

The Programming Language "*immediate C*"

- a language for the “Internet of Things”

Programming Manual

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

Copyright © 1985-2019 John E Wulff

SPDX-License-Identifier: GPL-3.0+ OR Artistic-2.0

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

contact the author at
immediateC@gmail.com

\$Id: iC_prog.odt 1.10 2019/10/13

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 of *iC*, which is compiled into pure *C* code by a pre-compiler, just like *yacc*. A fast executable is made by compiling and linking the generated *C* code. The *iC* language uses the syntax of *C* both for its declarative statements by giving meaning to statements that have no semantic support in *C* and obviously for its embedded *C* statements. This should make it very easy to learn *iC* for anyone familiar with *C* or its derivatives *C++* or *Java*. The only parts which may be unfamiliar to programmers are the functionality and use of some of the built-in function blocks. These are based on the well known family of TTL hardware building blocks for creating hardware digital and analog circuits, which will be described in detail in this manual.

iC is similar to Hardware Description Languages (HDL), but it is aimed at generating fast executables on any computer capable of running *C* and **not** for designing hardware. It is also similar to Programmable Logic Controller (PLC) languages, but it does not require specialised PLC hardware and is much faster than PLCs. *iC* is a simple language, which is based on the same concepts as logical and analog IC circuits, electromechanical relays, operational amplifiers and is capable of building control systems by combining the equivalents of such elements with real inputs and outputs from the “Internet of Things”.

Since *immediate C* is an extension of *C*, in a similar way that *C++* is in extension of *C*, using the same declaration syntax, same operators and similar variables as *C*, 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, in other words any activities in the “Internet of Things”. 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. Wikipedia 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 I/O 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, which is much easier to express in *iC* than in a regular instruction flow language such as *C* or *Python*, which require an event loop with much overhead and poor performance.
- 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. Watch points allow breaks in program execution for debugging. Of course *iC* program sources can be generated with any editor. Syntax high lighting for *iC* has been provided for *vim* and for printing under Linux for *a2ps*.

Following the example of K&R in “*The C Programming Language*” this manual is organized similarly (permission kindly granted by Brian Kernighan):

[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](#) 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 5](#) covers function blocks and program structure – external variables, scope rules, multiple source files and so on.

[Chapter 6](#) 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 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 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

Table of Contents

Preface	2
1 A Tutorial Introduction	7
1.1 Getting started	7
1.2 immediate Logical Expressions	8
1.3 immediate Variables and Arithmetic Expressions	9
1.4 Logical inversion	10
1.5 Symbolic Constants	11
1.6 Delayed execution	11
1.7 Logical Exclusive Or	12
1.8 Built-in Function Blocks	12
1.9 Counting	13
1.10 User defined Function Blocks	15
1.11 Function Block Arguments	16
2 Input and Output	17
2.1 Communication between iC apps	18
3 immediate Data Types, Expressions and Assignments	19
3.1 iC Variable Names	19
3.1.1 iC Keywords	19
3.1.2 additional C Keywords	19
3.1.3 iC Pragmas	19
3.1.4 iC built-in Function Blocks	19
3.1.5 iCa Keywords	19
3.2 iC Data Types and Sizes	19
3.3 iC Expressions	20
3.4 Operators in iC expressions	20
3.4.1 Arithmetic and Relational Operators	20
3.4.2 Bitwise integer Operators	20
3.4.3 Bit Operators	20
3.4.4 Logical Operators	21
3.4.5 Conditional Operators	21
3.5 iC Assignments	21
3.6 Constants and Constant expressions	21
3.7 C Variables in iC Expressions	22
3.8 C Functions and Macros in iC Expressions	22
3.9 External Variables and Scope	22
3.10 immC Arrays	23
4 Literal Blocks, immediate Conditional Statements and Pragmas	25
4.1 Literal blocks	25
4.2 immediate conditional if else statement	26
4.3 immediate switch statement	26

4.4	Pragmas	26
4.5	Comments	27
5	immediate Function Blocks	28
5.1	immediate Function Block Definition	28
5.2	immediate Function Block Call	30
6	Built-in Function Blocks	32
6.1	Unclocked memory elements	32
6.1.1	Unclocked flip-flop or LATCH	32
6.1.2	FORCE function	32
6.2	Race conditions, Glitches and Clocking	33
6.3	Clocked digital memory elements	34
6.3.1	Clocked SR flip-flop	34
6.3.2	Clocked JK flip-flop	35
6.3.3	Clocked SRX flip-flop	35
6.3.4	Mono-Flop ST(set, timer, delay)	35
6.3.5	Clocked D flip-flop	36
6.3.6	D flip-flop with Set and Reset	36
6.3.7	Clocked LATCH function DLATCH	36
6.4	Edge detector functions RISE, FALL and CHANGE	36
6.5	Clocked analog memory element	37
6.5.1	Clocked Sample and Hold function SH	37
6.5.2	Sample and Hold with Set and Reset	37
6.6	Clock Signals and Clock functions	38
6.6.1	Built-in immediate clock iClock	38
6.6.2	CLOCK function	38
6.6.3	TIMER function	39
6.6.4	TIMER1 function	40
6.7	Timing and miscellaneous inputs	40
6.8	Example programs using clocked functions	41
6.8.1	A divide by 10 Moebius ring counter	41
6.8.2	A divide by 16 binary counter	41
6.8.3	A state machine showing running lights	42
7	Arrays and the pre-compiler <i>immac</i>	43
7.1	Immediate Arrays	43
7.2	Use of immediate Arrays	43
7.3	Implementation of immediate Arrays	43
7.4	FOR loops	45
7.5	IF ELSE control statements	46
7.6	iCa index expressions	47
7.6.1	Multi-dimensional index syntax	48
7.7	Differences between iC and iCa code	49
7.8	immac Macro facility	50
7.8.1	Alternative immac Macro options	51

8	<u>I/O drivers and iCserver</u>	52
8.1	<u>iCserver</u>	53
8.2	<u>Bernstein Chaining</u>	55

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**) with the left mouse button, 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. (This way of declaring and using Literal Blocks was taken over directly from yacc). 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 the **printf** function 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"); }
```

Exercise 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.

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



1.2 immediate Logical Expressions

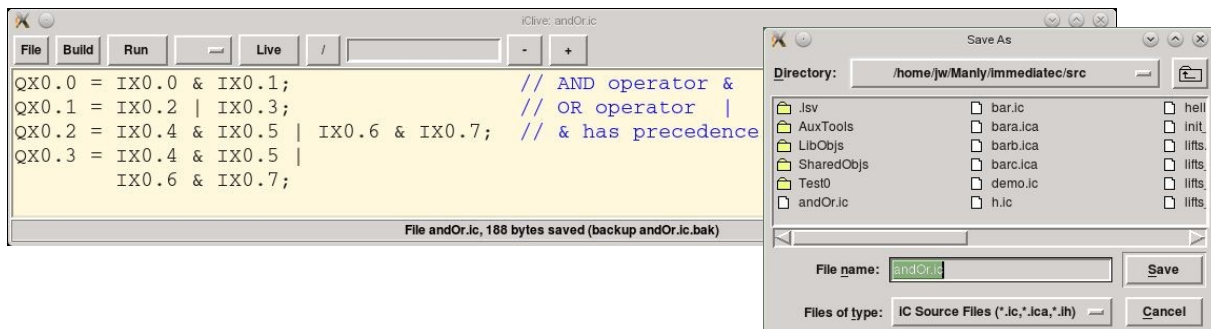
The next *iC* program **andOr.ic** explores the use of the logical operators AND and OR to act on external outputs **QX0.0** to **QX0.3**. These statements can be placed in any order in the *iC* program without changing its function.

```
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:

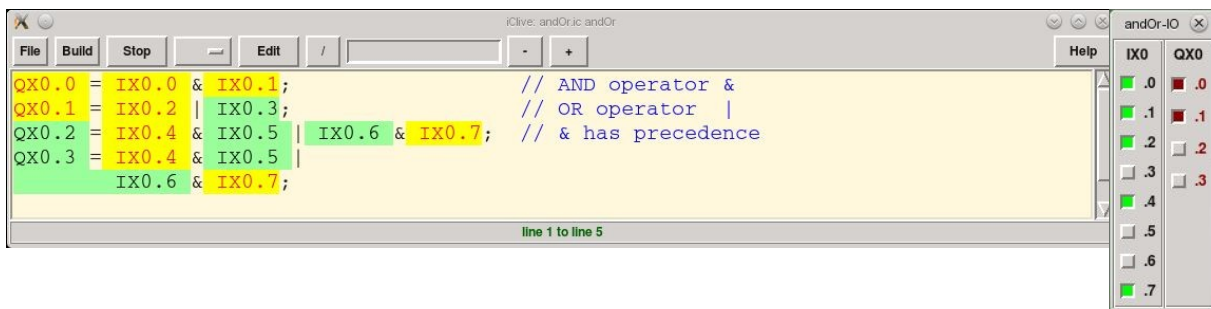
```
QX0.3 = IX0.4 & IX0.5 |           // AND OR in the style
      IX0.6 & IX0.7;               // of PLC Ladder Logic
```



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 > SaveAs**, 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 **0** or **LO** and **yellow/red** for **1** or **HI**.



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.

Exercise 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.

Exercise 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**.

Exercise 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 $^{\circ}\text{F} = ((^{\circ}\text{C} \times 9) / 5) + 32$ to convert an external analog input representing $^{\circ}\text{Celsius}$ to an analog output representing $^{\circ}\text{Fahrenheit}$. 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. IEC-1131 input names start with the letter **I**, IEC-1131 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* an *immediate* variable *declaration* usually 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 *immediate* value types available in *iC* are **imm bit** and **imm int**. Type **imm bit** declares variables capable of holding the values **0** or **LO** and **1** or **HI** only. The words boolean, false and true were avoided deliberately, because they have a different semantic bias in languages where they are used (truth of a test rather than a single bit object). Type **imm int** hold numeric signed integers in the normal C way.

Assignment statements in which the right hand side is a single variable is an *alias* in *iC*. (An *alias* is simply an alternate name for the same object). Aliases are particularly useful for giving meaningful names to external input and output IEC-1131 variables as shown in the following code:

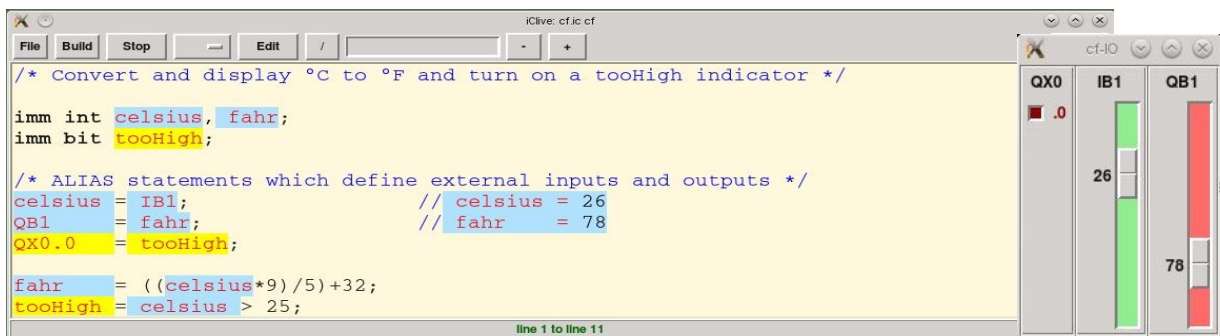
This is the full program **cf.ic**

```
/* 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; // correct order of mult and div
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 numeric 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 numeric 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; // incorrect order of mult and div
```

That expression will give the same result of 77 for all **celsius** values from 25 to 29. *immediate floating point variables* have not been implemented in *iC*, although they would be 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.

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

Exercise 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 for the latter in *iC*.

The following program **urn.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; // ready when 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 Symbolic Constants

The C language provides for symbolic constants with `#define` lines, which are processed by the C preprocessor. These are available as a matter of course in C code embedded in literal blocks and conditional statements. Symbolic constants are useful to hide magic numbers – it is bad practice to use numbers in expressions, which may change and cause problems if the same number is used in several places. *immediate C* has its own pre-processor **immac**, which provides for `%define` lines in *iC* code with the same syntax and functionality as `#define` lines in C. Nevertheless symbolic constants in *iC* are better expressed by the *alias* mechanism making `%define` lines superfluous. The *iC* pre-processor also handles `%include <file>` lines for *immediate C* code and conditional compilation with `%if` lines with all its variations, just like the C pre-processor.

An *immediate* assignment of a numeric value or even a constant expression (which is evaluated at compile time) is an *alias* of the numeric constant or evaluated constant expression, which makes it a good symbolic constant. In the last program we could provide the following two aliases:

```
imm int UrnCapacity = 90;           // alias of a constant
imm int BoilingPt   = 100 - 3;     // provide for sensor tolerances

imm bit waterLo     = waterLevel <= UrnCapacity;
imm bit tempHi      = temperature >= BoilingPt;
```

Alias statements do not generate any code in *iC*. They provide syntactic sugar during compilation.

1.6 Delayed execution

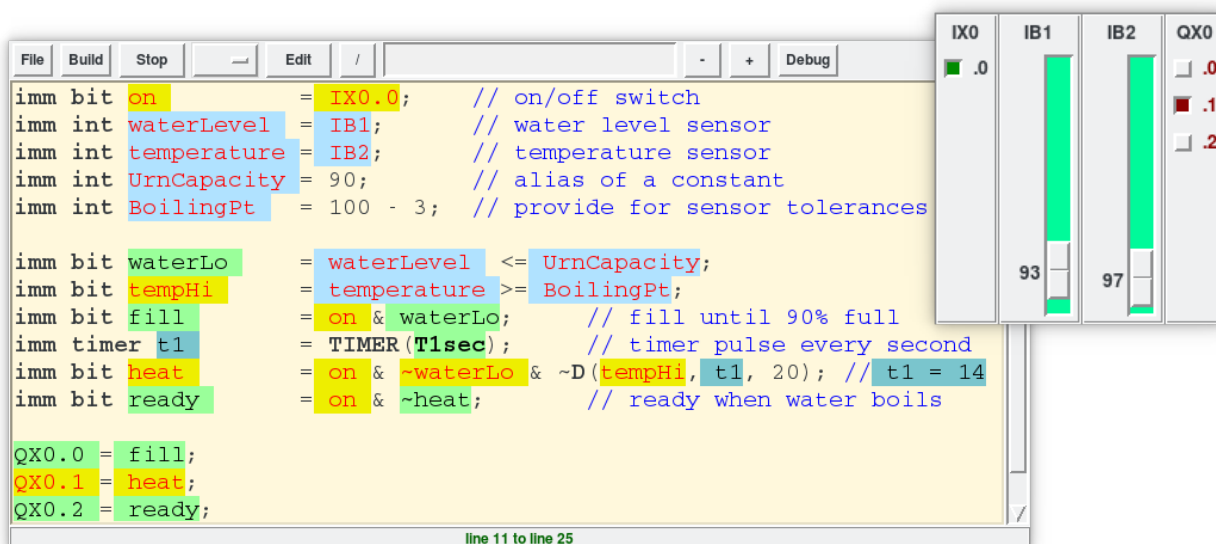
In the above change to the definition of **BoilingPt** we have allowed for tolerances in the water temperature sensor, which is proper engineering practice. The program as it now stands would never bring the water to the boil. A way to overcome this, is to keep heating the water for a short time after the sensor has indicated it has reached near boiling point temperature. To do this *immediate C* provides a mechanism to delay changes of state in logic and arithmetic signals by a given amount – usually a certain amount of time. For logic signals the delay can be for turning on or off as follows:

```
imm timer t1      = TIMER(T1sec);    // timer pulse every second
imm bit delayedOn = D(in, t1, 10);   // on delayed by 10 seconds
imm bit delayedOff = ~D(~in, t1, 20); // off delayed by 20 seconds
```

The full explanation of this mechanism will be given in chapter xx. For the urn program we want the heating to continue after the **tempHi** sensor detects near boiling temperature, which is a turn off delay (the turn on for heating is `~tempHi` so the input to the turn off delay is `~~tempHi == tempHi`). Heating will continue for 20 seconds after 97°C has been reached.

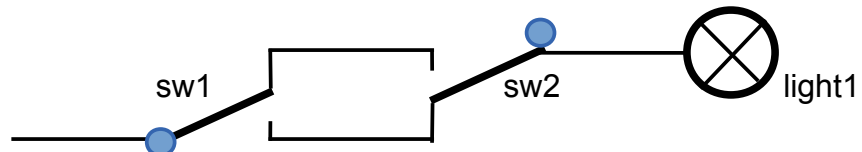
```
imm bit heat      = on & ~waterLo & ~D(tempHi, t1, 20); // t1 =
imm bit ready     = on & ~heat;                       // ready when water boils
```

Exercise 1-8. Incorporate the changes in the last two sections into the program **urn.ic** Build and Run it with *iClive*. Turn on (IX0.0) and vary the **waterLevel** (IB1) and **temperature** (IB2) sliders to near 100 and watch heating start and then the timer t1 counting down to 0, at which point **ready** (QX0.2) come on.



1.7 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;
```

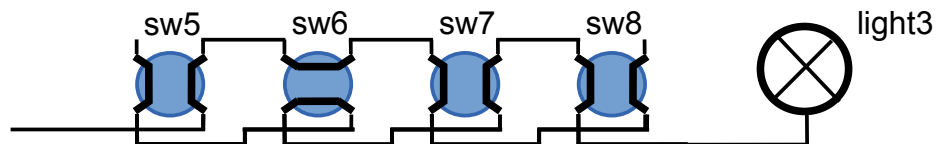
The expression above is equivalent to a logical *exclusive or*, which expresses the above functionality more simply as follows:

```
light2 = sw1 ^ sw2;    // sw1 or sw2 but not both
```

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.

1.8 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);
```

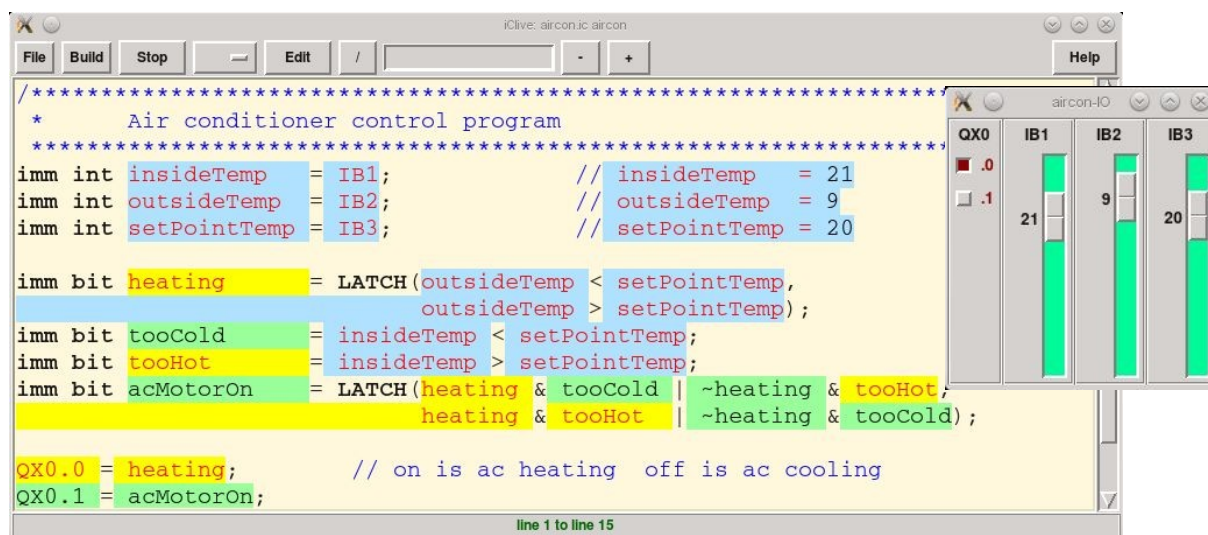
This function block is the simplest flip flop or memory element. When **set** is **1** and **reset** is **0** the output of **LATCH** goes to **1**; when **reset** is **1** and **set** is **0** the output of **LATCH** goes to **0**. **LATCH** remembers its previous state when **set** and **reset** are 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 **0** or **LO** for cooling mode and **1** or **HI** for heating, and **QX0.1**, which turns the motor of the compressor on and off.

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. **~name** is an inverting alias of **name**. This can be used to advantage to provide better visual meaning by adding an inverting alias **cooling = ~heating** 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.

Exercise 1-9. 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.

Exercise 1-10. 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.

1.9 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;    // really bad ERROR
```

When compiling, this statement produces the following error message:

```
*** 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 reason 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. In C code it acts just like a global variable. 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 initialised in iC code

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

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


1.10 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 at compile time 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 block¹ 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)
{
    declarations
    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](#)).

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

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 **setPointTemp**².

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 that 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.

1.11 Function Block Arguments

Since *iC* function blocks are cloned when used, each real (as opposed to formal) 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 yet. It must 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.

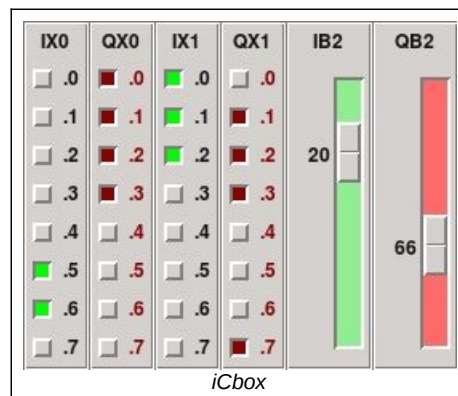
Inputs start with the letter **I**, outputs with the letter **Q**. These are followed by a second letter which defines the type of the input or output. **X** defines unsigned bytes of 8 single bit I/O variables. **B** defines unsigned numeric byte I/O variables, **W** signed 16 bit word I/O and **L** 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.



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 numeric 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; }
```

Exercise 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 for I/O and not formal for function parameters 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 numeric 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 numeric 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*, *C* and *iCa* keywords, *iC* pragmas, *iC* built in function names and names starting with “iC”, which are used by *iC* internally.

3.1.1 *iC* Keywords

assign	bit	clock	const
else	extern	if	imm
immC	int	return	sizeof
switch	this	timer	void

3.1.2 additional C Keywords

asm	auto	break	case
char	continue	default	do
double	enum	float	for
fortran	goto	long	register
short	signed	static	struct
typedef	union	unsigned	volatile
while			

3.1.3 *iC* Pragmas

use	no	
alias	list	strict

3.1.4 *iC* built-in Function Blocks

CHANGE	CLOCK	D	DLATCH
DR	DS	DSR	FALL
FORCE	iClock	JK	LATCH
RISE	SH	SHR	SHSR
SR	SRR	SRT	SRX
ST	TIMER	TIMER1	

3.1.5 *iCa* Keywords

ELSE	ELSIF	FOR	IF
------	-------	-----	----

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 <i>iC</i> code and mainly used in <i>immediate</i> logical expressions
imm int	is a variable whose size is the native size of a C signed int variable assigned in <i>iC</i> code and mainly used in <i>immediate</i> 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.

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

The other two *immediate* types 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.

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 C functions, which contain at least one *immediate* value variable or a function block call. All *immediate* expressions may contain constants, although they are fairly useless and not common in logical expressions. **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 of the *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 Operators in *iC* expressions

Most operators available in C may be used in *immediate* expressions. The precedence of the operators is the same as in C. Some C operators are not valid for *immediate* expressions, because the semantics in *iC* are different. These are the increment and decrement operators ++ and --, as well as assignment expressions += -= *= etc. Structure and pointer operators -> .(dot) &(address of) and *(pointer dereference) are also not allowed. These restrictions do not apply to embedded C code in literal blocks and *immediate if else* or *switch* statements, which will be introduced later.

Array variables and index expressions using [] are available with the Array extensions of the language either as **immC** Arrays or using **imm** variables using the pre-compiler **immac** (called automatically). See [section 7](#).

3.4.1 Arithmetic and Relational Operators

The binary arithmetic operators + - * /, the modulo operator %, as well as unary - and + operate on integer numeric values, usually of type **imm int**, and yield numeric results of type **imm int**. The same applies to the shift operators << and >>. If one or both of the operands used with one of these operators is type **imm bit**, automatic type conversion takes place. Values of type **imm bit** are converted to the **int** values 0 or 1 corresponding to the values of the **bit**. The relational and equality operators <, <=, >, >=, ==, != and the unary **not** operator ! also have numeric operands, but these operators yield **imm bit** results by default.

Immediate arithmetic, relational and bitwise integer expressions with numeric operands may contain constants, as well as *immediate* operands.

3.4.2 Bitwise integer Operators

If both operands of the binary operators &, |, ^ or the single operand of operator ~ are numeric values of type **imm int** or constants, these operators carry out bitwise manipulation on their integer operands – just like in C. The result is an **imm int** numeric value.

3.4.3 Bit Operators

If one or both of the operands of the binary operators &, |, ^ or the single operand of operator ~ are of type **imm bit**, these operators carry out the bit manipulation operations **and**, **or**, **exclusive-or** and **not** on **imm bit** objects. The result is an **imm bit**. Any operands of type **imm int** are converted to **imm bit**. The numeric value 0 converts to 0 (**LO**), any other numeric value converts to 1 (**HI**). The bit operators are used frequently in *immediate C*, since bit manipulation is very common in event driven systems – more so than in algorithmic programs written in conventional languages like C, which does not even provide a type *bit*. Such logical bit expressions in *immediate C* may not contain any *non-immediate* variables. Constants are allowed, although they do not make much sense. They either do not change a variable e.g. **a & 1 === a**; **b | 0 === b** or they produce another constant e.g. **c & 0 === 0**; **d | 1 === 1** and **~1 === 0**.

3.4.4 Logical Operators

The logical connectives **&&** and **||** are executed as arithmetic expressions, when one or both of the operands are of type **imm int**. Evaluation is from left to right, and evaluation stops when the truth or falsehood of the result is known – just like in C. The result is of type **imm bit** by default. The unary **not** operator **!**, operating on an **imm int** operand produces an **imm bit** result.

The operators **&&**, **||** and **!** with only **imm bit** operands are interpreted by the compiler exactly like the bit operators **&**, **|** and **~**, although there is really no point. Since evaluation does not stop when the result is known, the use of **&&** and **||** and **!** in expressions where all operands are **imm bit** is deprecated and causes a warning if **no strict** and an error if **use strict** (which is the default).

3.4.5 Conditional Operators

The operators **? :** implement conditional expressions, just like in C, which are evaluated as a whole in an arithmetic context. The conditional expression

expression_1 **?** expression_2 **:** expression_3

is a valid *immediate* arithmetic expression, which is triggered by a change in any *immediate* variable in any of the three sub-expressions.

3.5 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. 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.

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 executable code. The *alias* name on the left hand side is simply an alternative name at compile time 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.

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. Attempts at multiple *immediate* assignments are flagged as hard compile errors.

Expressions that occur in C code triggered by *immediate* conditional *if else* or *switch* statements or in C functions 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 in *iC* code 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.

3.6 Constants and Constant expressions

Apart from the bit constants **L0** and **HI**, only integer constants of type **int** can be used in *iC*. Constants in *iC* 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. The value of an integer constant can be specified just like in C as decimal e.g. **1275**, octal or hexadecimal. A leading **0** on an integer constant means octal; a leading **0x** or **0X** means hexadecimal. A character constant is an integer, written as one

character in single quotes such as **'a'**. Constants in logical bit expressions may be **0** or **1**, which are equivalent to **LO** or **HI**, which are special bit constants that do not change.

If an expression consists only of constants or **const int** parameters in a function block and no *immediate* variables it is a constant expression evaluated at compile time. Constant expressions may be assigned to value variables of type **imm int** or **imm bit** (after conversion to bit), which become aliases of the constant value of the expressions, executed at compile time, which obviously never change and are themselves constants. Constant expressions may be used to index members of **immC** arrays in *iC* code, to initialise **immC** variables following their declaration (similar to global initialisation in C) and in function block calls to satisfy **const int** formal parameters.

3.7 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 its 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.

3.8 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.

3.9 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 in C. 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 as parameters or inside the braces of a function block definition.

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 evaluated at compile time.

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 a 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: **extern** variables, which have not been subsequently defined in this module are assumed to be assigned in another module. Because of the single assignment rule these variables may not be assigned in this module. It causes a multiple assignment error.

3.10 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 bit bx, by, bz;           // declared immC variables
immC bit bb[] = { bx, by, bz }; // array of pre-declared variables
immC bit cc[3];               // immC bit cc0, cc1, cc2; generated
                                // and declared automatically

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

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 **immacc**, 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 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] };
extern immC int aa[3];
```

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 variable indexed references to **immC** array members are only possible in C code, the **sizeof** test in C code is appropriate. **sizeof** may be used to test index values to produce own error strategies.

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 cc[] – and has all the properties of a simple **immC** variable.

4 Literal Blocks, immediate Conditional Statements and Pragmas

An *immediate* conditional **if else** statement and an *immediate* **switch** statement are the only control constructs available in *iC*. The syntax of both statement types is similar to their C counterpart, except that braces around the C statements are mandatory. In particular an *else if* is not allowed, since the *if* after the *else* would have been part of the C statement controlled by the *else* part of the whole *immediate if* statement, which would be very confusing.

```

if (imm_bit_expression) { C_statement_1 }
if (imm_bit_expression) { C_statement_1 } else { C_statement_2 }
switch (imm_int_expression) { C_statement }

```

These are valid *immediate* statements when they occur in *iC* code. The controlling expression in each case must be an *immediate* expression. The controlling expressions in *immediate* conditional **if else** or **switch** statements are synchronized by a clock. The default clock is **iClock**, when no specific clock is coded (as in the above examples). Other clocks or timers may be specified as explained in [section 4](#). In all cases any change in the controlling *immediate* expression, synchronized by the controlling clock, triggers execution of the C statements. The actual execution of the C statements triggered by a conditional expression is deferred till after the clock cycle has completed. It is the first action of a new combinatorial scan after a clock cycle. This is necessary, because execution of the C code may modify **immC** variables, whose change must be allowed for in a combinatorial scan.

The *immediate* conditional **if else** and **switch** statements open the way to trigger the execution of short C fragments on particular events. These events are either rising or falling edges of bit values or changing numeric values. If more than a fragment of C code is involved, it is good practice to code this in a C function in a literal block, and to call that function in the *immediate* control statement. Long blocks of C code would make the purpose of those statements unclear. Depending on the time critical nature of the application, C code should not take too long to execute, because during the execution of such C fragments the processing of other immediate events is held up. Consider forking blocks of C code.

4.1 Literal blocks

Literal blocks are sections of C code enclosed in special braces **%{** and **%}**. They may occur before, between and after any *immediate* statement. Literal blocks are copied verbatim to the front of the generated C output code (without the special braces). Literal blocks are useful to declare any C variables, define macros and to declare and define auxiliary C functions to support the application. Since *iC* Version 3 any C pre-processor commands such as **#include**, **#define** or **#ifdef** etc. in a literal block are written in standard C form.

```

%{
    #include <math.h>    /* standard C-pre-processor syntax */
    int x, y, z;         /* declarations in a literal block */
    int abs(int);        /* C function declaration */
%}

```

Literal blocks and their embedded C pre-processor commands are resolved during C compilation, which follows the *iC* compilation. Pre-processor commands for the *iC* sections of code are **%include**, **%define** or **%ifdef** etc. These are resolved before the *iC* compilation.

The run-time system will call the function **iCbegin()** when an *iC* application is started before any *immediate* processing. This function can be provided by the user in a literal block. If it is not provided, a nearly empty function **iCbegin()** returning 0 is provided by the system. User implementations should return 1. Uses of **iCbegin()** are to initialise **immC** variables and additional –help output. It may even contain a **fork()** call to spawn a child process, which will run in parallel with normal *immediate* processing. This opens up the way to build mixed applications using conventional multi-process or multi-threaded control strategies in parallel with *immediate* C code, which leaves a lot of CPU time to do other things.

The complementary function **iCend()** is called by the run-time system when an *iC* application is terminated externally (*iC* applications never terminate by themselves, unless **iC_quit()** is called in embedded C code). **iCend()** could be used to free memory allocated with *malloc* or *new*.

```

%{
    int iCbegin() { ...; return 1; } /* optional C initialisation */
    int iCend()   { ...; return 1; } /* optional C termination */
%}

```

If the code in literal blocks, or code in C blocks controlled by an *immediate if else* or *switch*, is specifically C++ code, then the generated code must be compiled by a C++ compiler. The Code generated from the *iC* statements is pure C code.

4.2 immediate conditional if else statement

For an *immediate if* and optionally *else* statement, the controlling expression is a clocked *immediate bit* expression in parentheses. If not, it is converted from **int** to **bit** automatically.

```
if (imm_bit_expression) { C_statement_1 } else { C_statement_2 }
```

A **LO** to **HI** transition or rising edge causes C_statement_1 to be executed. A **HI** to **LO** transition or falling edge causes C_statement_2 to be executed (if an *else* is coded). The C_statements are embedded C compound statements, **not** *immediate* statements.

```
%{
int a, b, c;                /* C declarations in a literal block */
void reset(void);          /* C function declaration */
%}

imm bit sw1, sw2, sw3;      // immediate declarations
imm clock cl;              // use cl rather than iClock

if (sw1 & sw2 | sw3, cl) { /* imm controlling expression */
    a = 1; b = 12; c = -2; /* C code executed on rising edge */
} else {
    reset();                /* C code executed on falling edge */
}
```

4.3 immediate switch statement

For the *immediate switch* statement, the controlling expression is a clocked *immediate int* expression in parentheses. If not, it is converted from **bit** to **int** automatically (rare).

```
switch (imm_int_expression) { C_statement }
```

The C_statement is an embedded compound statement, which has the usual form of a C switch statement with case labels. Any change in the controlling expression triggers the switch statement. The value of that expression after the change is applied to the switch and the selected case is executed.

```
%{ enum Fuzzy { OFF, DIM, MEDIUM, BRIGHT }; %}    // literal
block
switch (brightness, cl) {
    case OFF:    lightVoltage(0);    break;
    case DIM:    lightVoltage(10);   break;
    case MEDIUM: lightVoltage(18);   break;
    case BRIGHT: lightVoltage(24);   break;
    default:    lightVoltage(24);   break;
} // end of immediate switch statement
```

4.4 Pragmas

Pragmas affect the compilation phase of an *iC* program. Pragmas are introduced by the keywords **use** and **no**.

```
use   turns a pragma option on
no   turns it off
```

Currently three pragmas are implemented in *immediate C*: **alias**, **strict** and **list**.

```
use alias;           // equivalent to -A command line option
no alias;           // turn alias option off

use strict;         // equivalent to -S command line option
                     // default since iC Version 2
no strict;          // turn strict option off (deprecated)

use list;           // restore listing output from the next line - default
no list;            // suppress listing output from the next line
```

1. The **alias** pragma or -A command line option forces the compiler to generate a node for each alias in the generated C code (default is to generate no node). This is needed in two circumstances:
 - It is required, if an *iC* source refers to an alias in another *iC* source by an **extern** reference. Since all references to aliases are normally removed from the compiled code, the C object modules, which are generated from such code could not be linked. With the **use alias** option, the code can be linked and the remaining aliases are resolved at start up.
 - The **use alias** option is also useful for debugging. Only when it is set, are alias names displayed as active words by *iClive*.
2. The **strict** pragma or -S command line option (which is the default since Version 2) forces the compiler to expect a declaration of all *immediate* variables, before it is used or assigned in an *iC* statement. With **no strict** (deprecated), an **imm bit** variable is assumed for any undeclared value variable. Similarly an assignment to an undeclared name from a **CLOCK()** or **TIMER()** function call produces a default **imm clock** or **imm timer** variable. Such laxness is OK for small single source projects, but can lead to problems with larger projects. I had a case in a large project, where I had declared a number of **imm int** variables and mistyped one of them, so the correct name was not declared. This name was then assigned - but converted to **imm bit** and then back to **imm int** when used, leading to incorrect arithmetic. As noted earlier, C functions and macros should be declared **extern** with their correct parameter ramp and return value. When **strict** is active, error messages are output if an undeclared C function or macro is called in an *immediate* C expression.
3. The **no list** pragma suppresses listing output from the next line until a **use list** statement starts listing output again. This is mainly used to suppress listings of function block definitions in %include files, which may be regarded as clutter. Typical use:

```
no list;      // %include "adconvert.ih"
%include "adconvert.ih"
use list;
```

Listing output is the **no list** line only. The comment is recommended, telling what will not have been listed, which is the whole of the file `adconvert.ih` and the **use list** line.

Several options may be turned on or off together in one pragma call: e.g. **use alias strict**;

The scope of *iC* pragmas is a file. If a pragma is enabled in one file it carries over to an included *iC* header file. If on the other hand a pragma is changed in a header file, it reverts to its previous value in the *iC* file after the %include statement, which includes the header file. This makes sure that sloppy **no strict** *iC* programs, which include a header file, which uses "**strict**" syntax, will not report errors, because they do not follow the "**strict**" syntax. This scope feature can only be used successfully with the **strict** and **list** pragmas, since **use alias** only comes into effect during C code generation – at this point the complete source has been parsed. This means **use alias** should definitely be used once in *iC* programs, which consists of several parts with extern references between them. Other single source *iC* programs can **use alias**, which generates slightly larger code, but which can be debugged without recompiling with the -A flag

4.5 Comments

C style comments `/* ... */` can be used anywhere between tokens of *iC* programs. C++ style comments may be used at the end of *iC* lines. `// ...`

Some older C compilers do not support C++ style comments, so their use in literal blocks and C statement blocks controlled by **if else** or **switch** statements may lead to portability problems.

5 *immediate* Function Blocks

Functions are commonly called function blocks in the PLC world, because they act more like functional blocks or templates rather than functions in the instruction flow sense, where a function evaluates a sequence of instructions whenever it is 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 function or function block call. Only standard IEC-1131 I/O variables may be used in a Function Block without being declared, although they may only be used as inputs, since any assignment to an output variable such as **QX0.0** inside a Function Block would lead to a multiple assignment, once the Function Block is used more than once. Another way to look at an *immediate* Function Block is like a higher level or LSI integrated circuit, which has connections into the system and provides a certain complex functionality with many internal components and connections.

5.1 *immediate* Function Block Definition

All *immediate* Function Blocks, except built-in Function Blocks, must be defined before they are used. Since the definition of a Function Block does not itself generate any C Code on compilation it can be and usually is defined with its code body in a header file, if multiple source files are used for a project. For small projects with a single source file Function Blocks can be defined at the start of the source file.

immediate Function Block definitions are very similar to C functions, although there are significant differences in detail. The definition of an *immediate* Function Block consists of a return value type, a Function Block name, a comma separated parameter list in parentheses and a function body in curly braces, e.g.

```
imm bit fall(bit f, clock c) { this = RISE(~f, c); }
```

The return value may be one of 5 types:

```
imm bit
imm int
imm clock
imm timer
imm void           // which means no value is returned
```

The **imm** modifier is mandatory for the return type – it identifies an immediate Function Block Definition syntactically. The Function Block name can be any valid name starting with a letter followed by any number of alphanumeric characters or underscores. A leading underscore is possible, but should be avoided. The name must be distinct from all other immediate variable names in a project. The individual formal parameters in the parameter list must be of the following types:

```
imm bit           // or simply bit           // imm is implied
imm int           // or           int
imm clock        // or           clock
imm timer        // or           timer
const int        // call parameter must be a constant expression
```

It is also possible to specify **immC bit** or **immC int** arrays in the formal parameter list as follows:

```
immC bit bb[10] // or           bit bb[10] // immC is implied
immC int aa[]   // or           int aa[]   // size is optional
```

The **imm** modifier (or **immC** for arrays) is optional for parameters in a parameter list. The variable declared is nevertheless immediate. 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 determined in the Function Block is to be assigned. In this case the type of the parameter must be preceded by the keyword **assign**.

```
assign imm bit // or           assign bit
```

Array parameters cannot be assigned. If the size in square brackets of an array parameter is left out, that position can be filled by an array of any size – there is one drawback – no indexed array references to that array can be made in the *iC* code of the Function Block.

The body of a Function Block is one or more immediate statements defining the functionality of the block encoded in curly braces. Immediate variables internal to the function must be declared before use in the Function Block. Parameter names and internal variable names are in a separate name

space for each function block, which is also separate from the global name space. If a Function Block is not **imm void** the body must contain a **return** statement. The semantics of the **return** statement is the assignment to the variable to which the Function Block is assigned, when it is called. This variable, which is identified by the keyword **this**, may be used in other expressions inside the Function Block. The preferred way to write the **return** statements is:

```
this = some + immediate + expression; // preferred return syntax
```

The usual C syntax may also be used, but does not make the action as clear:

```
return some + immediate + expression; // deprecated earlier syntax
```

The **return** statement need not be the last statement in the Function Block definition – its position does not influence when it is executed – that is controlled purely by changes in the values of the variables making up the **return** statement – something which holds for all **immediate** statements. This situation is more clearly expressed by the assignment to **this**. An **imm void** Function Block has no **this** variable, may not contain a **return** statement and may not be assigned when called.

Each **assign** parameter must occur on the left side of an assignment statement in the Function Block. The values of **assign** parameters may be used inside the Function Block. Each variable declared inside the Function Block must also be assigned in the Function Block. Variables declared **extern** outside or inside the Function Block may not be assigned to inside the Function Block. As is the case with I/O variables (which are implicitly extern). **extern** variables may only be used as values inside the Function Block. They may not be declared again as local inside the Function Block. Variables declared **extern** in a Function Block may be declared after the definition of the Function Block in the *iC* code following the definition. This declares that the variable will be assigned in this module. A variable with the same name as an **extern** variable may be declared locally in another Function Block, but it is a different formal variable local to that Function Block.

All **immediate** statement types – assignments, *if else*, *switch*, Built-in Functions and other user defined Function Block calls may be used in Function Block definitions. Function Blocks may be nested to any depth as long as Function Blocks are used, which have previously been defined. This implies that Function Blocks cannot be called recursively, either directly or indirectly. Function Blocks may be very simple one line definitions or complex systems with hundreds of parameters. Several examples follow:

The SRX flip-flop is built into the compiler, but defined in just this way during initialisation of the compiler. Since Version 2 of the compiler, all built in functions are defined as Function Blocks.

```
/* SRX flip-flop defined as a function block */
imm bit srx(imm bit set, imm clock scl,
             imm bit res, imm clock rcl)
{
    this = SR(set & ~res, scl, res & ~set, rcl);
}
```

The CountClk function adds 'increment' to 'this' for every occurrence of 'clk':

```
imm int CountClk(imm clock clk, imm int increment)
{
    this = SH(this + increment, clk);
}
```

The CountBit function adds 'increment' to 'this' for every rising edge of 'step':

```
imm int CountBit(imm bit step, imm int increment)
{
    this = CountClk(CLOCK(step), increment); // nested call
}
```

The Count function adds 1 to 'this' for every rising edge of 'step':

```
imm int Count(imm bit step)
{
    this = CountBit(step, 1); // nested twice
}
```

The SelectClk function selects either a 100 ms or a 1 second clock with variable 'second':

```
imm clock SelectClk(imm bit second)
{
    this = CLOCK(T100ms & ~second | T1sec & second );
}
```

The following function block ADConvert assigns the conversion of int val to 8 assign bit variables b0 to b7 passed as parameters (imm is implied for value and assign parameters).

```

/* Analog to digital conversion of a byte value */
imm void ADConvert(int val,           // input parameter
                    assign bit b0,   // output assign parameters
                    assign bit b1,
                    assign bit b2,
                    assign bit b3,
                    assign bit b4,
                    assign bit b5,
                    assign bit b6,
                    assign bit b7,
)
{
    b0 = val & (1 << 0);           // assignments to outputs
    b1 = val & (1 << 1);
    b2 = val & (1 << 2);
    b3 = val & (1 << 3);
    b4 = val & (1 << 4);
    b5 = val & (1 << 5);
    b6 = val & (1 << 6);
    b7 = val & (1 << 7);
}

```

Note: the parameter list may have a trailing comma before the closing parentheses. This is generally the case for comma separated lists in *iC* and makes it easier to edit the lists and copy parameters when written vertically, which is useful for large parameter lists.

The *iC* compiler builds a template of the Function Block, replacing each parameter and internally declared variable by the name of the Function Block followed by '@' and the formal parameter or declared variable name. This strategy ensures a private name space for each Function Block. When called, the template is copied, with each formal parameter replaced by its real parameter and internally declared variables replaced by the formal name with the '@' replaced by an underscore '_' followed by an instance number and another underscore. The instance number scheme ensures that there is no clash of compiler generated variable names (even for separately compiled modules).

5.2 immediate Function Block Call

An *immediate* Function Block is called in a similar fashion to a C function, again with some significant differences. In practice *immediate* Function Blocks are not called. When the compiler encounters a Function Block call, the pre-compiled Function Block, which is a template, is cloned, with all calling parameters and internal variables replacing the formal parameters and formal internal variables in the template. The resulting real network of individual nodes associated with the call will then be used at run-time like the network of nodes generated from all other immediate statements.

If an **imm void** function is encountered it looks like a subroutine call:

```

ADConvert(IB1,
          QX0.0, QX0.1, QX0.2, QX0.3,
          QX0.4, QX0.5, QX0.6, QX0.7,
);

```

This statement will assign bits 0 to 7 of **IB1** to **QX0.0** to **QX0.7** whenever **IB1** changes.

A Function Block with a return value 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 a parameter list. An **imm bit** Function Block may be used as an **imm int** value and vice versa – appropriate conversion takes place. **imm clock** and **imm timer** Function Blocks can either be assigned to correctly declared **clock** or **timer** variables or else used as a **clock** or **timer** in a parameter list.

```

/* count every rise of IX1.0 */
imm int count = Count(IX1.0);

/* selects 1 sec when IX1.7 is on else 100 ms */
imm clock clk = SelectClk(IX1.7);

```

Real parameters of type **imm int** and **imm bit** may be mismatched with their formal parameter types – value and assign parameters in the call will be forced to their formal type. **assign** parameters of type **imm clock** and **imm timer** must match – so must a parameter of type **imm timer**. Real **immC** Array parameters are only the name of a previously declared **immC** Array of the same type as the formal parameter. The size must also match unless the formal parameter did not specify a size.

The handling of formal **imm clock** parameters is more complicated, allowing the use of default clocks. Positions for formal **imm clock** parameters which do not immediately follow another formal clock parameter are handled as follows:

1. The position may be filled by a real **imm clock** parameter.
2. The position may be filled by a real **imm timer** parameter followed by an optional **imm int** delay (if delay is left out it will be set to 1).
3. The position may be left out altogether, in which case the next clock or timer parameter (including its delay) on the right, separated by at least one non clock formal parameter, will be replicated for the position. If there is no real clock parameter following on the right, **iClock** will be used.

On the other hand the second of two consecutive formal clock parameters must be matched by a real clock or by a real timer parameter optionally followed by an **imm int** delay parameter. If the first of the formal clock pair is not matched by a real clock or timer parameter, it and all unmatched formal clock parameters to the left will be set to **iClock**.

These rules for optional clock parameters are the same as for the clocked built-in functions **D**, **SR**, **SRR**, **SH**, **SHR**, **SHSR**, **RISE**, **CHANGE**, **CLOCK**, **TIMER** and **TIMER1** as well as for the **if** and **switch** statements.

Real **timer** parameters for formal **timer** parameters cannot be extended by a delay – the delay used is determined in the Function Block with delay(s) associated with formal **timer** parameter(s) in the code of the Function Block.

Formal parameters of type **const int** must be filled by a constant value or constant expression when called. **const int** parameters can be used in Function Blocks as initialiser values for **immC** variables and index values for **immC** array members, which must be constants. They can also be used as timer delay values and generally in immediate arithmetic expressions.

The following are calls of the **SRX()** Function Block with two formal clock parameters – one each for set and reset and the **ST()** function block with two consecutive formal clock parameters – one optional for set and the second a non optional delayed self reset timer or clock.

```

imm clock clk0 = CLOCK(IX1.0), clk1 = CLOCK(IX1.1);
imm timer t    = TIMER(IX1.2);
imm bit s, r;
imm bit m1 = SRX(s, clk0, r, clk1);    // uses individual clocks
imm bit m2 = SRX(s, t, 3, r, t, 5);    // individual timer delays
imm bit m3 = SRX(s, r, clk1);          // same clock for s and r
imm bit m4 = SRX(s, r, t, 5);          // one timer for s and r
imm bit m5 = SRX(s, clk0, r)           // default iClock for r
imm bit m6 = SRX(s, iClock, r, clk1);  // must specify iClock here
imm bit m7 = SRX(s, r);                // default iClock for both

imm bit m8 = ST(s, clk0, t, 5);         // t is not optional
                                           // because it fills 2nd formal clock
imm bit m9 = ST(s, t, 5);              // iClock for s - t is not optional

```

6 Built-in Function Blocks

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. Clocking overcomes the negative effects of race conditions.

6.1 Unlocked memory elements

There are two unlocked memory elements in *iC*, the **FORCE** function and the **LATCH** function, which was already used in earlier chapters.

6.1.1 Unlocked flip-flop or LATCH

The unlocked 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.

6.1.2 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.

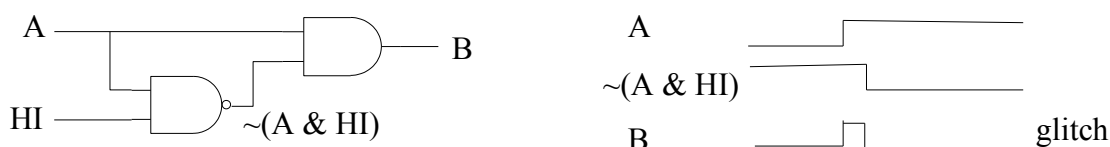
The **LATCH** function is generated by the more fundamental **FORCE** function as follows:


```
imm bit LATCH(bit set, bit reset)
{ this = FORCE(this, set, reset) }
```

Feedback of its own output 'this' 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.

6.2 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 *immediate C*, 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 gate on the other input. In *iC* this can be tested with the statement $B = A \& \sim(A \& HI)$; (The 2nd input HI on the inverting AND gate is necessary, because in *iC* a simple inverter $\sim A$ would be an alias, which would be $B = A \& \sim A$. This expression is recognised by the *iC* compiler as an error, since the output is always LO). In theory the expression $A \& \sim(A \& HI)$ 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 B will also be HI. Once the LO signal $\sim A$ arrives through the inverting AND gate, the output of B becomes a correct LO. But for a short period the HI glitch on B may trigger memory elements like a LATCH if B 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 glitches³. 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*. After all combinatorial changes induced by one or more input events have been computed, a clock phase is started, which usually changes some logic values of slave outputs of clocked functions. This starts a new run of combinatorial actions, which is again followed by a clock phase. This sequence is continued 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 in *iC* 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 in *iC* take fractions of microseconds to execute on modern computers. Times between external events

³I 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.

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 %.

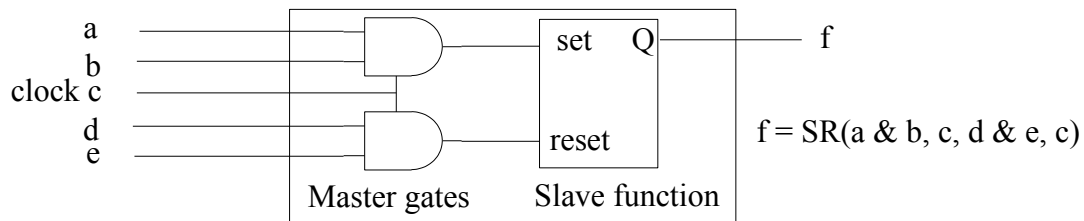
As pointed out earlier even software using multi-threaded or distributed programs can suffer from race conditions. This happens when two threads read and write the same shared data simultaneously. Mechanisms to control such sharing are Mutexes and Semaphores.

Controlling race conditions, glitches and synchronisation of multiple outputs in *iC* 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.

6.3 Clocked digital 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.



The Master gates of a clocked function do not act immediately on the Slave function, but instead are blocked by a clock. When the clock fires, the inputs to the Master gates are blocked and then the outputs of the Master 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 change of any Slave output can cause a change in any combinatorial Master gate expression to affect a Slave output, either directly or indirectly through other gates. The result is that 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).

6.3.1 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	reset	SR(set, sc, reset, rc)
S^n	R^n	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** monoflop 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 Q_{n+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.

6.3.2 Clocked JK flip-flop

Instead **JK** flip-flops were popular in integrated 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	reset	JK(set, sc, reset, rc)
J^n	K^n	Q^{n+1}
0	0	Q^n
1	0	1
0	1	0
1	1	$\sim Q^n$

6.3.3 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	reset	SRX(set, sc, reset, rc)
S^n	R^n	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.

6.3.4 Mono-Flop ST(set, 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** monoflop, 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.

6.3.5 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.

6.3.6 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);
```

6.3.7 Clocked LATCH function DLATCH

A final digital memory element in *iC* is a clocked LATCH, which is implemented as an unlocked 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. It is implemented as follows:

```
imm bit DLATCH(bit set, bit reset, clock c)
{ this = D(FORCE(this, 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 if the logic of the set and reset inputs is very simple and is guaranteed not to have glitches. Unlike the other clocked memory elements, **DLATCH** may not have separate clocks on its set and reset inputs. To properly synchronise a number of memory elements the same clock must be used for all inputs anyway.

6.4 Edge detector functions RISE, FALL and CHANGE

It is often useful to generate a pulse on the rising and/or falling edge of a logical signal or on a change of a 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**.

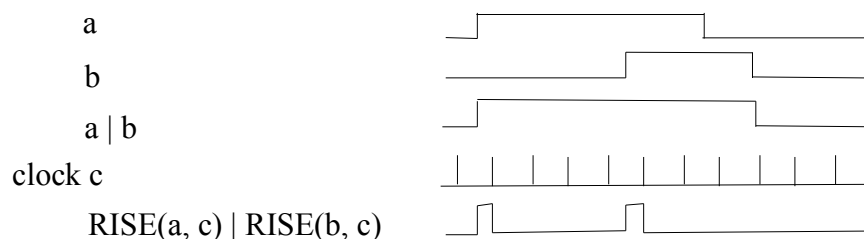
```

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 rising input 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.

6.5 Clocked analog memory element

iC has one clocked analog memory element.

6.5.1 Clocked Sample and Hold function SH

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 inputs 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);

```

6.5.2 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);

```

6.6 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 appear in expressions with logic or numeric values. Any attempt to do so generates a hard error message. Clocks may only be used as clock parameters in Function Block calls. 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 Function Blocks 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 Function Blocks, 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**

6.6.1 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 must 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_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 conventional PLC’s 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.

6.6.2 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:

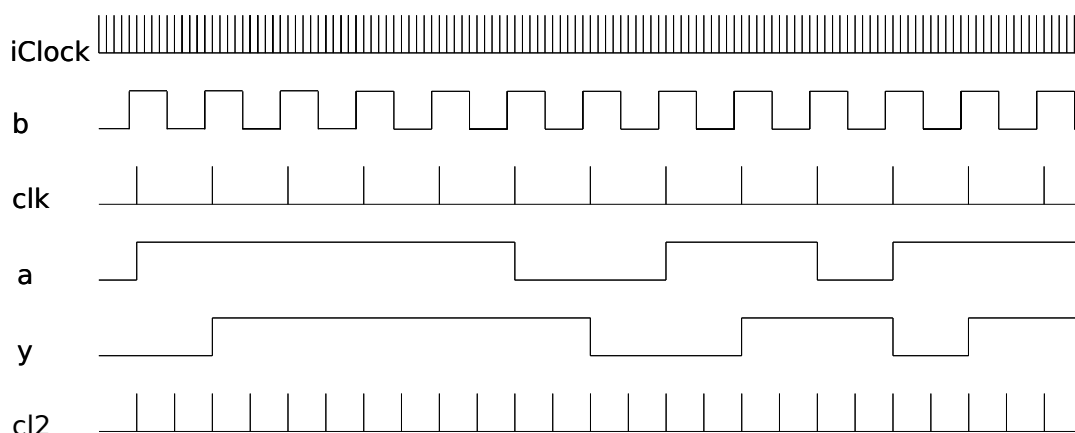
```
imm clock clk = CLOCK(b); // ‘clk’ on the rising edge of b
                        // clocked by next ‘iClock’(default)
imm bit y = D(a, clk); // D flip-flop clocked by ‘clk’
```

```

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.

6.6.3 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 clocked 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.

```

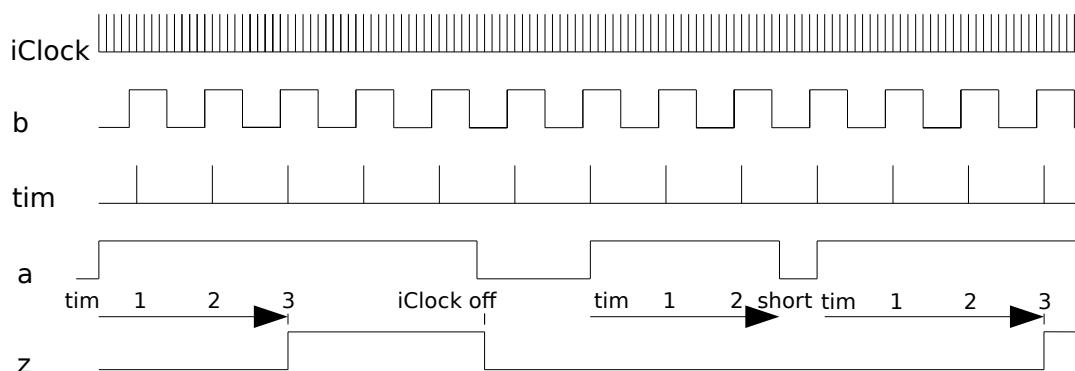
imm timer TIMER(bit in, clock c);    or
imm timer TIMER(bit in1, clock c1, bit in2, clock c2);

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:



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.

6.6.4 *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.

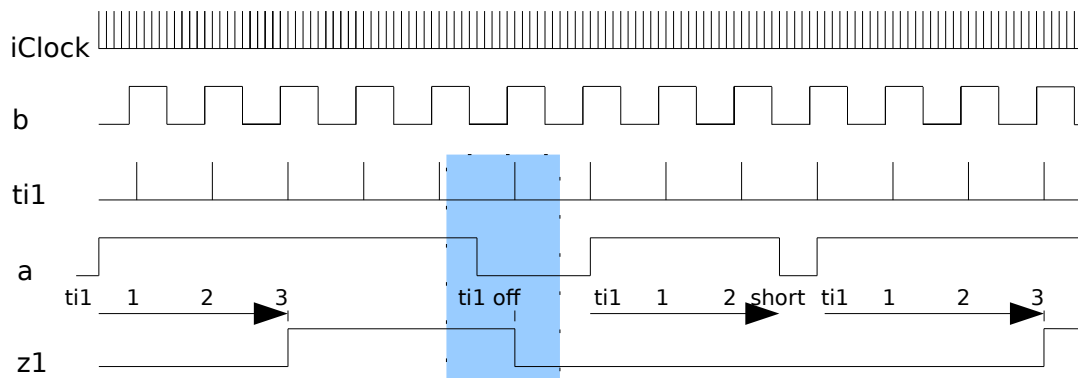
```

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.

6.7 *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 or T10ms    // 10 ms, 5 ms on, 5 ms off
TX0.4 or T100ms   // 100 ms, 50 ms on, 50 ms off
TX0.5 or T1sec    // 1 second, 500 ms on, 500 ms off
TX0.6 or T10sec   // 10 seconds, 5 seconds on, 5 seconds off
TX0.7 or T1min    // 1 minute, 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(T100ms);           // clock every 100 ms
imm timer tim500ms = TIMER(T1sec, ~T1sec); // timer 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

```


The aliases **T10ms**, **T100ms**, **T1sec**, **T10sec** and **T1min** for the IEC-1131 names **TX0.3** – **TX0.7** as well as **E0I**, **STDIN**, for **TX0.0**, **TX0.1** are compiler generated when those words are used in expressions. (**T1ms** for **TX0.2** has been reserved for implementations with a higher speed real time operating system like RTLinux). **L0** is a compiler generated **imm bit** variable with no input and a constant bit 0 output. **HI** is generated as the alias of **~L0** with a constant bit 1 output. They are all keywords in the *iC* language and may not be declared a second time. Bit constants **L0** and **HI** are provided to fill unneeded **bit** call parameters required by a function block.

The rising edge of **E0I** (end of initialisation) is guaranteed to be the first input to the system and can be used for initializing user constructs. It starts **L0** and then is **HI** for the remainder of the program (forever as far as applications are concerned)

Keyboard or other input received from standard input (stdin) causes an interrupt every time a line terminated by a Newline has been received. This interrupt causes **STDIN** to pulse **HI** for one **iClock** period. The data from stdin is available in the global C array `char iC_stdinBuf[]`.

6.8 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.

6.8.1 A divide by 10 Moebius ring counter

This is a very simple counter using 5 SR flip flops and 10 two input AND gates to decode the 10 outputs. It was popular in the first large control computer I built in the mid 60's, when only clocked SR flip flops, inverters and logic using germanium transistors and diodes were available. (moebiusSR.ic)

```

imm clock c0 = CLOCK(IX0.0); // input to be counted
imm bit m0, m1, m2, m3, m4;

m0 = SR(~m4, m4, c0);
m1 = SR( m0, ~m0, c0);
m2 = SR( m1, ~m1, c0);
m3 = SR( m2, ~m2, c0);
m4 = SR( m3, ~m3, c0);

QX0.0 = m0 & ~m1;           // 0
QX0.1 = m1 & ~m2;           // 1
QX0.2 = m2 & ~m3;           // 2
QX0.3 = m3 & ~m4;           // 3
QX0.4 = m4 & m0;            // 4
QX0.5 = ~m0 & m1;           // 5
QX0.6 = ~m1 & m2;           // 6
QX0.7 = ~m2 & m3;           // 7
QX1.0 = ~m3 & m4;           // 8
QX1.1 = ~m4 & ~m0;          // 9

```

The actual Moebius sequence is much simpler to generate and easier to visualise with D flip flops.

```

m0 = D(~m4, c0);             // (moebiusD.ic)
m1 = D( m0, c0);
m2 = D( m1, c0);
m3 = D( m2, c0);
m4 = D( m3, c0);

```

This counter is much simpler than the full binary counter, which follows.

6.8.2 A divide by 16 binary counter

This counter uses only 4 flip flops but many more gates. It would be even more complicated for a divide by 10 counter, which is left as an exercise. (binarySR.ic)

```

imm clock c0 = CLOCK(IX0.0); // input to be counted

```

```

imm bit m0 = SR(~m0, m0, c0);
imm bit m1 = SR( m0 & ~m1, m0 & m1, c0);
imm bit m2 = SR( m0 & m1 & ~m2, m0 & m1 & m2, c0);
imm bit m3 = SR( m0 & m1 & m2 & ~m3, m0 & m1 & m2 & m3, c0);

QX0.0 = ~m0 & ~m1 & ~m2 & ~m3; // 0
QX0.1 = m0 & ~m1 & ~m2 & ~m3; // 1
QX0.2 = ~m0 & m1 & ~m2 & ~m3; // 2
QX0.3 = m0 & m1 & ~m2 & ~m3; // 3
QX0.4 = ~m0 & ~m1 & m2 & ~m3; // 4
QX0.5 = m0 & ~m1 & m2 & ~m3; // 5
QX0.6 = ~m0 & m1 & m2 & ~m3; // 6
QX0.7 = m0 & m1 & m2 & ~m3; // 7
QX1.0 = ~m0 & ~m1 & ~m2 & m3; // 8
QX1.1 = m0 & ~m1 & ~m2 & m3; // 9
QX1.2 = ~m0 & m1 & ~m2 & m3; // A
QX1.3 = m0 & m1 & ~m2 & m3; // B
QX1.4 = ~m0 & ~m1 & m2 & m3; // C
QX1.5 = m0 & ~m1 & m2 & m3; // D
QX1.6 = ~m0 & m1 & m2 & m3; // E
QX1.7 = m0 & m1 & m2 & m3; // F

```

The binary sequence is quite difficult to generate using D flip flops. Here is an implementation developed using.

```

imm bit m0 = D(~m0, c0); // (binaryD.ic)
imm bit m1 = D( m1 ^ m0, c0);
imm bit m2 = D( m2 & ~m1 | m1 & (m2 ^ m0), c0);
imm bit m3 = D( m3 & ~m2 | m3 & ~m1 | m2 & m1 & (m3 ^ m0), c0);

```

All these counters are not very useful in actual control systems. They simply show how simple state sequences can be generated using flip flops.

6.8.3 A state machine showing running lights

Another state machine, which is often shown at trade fairs is a set of 8 running lights which go on and off up and down in sequence. It is an effective display, both on hardware lights for physical I/O cards and their simulation with iCbox. (bar.ic)

```

imm timer t = TIMER(T100ms); // 100 ms time base
imm bit b0 = D(~b0, t, IB1); // IB1 changes clock rate
imm clock c0 = CLOCK(b0);

imm bit m0, m1, m2, m3, m4, m5, m6, m7, m8;

m0 = SR(~m8, m8 & ~m1, c0);
m1 = SR(~m8 & m0, m8 & ~m2, c0);
m2 = SR(~m8 & m1, m8 & ~m3, c0);
m3 = SR(~m8 & m2, m8 & ~m4, c0);
m4 = SR(~m8 & m3, m8 & ~m5, c0);
m5 = SR(~m8 & m4, m8 & ~m6, c0);
m6 = SR(~m8 & m5, m8 & ~m7, c0);
m7 = SR(~m8 & m6, m8, c0);
m8 = SR(~m8 & m7, m8 & ~m0, c0);

```

7 Arrays and the pre-compiler *immac*

Arrays in conventional instruction flow languages are a named collection (often of fixed length) of similar variables, which are accessed by an index expression, e.g. `a[5]`. Each such entity is an individual object, but in instruction flow languages the index is often a variable, which is manipulated in a loop and references to the individual indexed entities occur sequentially, as in the following C example:

```
for (n = 0; n < 4; n++) {           // plain C code
    a[n] = b[n] * c[n];
}
```

7.1 Immediate Arrays

In data flow languages like *immediate C*, loops at run-time are meaningless. Each *immediate* variable is an entity, which is controlled by one assignment statement. The variable changes, when a variable in the expression of the controlling statement changes and not when some loop runs. It is well to remember, that *immediate* variables and their controlling expressions are more like IC building blocks connected in a static network. In that sense *immediate* Arrays are like hardware registers containing a number of similar hardware objects, which act out their individual function inside the hardware IC register.

Such arrays may be defined in *immediate C*, but each entity acts individually at run-time, which means that an individual *immediate* object must be generated for each *immediate* array member.

7.2 Use of immediate Arrays

Arrays in conventional languages as well as in *immediate C* give programmers extra capabilities to express themselves. These fall into several distinct categories:

1. Arrays allow the writing of repeated similar statements as one statement – this saves a lot of writing, but could also be done without arrays.
2. Additionally arrays allow the parametrisation of the array length, both within the program source and in the command line of the compiler call, which is probably more important. For *immediate C*, this makes possible the writing of control programs in which the number of control elements or groups is variable and the actual number is not bound until compile time. This would not be possible without arrays in the language.
3. Arrays are also useful to select another variable in one indexing operation. If the index is itself a variable, this sort of operation can only be done in embedded C code in *immediate C* using **immC** variables whose changes can act back on normal *iC* code. To allow this sort of fast access, **immC** Arrays have been implemented in *iC* – they were introduced earlier in [section 3.9](#). Note: **immC** arrays are not part of the extended *iCa* language compiled by *immac*.
4. The definition of dynamic arrays, whose sizes change at run-time, is meaningless for a data flow language and is therefore not possible in *immediate C*.

An example of the usefulness of arrays in the language would be an *iC* program controlling lifts or elevators in a building. The number of floors varies from building to building – so do the number of parallel lifts, which may be required. With arrays, a single *iC* program can be written, which can be compiled for a different number of floors and a different number of parallel lifts as follows:

```
immac -P FLOORS=12 -P LIFTS=2 liftControl.ica
```

7.3 Implementation of immediate Arrays

Since each immediate array member is an individual immediate object at run time, it is important for debugging with *iClive* to be able to have a listing showing each individual array member – not just its collective form, e.g. `a[N]`. To achieve this, an *iC* program containing arrays is translated by the pre-compiler *immac* to *iC* code without arrays. This is a simple text operation in which macros are expanded, loops are unrolled and index expressions are evaluated.

The *iC* language with arrays has four additional language extensions:

1. C or Perl-style **FOR** loops, which define a loop variable and a range.
2. C or Perl-style **IF**, **ELSE** and **ELSE IF** statements (**ELSIF** is a synonym for **ELSE IF**)
3. Index expressions in square brackets, which allow the definition of array variables – usually in a **FOR** loop.

4. Macro definitions, which are processed directly by *immac*, which can be defined in two ways:

- in C-pre-processor style with **%%define** instead of **#define**, e.g.
%%define FLOORS 12
- in the command line, just like for a C compiler, but using -P instead of -D, e.g.
-P FLOORS=12

Macros will mostly be used inside the square brackets of an array variable or in the control line of a 'FOR loop', but they can be used anywhere in the *iC* code or in the definition of another **%%define** macro – macros may be nested. The above implies, that the *immac* pre-compiler could be used as a macro pre-processor for *iC* programs without any arrays at all.

iC programs containing the above four extensions are called *iCa* programs and should be written in a file with the extension .ica – the *immac* pre-compiler, written in Perl, translates an *iCa* program to an *iC* program with the extension .ic in which macros and 'FOR loops' are expanded and *immediate* array instances are converted to simple *immediate* variables. The following *iCa* snippet in file lift.ica

```
%%define FLOORS 4
FOR (N = 0; N < FLOORS; N++) {{
    imm bit liftTo[N] = up[N] | down[N];
}}
```

expands to the following *iC* file lift.ic when compiled by *immac*:

```
imm bit liftTo0 = up0 | down0;
imm bit liftTo1 = up1 | down1;
imm bit liftTo2 = up2 | down2;
imm bit liftTo3 = up3 | down3;
```

The 'FOR loop' is executed at compile time and generates repeated copies of the statement(s) in the compound statement controlled by the loop. This only makes sense, if there are elements in the loop statement(s), which are modified by index operations using the control variable of the 'FOR statement' – in the above example that is the variable **N**.

The translation of indices in square brackets is carried out in two steps:

1. The expression in square brackets is evaluated as a Perl integer expression.
2. The numeric value produced replaces the square brackets and the expression it contains.

In the above example the index expressions are simply the variable **N**. But the index expressions can be more complex. A feature of *iCa* indexing may seem strange at first, but it turns out to be very useful; the square bracketed index expression may be placed anywhere in a word, not only at the end of a word. It may even be placed on its own – in that case the expression is evaluated and becomes a suitably modified integer constant in an *iC* statement. The following example shows both:

```
FOR (N = 0; N < 7; N++) {{
    QB[N] = IB[N+1] * [N+2];
    QX[N/8].[N%8] = IX[N/8].[N%8] & IX[10+(N/8)].[N%8];    // out: [N]
}}
```

expands to :

```
QB0 = IB1 * 2;
QX0.0 = IX0.0 & IX10.0;    // out: 0
QB1 = IB2 * 3;
QX0.1 = IX0.1 & IX10.1;    // out: 1
QB2 = IB3 * 4;
QX0.2 = IX0.2 & IX10.2;    // out: 2
QB3 = IB4 * 5;
QX0.3 = IX0.3 & IX10.3;    // out: 3
QB4 = IB5 * 6;
QX0.4 = IX0.4 & IX10.4;    // out: 4
QB5 = IB6 * 7;
QX0.5 = IX0.5 & IX10.5;    // out: 5
QB6 = IB7 * 8;
QX0.6 = IX0.6 & IX10.6;    // out: 6
```

As shown above, index expressions may even be used in comments. This can be useful, because the expanded *iC* text must later be used for debugging with *iClive* – the original text with 'FOR loops' and index expressions is not meaningful for following the values of actual nodes at run-time. The above

example already gives a hint of how much writing can be saved. The way I/O bit variables following the IEC-1131 standard are expanded is particularly useful.

The *iCa* extensions to the *iC* language can be embedded as additional lines in regular *iC* code. A **%%define** macro definition may **not** be embedded in the middle of a line of *iC* code – not even between *iC* statements, which have been written in one line. This limitation is similar to the limitations imposed by the C pre-processor **cpp** on the C language.

7.4 FOR loops

'**FOR** loops' follow the syntax of C '**for** statements' with the difference, that the word **FOR** is upper case (to avoid clashes with '**for** statements' in embedded C code) and the controlled *iC* code **must** be enclosed in twin braces (single braces are required for *immediate switch* and *if else* statements as well as for function block bodies):

```
FOR (expr1; expr2; expr3) {{
    iC code, which is repeated under control of the loop
    or nested 'FOR loops'
}}
```

The only restrictions are:

1. Each '**FOR** statement' must define one (and only one) control variable, which is an **int** by default:

```
FOR (N = 0; N < 10; N++) or FOR (int N = 0; N < 10; N++)
```

The control variable is the first 'word' of expr1, which is not '**int**' i.e. **N** in the example. The word '**int**' in the second form is optional and can be written to remind programmers, that the control variable is an integer. The control variable cannot be declared anywhere else.

2. Other atoms in the three expressions must be either constant expressions or expressions which contain control variables of the current and/or outer '**FOR** loops'. All expressions may contain macro calls, which must expand to integer constants, strings or expressions containing valid **FOR** loop control variables. Under no circumstances may *immediate* variables be used in these expressions.
3. The names of control variables must be different from any *immediate* variable. It is highly recommended, that upper case names be used for '**FOR** loop' control variables. This and the upper case keyword '**FOR**' and the twin braces **{{ }}** make these code generating statements in the *iCa* language stand out from normal *iC* and C code.
4. The scope of the control variable of a '**FOR** loop' begins when the control variable is initialised in the '**FOR** statement' and ends with the final matching twin braces. The control variable is not valid outside of this scope. '**FOR** loop' control variables will never appear in the generated *iC* files (except as comments if the *immac* -a option is used).

Since *immac* is implemented as a Perl script, an alternate Perl type of '**FOR** loop' using a list in various forms may also be used.

```
FOR N (<Perl type list>) {{
    iC code, which is repeated under control of the loop
    or nested 'FOR loops'
}}
```

Similar restrictions to those above apply. The variable after the '**FOR**' is the loop control variable. It may optionally be preceded by the word '**int**'. The control variable is given each value of the 'Perl type list' for each iteration of the loop. Some powerful manipulations are possible with this form. Although a perlsh syntax is used in the second form of the **FOR** control statement, any variables in either form follow the C syntax for scalar variables – they are never preceded by a \$ as in Perl.

```
FOR int N (0 .. 3) {{ a[N], }}
```

internally generates the following Perl code (see optional .log file)

```
$out = “”; for my $N (0 .. 3) {$out .= “ a@{[$N]},”;} print $out;
```

which is executed as an eval to generate the following output:

```
a0, a1, a2, a3,
```

iC code embedded in twin braces is repeated without a LF, if the final braces are on the same line as the *iC* code. The same can be achieved by terminating an *iC* code line with a back slash '\', which looks as follows:

```
FOR int N (0 .. 3) {{
    a[N],\
}}
```

generates the same as above.

Lists in the second form of the '**FOR** loop' may be made up of decimal numbers or strings. Strings may be embedded in parentheses although lists of bare words will also be interpreted as strings.

```
imm int FOR N ("in", "out", "tmp") {{ fast_[N], }};
```

generates

```
imm int fast_in, fast_out, fast_tmp,;
```

The above *iC* declaration would have produced a syntax error until recently. The *iC* language has been extended to allow such comma separated lists to have a final comma before the semi-colon to end the statement. This is in line with other comma separated parameter lists, which may also have an extra comma at the end.

Again the same can be achieved with backslashes. The following (with barewords in the list) generates the same output as above, although this *iCa* snippet is not nearly as readable:

```
imm int\
FOR N (in, out, tmp) {{
    fast_[N],\
}}\
;
```

As shown above, lines terminated by a back-slash (\) are output without starting a new line – this make it possible to generate lists in a single line. This applies both inside a '**FOR** loop' and directly before and after a '**FOR** loop'. The end of the '**FOR** loop' would normally terminate such a generated list, unless the final brace of the '**FOR** loop' is also followed by a back-slash (\) as shown in the generated function block call statement in the last example above.

For those who don't like to see a comma followed by a semicolon ',' at the end of a declaration, a special characteristic of *iCa* index expressions can be used (see next paragraph). The value in square brackets may be strings as well as numbers, since they are actually generated by Perl code. To generate a variable length – single line – declaration, use the following:

```
imm bit FOR N (0 .. 5) {{ a[N][N < 5 ? "," : ";"] }}
```

generates

```
imm bit a0, a1, a2, a3, a4, a5;
```

Each execution of the second conditional index expression `[N < 5 ? "," : ";"]` in the loop generates a single comma, which is appended – the last execution of the index expression generates a semi colon.

The '**FOR** statements' for both types of '**FOR** loop' and the associated twin braces are not copied to the target except as comment lines, if the **-a** option is active for the *immac* compiler.

7.5 IF ELSE control statements

Sometimes it is necessary to suppress the output of code lines in a '**FOR** loop' or to supply one or more alternative output lines depending on some condition of the existing loop variables. This can be achieved with an '**IF**' or '**IF ELSE**' control statement. The syntax and semantics is identical to C '**if**' or '**if else**' statements – except that again the '**IF**' and '**ELSE**' keywords are upper-case not lower-case. Even one or more '**ELSE IF**' statements may follow an initial '**IF**' statement followed by a final (optional) '**ELSE**' statement. ('**ELSE IF**' may be written as '**ELSIF**' – it is translated to this form anyway to execute as Perl code). The '**IF**' conditional expression in parentheses may only contain existing '**FOR** loop' control variables and constants. No new control variable can be defined. Again *immediate* variables may not be used in these expressions. The *iC* or C code controlled by an '**IF**', '**ELSE IF**' or '**ELSE**' statement must be contained in twin braces (like the '**FOR** loop'). The following generates the same as the example in the previous section:

```
imm bit FOR N (0..5) {{ IF (N < 5){ a[N], } ELSE {{ a[N]; }} }}
```

7.6 *iCa* index expressions

Index expressions in *iCa* are expressions in square brackets involving loop control variables and integer or string constants. Unlike in other computer languages these 'index' expressions can be placed anywhere in the *iC* code – not just as an index of an array variable. *immediate* array variables cannot even be declared directly – they come into existence as simple immediate variables by evaluating the index expression and replacing the square brackets by the numeric or string result of that evaluation. The underlying simple *immediate* variables must of course be declared (unless not **strict** (which you wouldn't, would you)). Such a group declaration is best done as follows:

```
FOR (N = 0; N < 10; N++) {{
    imm bit a[N];
}}
```

Normally the square brackets are placed after a name, which then makes the array variables look like those in C. But there are special cases where the square bracketed index expression is placed somewhere else, as we saw in the earlier examples (computing IEC-1131 I/O variable names).

The semantics of index expressions is, that the expression in square brackets is evaluated during the execution of the *immac* compiler (written in Perl) as a Perl eval. The numeric or string result of the eval replaces the square brackets and the expression they enclose. When the index expression is a simple array reference, this generates a name followed by a number. The fact that evaluation of the index expressions is done by Perl means, that the expression syntax and semantics of Perl integer arithmetic apply, since **use int** is declared in the *immac* compiler. Since most arithmetic operators are the same for Perl and C, this is not of great consequence. One notable exception is the Perl exponentiation operator ******, which may be used in **FOR** loops and index expressions with good effect:

```
FOR (J = 0; 2**J < 16; J++) {{
    imm int mask[J] = [2**J];
}}
```

generates

```
imm int mask0 = 1;
imm int mask1 = 2;
imm int mask2 = 4;
imm int mask3 = 8;
```

Any *iC* or C code may have strings which contain the backslashed characters '\n' or '\t', which stand for a Newline or a Horizontal tab both in C or in Perl and also in *iC*. These special characters do not actually execute as a Newline or a Tab until the final machine code executes.

```
FOR (I = 0; I < 4; I++) {{
    printf("Hello world\t%d\n", [I]);
}}
```

generates

```
printf("Hello world\t%d\n", 0);
printf("Hello world\t%d\n", 1);
printf("Hello world\t%d\n", 2);
printf("Hello world\t%d\n", 3);
```

Not brilliant code but notice that '\t' and '\n' are correctly preserved in the generated *iC* code strings.

An exception to this rule are '\n' and '\t' characters contained in string expressions of an *iCa* index expression in square brackets. These '\n' and '\t' characters are converted to a Newline or Tab directly in the conversion from *iCa* to *iC* code. This allows the embedding of real Newlines or Tabs in lists of *iC* code generated by a **FOR** loop.

```
imm int trans = \
FOR (I = 0; I < 16; I++) {{
    IX[I/8].[I%8][I==16-1?"":I%4==3?" |\n\t\t":" | "]"
}}      // | NL TAB TAB after each group of 4 generates:

imm bit trans = IX0.0 | IX0.1 | IX0.2 | IX0.3 | NL
            TAB      TAB      = IX0.4 | IX0.5 | IX0.6 | IX0.7 | NL
            TAB      TAB      = IX1.0 | IX1.1 | IX1.2 | IX1.3 | NL
            TAB      TAB      = IX1.4 | IX1.5 | IX1.6 | IX1.7;
```

Normally index expressions occur in *iC* code in a '**FOR** loop'. I deliberately say *iC* code and not *iC* statements, because '**FOR** loops' are used not only to generate lists of statements, but also lists of

parameters – both for the definition and the call of function blocks, whose parameter lists can be varied at compile time. Another use is varying constant parameters. Inside a '**FOR** loop' or a nest of '**FOR** loops', the *iC* code use the '**FOR** loop' control variable(s) in the index expression(s) to make each repeated *iC* code line different.

For index expressions in *immediate C* code outside of a '**FOR** loop', the expression must be a constant expression – no variables are allowed (remember no '**FOR** loop' control variables are in scope anyway). Nevertheless an *iC* variable, which is used as an indexed array variable inside a '**FOR** loop' looks better if it follows the same syntax outside of the loop. The variable **a[1]** could of course be written as **a1** – this is the same immediate variable. But inside a loop it must be written as **a[N]** and only the varying value of **N** will generate **a0 a1 a2** etc.

Index expressions in embedded C code – either in a literal block or in a compound C statement controlled by an *immediate if else* or *switch* statement may have index expressions, but they are part of the C code and are not changed except index expressions, which contain an in-scope '**FOR** loop' control variable. This means that the translation of constant index expressions – as described in the previous paragraph - are not carried out in embedded C code. In the rare instances where such a translation is needed, it must be done manually – write **a1** instead of **a[1]**.

A special case in embedded C code occurs, if a numeric value generated by the control variable of a '**FOR** loop' must be placed inside the square brackets of a C array reference. This can be done by simply embedding the *iCa* index expression in the C index expression – e.g.:

```
if (IX0.0) {
    int Carray[3];                // start of embedded C code
    FOR (N = 0; N < 3; N++) {{
        Carray[[N]] = iCarray[N];
    }}
}
```

generates

```
if (IX0.0) {
    int Carray[3];                // start of embedded C Code
    Carray[0] = iCarray0;
    Carray[1] = iCarray1;
    Carray[2] = iCarray2;
}
```

As can be seen in the above example, *iCa* '**FOR** loops' may be embedded in C code – this is the reason why the keyword '**FOR**' was chosen instead of '**for**' – the C code may also contain C '**for** statements'.

To sum up, immediate arrays are not declared as such – variable names are used with index expressions in square brackets. The programmer must be aware that this generates simple *immediate* variables starting with the array name followed by a number. Such generated variable names cannot be used anywhere else – this would show up as a multiple declaration during *iC* compilation. If we use a one-dimensional array in an *iCa* program – e.g. **sa**, any array reference will simply have a number appended to the array name in the generated *iC* code.

```
i = 2,          sa[i]    generates sa2
i = 22,         sa[i+1]  generates sa23
```

7.6.1 Multi-dimensional index syntax

A special case are multi-dimensional arrays. If we use the standard C syntax to write a multi-dimensional array reference, e.g. **ma[i][j]**, and the *immac* pre-processor did not take special action, we would get the following erroneous compile output for the following pairs of index values:

```
i = 2,  j = 34    ma[i][j] would generate ma234 // NOT output
i = 23, j = 4     ma[i][j] would generate ma234 // NOT output
```

This would be unsatisfactory, because it is ambiguous – therefore *immac* inserts a letter **x** between adjacent numeric index expressions, producing the following correct output instead:

```
i = 2,  j = 34    ma[i][j] generates ma2x34
i = 23, j = 4     ma[i][j] generates ma23x4
```

This is no longer ambiguous. Any multiple index is separated by an **x**, which is easily recognised in the generated *iC* code as a member of a multiple-dimensional array – even the numeric index values can be recognised easily in the generated names.

Both in *C* and by analogy in *immediate C* with arrays (*iCa*), array names and the index expressions in square brackets (and of course the expressions in the square brackets) may be separated by spaces and tab's – as follows:

```
i = 2, j = 34    ma [ i ] [ j ]    still generates ma2x34
i = 23, j = 4   ma [ i ] [ j ]    still generates ma23x4
```

One caveat applies for **immac**: such an array name with all its subsequent square bracketed index expressions must be in the same line. (In *C* any sort of white space is allowed).

Another case where **immac** inserts an extra character are array names which finish with a numeral. This could also lead to ambiguity if special action were not taken:

```
i = 2,          sa9 [ i ]          generates sa9y2
i = 22,         sa9 [ i+1 ]        generates sa9y23
```

Although the way **immac** handles array names, which finish with a numeral avoids ambiguity, such names should be avoided, because in the generated *iC* code they look too much like expanded array names with an extra index, which could easily lead to clashes. To avoid this clash a **y** is inserted in this case.

String index expressions in square brackets, which contain a string value in parentheses, e.g.

```
[N < MAX ? ", " : ";"]
```

are not separated from an adjacent index expression by **x** or **y**.

In every case, the names generated from numerical indexed single- and multi-dimensional array references are well formed *iC* variables, which show their name and index value(s). The main thing to remember with array references is, that every array reference translates to a simple *iC* variable name, which shows up in the generated *iC* code, which will normally be a lot longer than the *iCa* code, but which can then be used for live debugging with **iClive**. The mental translation between indexed array references and the resolved *iC* names is so simple, that it should not cause any problems to the user.

7.7 Differences between *iC* and *iCa* code

Straight *immediate C* code is usually made up of short statements declaring the relationship between input and intermediate variables to output or intermediate variables - very similar to PLC code, which is easy to understand by technicians. It presents a clean picture of control expressions acting on control variables, which build up to a clear picture of the interactions with the plant to be controlled. This interaction is most clearly visible when a live display is active, where individual changes in the real plant parameters show up as state information in the code. This PLC style of coding is a very important aspect of producing *immediately* understandable and straight forward control programs. This was the most important design consideration for the *immediate C* language.

On the other hand *iCa* code with arrays introduces another level of algorithmic loops, control statements and indexing in the middle of *iC* code for generating larger parametrised blocks of *iC* code. Frankly the actual *iC* code required is hidden quite deeply and it requires a certain amount of skill when developing *iCa* code snippets, to simply concentrate on what is to be generated and adjust the looping and control algorithms accordingly. Translating the code with the **immac** compiler frequently is the best way to see what *iC* code is actually generated, which can then be checked to see if it is really the *iC* code envisaged. In fact I found it important to code a small block of *iC* code first to lay down the control strategy. Once that is fixed, repeating statements can be rolled into loops fairly easily. Comparing the generated code with the hand coded part using **diff** confirms that *iCa* loop and control algorithms are correct.

Here is a hand coded *iC* program segment which can then be made variable in length:

```
imm clock c0 = CLOCK(T1sec, ~T1sec);
imm bit m0, m1, m2, m3, m4, m5, m6, m7, m8;
QX0.0 = m0 = SR(~m8, m8 & ~m1, c0);
QX0.1 = m1 = SR(~m8 & m0, m8 & ~m2, c0);
QX0.2 = m2 = SR(~m8 & m1, m8 & ~m3, c0);
QX0.3 = m3 = SR(~m8 & m2, m8 & ~m4, c0);
QX0.4 = m4 = SR(~m8 & m3, m8 & ~m5, c0);
QX0.5 = m5 = SR(~m8 & m4, m8 & ~m6, c0);
QX0.6 = m6 = SR(~m8 & m5, m8 & ~m7, c0);
QX0.7 = m7 = SR(~m8 & m6, m8, c0);
QX1.0 = m8 = SR(~m8 & m7, m8 & ~m0, c0);
```

This is the required *iCa* code which is not as clear cut, but does generate blocks of any length:

```
%%define LAST 8          // iC code snippets are highlighted
imm clock c0 = CLOCK(T1sec, ~T1sec);
imm bit FOR (I = 0; I <= LAST; I++) {{ m[I], }};
FOR (I = 0; I <= LAST; I++) {{
  QX[I/8].[I%8] = m[I] = SR(~m[LAST]\
    IF (I == 0) {{["", "]}}, ELSE {{ & m[I-1],}} m[LAST]\
    IF (I == LAST-1) {{["", "]}}, ELSE {{ & ~m[(I+1)%(LAST+1)],}} c0);
}}
```

Saving that code as **genBar.ica**, the following call will generate the code below the call:

```
$ immac -P LAST=16 genBar.ica > genBar.ic
imm clock c0 = CLOCK(T1sec, ~T1sec);
imm bit m0, m1, m2, m3, m4, m5, m6, m7, m8, m9, m10, m11, m12, m13, m14,
m15, m16;
QX0.0 = m0 = SR(~m16, m16 & ~m1, c0);
QX0.1 = m1 = SR(~m16 & m0, m16 & ~m2, c0);
QX0.2 = m2 = SR(~m16 & m1, m16 & ~m3, c0);
QX0.3 = m3 = SR(~m16 & m2, m16 & ~m4, c0);
QX0.4 = m4 = SR(~m16 & m3, m16 & ~m5, c0);
QX0.5 = m5 = SR(~m16 & m4, m16 & ~m6, c0);
QX0.6 = m6 = SR(~m16 & m5, m16 & ~m7, c0);
QX0.7 = m7 = SR(~m16 & m6, m16 & ~m8, c0);
QX1.0 = m8 = SR(~m16 & m7, m16 & ~m9, c0);
QX1.1 = m9 = SR(~m16 & m8, m16 & ~m10, c0);
QX1.2 = m10 = SR(~m16 & m9, m16 & ~m11, c0);
QX1.3 = m11 = SR(~m16 & m10, m16 & ~m12, c0);
QX1.4 = m12 = SR(~m16 & m11, m16 & ~m13, c0);
QX1.5 = m13 = SR(~m16 & m12, m16 & ~m14, c0);
QX1.6 = m14 = SR(~m16 & m13, m16 & ~m15, c0);
QX1.7 = m15 = SR(~m16 & m14, m16, c0);
QX2.0 = m16 = SR(~m16 & m15, m16 & ~m0, c0);
```

The above also demonstrates how **-P LAST=16** has precedence over **%%define LAST 8**.

7.8 immac Macro facility

The pre-compiler **immac** provides a full macro facility very similar to that provided by the C pre-processor **cpp**. Object like macros without parameters as well as function like macros with parameters in parentheses are supported. The keyword to introduce an **immac** macro definition is **%%define** not **#define**; that is reserved for **cpp** or **immac -m**. The latter is an alternative to **cpp** and is used in conjunction with the full *iC* compiler **immcc** to resolve C type macro's in embedded C code fragments.

```
%%define LENGTH 4
```

The same macro term **LENGTH** could also be pre-defined in the command line with the **-P** option:

```
immac -P LENGTH=4
```

Unlike **cpp**, the definition in the command line has precedence over the definition with a **%%define** line in the program. This allows *iCa* programs to define default values for macro terms, which can be re-defined in the command line. It is an error to **%%define** a macro, which has been previously defined (except on the command line, in which case the new definition is ignored). The command **%%undef X** will undefine the macro X, which can then be re-defined. This is important if an internal definition is to have precedence over a (possible) command line definition – do a **%%undef** first. It is not an error to **%%undef** a non-existing macro.

Macros must be a word starting with a letter or underscore followed optionally by letters underscores or decimal digits (same as a C or *iC* identifier). It is highly recommended that letters in a macro are all upper case (same recommendation as for **cpp**). Macro replacements can be any sort of text, which may also include previously defined macros. For replacement as index values, they should of course reduce to numeric values or string constants.

```
%%define WIDTH (5+1)          /* C comment */
%%define AREA (LENGTH * WIDTH) // C++ comment
```

If a replacement text is longer than one line, each line except the last must finish with a backslash \. As shown above **%%define** lines may be terminated with a C or C++ comment. Replacement texts may also contain embedded C comments, which will be replaced by a single space on expansion. Multiple spaces will be replaced by one space (same as **cpp**). As with 'FOR loop' control lines, a C comment must finish on the **%%define** line. Replacement texts for function like macros should contain at least one sample of each parameter text. If not a warning will be issued.

Parameters may be 'stringified' in the replacement by preceding them with a single #. Two parameters or indeed any words may be concatenated by placing ## between them. Every effort has been made to obtain the same translations for replacement texts as those obtained by using **cpp**.

There are some deliberate minor differences. Replacements which resolve to a constant arithmetic expression involving only the operators + - * / and % as well as () *decimal integers* and *spaces* are evaluated in the definition. This brings error messages a little closer to the source of any erroneous constant expression. The final result is the same though.

For the 2nd macro above **immac** translates **%%define AREA** to **48** whereas **immac -m** and **cpp** translate **#define AREA** to **(8 * (5+1))**.

The **%%define** lines are not copied to the target except as comment lines, if the -a option is active for the **immac** compiler.

Macro replacements may be made in all parts of the *iCa* code. They are of course particularly useful to parametrise the termination of a 'FOR loop' and hence the number of blocks of *iC* code, which is generated by the 'FOR loop'.

File inclusion with **%%include "file"** and conditional compilation with **%%ifdef**, **%%ifndef**, **%%if**, **%%elif**, **%%else**, **%%endif** and **%%error** are also supported using the same rules as **cpp**. The word **defined** in an **%%if** or **%%elif** expression has the usual **cpp** meaning - it is set to 1 (true) if defined else 0 (false). Identifiers in such an expression which are not defined in a previous **%%define** or -P are also set to 0 (false).

7.8.1 Alternative *immac* Macro options

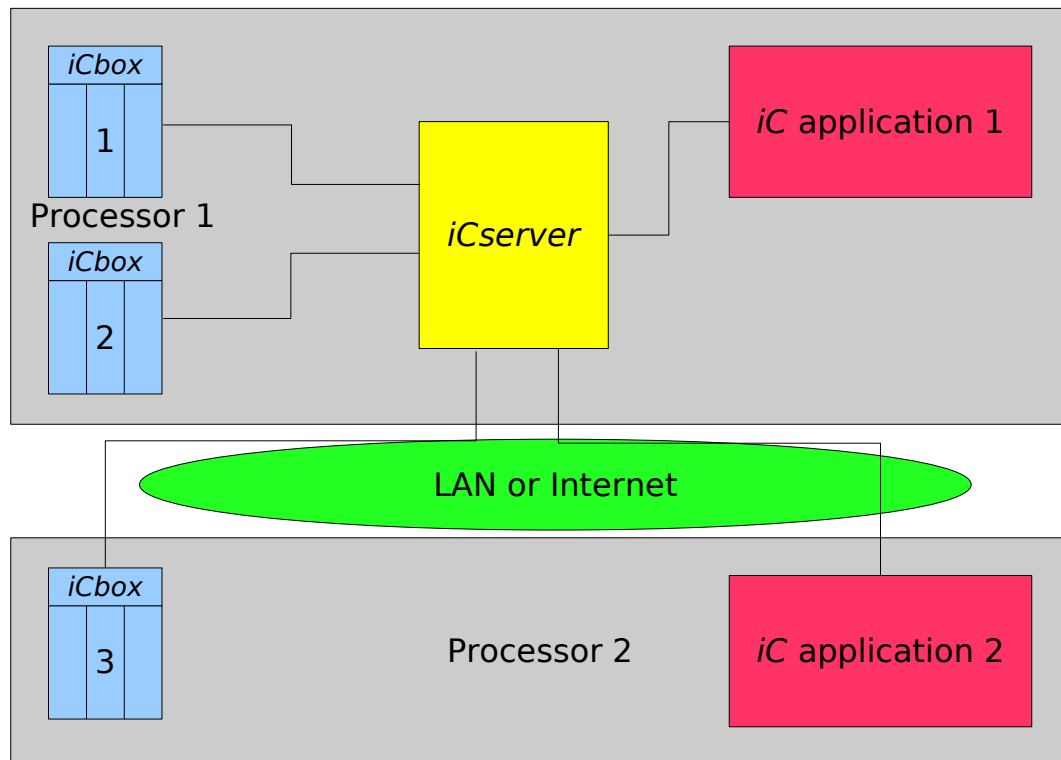
Calling **immac** with the -m option, it acts as a straight 'cpp style' macro processor handling **#define**, **#undef**, -D, -U, **#include**, **#if**, **#else** etc. No *iCa* constructs are translated in this mode. Every attempt has been made to make **immac -m** equivalent to **cpp**. This option is used as a pre-compiler for generated C code when compiling *iC* programs with **immcc**.

Calling **immac** with the -M option, it acts as a straight macro processor for *iC* code handling **%define**, **%undef**, -D, -U, **%include**, **%if**, **%else** etc. Again no *iCa* constructs are translated in this mode. This option is used as a pre-compiler for *iC* code when compiling *iC* programs with **immcc**.

Calling **immac** with the -Y option, it acts as a special macro processor handling **%if**, **%else** etc. directives only. This enables optional compiling for yacc, bison or flex; **%define** directives are left untouched – they are used as macros directly by bison.

8 I/O drivers and *iCserver*

This chapter describes 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***.



Most input and output in the *iC* system is via short messages transported by TCP/IP connections between *iC* components, which may be *iC* applications or I/O drivers. All TCP/IP I/O messages are routed through a central server called *iCserver*. Both *iC* applications and I/O drivers are clients of *iCserver*, which is started by the first client, unless it is already running. On start-up, each client opens a TCP/IP connection to *iCserver* and registers itself with *iCserver*. More importantly the client registers each input or output word it wants to send or receive by its IEC-1131 name (bit I/O's are grouped into one 8 bit byte for transmission). The client also states, whether it is a Sender or Receiver for the named IEC-1131 word. *iCserver* allocates a channel number for each word which is registered (unless two different words are equivalenced, which will be discussed later). Only *iCserver* channel numbers, which are small integers, are used in the actual I/O messages, which are a comma separated list of pairs as follows:

<channel number>:<value>

Example for 4 simultaneous, but independent values on channels 10, 12, 14 and 22:

10:2,12:128,14:0,22:500

Values are also decimal integers. *iCserver* monitors that only one Sender is registered for each IEC-1131 name. There may be more than one Receiver per name. IEC-1131 output names for *iC* applications, e.g. **QX0** or **QB1** are Senders, whereas they are Receivers for I/O drivers. The reverse is true for IEC-1131 input names, e.g. **IX2** or **IW3**, which are Receivers for *iC* applications and Senders for I/O drivers. I/O drivers may be virtual drivers e.g. **iCbox**, I/O for GUI driven canvases, e.g. **iClift** or real I/O drivers e.g. **iCpiFace** for one or more PiFace boards and the GPIO's on a Raspberry Pi.

For the Raspberry Pi there is an alternative driver to **iCpiFace**, which completely by-passes *iCserver*. Instead of sending TCP/IP messages, this driver is implemented in the *iC* run-time library, linked to an application and connects input and output events generated by the application directly to the outputs and inputs of the PiFace boards and the GPIO's on a Raspberry Pi. This type of real I/O is 10 to 50 times faster than I/O via *iCserver*, although that is already fast compared with relay logic or PLC's (0.9 ms on an Rpi 2B, 2.8 ms on an Rpi B or B+).

A similar direct driver for the *Interbus* system for Phoenix Contact industrial I/O devices and another driver for a *Fieldbus* system had been written and tested successfully during early development of the *iC* system but were abandoned when interest for industrial applications was not forthcoming.

Currently there are plans to provide a driver as well as direct connection to *iCserver* for JSON messages to connect to external devices. These will look as follows in JSON for the same example as above:

```
{“IX0.1”:true,“IX2.6”:false,“IX2.7”:true,“IB1”:0,“IW3”:500}
```

Registration from JSON capable apps has yet to be designed. The *iC* network protocol is very similar to JSON, but shorter and faster, because of the use of channel numbers and hence will not be abandoned. It was designed and implemented about 10 years before JSON first arrived.

8.1 *iCserver*

iCserver acts as a router for a number of *iC* clients in a network, who send data to each other. Each client connects via TCP/IP to *iCserver* on a specified port (8778 is the default at the moment). Only one *iCserver* on one port may run in a network. It is possible to run several *iCservers* on different ports. Clients can connect either via 'localhost' (default for *iC* clients) when they run on the same processor as *iCserver* or via the host address of the processor *iCserver* is running on. On connection each client registers itself with *iCserver*.

Clients for *iCserver* are *iC* control applications, I/O clients such as *iCbox* or similar real I/O clients and debugging programs such as *iClive*. These clients either send or receive data values from and to named I/O locations or debugging information. As far as *iCserver* is concerned I/O locations could have any name, but the *iC* language calls for I/O names or addresses according to the IEC-1131 standard. Data values can be 8 bit bytes (e.g.: **IB1 QB9**), 16 bit words (**IW2 QW10**), 32 bit long words (**IL4 QL12**) or 64 bit huge words (**IH8 QH16**), although huge words have not been implemented in any client so far. Bit values like **IX0.0 IX0.1 QX8.2 QX8.3** are always transmitted as bytes - in this case via **IX0** and **QX8**, which are the names used for registration. Whenever any bit in the byte changes, the whole byte is transmitted. Each client registers the I/O names it requires on connection to *iCserver*. Each unique name is stored in a Hash in *iCserver*, whose value is a channel number, which is used for all actual data transfers. The Hash is only required for registration. Each channel allows the naming of one Sender for data on the channel (or I/O name) and one or more Receivers for the data. A detailed description is in the Specification in the *iCserver* man page.

Additional functionality in *iCserver*.

- a) Equivalences - or interconnection of different I/O addresses in *iCserver*. This option puts two or more entries in the Hash and assigns them a common channel number. Send or receive entries associated with the channel are undefined at this stage. Then when registration of one of the equivalence names occurs, the common channel number is used. With this option different IEC-1131 addresses can be assigned to the same channel, thereby making them equivalent or interconnecting them.

This functionality is required when the output of one *iC* control application is to be the input for another *iC* control application (often a different instance of the same application). Several equivalences may be specified. The order of the equivalence is not relevant (it is not an assignment). For correct autovivification outputs should be named first though.

Example 1:

```
iCserver -e QX7-0=IX7-1,QX7-1=IX7-0
```

This connects **QX7** of instance 0 with **IX7** of instance 1 via a common channel and **QX7** of instance 1 with **IX7** of instance 0 via another common channel. Equivalencing an output and an input of the same instance is possible but rather useless and much slower than using internal variables (it may be useful for testing).

Equivalencing is also required if one external I/O source must be connected to the input of more than one *iC* control application.

Example 2:

```
iCserver -e IX8=IX8-0=IX8-1,IX9=IX9-0=IX9-1
```

This sets up common inputs **IX8** and **IX9** from an I/O driver to two instances of the same control app. Naming the other inputs with the same base IEC-1131 address is not necessary but highly advisable for transparency in the documentation. The first address is used for autovivifying an *iCbox* if it does not exist already. Autovivification does not take place for a channel, until a receiver has been registered for that channel and all other registrations have taken place.

Equivalencing two or more output addresses (**Q...**) will lead to an error, if both output addresses register as senders in an *iC* control application - this would lead to two or more

senders on the same channel. When the second or later output sender registers, the error will be reported. A similar error will be reported if a second external input device in an equivalence chain is registered as a second sender. This would happen if both **IX8-0** and **IX8** were started as an *iCbox* after the equivalence statement in Example 2 above. In rare cases two outputs may be legitimately equivalenced if an *iC* application uses a certain output name and a real output with a different name must be used to accept that output. This practice is highly deprecated, because transparency in the documentation is lost.

Formally equivalences consist of two or more IEC base identifiers followed by an optional 1 to 3 digit instance specifier separated by an equal sign '='. Several equivalences may be specified in a comma ',' separated list or several equivalence parameters may be used.

It is not allowed to equivalence IEC identifiers of different types, since the consequences are not what is expected.

Equivalences can also be defined later by an *iC* app for input IEC variables in that app before the variables are registered.

b) Autovivification

-a option - automatic startup of one or more *iCbox* widgets. When a control application registers its I/O's, '*iCserver* -a' starts up a matching '*iCbox*' for all complementary I/O's, which have not already been registered. With the -a option, clients must be started in a particular order:

- i) **iCserver -a** # always first anyway.
- ii) any manually started I/O's with real I/O or specific ranges etc. or because of equivalences (optional).
- iii) *iC* control application(s), which causes *iCserver* to autovivify any missing I/O's as *iCbox* virtual IO's with appropriate ranges for each app.

Alternatively starting an *iC* -d option - automatic startup of one *iCbox* -d When a control application registers its I/O's, '*iCserver* -d' starts up a matching '*iCbox* -d' for all complementary I/O's for monitoring. Outputs are the same but inputs will only display their value and cannot be changed. With the -d option, clients must be started in a different order:

- i) **iCserver -d** # always first anyway.
- ii) one control application, which causes *iCserver* to autovivify all I/O's (which are all missing their complementary senders and receivers at this stage) as an '*iCbox* -d' with appropriate ranges.
- iii) any manually started I/O's with real I/O or virtual I/O. Care must be taken to ensure that all missing inputs for the app are present, since no further missing I/O's are autovivified. Missing outputs are optional.

-A <cmd> - automatic startup with <cmd>. Usually <cmd> is *iCbox* with extra options e.g. -A '*iCbox* -C19'. Startup and calling order is the same as for the -a option unless the -d option is also used, in which case the -d option applies.

Note: with the -a -d and -A option care is taken to only autovivify the first member of an equivalence list, which is the sender of that list if it is an output or will become the sender if it is an input.

c) -g - automatically start *iClive* xxx.ic when SCxxx registers.

-G <dbg> - automatically start <dbg> xxx.ic when SCxxx registers. Usually <dbg> is *iClive* with options e.g. -G '*iClive* -t'

d) -r option - reset registered receivers when sender disconnects i.e. reset outputs of an app when it shuts down (default no change)

e) -k option - if a sender registers with the same name as one already registered, kill the previously registered sender, rather than reporting an error. This allows a recompiled version of an *iC* application to be started, while an old version is still running - the old one will quietly be killed. This should not be done in a production system.

Note: when a control application exits (disconnects from *iCserver*) the I/O's are not disconnected. They can be re-used by a restarted similar control application. If the restarted control application uses different I/O's a new *iCbox* is autovivified for any extra I/Os. This situation could be confusing and it would be better to start again by stopping *iCserver*. When *iCserver* exits, all connected clients are disconnected and closed.

- f) `-R <aux_app>[<aux_app_argument> ...]` # must be last arguments. Start a Bernstein chain of *iC* application which are each initialised and then started in parallel - in this case with *iCserver*.

Example 3:

```
iCserver -R iCbox X0-X3 X10 -R sort
```

- g) client calls (deprecated) - a list of calls with their parameters allow *iCserver* to start a number of clients as separate processes like a shell. These are usually all the control application(s) and I/O client(s) to make up a complete control system. By providing this functionality in *iCserver*, all the information for starting a control system is grouped in one place.

Client calls consist of a program path optionally followed by a space separated list of parameters. Calls with parameters must be quoted on the command line and the optional INI file to make them a single parameter for *iCserver*.

Example 4:

```
iCserver 'iCbox -n sorter-I0 IX0 QX0 QX1' sorter
```

This starts *iCserver* and two clients - *iCbox* and the control application 'sorter'. Since correct initialisation of the client processes started in parallel is not well controlled leading to bad Autovivication, Bernstein chaining with the -R option is a better choice.

- h) `-f <option_file>` - execute a file with *iCserver* options at startup. For very large equivalence tables a file defining equivalences and possibly other *iCserver* switches and options can be used.

```
iCserver -f <option file>
```

Format of the option file:

```
<equivalence line>
<equivalence line>
...
<other options>
```

- i) Example of an option file (same as Example 1, 2 and 4 above with extra options):

```
QX7-0 = IX7-1 QX7-1 = IX7-0 # equivalences joining 2 instances
IX8 = IX8-0 = IX8-1      # input equivalences
IX9 = IX9-0 = IX9-1
'iCbox -n sorter-I0 IX0 QX0 QX1' # quoted client call
sorter                        # plain client call
-ak                           # extra switches for iCserver
```

Individual equivalences and client calls must be written without spaces in the command line unless client calls are quoted. In the INI file white spaces before and after the = may be used in equivalences

Comments in the option file are started with #

A detailed description of command line options are available with '*iCserver -h*' or in the *iCserver* man page, best displayed by:

```
iCman iCserver
```

The *iCserver* man page also includes the full specification of registration and data messages between *iCserver* and its clients, which may help understanding the *iC* network and how it should be configured.

8.2 Bernstein Chaining

If several different *iC* applications, or different instances of the same application are to be started together, they must all run in parallel as separate processes (and in parallel with *iCserver*, *iCbox* and *iClive*). This is difficult to achieve with shell commands.

THESE SHELL COMMANDS DO NOT WORK CORRECTLY!

```
bar; bar -i1      # does not start bar -i1 until bar stops
bar &; bar -i1    # is a shell syntax error
bar &             # this starts processes in parallel
bar -i1          # but initialisation sequences clash
                  # because initialisation starts in parallel
```

None of these is what we want., because each *iC* application as well as each I/O driver consists of two parts:

1. an initialisation sequence, which includes registration with *iCserver*, which must be completed for each app before the next app is started.
2. a series of interrupt driven actions, which are independent for each app and must be run in parallel so that all events triggered by interrupts are handled correctly.

To achieve the desired result, *Bernstein chaining* has been implemented with the -R option for every *iC* app, for all *iC* drivers and for *iCserver*.

THIS WORKS FOR THE PREVIOUS NON-FUNCTIONAL CASES

```
bar -R bar -i1    # starts bar and bar -i1 in parallel
                  # with well sequenced initialisation
```

With *Bernstein Chaining* each app e.g. 'bar' is fully initialised and registered with *iCserver* before the -R option starts the next app and forks it to run in parallel with the previous app. Here is a longer example:

```
bar -l -R bar -i1 -R bar -i2 -R bar -i3    # starts bar -l then:
      iCserver -z -ak                      # unless already on
      iClive -z bar.ic                     # triggered by bar -l
      bar -z -i1 -R bar -i2 -R bar -i3     # 2nd app in chain
      bar -z -i2 -R bar -i3                # 3rd app in chain
      bar -z -i3                           # 4th app in chain

                                           # by auto-vivification iCserver starts:
      iCbox X0 B1 X2                       # for bar
      iCbox X0-1 B1-1 X2-1                 # for bar -i1
      iCbox X0-2 B1-2 X2-2                 # for bar -i2
      iCbox X0-3 B1-3 X2-3                 # for bar -i3
```

Only the first app in the chain (which is 'bar' in this case) has keyboard input. It can be stopped by typing 'q'. This in turn stops *iCserver*, which stops all other apps in the chain. All chained apps have the -z option, which blocks keyboard input. Another way to stop the whole chain is to click the (X) button of any *iCbox*, which stops *iCserver*, which in turn stops all *iC* apps registered with it.

Chaining is important for driver calls with real I/O arguments. For the Raspberry Pi these are calls to *iCpiFace*, *iCpiPWM* and *iCtherm*, which all support Bernstein chaining with the -R option.