which has two inputs **IB1** and **IB2** from external thermometers for inside and outside temperature. Another input **IB3** provides the desired room temperature. Two bit outputs act on the air conditioner - **QX0.0**, which is **lo** for cooling mode and **hi** for heating, and **QX0.1**, which turns the motor of the compressor on.

The first thing to do is to give meaningful names to the external inputs with alias statements, followed by control statements, which are mostly expressions combined with the declaration of an immediate variable. The alias statements to give meaningful names to the outputs come last. These are reversed. IEC-1131 output names are aliases for meaningful computed variables, whereas for inputs IEC-1131 names are variables with changing values and the meaningful names are their aliases.

```
imm int insideTemp   = IB1;        // insideTemp   =
imm int outsideTemp  = IB2;        // outsideTemp  =
imm int setPointTemp = IB3;        // setPointTemp =

imm bit heating      = LATCH(outsideTemp < setPointTemp,
                             outsideTemp > setPointTemp);
imm bit tooCold      = insideTemp < setPointTemp;
imm bit tooHot       = insideTemp > setPointTemp;
imm bit acMotorOn    = LATCH(heating & tooCold | ~heating & tooHot,
```

```
                                              heating & tooHot  | ~heating & tooCold);
        QX0.0 = heating;          // on is ac heating  off is ac cooling
        QX0.1 = acMotorOn;
```


*Live display of **aircon.ic***

An unusual aspect of the Live display is the fact that inverted variables show their logic state after inversion. The variable **heating** is displayed **hi**, whereas **~heating** is displayed **lo**. This makes inspection of live AND and OR expressions very natural. **heating & tooHot** is obviously **hi**, whereas **~heating & tooHot** is obviously **lo**. Similar arguments apply to OR expressions. All consecutive variables in an OR expression must show the **lo** colour for the whole expression to be **lo**.

Internally **imm bit** variables always have two outputs – the non-inverted or normal output and the inverted output. There is no computational overhead in doing inversion. The **~name** is an inverting alias of a **name**. This can be used to advantage to provide better visual meaning by adding an inverting alias statement to the above code (which causes no run time overhead).

```
        imm bit cooling       = ~heating;
        imm bit acMotorOn     = LATCH(heating & tooCold | cooling & tooHot,
                                      heating & tooHot  | cooling & tooCold);
```

In the state shown in the live display above, the outside temperature is 9°C and the desired temperature is 20°C, which calls for heating, which is provided by the first LATCH call, whose set input is hi, because **outsideTemp < setPointTemp** is true, which is **hi** or **1** in *iC*. Two intermediate variables **tooCold** and **tooHot** are used, because they are both used twice in the second LATCH call, which turns the aircon motor on for heating when the inside temperature is too low and off again when it is too high. The above statements provide a hysteresis of 2°C. With a set point of 20°C heating is turned on when the inside temperature falls to 19°C and turns off when it reaches 21°C.

In cooling mode, which applies, when the outside temperature rises above the set point temperature, the opposite changes in temperature control the aircon motor.

immediate function blocks can also be user-defined. This will be covered later in this chapter.

Excercise 1-8. Run the program **aircon.ic** in *iClive*. Vary all 3 inputs and check that the outputs control heating/cooling and the motor correctly. Have a look at the listing produced by the ***immcc*** compiler by pressing [File] > aircon.lst. Find the assignment statement for **acMotorOn** (Tip: press the search button [/], type acMotorOn in the search box next to the search button and press the search button again). There are 7 expression nodes like logic symbols in a hardware logic diagram listed under the statement. Inputs are on the left with a possible inversion followed by the logic symbols of the node and the output name. The statement is broken up into intermediate nodes. [File] > aircon.ic gets you back to the source.

Excercise 1-9. Save **aircon.ic** as **airconx.ic**. Modify **airconx.ic** by introducing the alias **cooling** for **~heating** as shown above. Build and Run this version and show its listing. The last 4 auxiliary expressions should be identical to the listing of **aircon.ic** showing the variable name **~heating**, which is used for execution, and not its alias **cooling**, which is just a bit of syntactic sugar.

1.7 Counting

Counting is very important for all types of *iC* programs and implementing counters in *iC* opens up a number of aspects which are different from ordinary imperative programming. What you **cannot** do is simply increment an *immediate* variable like this:

```
imm int badCounter   = badCounter + 1;
```

This statement produces the following Error:

```
*** Error: input equals output at gate: badCounter
```

The problem is, that the *immediate* variable **badCounter** would change due to the addition and would be scheduled immediately for another addition – if left like that the CPU would be in an infinite loop with **badCounter** never catching up with itself. Also what are we counting? The basic assumption for imperative languages is that we increment every time the algorithm executes the statement. This does not hold for declarative languages. Worse still is:

```
imm int badCounter++;        // causes a syntax error
```

The **++** and **--** operators as well as all *C* assignment operators **+=**, **-=** etc are not allowed for *immediate* variables declared with **imm** for the same reasons outlined above.

What we can do is to declare a special kind of *immediate* variable with the type modifier **immC** instead of **imm** in front of the two possible *immediate* value types **int** or **bit**. An *immediate* **immC** variable must be declared in *iC* code. It may optionally be initialised with a constant expression as part of the declaration, just like a *C* global variable. After that it can only be assigned in *C* code – but there it can be assigned in more than one *C* statement in the normal imperative manner. Apart from that an **immC** *immediate* variable has all the properties of other *immediate* value variables – it can be used as a value in *immediate* expressions, whose execution will be triggered when that **immC** variable is modified in a *C* statement by an assignment.

To test these ideas let us extend the air conditioner control program to **aircony.ic** with the following added feature: instead of taking the set point temperature from an analog slider we provide two buttons **raiseTemp** and **lowerTemp** to adjust the set point temperature in 1°C steps as is usual in air conditioner remote control units. For this we will need a counter which counts up and down. There are several ways to do this. An obvious way is to use an **immC int** variable for the counter **setPointTemp** and do the counting in *C* code as follows:

```
immC int setPointTemp = 20;   // immC variables are declared and
                              // optionally initialise in iC code

imm  bit raiseTemp = IX0.0;   // push-button aliases
imm  bit lowerTemp = IX0.1;

if (raiseTemp) { setPointTemp++; }  // outside raise button pressed
if (lowerTemp) { setPointTemp--; }  // outside lower button pressed
```

As explained in the program **hello.ic** an *iC* **if** statement executes a block of *C* code enclosed in braces when the variable in parentheses after the **if** (the condition) goes **hi**. The single increment or decrement *C* statements **setPointTemp++** or **setPointTemp--** are executed each time one of the buttons is pressed.

Here is the extended version **aircony.ic** of the program:

```
imm  int insideTemp  = IB1;        // sense insideTemp   =
imm  int outsideTemp = IB2;        // sense outsideTemp  =
imm  bit raiseTemp   = IX0.0;      // remote control push-buttons
imm  bit lowerTemp   = IX0.1;

immC int setPointTemp = 20;        //        setPointTemp =

if (raiseTemp) { setPointTemp++; }  // outside raise button pressed
if (lowerTemp) { setPointTemp--; }  // outside lower button pressed

imm bit heating      = LATCH(outsideTemp < setPointTemp,
                             outsideTemp > setPointTemp);
imm bit cooling      = ~heating;
imm bit tooCold      = insideTemp < setPointTemp;
imm bit tooHot       = insideTemp > setPointTemp;
imm bit acMotorOn    = LATCH(heating & tooCold | cooling & tooHot,
```

```
                              heating & tooHot  | cooling & tooCold);

    QX0.0 = heating;        // on is ac heating  off is ac cooling
    QX0.1 = acMotorOn;
    QB1   = setPointTemp;   // remote control set point indicator
```

1.8 User defined Function Blocks

Unlike in *C* or other imperative languages, where a *function* evaluates a sequence of instructions whenever it is called, *function blocks* in *immediate C* act more like templates, which are cloned every time they are called (actually they are used, not called, but it is easier to think of them as being called). An *immediate* function block is a separate *immediate* subsystem with *immediate* parameters which are its inputs and outputs from other section of the *immediate* system, optional internal *immediate* variables, which must be declared inside the function block and an optional *immediate* return value, which may be used like any other immediate value – in an expression – assigned to an immediate variable or used as an input parameter in a built in or user defined function block call.

Like in *C* a function block provides a convenient way to encapsulate some computation, which can then be used without worrying about its implementation. Like in *C* the use of function blocks is easy, convenient and efficient.

So far we have only used the LATCH function block, which is a built-in function block[1] provided by the *iC* system. Let us encapsulate the counter used in the previous section in a function block and use it in another version **airconz.ic** of the air conditioner program.

```
    /******************************************************************
     *     Up/Down counter with initialisation at compile time
     ******************************************************************/
    imm int upDownCounter(bit up, bit down, const int ini)
    {
        immC int counter = ini;         // declare and initialise counter
        if (up)    { counter++; }        // increment counter
        if (down)  { counter--; }        // decrement counter
        this = counter;                  // return the counter value
    }


    /******************************************************************
     *     Air conditioner control program
     *     with raise and lower set point buttons and set point indicator
     ******************************************************************/
    imm int insideTemp  = IB1;          // sense insideTemp   =
    imm int outsideTemp = IB2;          // sense outsideTemp  =
    imm bit raiseTemp   = IX0.0;        // remote control push-buttons
    imm bit lowerTemp   = IX0.1;        // show  setPointTemp =

    imm int setPointTemp = upDownCounter(raiseTemp, lowerTemp, 20);

    imm bit heating      = LATCH(outsideTemp < setPointTemp,
                                 outsideTemp > setPointTemp);
    imm bit cooling      = ~heating;

    imm bit tooCold      = insideTemp < setPointTemp;
    imm bit tooHot       = insideTemp > setPointTemp;
    imm bit acMotorOn    = LATCH(heating & tooCold | cooling & tooHot,
                                 heating & tooHot  | cooling & tooCold);

    QX0.0 = heating;        // on is ac heating  off is ac cooling
    QX0.1 = acMotorOn;
    QB3   = setPointTemp;   // remote control set point indicator
```

An *iC* function block definition has the same form as a *C* function definition:

```
    imm-return-type function-block-name(parameter declarations)
    {
        declararions
```

---

[1] Built in *iC* function blocks are defined and used in the same way as user defined function blocks.

```
        statements
    }
```

The most significant difference is, that the return type must be an *immediate* type, either **imm int**, **imm bit**, **imm clock**, **imm timer** or **imm void** (the last three will be introduced in Chapter 3).

The function block **upDownCounter** is called once in the line

```
    imm int setPointTemp = upDownCounter(raiseTemp, lowerTemp, 20);
```

Each call clones the function block, replaces the real argument objects for the formal parameters in the definition and generates new nodes linked the same way as in the definition. The value returned by **upDownCounter()** is assigned to **setPointTemp²**.

The first line of **upDownCounter** itself,

```
    imm int upDownCounter(bit up, bit down, const int ini)
```

declares the type of the result that the function block returns as well as all parameter types and their formal names. The **imm** modifier is mandatory for the return type – it identifies an immediate function block definition syntactically. The **imm** modifier is optional for parameters in a parameter list. The declared parameters are nevertheless immediate, except **const int** parameters, which must be matched by a constant expression when called. Parameters may be either input value parameters, in which case only their type is written in the list or the parameter may be an immediate output to which a value from the function block is to be assigned. In this case the type of the parameter must be preceded by the keyword **assign** (This will be explained in more detail in Chapter xx).

The 'return' statement of an *iC* function block is an immediate assignment to a pseudo-variable called **this**, which is a place holder for the value in the expression the function block is used in. In our example the return statement is

```
    this = counter;
```

which simply returns the current incremented or decremented value of **counter** or as in this example is an alias of **counter**.

A function block need not return a value, but in that case it must be declared **imm void**. In all other cases a function block must return a value compatible with its declared return type. A function block with a return value must have a return statement (assignment to **this**) and must either be assigned to a suitable variable or else it must be used as a value of a suitable type in an expression or in an argument list. An **imm bit** function block may be used as an **imm int** value and vice versa – appropriate conversion takes place. Also a function block must have at least one statement. These rules are much stricter than the rules for *C* functions.

1.9 Function Block Arguments

Since *iC* function blocks are cloned when used and each real argument is an *iC* node, which is linked into the network of nodes cloned from the function block definition, the question whether arguments are passed by value or by reference, as in *C* and other computer languages is meaningless, except for **const int** arguments, which are passed by value as the result of a constant expression evaluated at compile time.

Each real immediate value argument of a function block call is either a simple *immediate* variable or an *immediate* expression, both of which are compiled to an expression node object, which is linked to the cloned internal nodes of the function block to form a subsystem of immediate expression node objects driven by the argument expression nodes. An **assign** argument must be the name of a previously declared **imm** variable, which has not been assigned. It will be assigned in the function block.

There is one other type of function block argument – an array of **immC** variables, which will be dealt with in Chapter xx. At this point it is worth mentioning that 'pointers' to *iC* variables are meaningless. The *iC* language can only deal with specific *iC* node objects declared with an **imm** or **immC** declaration or aggregations of **immC** variables in an array.

---

² In this particular example this is not quite true, because the return statement 'this = counter' makes 'this' an alias of 'counter', which makes 'setPointTemp' an alias of 'counter'. But the variable 'counter' is of type 'immC int' which make 'setPointTemp' type 'immC int'. 'imm' variables and 'immC' variables are the same as far as their value is concerned, so in practice there is no difference, except in this special case we could assign to 'setPointTemp' in another C statement. But that would be very bad form and would break the code if 'upDownCounter' is modified to return an expression of type 'imm int'.

2   Input and Output

This chapter describes the I/O interface, which is the only unusual feature of the language. A rationale for the reasons this form was chosen is provided. It is covered first because it is so central to all *iC* programs.

External input and output names in *iC* follow the IEC-1131 standard. This was the standard for PLC's when I worked as a software engineer developing firmware for PLC CPU's in the 80's. Unfortunately that standard was renamed IEC-61131 in 1993 and was changed considerably – in particular the following naming conventions were no longer included as standard. Nevertheless they are still widely used in industry and provide a sensible way to identify sources and sinks of external data in control software with physical terminals in I/O racks. I have extended this usage in *iC* to use IEC-1131 names as a common naming convention for sources and sinks of data between any type of app making up a larger network of communicating *iC* applications. This includes I/O drivers for real I/O, *iCbox* – a simulated I/O driver, GUI wrappers, which also provide sources and sinks of external data and the actual *iC* executables themselves.
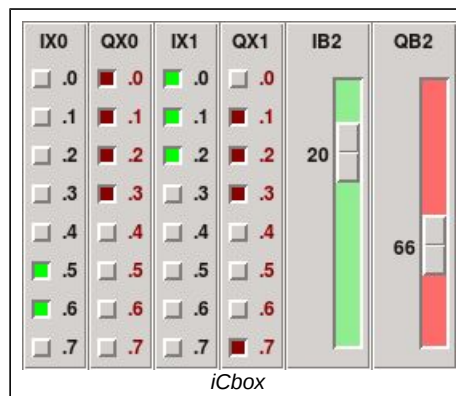
Inputs start with the letter **I**, outputs with the letter **Q**. These are followed by a second letter which defines the type of the input or output. **X** defines unsigned bytes of 8 single bit I/O variables. **B** defines unsigned numerical byte I/O variables, **W** signed 16 bit word I/O and **L** signed 32 bit long word I/O variables. The letters **H**, **F** and **D** have been reserved for 64 bit long long or huge integers, 32 bit floating point and 64 bit double precision floating point variables. None of these last three have been implemented yet. The 2 capital letters are followed by a number, which defines the address index of the variable in the I/O field. For bit I/O variables the address is followed by a full stop and a number in the range 0 to 7, marking the bit address of the actual bit variable in the addressed I/O byte. The maximum address index that can be used depends on the implementation of the driver and the underlying hardware. Addresses in the I/O field may be used for bit, byte, word or long word I/O. If all of these are in the same physical address space, care must be taken not to overlap different types of I/O. In the case 16 and 32 bit word I/O variables the byte addresses used may need to be on a 16 bit word or a 32 bit long word boundary respectively. The *iC* compiler can generate warnings if I/O fields overlap. In the default case, each size variable is assumed to be in its own address space and the address of each variable is simply an index into each of these address spaces.

Here are some examples of IEC1131 names:

```
IX0.0     bit 0 of input byte 0 - pre-declared as imm bit
IX0.1     bit 1 of input byte 0   ...
IX0.7     bit 7 of input byte 0

IX1.0     bit 0 of input byte 1
IX1.1     bit 1 of input byte 1   ...
IX1.7     bit 7 of input byte 1

QX0.0     bit 0 of output byte 0 - pre-declared as imm bit
QX0.1     bit 1 of output byte 0   ...
QX0.7     bit 7 of output byte 0

QX1.0     bit 0 of output byte 1
QX1.1     bit 1 of output byte 1   ...
QX1.7     bit 7 of output byte 1

IB2       input byte 2 - pre-declared as imm int (8 bit input)

QB2       output byte 2 - pre-declared as imm int (8 bit output)
```

The IEC-1131 names above define the physical addresses of inputs and outputs in the I/O field. Standard practice for PLC I/O electronics is to package I/O units in narrow plug in units, which are labelled as shown on the right. The program *iCbox*, which is a simulated I/O widget, emulates this scheme, showing the relationship of physical addresses to their IEC-1131 names.

For more readable applications it is   highly recommended, that alternate descriptive names are defined for IEC-1131 input and output names. This would normally be done in a table of alias assignments at the start of an iC program. One advantage of this scheme is, that if an input or output is physically moved to another I/O pin, only 1 statement in the source needs to be changed.

IX0	QX0	IX1	QX1	IB2	QB2
.0	.0	.0	.0		
.1	.1	.1	.1		
.2	.2	.2	.2	20	
.3	.3	.3	.3		
.4	.4	.4	.4		
.5	.5	.5	.5		66
.6	.6	.6	.6		
.7	.7	.7	.7		

iCbox

IEC-1131 names are pre-declared *immediate* variables. **IX0.0** and **QX0.0** etc are of type **imm bit**, whereas **IB0**, **QB0**, **IW0**, **QW0**, **IL0**, **QL0** etc are all of type **imm int**. All declared **imm int** variables have the native **int** size provided by the *C* compiler used to compile the output of the ***immcc*** compiler, which is usually 32 bits. All arithmetic is carried out with signed native integers, except that the byte numerical I/O variables **IB0**, **QB0** etc are **unsigned char**. The usual *C* automatic conversion of an **unsigned char** to a **signed int** is used to transfer values to and from the default **signed int** used for arithmetic.

IEC-1131 input and output variables are pre-declared for *iC* and *C* code and normally do not need to be declared except for the following cases:

- An **extern imm** type declaration of an IEC-1131 input variable or an **extern imm** or **extern immC** type declaration of an IEC-1131 output variable is needed if the same input or output variable is going to be used in more than one module.

- An **imm** type declaration of an IEC-1131 input variable or an **imm** or **immC** type declaration of an IEC-1131 output variable is needed if that variable has been declared with an **extern imm** declaration or in the case of an output variable with an **extern immC** declaration in this source module (usually in an included .ih header), which means its storage is going to be defined in this module. For an output variable this also means that the output variable must be assigned in this source unless it is declared **immC**, in which case *C* assignment in this source is optional.

- An IEC-1131 output variable, which is to be assigned in C code must be declared **immC** independent of whether it was declared **extern** or not.

- IEC-1131 input variables can never be declared **immC**, because they are value variables which can never be assigned either in *iC* or *C* code. Their values are determined in another app.

These rules for input and output variables are the same as for ordinary immediate variables, except that IEC-1131 I/O variables which have not been declared **extern imm** do not need to be declared at all (pre-declared when used in *iC* code and *C* code by default) except that IEC-1131 output variables which are to be assigned in *C* code must be declared **immC** like ordinary immediate variables which are to be assigned in *C* code.

```
extern imm int QB2;    // to avoid multiple assign error
imm int QB2 = IB2 * 2; // QB2 must be declared imm int

extern imm bit QX0.2;           // to avoid multiple assign error
imm bit QX0.2 = IX0.2 & IX0.3;  // QX0.2 must be declared imm bit

immC int QB1;          // must be declared immC int to allow C assign
immC bit QX0.1;        // must be declared immC bit to allow C assign
if (IX0.0) { QB1 = IB1; QX0.1 = IX0.1; }
```

Excercise 2-1. Write two short iC source programs **a.ic** and **b.ic** in which some *immediate* I/O variables assigned in **a.ic** are used in **b.ic**. Tip: to build the executable **a** execute

```
$ iCmake -l a.ic b.ic
```

2.1 Communication between iC apps

An app written in *immediate C* normally requires a driver program to supply or sink the IEC-1131 variables used in the app, although this driver code has been incorporated directly in the support library for some real I/O hardware to gain a significant speed advantage (real means physical in this manual and not floating point). Only the GPIOs and the PiFace extension board for the Raspberry Pi computer have direct high speed drivers, which can be linked directly to an app. All other drivers and GUI wrappers (represented by an app called *iClift* in the distribution) use TCP/IP communication via a special program called *iCserver* to forward event data to and from *iC* executables. The physical channels for this TCP/IP communication can be **localhost** (**127.0.0.1**) for communication between *iC* apps and *iCserver* running in parallel on the same CPU. For apps running on other hosts on the same local area network (LAN) or generally anywhere on the internet, the IP address name or 4 part numerical IP identifier of the host that *iCserver* is running on can be specified by apps to register as clients with *iCserver*. The IP port used is 8778. When coming from outside a LAN, this port must be allowed to pass messages through any firewall (a different port number can be specified if 8778 is a problem).

3 immediate Data Types, Expressions and Assignments

This chapter deals with data types additional to *C*, and the way operators and expressions are handled with these new data types.

3.1 iC Variable Names

Names of variables in *iC* follow the same pattern as in *C* - letters and numerical digits; the first character must be a letter. The underscore "_" counts as a letter and "$" counts as a number, although the use of "$" is deprecated, because not all *C* compilers can handle it. Upper and lower case letters are distinct. There is no limit to the length and case of *iC* variables. Only global variables used in embedded *C* code have a limit, although most *C* compilers do not seem to impose a limit these days. The only restriction on *iC* variable names are *iC* and *C* keywords like **if**, **else**, **imm**, **bit, int**, etc and names starting with "iC", which are used by *iC* internally

3.2 iC Data Types and Sizes

Immediate C has six data types for use in *immediate* expressions, four of which are value variables:

imm bit	is a single bit variable assigned in *iC* code and mainly used in *immediate* logical expressions
imm int	is a variable whose size is the native size of a *C* **signed int** variable assigned in *iC* code and mainly used in *immediate* arithmetic expressions
immC bit	is a single bit logical variable which can only be assigned in C code
immC int	is an **int** sized arithmetic variable which can only be assigned in C code

All **imm** and **immC** value variables can be used in both *iC* and *C* code. Only assignment is restricted.

The other two are special types used for synchronising the change of groups of *immediate* variables to avoid timing races and for producing timed or counted delays:

imm clock	synchronises the change of a group of variables
imm timer	delays the change of a variable by a fixed or computed amount.

All these types are implemented with objects, which have extra members to implement the immediate execution strategy in addition to the **bit** or **int** values. For the actual logic or arithmetic these extra members are irrelevant.

There is a last *immediate* pseudo type:

imm void	used only to declare a function block without a return value.

3.3 iC Expressions

Immediate expressions are arithmetic or logical expressions external to all functions, which contain at least one *immediate* value variable or function block call. *Immediate* arithmetic expressions may also contain constants, whereas *immediate* logical expressions, containing only **imm bit** variables, may not. **An *immediate* expression is re-evaluated whenever the value of one of the *immediate* variables it contains has changed (and only then)**. This is the core the of *iC* event-driven strategy.

Immediate expressions are most often assigned to variables declared **imm int** or **imm bit**, which can be used in other *immediate* expressions. Each such assignment causes all *immediate* expressions containing the *immediate* variable just assigned to be re-evaluated. *Immediate* expressions may also be used as value parameters in an immediate function block call, which usually causes immediate assignments in the *iC* code cloned by the function block call or its return – all of which propagate to other *immediate* expressions and finally to *immediate* outputs.

3.4 iC Assignments

Immediate assignments are assignments of *immediate* expressions to *immediate* value variables. If the value of the expression just computed has not changed from its previous value, nothing happens in the assignment and no follow on expressions are affected. Only value changes to an *immediate* variable are detected in the assignment and this event triggers the re-computation of all *immediate* expressions, in which the *immediate* variable, which has just changed, is a member. This is made possible, because each *immediate* variable object has a list of pointers to every *immediate* variable, whose assignment expression is directly modified by the current *immediate* variable. This strategy ensures that all *immediate* variables are kept up to date with the minimum amount of computation.

Like in *C*, an *immediate* assignment is also an *immediate* expression, which means that assignments embedded in expressions are allowed. *immediate* assignments can be combined with the declarations of *immediate* variables, but such declaration assignments are not an expression.

Assignments of *iC* expressions to *immediate* variables obey the single assignment rule, a rule which applies generally for data flow systems. Any *immediate* variable may only be assigned in one *immediate* assignment. If multiple *immediate* assignments were allowed there would be a conflict between the current values of the different expressions being assigned to the same variable. Therefore attempts at multiple *immediate* assignments are a hard compile error.

Expressions that occur in *C* code triggered by *immediate* conditional *if else* or *switch* statements or in *C* functions or in literal blocks may contain *immediate* value variables. These expressions are not *immediate* expressions and are not triggered by the variables in the expression. Instead they are executed following conventional instruction flow in the *C* code. When such an expression is executed in the *C* code, the current value of any *immediate* variable is used in standard instruction flow manner.

Immediate variables may even be assigned in *C* code embedded in *immediate* conditional *if else* or *switch* statements or in literal blocks. Such an assignment is **not** an *immediate* assignment – the value is changed when the *C* statement is executed. Nevertheless any change in the *immediate* variable assigned in the *C* code will trigger *immediate* expressions that contain that variable. Several such assignments to the same *immediate* variable may be made in different sections of *C* code. Every new assignment changes the variable in accordance with the intended algorithm. *Immediate* variables assigned in *C* code must be declared as `immC bit` or `immC int` in an *iC* code section. An *immediate* variable that is assigned in *C* code may not also be assigned in an *immediate* assignment.

3.5 Constants and Constant expressions

Only integer constants of type `int` can be used in *iC*. The value of an integer can be specified in decimal like `1234,` octal or hexadecimal. A leading `0` (zero) on an integer constant means octal; a leading `0x` or `0X` means hexadecimal. A character constant is an integer, written as one character within single quotes such as `'x'`. iC constants follow the same rules as for constants in *C*, except that modifiers for sizes other than `int` as well as floating point constants are not supported.

If an expression consists only of constants and no *immediate* variables it is a constant expression evaluated at compile time. Constant expressions may be assigned to a variable declared `imm int`, which becomes an alias of the constant value of the expression, executed at compile time, which obviously never changes and is itself a constant. Constant expressions may be used to index members of `immC` arrays in *iC* code and to initialise `immC` variables following their declaration (similar to global initialisation in *C*).

3.6 C Variables in iC Expressions

Plain *C* `int` variables can be used in *immediate* arithmetic expressions, but their use in this way is deprecated, since any change in such a *C* `int` variable does not trigger re-execution of the expression when it's value changes. To alert programmers, any plain *C* `int` variable to be used in *iC* expressions must be declared in *iC* as follows:

```
extern int var;              // C variable to use in an imm expression
```

One possible use of a plain *C* variable is one which holds the value of a command line term, which never changes after starting the program. A better choice is an `immC` variable, which can be changed in *C* code if that were necessary.

3.7 C Functions and Macros in iC Expressions

C functions and macros to be used in *iC* expressions must also be declared in *iC* as follows:

```
extern int rand();        // C function with no parameters
extern int rand(void);    // alternative syntax for no parameters
extern int abs(int);      // C function or macro with 1 parameter
extern int min(int, int); // C function or macro with 2 parameters
```

It is easy to mistype the name of an *iC* function block call, which then looks like a C function call. Unless declared `extern` such a non-defined function block call will be compiled without error as a *C* function call. Such an error is not discovered until link time. By forcing `extern` declarations clean error messages are produced at *iC* compile time and the extra effort is not great.

When a *C* function or macro is called in an *immediate* expression, a check is also made, that the number of parameters is the same as in the `extern` declaration. An error message is issued if not correct. No check is made for *C* function calls in *C* fragments controlled by `if else` or `switch`

statements or other literal *C* code, since the compilation is handled by the follow up *C* compiler, which relies on its own function declarations with modern *C* compilers. This does mean that the correct #include files for any *C* library functions to be used in *iC* code must also be mentioned in a literal block.

As can be seen above, only *C* **int** variables and *C* functions returning an **int** value and having only **int** parameters may be used in *iC* expressions. For any other type *C* variable or function a suitable *C* wrapper function, which casts all values to **int**, must be defined in a literal block.

3.8 External Variables and Scope

The *C* language makes a distinction between "external" objects, which are either variables or functions and "internal" objects, which are variables and arguments inside functions. External variables are defined outside of any *C* function, and are thus potentially available to many functions. Functions themselves are always external. By default, external variables and functions have the property that all references to them by the same name, even from functions compiled separately, are references to the same thing (quoted from K&R).

This distinction holds for any *C* code in an *iC* program. But in straight *iC* code all *immediate* variables are "external" by the above definition, except that formal parameters and variables declared in an *iC* function block definition fall into a different category altogether. Since *iC* function blocks are templates, which do not compile into any code objects until they are cloned in a function block call, the formal parameters and variables declared in a function block definition are "virtual" objects, which do not become real external objects until a function block is cloned. Virtual *iC* objects used in a function block definition have similar scope to internal variables in a *C* function definition – they are only defined inside a function block.

In practice the scope rules for *C* and *iC* variables and function (blocks) are similar, which was one of the design aims for the *iC* language. The main difference is, that all non-virtual *iC* variables are external because they exist outside of any *C* function. Syntactically *iC* variables are like *C* global variables with an initialiser expression assignment. In *C* this initialiser expression must be a constant expression, which is evaluated and assigned at compile time. The same is true for **immC** variables, which can then only be modified in *C* code. For **imm** variables the expression for the single *immediate* assignment stays active - it is re-evaluated whenever an **imm** or **immC** variable in that expression changes, unless the expression is a constant expression, in which case the **imm** variable becomes an alias of a constant.

Like in *C*, *iC* programs need not all be compiled at the same time; the *iC* source text of the program may be kept in several files. Each such *iC* source module is separately compiled into a *C* file with the **immcc** compiler, which are then compiled by the *C* compiler and linked with other compiled modules and the *iC* run-time library into a machine code executable. Also like in *C*, each immediate variable must be defined in one, and only one source module as follows:

```
imm bit heat;
```

although it is recommended to combine the variable definition with the *immediate* expression assignment required for the functionality of the program as follows:

```
imm bit heat = on & ~waterLo & ~tempHi;
```

Sometimes the functionality is circular, in which case a variable must be defined without an assignment before it is used. The defining type **imm bit** may be repeated when that variable is finally assigned, as long as the type matches the previous definition. It is recommended that all immediate assignments are preceded by their defining *immediate* type. A trial compilation will report any variables, which have been used before they have been defined as *undefined*. A simple definition can then be placed near the beginning of the program for those *undefined* variables. (None of this is necessary if **no strict** is used, but this can lead to subtle errors and is highly deprecated).

An immediate variable defined in another *iC* source module must be declared **extern** in the source it is used in, just like in *C*.

```
extern imm bit waterLo, tempHi;
```

The rules for **extern** variables in *iC* are the same as in *C*. Such **extern** variables can be used in any *immediate* expression without being defined or assigned in the current module. The expectation is, that they will be defined and assigned in another module, which will be linked to this module. There is one difference to *C* though. Because of the single assignment rule **extern** variables, which have not been subsequently defined in this module, may not be assigned in this module. It causes a multiple assignment error.

3.9 immC Arrays

immC Arrays are arrays of **immC bit** or **immC int** variables of the same type as its members. Just like ordinary **immC** variables, indexed references to an **immC** Array may be used as immediate values in both *iC* and *C* code, but they may only be assigned and changed in *C* code – either in **if else** or **switch** *C* code fragments or in literal blocks. Another limitation is, that **immC** Array indexed value references in *iC* code may only use a constant expression index. Such an indexed **immC** Array element is an alias for the **immC** member referred to and as such simply provides some syntactic sugar. In the example below, bb[0]is the same as bx - it simplifies coding though. Whole **immC** Arrays may be passed by name in a Function Block call, if the Function Block definition specifies an **immC** array in that position in its formal parameter list.

immC Arrays are declared in *iC* code – either with or without a list of named members.

```
immC int aa[3];               // immC int aa0, aa1, aa2; corres-
                              // ponding to aa[0] aa[1] and aa[2]
                              // are automatically generated

immC bit bx, by, bz;          // the immC members in an initialiser
immC bit bb[] = { bx, by, bz }; // list may be pre-declared
immC bit cc[3];
```

A declaration of an **immC** Array without a member list must specify a size. The member names automatically generated are the name of the array followed by a number equal to the index. (This follows the same pattern as **imm** Arrays resolved by *immac* , which will be introduced in chapter xx. This choice was deliberate). Multi-dimensional **immC** arrays have not been implemented.

For an array with a member list the size specification is optional, but must equal the number of members in the list if it is specified. The names in the member list can be any previously declared immC variable – they may even be indexed references of a previously declared **immC** Array. If not previously declared, the members are generated in the array declaration, just like automatic members.

```
immC bit ccr[3] = { cc[2], cc[1], cc[0] };  // reverse of cc[3]
```

immC Arrays may be used in another source if they have been previously declared **extern**. The **extern** declaration must match the final declaration exactly. The size must match and if a member list is provided it must also be provided identically in the **extern** declaration. Only that way can the members of an **immC** Array be used correctly both in *iC* code and *C* code of another source file.

```
extern immC int aa[3];

extern immC bit bx, by, bz;
extern immC bit bb[] = { bx, by, bz };

extern immC bit cc[3];
extern immC bit ccr[3] = { cc[2], cc[1], cc[0] };
```

An **immC** Array knows its own size and a run time warning occurs if an indexed reference is not within the size range of the array. An indexed reference, which is out of range returns **bit** or **int** 0.

immC Arrays may be passed as formal parameters in a function block definition (xx). A formal array parameter is a name followed by square brackets which either contain a numeric size or is empty. If a size is given (b[4]), the call to that function block must provide a previously declared array of exactly that size. In this case *iC* code in the function block can also access the array. If no size is specified (a[]), any size array can be provided in the call. That array can only be accessed in C code in the function block. It is up to the C code algorithm to make sure that index values are within range.

The built in *iC* operator **sizeof** *array* returns the number of elements of an **immC** *array* (not its size in bytes)*.* The **sizeof** operator works best in *C* code fragments where its value is dynamic at run time. It also works in *iC* code, where its value is determined at compile time. A difference occurs in function blocks which have been passed an array of indeterminate size (a[]) as a parameter. Only the **sizeof** operator in *C* code will return the actual size of the array passed in a call. Since in this situation only indexed references in *C* code fragments are possible, the **sizeof** test in the *C* code is appropriate.

sizeof may be used to test index values to produce own error strategies. The following must be true:

```
index < sizeof array
```

To sum up: each use of an indexed **immC** array member like cc[2] in *iC* code or cc[x] in *C* code is itself an **immC** variable – namely the indexed **immC** member of the array – and has all the properties of a simple i**mmC** variable.

4   Built-in Functions

iC has a number of built-in functions, which are so central to the operation of the system, that they have been made a part of the language. They are implemented as efficient building blocks in the supporting run time package. All built-in functions are defined internally as pre compiled Function Blocks. (parameter types shown are all immediate – the keyword **imm** is optional for Function Block definitions and is left out in this description for clarity). All except the **LATCH** and the **FORCE** functions are clocked, which is analogous to similar functionality in hardware IC's. ¿ Why use Clocking ?

4.1   Race conditions, Glitches and Clocking

A race condition is an undesirable situation that occurs when a device or system attempts to perform two or more operations at the same time, but because of the nature of the device or system, the operations must be done in the proper sequence to be done correctly. Race conditions manifest themselves as timing races between different events in electro-mechanical relay logic, electronic switching circuits as well as in computer software, especially multi-threaded or distributed programs. Timing races can also occur in *iC*, because it is an event driven system. All these systems take a small, but not negligible amount of time to execute their various actions. This leads to the situation, where an event signal may be processed by several elements on different paths in a network. When a design specifies that the signals triggered by one event come together again, the timing through these elements may lead to a timing race, where the signal through one path may come before or after that same signal processed through another path. Under unfavourable conditions this leads to a short erroneous output, which is known as a glitch.
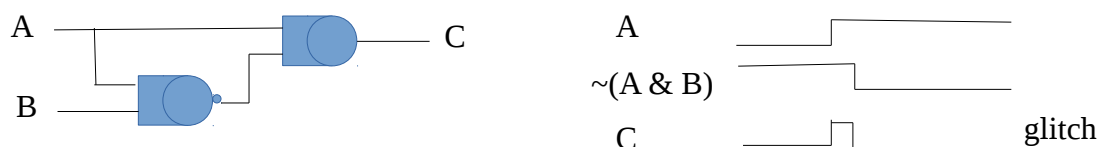
The simplest example to demonstrate a timing race is a two-input AND gate fed with a logic signal A on one input and the same signal passed through an inverting AND gate on the other input. In *iC* this can be tested with the statement C = A & ~(A & B); (The2nd input B on the inverting AND gate has been added and is always hi – it has been included, because in *iC* a simple inverter on A would be an alias, which would make the inputs for C = A & ~A. This condition is recognised by the *iC* compiler as an error, since the output is always lo). In theory the output C = A & ~A should never be **hi**. However for both electronic logic and *iC*, changes in the value of A take longer to propagate to the second input through the inverting AND gate than the first when A changes from **lo** to **hi**. This results in a brief period during which both inputs are **hi**, and so the output of gate C will also be **hi**. Once the **lo** signal ~A arrives through the inverting AND gate, the output of C becomes a correct **lo**. But for a short period the **hi** glitch on C may trigger memory elements like a LATCH if C is connected directly to their set or reset input.

Design techniques such as Karnaugh maps encourage designers to recognize and eliminate race conditions before they cause problems. This was the only way to deal with race conditions in electro-mechanical relay circuits.

Fortunately in the late fifties John Sparkes invented a method called Clocking or synchronous logic for electronic logic circuits, which completely eliminates the effects of glitches[3]. With Clocking, memory elements such as RS flip flops have an extra clock input in addition to their normal set and reset input. The effect of the clock is to hold up the output of any clocked memory element synchronised by the same clock until all combinatorial logic – including all glitches have settled down. Clocking also ensures that the outputs of a number of clocked memory elements never change between clock pulses, which ensures that the next state of a memory element after a clock can never affect the logic during the current clock period. For clocked electronic logic circuits there is a minor penalty. The frequency of the clock must be slow enough so that all combinatorial actions have completed between two clock pulses.

Clocking has been used to good effect in the design of the *immediate C* language. It has been implemented as follows in *iC. A*fter all combinatorial changes induced by one or more input events have been computed, a clock phase is started, which usually changes some logic values. This starts a new run of combinatorial actions, which is again followed by a clock phase. This sequence is continued

---

[3]I was fortunate to be introduced to clocked logic in 1964 at the 'British Telecommunications Research Laboratory' where John Sparkes made his invention. I used clocked logic with Germanium Transistor and Diode circuits to design a control computer for a large mail sorting machine. This was well before clocked logic became popular with DTL and TTL integrated circuit chips.

until there are no more changes to compute. Only at this point are external outputs sent. After this the *iC* system waits for further input events at which point the cycle is repeated.

Because clock phases follow immediately on completed combinatorial action phases there is no timing penalty for using clocking in *iC*. It is worth pointing out here, that combinatorial and clock actions take fractions of microseconds to execute on modern computers. Times between external events in a system to be controlled by an *immediate C* program are usually in the range of 50 ms to seconds, minutes or even hours. For even the fastest inputs the CPU loading of an iC program is rarely more than 1 %.

As pointed out earlier even software using multi-threaded or distributed programs can suffer from race conditions. This happens when two threads read and write the same shared data simultaneously. Mechanisms to control such sharing are Mutexes and Semaphores.

Controlling race conditions, glitches and synchronisation of multiple outputs is relatively straightforward and very safe, if the rules and calling sequences for clocked function blocks described in the rest of this chapter are followed. These rules are identical to those used by clocked integrated circuit elements, which have proved immensely successful for implementing modern computing circuits. The use of clocking is certainly much simpler than the rules for using Mutexes in multi-threaded programs.

4.2 Clocked memory elements

Following the usage in hardware integrated circuits a number of different clocked memory elements have been implemented in *iC*. These are the D flip-flop with and without Set and Reset, the SR flip-flop and the JK flip-flop. An unusual memory element is a 'Sample and Hold', which is a direct analogy of the clocked **D** flip-flop for numeric values.

All clocked built in memory elements follow the Master Slave principle, which is also the way clocked memory elements are realised in hardware.

$$C = SR(A \,\&\, B, c, D \,\&\, E, c)$$

Master gates   Slave function
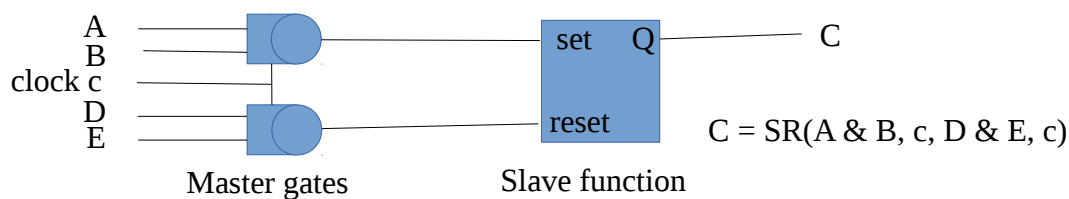
The Master gates of a clocked function do not act immediately on other gates, but instead are linked to a clock. Only when the clock fires, which happens in the clock phase, do the outputs of those gates act on the Slave function, which is expressed as a Truth Table as shown below for the different clocked functions. During the clock phase no changes of any Slave outputs can change any combinatorial expressions acting on Master gates, either directly or indirectly through other gates, which means the state of all Slave outputs at the end of a clock phase reflect the state of the Master gates at the beginning of that clock phase (when incidentally all glitches have been resolved).

4.2.1 Clocked D flip-flop

The simplest clocked flip-flop is the D flip-flop or delay memory element, a function having a single logic input, a clock input and an output equal to the input in the previous clock period.

```
imm bit D(bit expr, clock c);   or
imm bit D(bit expr);         /* default iClock used as clock */
```

The following truth table describes the D flip-flop:

expr D^n	D(expr,c) Q^{n+1}
0	0
1	1

The **D** flip-flop has become the most commonly used clocked flip-flop in hardware design. Its application is called for, when several bit expressions must produce synchronized outputs, so that any further logic done with these outputs does not suffer from timing races. A typical example is the implementation of a state machine. The **D** flip-flop is also a 1 bit memory element, which can store

information from one clock period to the next. The **D** flip-flop is called for in any design where feedback is involved. The use of the clocked **D** flip-flop in *iC* will probably fall into a similar pattern.

For all clocked built in functions with more than one input value parameter, each such parameter may have its own clock. If a clock parameter is supplied it applies to all value parameters on its left, which do not have their own clock. If no clock parameter is specified, the built in **iClock** is used.

4.2.2 Clocked SR flip-flop

The memory element that is represented in most PLC instruction sets is the R-S flip-flop. This flip-flop has two logic inputs. The rising edge of the set input puts the flip-flop in the "one" state and the rising edge of the reset input puts the flip-flop in the "zero" state. Many books on switching theory describe a simple unclocked latch memory element by the name R-S flip-flop. Following the usage for PLC's in IEC-1131, and because the set parameter precedes the reset parameter in the calling sequence, the clocked Set-Reset flip-flop was named **SR** flip-flop in *iC*:

```
imm bit SR(bit set, clock sc, bit reset, clock rc);
```

set S^n	reset R^n	SR(set,sc,reset,rc) Q^{n+1}
0	0	Q^n
0/1	X	1
X	0/1	0
1	1	Q^n

A version with one set input and two reset inputs is provided (mainly to implement the full **SRT** mono-flop as a function block).

```
imm bit SRR(bit set, clock sc, bit reset1, clock rc1,
                                bit reset2, clock rc2);
```

The **SR** flip-flop implemented in *iC* differs marginally from the classical R-S flip-flop described in the literature, which has the disadvantage that Qn+1 is undefined for R and S both "one". The design rules for the R-S flip flop state that R and S must never be "one" together. Since this would cause unwarranted confusion the implementation with the above truth table was chosen, which gives identical results with designs following the rules of the classical R-S flip-flop. If the rule of both inputs "one" is ignored, the results are still easy to interpret. For the above reasons clocked R-S flip-flops are rare as integrated circuits.

4.2.3 Clocked JK flip-flop

Instead **JK** flip-flops were made in hardware. They toggle their output on every clock pulse, when J and K are both "one". In recent years even these have not been listed in the IC data books. A **JK** flip-flop has been implemented in *iC*:

```
imm bit JK(bit set, bit reset, clock c);
equivalent to  SR(set & ~Q, reset & Q, c);
```

set J^n	reset K^n	JK(set,sc,reset,rc) Q^{n+1}
0	0	Q^n
1	0	1
0	1	0
1	1	$\sim Q^n$

4.2.4 Clocked SRX flip-flop

In practice the simple clocked **SR** flip-flop can be difficult to control under the following conditions:

A `0/1` set transition has occurred which sets the flip-flop and some time later a `0/1` reset transition occurs which resets it, while set is still a `1`. Even if reset goes back to `0`, the set input is not active

again until it goes back to `0` and then to `1` again. This works well in many situations, but can be counter intuitive. For this reason the **SRX** flip-flop or the **JK** flip-flop can be used more effectively.

```
    imm bit SRX(bit set, clock sc, bit reset, clock rc);
     equivalent to  SR(set & ~reset,sc, reset & ~set,rc);
```

set S^n	reset R^n	SRX(set,sc,reset,rc) Q^{n+1}
0	0	Q^n
0/1	0	1
0	0/1	0
1	1	Q^n
1\0	1	0
1	1\0	1

When both set and reset are `1`, then both internal S and R inputs are `0`. If there is a `1\0` transition on either `set` or `reset`, then the alternate input has a `0/1` transition, which sets or resets Q.

4.2.5 D flip-flop with Set and Reset

D flip-flops may have an optional set or reset input or both, as well as the D input. The names of these variants indicate which parameters are required (clocks are optional):

```
    imm bit D(  bit expr, clock c);        /* simple D flip-flop */
    imm bit DS( bit expr, clock c, bit set, clock sc);
    imm bit DR( bit expr, clock c, bit res, clock rc);
    imm bit DSR(bit expr, clock c, bit set, clock sc,
                                   bit res, clock rc);
```

4.2.6 Mono-Flop ST(set, timer, delay) or SRT(set, reset, timer, delay)

The Mono-Flop, or **ST**() function is a modified **SR** flip-flop, in which the output is internally connected back to a timed reset input. This internal reset is usually clocked by a **TIMER,** which is controlled by a delay parameter. The delay parameter may have a fixed or variable numeric value. The **ST** mono-flop output is reset, when the number of **TIMER** ticks corresponding to the value of "delay", from the moment when the **ST** was set, has occurred.

```
    imm bit ST(bit set, clock sc, timer tim, int delay); or
    imm bit ST(bit set, clock sc, clock tc);
```

The **SRT** mono-flop has an additional reset parameter, which can reset the mono-flop prematurely. The **SRT** mono-flop is based on the **SRR** flip flop, which has two reset inputs.

```
    imm bit SRT(bit set, clock sc, bit res, clock rc, clock tc);
```

Instead of clocking with a delay **TIMER**, any clock may be used as the last parameter of the **ST** mono-flop, which is then reset on the next clock pulse after it has been set. The last timer, delay or clock must be specified – it may be **iClock** in which case a thin pulse is produced - one fundamental clock period wide. Both set (and reset in the case of **SRT)** can have clock parameters – default is **iClock** if none are provided.

4.2.7 Clocked Sample and Hold

This function is a direct analogy of the clocked **D** flip-flop for numeric values. The numeric output of the **SH** function equals the numeric input in the previous clock period.

```
    imm int SH(int arithmeticValue, clock c);
```

The sample and hold function can be used to sample fast changing numeric outputs at a constant clock rate. Other uses are the implementation of many useful constructs such as state machines, counters and shift registers, to name a few.

```
imm int count = SH(count + 1, c);      // count clock c pulses
// shift register with b as input in the least significant bit.
imm bit b;                             // b assigned somewhere
else
imm int shift = SH((shift << 1) + b, c);
```

4.2.8 Sample and Hold with Set and Reset

The Sample and Hold function also comes with either reset or set and reset inputs. When the reset input is clocked, the output is set to all 0's. By analogy when the set input is clocked the output is set to all 1's. The inputs `set` and `reset` are **imm bit** expressions; whereas the first input `arithmeticValue` and the output are **imm int**.

```
imm int SHR( int arithmeticValue, clock c, bit res, clock rc);
imm int SHSR(int arithmeticValue, clock c, bit set, clock sc,
                                  bit res, clock rc);
```

4.3   Unclocked memory elements

There is one unclocked memory element in *iC*, the LATCH function, which was already used in earlier chapters.

4.3.1 Unclocked flip-flop or LATCH

The unclocked R-S flip-flop is the **LATCH** function with the following calling sequence:

```
imm bit LATCH(bit set, bit reset);
```

The following truth table describes the **LATCH** function:

set	reset	LATCH(set,reset)
		Q
0	0	Q
1	0	1
0	1	0
1	1	Q

The **LATCH** function is particularly fast and efficient, using only a single gate node. It is of course possible to program a similar latch function with a pair of cross coupled OR gates. In *iC* this looks as follows:

```
imm bit set, reset, Q, Qbar;
Q    = set & ~reset | ~Qbar;
Qbar = reset & ~set | ~Q;
```

The disadvantage of this implementation is the fact that four gate nodes are used and that its function as a latch memory element is hidden. **LATCH** clearly shows its function.

4.3.2 FORCE function

Closely related to the **LATCH** function is the **FORCE** function with the following calling sequence and truth table:

```
imm bit FORCE(bit arg1, bit on, bit off);
```

arg1	on	off	FORCE(arg1,on,off)
0	0	0	0
1	0	0	1
X	1	0	1
X	0	1	0
0	1	1	0
1	1	1	1

The **FORCE** function passes the value of `arg1` to the output if both `on` and `off` are `0` (or both are `1`). If

only `on` is `1` then the output is forced to `1`, independent of the value of `arg1`. Conversely if only `off` is `1` then the output is forced to `0`. This function is useful for testing.

The **LATCH** function is generated by the more fundamental **FORCE** function as follows:

```
(temp = FORCE(temp, set, reset))
```

Feedback of its own output 'temp' is used to hold that value at its input, unless the 'on' or 'off' inputs force the output to a different value, which is then maintained.

4.3.3 Clocked LATCH function DLATCH

A final memory element in *iC* is a clocked LATCH, which is implemented as an unclocked FORCE function as a Master input to a clocked D flip flop with feedback from the output of the D flip-flop to the FORCE function:

```
imm bit DLATCH(bit set, bit reset, clock c);
equivalent to   (temp = D(FORCE(temp, set, reset), c));
```

DLATCH will not trigger on glitches on its set and reset inputs, whereas **LATCH** will. This means that **LATCH** should only be used of the sequencing of the set and reset inputs is very simple and is guaranteed to not have glitches. Unlike the other clocked memory elements, **DLATCH** may not have separate clocks on its set and reset inputs. To synchronise a number of memory elements the same clock must be used for all inputs anyway.

4.4 Edge detectors

It is often useful to generate a pulse on the rising and/or falling edge of a logical signal or on a change of numeric value. These pulses should turn off at the next clock. Edge detectors can be generated as composites of a D flip=flop and another gate, but since these operations are quite important, the following more efficient functions are implemented in *iC*.

```
imm bit RISE(bit expr, clock c);      // pulse on rising edge
imm bit FALL(bit expr, clock c);      // pulse on falling edge
imm bit CHANGE(bit expr, clock c);    // pulse on both edges
```

The **CHANGE** function is also implemented for arithmetic expressions. The output is nevertheless of type **imm bit**.

```
imm bit CHANGE(int arithExpr, clock c); // pulse on every change
```

The **bit** output pulses every time `arithExpr` changes, qualified by the clock `c`. The clock limits the rate at which changes are recognized. This is often useful with numeric values, which may change at a high rate, and a slower sampling rate is called for.

The pulse outputs of all edge detectors are just long enough, so that they catch the next clock pulse after the edge, but only that one clock pulse – not more. When the output of an edge detector is used directly or indirectly as input of another clocked function with the same clock, correct synchronization is achieved. Edge detectors are needed when the rising or falling edges of a number of signals which overlap need to be combined.
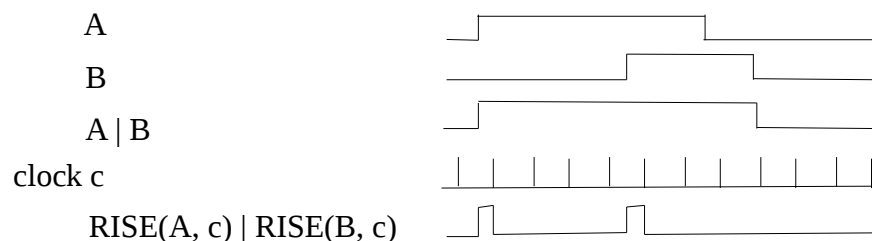
As shown in the diagram, A | B has only one rising edge, whereas RISE(A, c) | RISE(B, c) has two rising edges, which is what is normally required.

Note: there is a significant difference between the output of the **RISE** function and the output of the **ST** mono-flop. The output of the **RISE** function turns on with the rising input signal and turns off again on the next clock. The output of the mono-flop turns on with the next clock after the set signal and turns off with the next clock after that, which is one clock pulse later, assuming the same clock is used for set and internal reset. When the two clocks are different, which is usual for **ST** mono-flops, the case is different again.

4.5   Clock Signals and Clock functions

There are two types of clock signal, '**imm clock**' and '**imm timer**'. It is important to realize that clock signals are not of the same type as logic or numeric value signals of type '**imm bit**' or '**imm int**'. Clock signals are declared as follows:

```
imm clock myClock;
imm timer myTimer;
```

Under no circumstances may clocks be interconnected with logic or numeric values. Any attempt to do so generates a hard error message. Clock signals in *iC* are best thought of as timeless pulses, whose occurrence marks the separation of one clock period from the next along the time axis. All clocked functions in *iC* follow the *Master-Slave* principle. The *Master* element in a D flip-flop follows the input. The output of this *Master* gate is transferred to the *Slave* element during the active phase of the next clock pulse. The output of the *Slave* element is the output of the D flip-flop. All *Master-Slave* transfers during one particular clock pulse are completed before more combinatorial bit or arithmetic expressions are executed. This insures that the outputs of all functions, which are synchronized by the same clock, change simultaneously as far as the input logic is concerned.

Clock signals can come from four different sources:

1. The built-in **iClock**, which is signal type   **imm clock**
2. The **CLOCK** function, which generates type   **imm clock**
3. The **TIMER** function, which generates type   **imm timer**
4. The **TIMER1** function, which also generates type **imm timer**

4.5.1 Built-in immediate clock `iClock`

There is a built-in *immediate* clock with the name **iClock**. This clock runs at the highest system rate. Syntactically **iClock** is used as the default clock, when no other clock is specified. It may also be specified by the name **iClock** when no default clock is allowed by the syntax of a function call.

```
x = SR(set, reset);    // set and reset clocked by built-in iClock

y = SR(set, iClock, reset, rc); // clock for the set argument
                                // must be named if different
                                // from the reset clock rc
```

iClock introduces a clock phase immediately after every completed run of combinatorial actions, which have linked a Master gate of a clocked function to the special clock list **iC_c_list**, which is the action list for **iClock**. Because secondary clocks either use **iClock** by default, or another clock that is eventually clocked by **iClock**, all clocks (and timers) are synchronous with **iClock**. The execution of *immediate* logic is triggered by some input, which causes evaluation of follow up statements, until no more changes occur. **iClock** generates a clock pulse after every such burst of activity in the logic. **iClock** has the same significance for *immediate* logic as the "end of program cycle" in a conventional PLC. The main difference is, that for a conventional PLC all statements in the program are executed for each program cycle. For *immediate* logic only the changes triggered by one or at most a few simultaneous inputs are executed for each program cycle. This typically takes a few microseconds at most for a modern processor. There are support tools which can measure and display this time in microseconds.

4.5.2 CLOCK function

The second source of clock signals is the **CLOCK** function, which has one or two logic inputs – each with an optional clock input. The **CLOCK** function produces an output **clock** pulse during the active phase of the input clock, which follows a 0 to 1 transition of one of the logic inputs. If no **clock** input is specified, **iClock** is used. All **CLOCK** outputs are synchronous with their input clock, and ultimately with **iClock**. The following are the calling profiles for the **CLOCK** function:

```
imm clock CLOCK(bit in,  clock c);    or
imm clock CLOCK(bit in1, clock c1, bit in2, clock c2);
```

The following are examples of calling the **CLOCK** function and using the **clock** output:

```
imm clock clk = CLOCK(b);   // 'clk' on the rising edge of b
                            // clocked by next 'iClock'(default)
imm bit   y   = D(a, clk);  // D flip-flop clocked by 'clk'
```

```
imm clock cl2 = CLOCK(b,~b); // clock on rising and falling edge
                             // of b, both clocked by 'iClock'
```

The following diagram shows the timing relationship between **iClock** and input **b** to the output **clock**
clk generated by the **CLOCK()** function, the timing of clocking y = **D**(a, clk) with **clk**, and the
timing of generating **cl2** with the function above.

iClock

b

clk

a

y

cl2
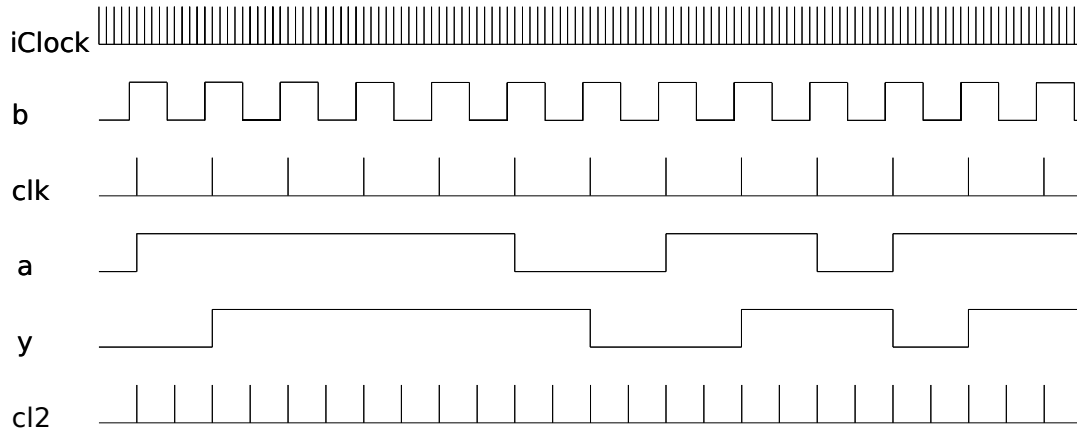
4.5.3 TIMER function

The third source of clock signals is the **TIMER** function, which also has one or two logic inputs – each
with an optional **clock** input. The output generated by the **TIMER** function are of signal type **imm**
timer and are generated in precisely the same way and at the same time as **clock** pulses from a
CLOCK function with the same inputs. **timer** pulses differ from **clock** pulses in the way they are used.
Input parameters of type **timer** are followed by an optional delay parameter, which may be a constant
value or an arithmetic expression (if missing a value of 1 is used). The current value of the delay
expression is read on the rising edge or change of the associated input, and the result **n** is used to
count **timer** pulses. The output is changed by the **n**th **timer** pulse after the changing input. Use of a
clock rather than a **timer** changes the output of a function on the next **clock** after a change in
input. If the delay value **n** of a **timer** call is 0 - or on the falling edge of a logic input for a function other
than the **SH**, **CHANGE** or **switch** function - the output is changed immediately by the next **iClock**. For
a **SH**, **CHANGE** or **switch** function the input is usually arithmetic and those functions are timed on all
changes of input, even if they are a logic input, which is possible for the **CHANGE** function.

```
imm timer TIMER(bit in,  clock c);     or
imm timer TIMER(bit in1, clock c1, bit in2, clock c2);

imm timer tim = TIMER(b);     // 'tim' on the rising edge of b
                              // clocked by next 'iClock'(default)
imm bit   z   = D(a, tim, 3);// D flip-flop clocked by 'tim',
                  // turn on delayed by 3 'tim' pulses,
                  // immediate turn off clocked by 'iClock'
```

The following diagram shows the behaviour of a **TIMER()** generated **timer** for different length's of
input **'a'** relative to the **timer 'tim'** pulses:

iClock

b

tim

a

tim 1   2   3   iClock off   tim 1   2 short tim 1   2   3

z
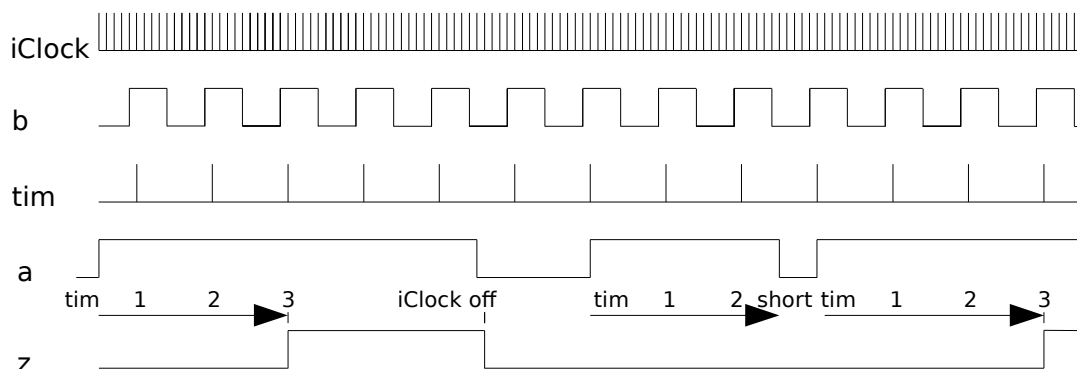
A **D** flip-flop clocked with a **timer** generates a function with turn on delay. If the logic input to such a delay element turns off before the delay time is up, the output never turns on. This is a very useful function to implement time-outs, which are notoriously difficult to implement by conventional means.
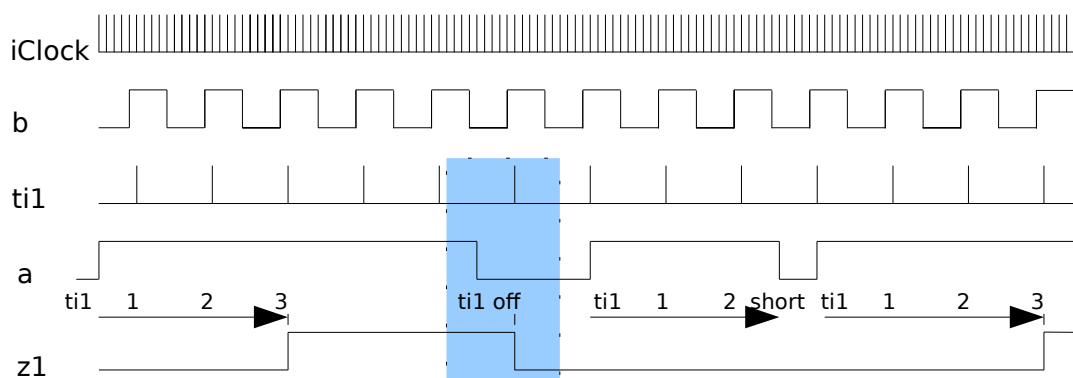
4.5.4 TIMER1 function

The fourth source of clock signals is the **TIMER1** function, which is very similar to the normal **TIMER** function. The signal type generated is **imm timer** – the same as the type generated by a normal **TIMER**. The only difference is the way in which a 0 delay and the falling logic input is handled, when a **timer**, generated by the **TIMER1** function controls a clocked function. A 0 delay is handled like a delay of 1 – turn on is at the next **timer** pulse. On the falling edge of the input the output is clocked on the next **timer** pulse, rather than by the next **iClock**, which is the case for **TIMER** generated **timer** signals unless the input is to an **SH**, **CHANGE** or **switch** function, in which case the falling edge is also timed – just like for the **TIMER** function. A **TIMER1** generated **timer**, used with a delay of 1 (or 0), functions identically to a **CLOCK** generated **clock** signal, except there is a small, but significant amount of overhead in handling **timer** signals. For this reason **CLOCK** functions are to be preferred – their use is very fast.

```
imm timer TIMER1(bit in,  clock c);   or
imm timer TIMER1(bit in1, clock c1, bit in2, clock c2);

imm timer ti1 = TIMER1(b);   // 'ti1' on the rising edge of b
                             // clocked by next 'iClock'(default)
imm bit   z1  = D(a, ti1, 3);// D flip-flop clocked by 'ti1',
                 // turn on delayed by 3 'ti1' pulses,
                 // turn off clocked by next 'ti1'
```

The following diagram shows the different turn-off handling for a **TIMER1** generated **timer** (in the shaded area):

CLOCK, **TIMER** and **TIMER1** functions have optional clock inputs, which may come from other **CLOCK** or **TIMER** functions. All **CLOCK**, **TIMER** or **TIMER1** outputs are synchronous with their input clock(s). This absolute synchronisation is an important aspect of the robust performance of clocked *immediate C* applications. The cascading of clocked functions allows the realization of many useful applications.

4.6 Timing and miscellaneous inputs

To allow programs to work with real time, the following timing inputs have been provided as internal inputs in *iC*:

```
TX0.3    10 milliseconds    // 5ms on, 5ms off
TX0.4    100 milliseconds   // 50ms on, 50ms off
TX0.5    1 second           // 500ms on, 500ms off
TX0.6    10 seconds         // 5 seconds on, 5 seconds off
TX0.7    60 seconds         // 30 seconds on, 30 seconds off
```

These are **imm bit** inputs, not **imm clock** signals. They are mainly used to generate clocks or timers, which are synchronous with real time. For example:

```
imm clock clk100ms = CLOCK(TX0.4);       // every 100 ms
imm timer tim500ms = TIMER(TX0.5, ~TX0.5); // every 500 ms
```

The following miscellaneous internal inputs will be discussed in later examples.

```
      TX0.0 or EOI       // off during initialization, then always on
      TX0.1 or STDIN     // notification of a line of standard input
      TX0.2 or LO        // bit input which is always lo (0 in C code)
     ~TX0.2 or HI        // bit input which is always hi (1 in C code)
```

The aliases `EOI`, `STDIN`, `LO` and `HI` are compiler generated when those words are used in expressions. They are thus keywords in the *iC* language and may not be declared a second time.

4.7 Example programs using clocked functions

So far in this chapter the calling profiles and functionality of the iC built-in functions have been listed. The following examples explain the way these functions are used. It must be stressed again, that the way they are used is exactly the same as the use of similar IC function modules in hardware electronic logic design. There is a lot of literature on this subject, which will help programmers to come up to speed in this area. On the other hand the following examples will show how clocking designs are organised and clocking is used.

4.7.1 To be continued