B-CON, IEC1131 Programming Language

Getting Started Guide

V. 1.00 / Feb 2006 / Doc 40029

Brodersen Controls A/S ● Industrivej 3 ● DK-4000 Roskilde ● Tel: +45 46 74 00 00 ● Fax: +45 46 75 73 36
E-mail: bc@brodersencontrols.com ● Internet: www.brodersencontrols.com

1. Introduction

B-CON is a PC programming language, designed to allow engineers to create control sequences and configuration software for the Brodersen Control range of Telemetry and Bitbus products.

B-CON can be used by both the engineer, who may not be formally educated in programming, and the experienced programmer alike. B-CON unlocks the power of System 2000 modules, enabling them to perform complex control functions, alarm monitoring and in the case of the RTU8 compact outstation, the logging of data.

The IEC 1131 standard for programming software defines a range of tools with the basic idea of achieving some common standards between different hardware/software vendors. B-CON uses the IEC Instruction List standard.

The B-CON software package is run from a PC, where the program is created, compiled into code, capable of running in the System 2000 modules, tested for correct operation and finally downloaded into the module.

In the development of B-CON, great importance has been attached to achieving an efficient programming cycle, editing, compiling and testing.

The creation of a programme consists of the following steps:

The programme is created using the instruction list format in the editor, to create the source text.

The source text is then translated into binary code by the compiler, which will perform two passes on the source text to check for errors.

The resulting code can then be tested for correct operation using the test mode.

The finished code is then downloaded to the module.

B-CON programmes can be downloaded into System 2000 slave modules, remote slave and master modules and the RTU8 range of compact outstations.

2. Installing the software

B-CON for Windows will run under Windows 95, 98, 2000, ME or NT and the minimum specification for the PC is dictated by Windows rather than the B-CON software.

In general terms the PC should be at least a 200Mhz Pentium machine, with a hard disk, VGA or better display and have at least 32 Mbytes of RAM.

The B-CON software will be automatically installed on you PC, when you install the IOTOOL32 suite of software from your CD.

Definitions

1 bit = the basic logic unit which can have a value of 0 (logic false) or 1 (logic true)

1 byte = 8 bits

1 word = 16 bits or 2 bytes

Digital Input (DI) - each DI is 1bit

Digital Output (DO) - each DO is 1bit

Analogue Input (AI) - each AI is 16 bits or 1 word

Analogue Output (AO) - each AO is 16 bits or 1 word

Using B-CON for the first time.

This part of the manual is a simple step by step exercise in making a B-CON programme and installing the programme in a module. The points addressed in this exercise are discussed in detail in the B-CON User Manual. Experienced users may naturally skip this guide, but if B-CON is new to you, it is strongly recommended that you follow the exercise.

B-CON is started from Ioexplorer. You will find a button on the toolbar, activate the button and the B-CON editor will appear. The editor has a standard Windows type screen, with pull-down menus at the top.

A full explanation of the pull-down menus is given in the B-CON User Manual, but you will probably recognise the format as being similar to your word processor.

You now need to connect the module, to be configured, to the PC. This will either be with an RS232 serial link, in the case of an RTU8 or other Modbus module, or via the Bitbus PC card and RS485 in the case of Bitbus modules. See the documentation with your hardware for details of making this connection.

For our example in getting started we will assume that the node to be configured is an RTU8 compact outstation with 16 digital inputs, 8 digital outputs and 4 analogue inputs.

Setting the communication, device and programme options.

The first step is to tell B-CON the location of the node.

From the "Options" menu, select "Communication and Program" and in the box that appears, ensure that the small box labelled "System 2000 I/O Drivers" is ticked.

The 3 options at the bottom of the box should be left at their default values:

The first is to select the "Scan Time" in milli-secs. This is the time interval that the end programme will run at in the hardware module. Default is 100ms.

The second option is to define the module as a master or slave. Default is slave.

The third option is to select whether the De-bug option is required. De-bug is a tool for finding faults in the programme that you create and is explained in the B-CON User Manual. Default is "no". Click "OK".

You will be now be presented with the familiar Windows "Save as" screen. Insert the name you wish to call the file (for storage on the PC) being sure to retain the .PGM extension. Select an alternative directory if you wish and click "Save".

Another box will appear entitled "Device Configuration" in which we tell B-CON the module address and I/O configuration.

In this box insert the address of the node ie. The address set on the module, see the documentation with your hardware for details of setting the hardware address. For an RTU8 this address should be in the range 1 - 32. So set the address to 1 in both this box and on the RTU8.

The remaining boxes are for defining the actual number of inputs and outputs the node has, both in itself and any attached expansion modules. These are entered in "words" (1 word = 16 bits) therefor for our exercise 16DI = 1 word, 8DO = 1 word (even though there are only 8DI) and 4AI = 4 words. You can now enter these values in the relevant boxes, but instead of doing it manually, you can just click the "Update" button and B-CON will automatically interrogate the module and make the entries. Then click "Save".

You are now ready to start making your B-CON programme, but before we do that let's look at how we identify (address) the individual hardware inputs and outputs in the B-CON programme.

I/O addressing

Because the modules utilise an 8 bit micro-processor, which handles data 8 bits (1 byte) at a time, the I/O is addressed in bytes (8 bits) or words (2 bytes or 16 bits).

Please note that the physical inputs/outputs in the module are labelled from 0 and not from 1. This convention is also used within B-CON, thus 1 byte of DI will be addressed 0 – 7, not 1 to 8.

Digital inputs have an address between 0 - 1999. In our RTU8 we have 16DI or 2 bytes of inputs and the first byte of inputs have the address 0 and the second byte an address of 1. Byte 0 is physical input 0 - 7 on the module and byte 1 is physical input 8 - 15.

Digital outputs also have an address between 0 - 1999. In our RTU8 we have 8DO or 1 byte of outputs. This byte of outputs have the address 0 which are physical outputs 0 - 7 on the module.

Inputs and outputs have the same address, but they are always identified as an input or output and thus there is no conflict.

Analogue inputs have an address between 2000 - 3998 and each analogue input uses 2 bytes or 1 word. Thus in our RTU8 we have 4AI or 4 words of inputs and the first input has the address 2000, the second input an address of 2002 and so on, corresponding to physical analogue input 0, 1 etc. on the module.

It is import to understand that when dealing with words, the address of a word must have an even number e.g. 2000, 2002, 2004 etc. This is because the registers are numbered as bytes and there are 2 bytes to each word. Just remember that it must be an even number.

Likewise analogue outputs also have an address between 2000 - 3998, but as our RTU8 in this example does not have any analogue outputs, we won't worry about that for the time being.

Making the B-CON programme

The actual B-CON programme is made as a text file (the source text) on the main screen, just as you would make a document in a word processor. For the purpose of our exercise the text shown below in UPPERCASE is the text you need to type on the screen, the text can actually be entered in lowercase if you wish.

Click on the top of the screen working area to write the first line of text.

In order to make the local I/O of the module available to the operational PC that will be eventually connected to the module, we need to first make a definition of the number of inputs the PC is allowed to read. The definition is made in words. A definition of physical digital or analogue outputs is not required, as the PC is not allowed to directly access outputs, but more of that later.

So for now type the following:

```
#DEFINE DI_CNT 1
#DEFINE AI_CNT 4
```

Be sure to indent and use the #, _ and insert spaces.

The operational PC will now be able to read the values of the 16DI and 4AI.

So what programme should we now write? Lets make a very simple programme which will copy the first 8 digital inputs to the 8 digital outputs, so that if input 0 is driven (logic true) then output 0 will also be set to give an output (logic true). The same happening for input 1 and output 1, input 2 and output 2 etc.

Under the define statements, insert a blank line and then type:

MAIN:

This is just syntax, it means the start of the main part of the programme and must always be placed before the programme text, without an indent.

Your screen should now display:

```
#DEFINE DI_CNT 1
#DEFINE AI_CNT 4
```

MAIN:

It is good practice to save your work periodically, so from the "File" menu click "Save" and save your work thus far.

The source text is largely made of lines containing operators and operands.

Operators are commands. Operands are modifiers to the operand.

One of most used operators is LD (Load) followed by a modifier which identifies exactly what is to be loaded into the ACC. The ACC is a small area of memory (it can be a bit, byte or word) which is used to temporarily store data which is going to be, or has been processed.

In order to copy the first 8 digital inputs to the 8 digital outputs, we must first load the value of the 8 inputs into the ACC, so type:

```
LD    BI0
```

Be sure to indent and insert a tab between the operator (LD) and operand (BI0).

The operand BI0 can be broken down into 3 parts, B meaning that we are to load a byte of data, I meaning its an input and 0 which identifies which byte to load. Thus we are loading the Byte - Input - 0 which is the value of the first eight physical digital inputs in the module.

Now that we have loaded the value of the inputs into the ACC, we must now move this value to the outputs, which we do with a ST (Store) operator. ST is the opposite of LD. Type:

```
ST    BO0
```

The operand BO0 can also be broken down into 3 parts, B meaning that we are to store a byte of data, O meaning its an output and 0 which identifies which byte to store. Thus we are storing the value in the ACC to Byte - Output - 0 which is the value of the first eight physical digital outputs in the module.

Now we need a bit of syntax to end the programme so type:

```
EP
```

EP just means end of programme.

Your programme should now look like this:

```
#DEFINE DI_CNT 1
#DEFINE AI_CNT 4
MAIN:
    LD    BI0
    ST    BO0
    EP
```

Congratulations, you have made your first B-CON programme. All you need to do now is compile the source text into code and download it into the module.

Compiling

Having now created your programme, first save it to disk, then from the "Compile" menu select "Compile". A box will pop up in the middle of the screen to inform you of the progress of the compilation and to report any warnings or errors. You will have to be quick to see it in this example as such a short programme will not take long to compile.

If you have made any mistakes the box will remain on the screen and you can see how many errors you have made. Clicking "OK" will then bring up a screen giving you a clue as to what the error is.

Lets put an error into the programme so that we can see this feature.

On line:

```
LD    BI0
```

remove the B before the I0 and add a period after the 0 followed by another 0, so that the programme now looks like this:

```
#DEFINE DI_CNT 1
#DEFINE AI_CNT 4

MAIN:
    LD    I0.0
    ST    BO0
    EP
```

Run the compiler again and you will see that 1 error has been detected. Click "OK" and you will be informed of the programme details, the number of the line in which the error is situated and what the error is. In this case "Type of Operand", which have been mismatched.

More on Operators and Operands

Operators

So far we have used only 2 operators, LD and ST. B-CON has many operators, a full list of which, and a detailed explanation of each, is printed in the B-CON User Manual. We shall use some more operators in further exercises below.

Operands

When we introduced into the programme our deliberate error, we were informed that the error was "Type of Operand". There are different types of operand and in this case they were mismatched. Operands can be explained as follows:

Operands are made up of three parts or labels:

- The "operand format" which can be a bit, byte (B) or word (W).

- The "data type" which can be an input (I), output (O), marker (M) or constant (C).

- The "value" or address which can be binary, decimal or hexadecimal.

For now will ignore markers and constants for the data type, these will be explained later, and we will ignore completely hexadecimal (if you understand hex you won't need these simple exercises).

So what went wrong above? With the instruction:

```
LD    I0.0
```

we loaded into the ACC the value of the first physical input in the module (input 0.0). This is a single bit. With the instruction:

```
ST    BO0
```

we then tried to store this single bit into a byte. This you can't do. Operands must be matched.

Handling data 1 bit at a time.

To handle a single I/O value or bit, the first part of the operand (the operand format) is omitted. Only the data type and value are included, but the value is extended by a period and an extra digit. The extra digit identifies the individual bit within the byte.

The operand "I0.0" can be broken down into:

I = input

0 = byte 0 (the first block (or byte) of 8 DI in the module)

.0 = bit 0 (the first DI in the second block (or byte) in the module)

And the operand "I1.0" can be broken down into:

I = input

1 = byte 1 (the second block (or byte) of 8 DI in the module)

.0 = bit 0 (the first DI in the second block (or byte) in the module)

examples of other valid operands are:

I0.5 - Input number 5 in the first byte of digital inputs.

I1.1 - Input number 1 in the second byte of digital inputs.

O0.0 - Output number 0 in the first byte of digital outputs.

O0.3 - Output number 3 in the first byte of digital outputs.

O1.2 - Output number 2 in the second byte of digital outputs.

Handling data 1 byte at a time

To handle a byte (8 bits) of I/O all 3 parts of the operand are used. The operand format, data type and value.

The operand "BI0" can be broken down into:

B = byte

I = input

0 = byte 0 (the first 8 or byte of DI in the module)

examples of other valid operands are:

BI2 - the second byte of digital inputs.

BO0 - the first byte of digital outputs.

Handling data 1 word at a time

To handle a word (16 bits) of I/O all 3 parts of the operand are used. The operand format, data type and value.

The operand "WI0" can be broken down into:

W = word

I = input

0 = word 0 (the first 16 or word of DI in the module)

examples of other valid operands are:

WO0 - the first word (16 bits) of digital outputs (our example only has 8DO)

WI2000 - the first analogue input in the module (the address of AIs start at 2000)

WI2004 - the third analogue input in the module

WO2000 - the first analogue output in the module
(our example does not have any AO)

Downloading

Now that you have compiled your source text into code, without errors, your next step is to download it into the module and start the programme running.

From the "Compile" menu, select download and a box will appear showing you the progress of the code being downloaded into the module and finally informing you that the download was successful. B-CON will always download the last source text that has been compiled.

If the download is not successful, check your serial/Bitbus connection to the module and also check that the DIL switches on the module are set correctly, on an RTU8 switch 9 should be off and switch 10 on.

Finally start the programme running in the module by selecting "Start" from the "Compile" menu. A box will appear to inform you that the programme is now running.

You will probably forget to do this at some time, so if ever your module appears not to be running correctly, check that the B-CON programme has been started. The "System" LED on the Module will be off if the programme has not been started.

Test your programme by driving the inputs on the RTU8 and watching the corresponding outputs light up.

Analogue values

Analogue values use a word, or at least 12 of the 16 bits in a word, to record the analogue value in digital form. This is called 12 bit resolution.

Using 12 bits the analogue value can range from 0 - 4095. Thus if the analogue input is a 0 - 10V input, when there is 0V on the input the digital value will be 0 and when there is 10V on the input the digital value will be 4095.

It is this digital value that is used in B-CON when handling analogue values.

Further exercise

Now lets get a bit more ambitious.

It is a good idea when making a B-CON programme to first make a functional specification, a list of what tasks the programme/module is required to perform.

Our functionality specification for this exercise is as follows:

If both DI 0 and DI 1 are true, set DO 0 true.

If either DI 0 or DI 1 are true, set DO 1 true.

If AI 0 is greater than 5V, set DO 2 true. (assume AIs are 0 - 10V input).

Start as we did before, with the define statements, main and ep, so that your screen looks like this:

```
#DEFINE DI_CNT 1

#DEFINE AI_CNT 4
```

MAIN:

EP

Our first requirement is: If both DI 0 and DI 1 are true, set DO 0 true. So type:

```
LD      I0.0
```

to load the value of DI 0 into the ACC. Now we need to also look at the value of DI 1, but we can't just load it, or it will overwrite the value of DI 0 in ACC. So we use another operator, the AND command, which allows you to load more than one value into the ACC. It actually puts the second value into an extension of the ACC, called a stack. So type:

```
AND     I0.1
```

to load the value of DI 1 into the stack. The AND operator will now compare the value in the ACC (DI 0) and the value in the stack (DI 1) and if both are true (1) then it will pace a bit value of one in the ACC. If both are not true, it will place a value of 0 in the ACC.

So now we have ascertained if both DI 0 and DI 1 are true and the result is in the ACC. We now have to set DO 0. So type:

```
ST      DO0
```

DO 0 will now be driven true, if both DI 0 & DI1 are true and we have fulfilled the first part of our functionality specification. Your screen should look like this:

```
#DEFINE DI_CNT 1
#DEFINE AI_CNT 4


MAIN:
    LD      I0.0
    AND     I0.1
    ST      O0.0
    EP
```

At this point save to disk and compile the programme, just to see if you have made any mistakes.

Our second requirement is, If either DI 0 or DI 1 are true, set DO 1 true.

This is achieved by using an OR operator, which works in a similar way to the AND statement except that it puts a 1 in the ACC if either of the inputs is true and a 0 in the ACC for all other combinations ie. Both true or both false. So now insert a blank line (breaking up the programme thus, makes it easier to understand) and type:

```
LD      I0.0
OR      I0.1
ST      O0.1
```

Your screen should look like this:

```
#DEFINE DI_CNT 1
#DEFINE AI_CNT 4
MAIN:
LD      I0.0
AND     I0.1
ST      O0.0
LD      I0.0
OR      I0.1
ST      O0.1
EP
```

Again save to disk and compile the programme, just to see if you have made any mistakes.

Our third requirement is, if AI 0 is greater than 5V, set DO 2 true.

The first thing we need to do is to load the value of AI 0 into the ACC. So type:

```
LD      WI2000
```

remember that analogue values use a word, so W must be put in front of the I and that the address of the first AI (AI 0) is 2000. We now need to see if the value of AI 0 is greater then 5V. So type:

```
GT      WC2048
```

the GT (greater than) operator looks at the value in the ACC and if it is greater then its operand (WC2048) then it will set a value of 1 into the ACC or if it is not greater than the operand it will set a value of 0 into the ACC. But what you say is this operand, WC2048. Where did that come from? WC2048 is the operand for 5V. Don't worry about it now, it will be covered in the very next section. Let's carry on with our programme. We now have a 1 or a 0 in the ACC as a result of the GT operator and we can use that to set DO2, just as we did the AND/OR operators. So type:

```
ST      O0.2
```

Your screen should look like this:

```
#DEFINE DI_CNT 1
#DEFINE AI_CNT 4
```

MAIN:

```
LD      I0.0
AND     I0.1
ST      O0.0

LD      I0.0
OR      I0.1
ST      O0.1

LD      WI2000
GT      WC2048
ST      O0.2
EP
```

Again save to disk and compile the programme.

Congratulations again, you have just made your second B-CON programme. Go ahead and download it into the module, just as you did with the first programme and test it.

Constants

When you have to compare an input to a fixed value, 5V in the above example, or you need to set an output to fixed value you need to use a constant.

The letter C designates the operand data type of a constant, in the same way as I is an input and O is an output. C can be also prefixed with the operand format of W for a word value, B for a byte value or the operand format omitted in the case of a bit value.

Thus the constant WC2048 used in the example above can be broken down into:

W	=	word
C	=	constant
2048	=	50% of the range of an analogue input (0 - 4095) or 50% of the 0 - 10V input = 5V.

A bit constant (C) can have a value of 0 or 1, a byte constant (BC) can have a value of 0 - 255 and a word constant (WC) can have a value of 0 - 65535. When a constant is used in connection with an analogue input or output the value will always be between 0 - 4095, because the AI/AO only uses 12 of the 16 bits in the word.

Please note that B-CON does not support a decimal point. All analogue values are integers, but can be signed + or -, see B-CON User Manual for details. As negative values are rarely used in B-CON programmes their use will not be further discussed in this guide.

Other valid constants include:

C1	-	bit value 1
C0	-	bit value 0
BC50	-	byte value 50
BC255	-	byte value 255
WC0	-	word value 0 (the minimum value of an AI/AO)
WC4095	-	word value 4095 (the maximum value of an AI/AO)

Choosing an operator

So far we have used only a limited number of operators, LD, ST, AND, OR and GT. There are many more operators available in B-CON, but how do we choose which one to use?

In most cases your functional specification will tell you which operator to use. You may have noticed with the above exercise that the type of operator was defined in the functional specification:

If both DI 0 and DI 1 are true, set DO 0 true.

(you used an AND operator)

If either DI 0 or DI 1 are true, set DO 1 true.

(you used an OR operator)

If AI 0 is greater than 5V, set DO 2 true.

(you used a GT operator)

Its easy if you start with a functional specification.

Remember some operators require 2 operands, the first is usually already in the ACC (Op.1) and the second is declared after the operator in the normal manner (Op. 2).

The most commonly used operators are set out below. A list of all operators is available in the B-CON User Manual.

Command (operator)	Format	Description
LD	all	operand is loaded into ACC
LDN	all	operand is negated and loaded into ACC
ST	all	ACC contents are loaded into operand
STN	all	ACC contents are negated and loaded into operand
AND	bit	logic AND function, result in ACC
ANDN	bit	OP2 negated, logic AND function, result in ACC
OR	bit	logic OR function, result in ACC
ORN	bit	OP2 negated, logic OR function, result in ACC
XOR	bit	logic XOR function, result in ACC
XORN	bit	OP2 negated, logic XOR function, result in ACC
JMP	bit	unconditional jump to label
JMPC	bit	jump to label if ACC is condition logic 1
JMPCN	bit	jump to label if ACC is condition logic 0
ADD	byte, word	arithmetical add operation, result in ACC
SUB	byte, word	arithmetical subtract operation, result in ACC
MUL	byte, word	arithmetical multiply operation, result in ACC
DIV	byte, word	arithmetical divide operation, result in ACC
GT	byte, word	compare greater than (>) ACC to operand, result in ACC
GE	byte, word	compare greater than or equal to (>=) ACC to operand, result in ACC
EQ	byte, word	compare equal to (=) ACC to operand, result in ACC
LT	byte, word	compare less then (<) ACC to operand, result in ACC

LE	byte, word	compare less than or equal to (<=) ACC to operand, result in ACC
LOG	all	Data logging function, only for the RTU8
MOV	all	move data from op1 to op2

Markers

In the exercises above we have only been loading the value of a physical input and storing to physical outputs, but there are other places you can load from and store to. They are called markers.

Markers are storage cells or registers within the memory of the module, into which values can be placed.

Altogether there are 1024 bytes of markers, designated as BM0 – BM1023 (BM = Byte Marker), and they can be used to store bit, byte or word values.

Examples:

M50.0

BM55

WM60 (even numbers only)

BM0 – BM39 are reserved for use by the module firmware and are not available to you, but BM40 - BM511 , which are reset at power on, can be used at will. BM512 - BM1023 are battery backed and can also be used at will. However BM600 – BM999 are used by the firmware, in RTU8s having a second serial port (Port B).

Markers are particularly useful when you are manipulating data, such as storing an input value which you wish to monitor for a change of status. In this case you could load the input value and store it to a marker. The next time you load the same value you can use a Compare operator to look at both the newly loaded value and the older value in the marker and decide if the input has changed status.

Control of Outputs

When we made the define statements above, it was mentioned that it was unnecessary to make such statements for digital or analogue outputs. The reason for this is that the operational PC should not be allowed to have direct access to the outputs in a module. If it

did have such access then conflicts could occur if the PC and the B-CON programme both tried to set an output at the same time.

To prevent such conflicts only the B-CON software should be allowed to have control of the outputs and the PC must pass instructions through the B-CON programme to the outputs. This is done by using YO registers.

So far we have only used DI/DO, AI/AO and marker registers, but there are other registers, namely Y and Z registers, both of which come as input (YI, ZI) and output (YO, ZO).

YI/YO registers are used as a storage space for derived values, in a similar way to a marker can be used, but the difference between a marker and a Y register is that the operational PC can read and write to a Y register, but not to a marker.

ZI/ZO registers are used in only a few of the System 2000 modules, such as Gateway modules where data imported into the module by a serial link is stored.

As far as the B-CON programme is concerned these Y and Z registers are treated exactly the same as D and A registers. ZI/ZO registers have an address from 4000 - 5999 and Y registers an address from 6000 - 7999.

Lets try using a Y register to allow the operational PC to set the outputs of RTU8. Insert the following lines in the programme you have already made:

```
#DEFINE YO_CNT 1
```

(with the other define statements) The define statement allows the operational PC to access the Y register.

```
MOV  BI6000  BO0
```

MOV is the Move operator and its has 2 operands. The first defines where to move the data from, in this case the first Y input register, and the second to define where to move the data to, in this case the first byte of digital outputs. The More operator actually copies the value to be moved. The value is not deleted from its original location. The above Move command is equivalent to:

```
LDMI6000
STBO0
```

The operational PC can now write data to the Y input register and the B-CON programme will move this value to the outputs. The reason we define a YO register but write to a YI input register is that the define statement is relative to the PC where the data will be placed in a YO register and then transferred to the module where it is placed in a YI input register. An output from the PC is an input in the module.

Your screen should now look like this:

```
#DEFINE DI_CNT 1
#DEFINE AI_CNT 4
#DEFINE YO_CNT 1

MAIN:
LDI0.0
AND  I0.1
STO0.0

LDI0.0
OR    I0.1
STO0.1

LDWI2000
GTWC2048
STO0.2

MOV  BI6000  BO0
EP
```

If you now look at the programme you have made, you may notice that there is a potential problem. The 3 store commands are setting the first 3 digital outputs and with the move command you have just given the operational PC the ability to do the same. If both tried to set the same output to a different status, a problem would occur.

To overcome this you could use a logic statement, such as an AND or OR operand, to decide which of the two has priority.

LOG operator (RTU8/DL modules only)

Let us now look at the use of an operator crucial to the use of an RTU8, the LOG command or operator.

The RTU8 has the ability to log data, with a time/date stamp into non-volatile memory within the module. This data can be retrieved periodically by the operational PC.

It is important that we are able to select when and what data is logged into the non-volatile memory. If we logged all the I/O values, all of the time we would soon fill up the memory, despite it being up to 480Kbytes in size.

To control the logging we use the LOG command, with which we can define when a log should start/stop and which data should be logged. Up to 32 different log files can be defined.

The LOG command is a bit different from the operators we have encountered so far in that it has four operands:

LOG op1, op2, op3, op4

Operand 1 (op1) is the trigger. This is a bit value which when set true will start the log file logging.

Operand 2 (op2) is the label or name of the log file. It is a BC (byte constant) value and can range from BC0 - BC31. 32 log files in total.

Operand 3 (op3) defines which data to start logging from. It is a word value, the first word of data you wish to log. This can be actual inputs or outputs or a marker.

Operand 4 (op4) defines how many words of data should be logged from the starting point defined in op3. It is a BC (byte constant) value, but its value defines words of data, not bytes.

The operands are separated by a comma (but not after op4), a space or tab.

Examples:

LOG I0.0, BC0, WI0, BC1

LOG =	operator
I0.0=	Trigger, log file will start logging when digital input 0.0 (the first DI) is set true.
BC0=	Label, this is log file zero, the first log file.
WI0=	Data start point. The first word of DI in the module.
BC1=	How many words of data to log from WI0, in this case one.

LOG m20.0, BC1, WI2000, BC4

LOG = operator
m20.0 = Trigger, log file will start logging when marker m20.0 (first bit in marker 20) is set true.
BC1 = Label, this is log file one, the second log file.
WI2000 = Data start point. The first AI in the module.

BC4 = How many words of data to log from AI 1, in this case all 4 AI.

LOG O0.0,BC2, WO0,BC1

LOG = operator
O0.0 = Trigger, log file will start logging when digital output 0.0 (the first DO) is set true.
BC2 = Label, this is log file two, the third log file.
WO0 = Data start point. The first word of DO in the module.
BC1 = How many words of data to log from WO0, in this case one.

Thus the above 3 log files will log all the status of all the inputs and outputs of the RTU8 when their trigger conditions are fulfilled.

Further Exercise

Our functional specification is:

If DI 1 is true

Log the status of all four AI

Type the following:

```
#DEFINE DI_CNT 1
#DEFINE AI_CNT 4
#DEFINE YO_CNT 1

MAIN:
LOG  I0.0  BC0  WI2000   BC4
EP
```

This simple programme will fulfil the functional specification. Go ahead and compile, download and test it using the RTU8 log upload facility, to upload the log file to the PC (see IOTOOL32 getting started guide).

One thing you might have noticed when testing the above is that once you start the log file logging, by setting DI 1 to true, all four analogue inputs will be logged every scan (100 msecs). This is find if you need to log to that resolution but if you don't need to make such a fast record how do you slow down the logging and conserve the data storage capacity?

The answer lies in a marker, M17, in which the RTU8 provides time interval signals.

M17	
Bit	Time interval
0	0.1 seconds
1	1 second
2	10 seconds
3	1 minute
4	10 minutes
5	1 hour
6	10 hours
7	not used

So if we use the command:

```
LOG  M17.2   BC0  WI2000   BC4
```

M17.2 will provide a pulse every 10 seconds to trigger the log file, which will then log the value of all four analogue inputs every 10 seconds. But this log file will now run continuously. We have lost the control of DI 1. How do we now get DI 1 to again start and stop the log file?

The answer lies in the next section. Jump operators.

Jump operators

Jump operators are very common in B-CON programmes. They give you the ability to jump over part of the programme i.e. Not execute part of it. They are the logic equivalent of "IF xxx then yyy".

There are 3 different jump operators:

JMP - unconditional jump to label

JMPC - jump to label if condition is logic true

JMPCN - jump to label if condition is logic false

JMP will always make a jump.

JMPC or JMPCN will read the value in the ACC and depending its condition a jump is or is not made to a pre-defined label.

Jump operators have one operand, the label.

A label is the point to which the jump is made. It is similar to the MAIN: instruction, which designates the start of the programme. In fact MAIN is a label, but a mandatory one. Labels can be any name you choose, except MAIN. They can be up to 8 characters in length, either letters or figures, but they must start with a letter and are terminated with a colon (:). They should not be indented.

Examples of jump statements:

```
JMP     LABEL1
JMPC    LABEL2
JMPCN   LABEL3
```

Lets solve the above problem using a jump operator. Start your programme as usual, load the value of DI 1 into the ACC and insert the log operator (for operation every 10 seconds):

```
#DEFINE DI_CNT 1
#DEFINE AI_CNT 4
#DEFINE YO_CNT 1

MAIN:
  LDI0.0
  LOG  M17.2   BC0  WI2000   BC4
  EP
```

Lets insert a label, called "label1" into our programme:

```
#DEFINE DI_CNT 1
#DEFINE AI_CNT 4
#DEFINE YO_CNT 1
```

```
MAIN:
   LDI0.0
   LOG        M17.2      BC0      WI2000    BC4

LABEL1:
   EP
```

Now lets insert our jump operator. Our functionality specification calls for the log file to be started if the value of DI 1 is true, therefore insert:

```
   JMPCN   LABEL1
```

between the Load and LOG operators. Note that you don't put a colon after the label when use you the label name as an operand. Your programme should now look like this:

```
   #DEFINE DI_CNT 1
   #DEFINE AI_CNT 4
   #DEFINE YO_CNT 1

MAIN:
   LD          I0.0
   JMPCN       LABEL1
   LOG         M17.2      BC0      WI2000    BC4

LABEL1:
   EP
```

The operation of this programme can be explained thus:

LD	I0.0	-	loads the value of DI 1 into the ACC
JMPCN	LABEL1	-	reads the value in the ACC and if that value is false will jump to LABEL1, missing out the LOG command. If the value in the ACC is true the jump will not be made and the next line, the LOG command will be executed.
LOG etc.		-	The LOG command.
LABEL1:		-	The label for the jump operator.

Go ahead, compile, download and test it using the RTU8 log upload facility, to upload the log file to the PC.

It is important to understand that jump statements are used to jump over parts of the programme and are not used to jump to them. For instance:

```
   #DEFINE DI_CNT 1
    #DEFINE AI_CNT 4
    #DEFINE YO_CNT 1
MAIN:
   LD  I0.0
   JMPC      LABEL1
LABEL1:
   LOG       M17.2     BC0       WI2000     BC4
   EP
```

In this programme using JMPC in place of JMPCN will make a jump to LABEL1 if the value of DI 1 is true and the LOG command will be executed, but if the value of DI 1 is false the jump will not be made and the programme will continue to be executed one line at a time, including the LOG command. Thus control of the LOG command from DI 1 will be lost.

Control of Dial-up (RTU8 and Modbus remote master and slave modules only)

RTU8 and Modbus remote modules have the ability to communicate via the Public Switched Telephone Network (PSTN) or GSM cellular systems . The operational PC may dial-up the module to upload data, access I/O or download new configuration programmes. The module may also dial-up the PC under predefined conditions, such as an alarm.

Such alarms and the resultant dial-up are controlled by the B-CON programme. The actual routines for dial-up and control of the attached modem are built-in to the firmware of the module and need no further action from the B-CON programme other then to start the communication.

This is controlled by a marker, BM2, into which a value is placed to initiate a dial-up.

Four values can be loaded into BM2:

0 - Standby

1 - Dial-up (PSTN)

2 - Hang-up

3 - Send SMS message, GSM modems only

(see SMS Getting Started Guide)

Try the following simple programme:

```
#DEFINE DI_CNT 1
#DEFINE AI_CNT 4
#DEFINE YO_CNT 1

MAIN:
  LD   BI0
  ST   BM2
  EP
```

This programme copies the digital value of the first 8 digital inputs into the BM2 register. If DI 0 is set true a dial-up will be instigated. Setting DI0 back to false and setting DI 1 to true will then cause a hang-up.

In most programmes it is only necessary to instigate the dial-up by placing BC1 in BM2. This value of BC1 need only be in BM2 for one scan, the firmware only needs to read it once. It is therefore good practice to reset BM2 at the start of each scan.

Use:

```
MOV   BC0  BM2
```

at the start of the source text, immediately after MAIN:

The firmware, via the com count will handle the hang-up. (see the IOTOOL32 Getting Started Guide)

There are other markers associated with PSTN/SMS communications including:

BM3 in which B-CON can select any one of the 50 telephone numbers, pre-stored in the module, as the number to dial.

BM4 which contains the status of the communications and from which B-CON can ascertain if the dial-up has been successful, data has been passed etc.

BM5 which is the communication counter, keeping a count of how many transmissions have been made in any one communication between the module and the PC. The firmware uses the communication counter

Making life easier

When making small programmes you may remember what each line of the instruction list means, but when making more complex and longer programmes, especially if they are written over a period of time, it is often difficult to recall just what you implemented in the programme.

In this case the "remark" (/) is particularly useful.

Putting a forward slash (/) before any text causes that text to be ignored by the compiler. It is only a notation.

So for the programme we created above we add notations as follows:

```
#DEFINE DI_CNT 1        /defines the number of DI
#DEFINE AI_CNT 4        /defines the number of AI
#DEFINE YO_CNT 1        /defines the number of YO
MAIN:                                   /start of programme
   LD        I0.0                       /load value of DI 1
   JMPCN     LABEL1                     /jump to Label1 if DI 1 = 0
   LOG       M17.2  BC0  WI2000 BC4     /log AI 1- 4 every 10 sec's.
LABEL1:                                 /label for JUMPN
   EP                                   /end of programme
```

It is good practice to use notations. It will make life a lot easier, doit!

There is another way of making life easier for yourself, by assigning names to inputs, outputs, registers etc., instead of using an address. This is done by using the same "define" statements we have already used.

Thus I0.0 (the first digital input) could be assigned the name DI1 with the following statement:

```
#DEFINE DI1  I0.0
```

similarly WI2000 (the first analogue input) could be assigned the name AI1 with the following statement:

```
#DEFINE AT1  WI2000
```

so that our programme could now look like this:

```
#DEFINE DI_CNT 1                /defines the number of DI
#DEFINE AI_CNT 4                /defines the number of AI
#DEFINE YO_CNT 1                /defines the number of YO
#DEFINE DI1  I0.0               /name DI1
#DEFINE AT1  WI2000             /name AI1
```

```
MAIN:                                   /start of programme
   LD       DI1                         /load value of DI 1
   JMPCN    LABEL1                      /jump to Label1 if DI 1 = 0
   LOG      M17.2  BC0                  AI1   BC4   /log AI 1- 4 every 10 sec's.
LABEL1:                                 /label for JUMPN
   EP                                   /end of programme
```

If you find it difficult to remember the I/O addresses this could be useful. It is particularly useful to define markers with a name, so that you can remember what they are used for.

End of the beginning

Having got this far and completed the simple exercises above, you will now be in a good position to understand how a B-CON programme is created. You will not be an expert programmer, but you will be well on the path to becoming so.

B-CON has many more features than the ones we have discussed above and these are explained in the B-CON User Manual. Some of these features are quite complex and at first sight you may find it difficult to understand them, but don't worry about that, the vast majority of B-CON programmes are written using only the simpler commands. You may never have cause to use the more complex commands.

B-CON is a programming tool for the engineer and professional programmer alike.