Table of Contents

1 Array.....1
- 1.1 Definition.....1
- 1.2 Example.....1

2 Begin.....2
- 2.1 Syntax.....2
- 2.2 Description.....2
- 2.3 Example.....2

3 Break.....3

4 Byte.....4
- 4.1 Definition.....4

5 Call.....5
- 5.1 Syntax.....5
- 5.2 Description.....5
- 5.3 Example.....5

6 Case.....6

7 Clone.....7
- 7.1 Syntax.....7
- 7.2 Description.....7
- 7.3 Example.....7

8 Const.....8
- 8.1 Syntax.....8
- 8.2 Description.....8
- 8.3 Example.....8

9 Continue.....9

10 Debug.....10
- 10.1 Syntax.....10
- 10.2 Description.....10
- 10.3 Example.....10

11 Declare.....11
- 11.1 Syntax.....11
- 11.2 Description.....11
- 11.3 Example.....11

12 Default.....12

13 Dup.....13
- 13.1 Syntax.....13
- 13.2 Description.....13
- 13.3 Example 1.....13
- 13.4 Example 2.....13

14 Elif.....14

15 Else.....15

16 Elseif.....16

17 Elsif.....17

18 End.....18
- 18.1 Syntax.....18
- 18.2 Description.....18
- 18.3 Example.....18

19 Float.....19
- 19.1 Definition.....19

20 For.....20

Table of Contents

21 Frame.....21
- 21.1 Syntax.....21
- 21.2 Description.....21
- 21.3 Example.....21

22 From.....22

23 Function.....23
- 23.1 Syntax.....23
- 23.2 Description.....23
- 23.3 Example.....23

24 Global.....24
- 24.1 Syntax.....24
- 24.2 Description.....24
- 24.3 Example.....24

25 Goto.....25

26 If.....26
- 26.1 Syntax.....26
- 26.2 Description.....26
- 26.3 Example.....26

27 Import.....28
- 27.1 Syntax.....28
- 27.2 Description.....28
- 27.3 Example.....28

28 Include.....29
- 28.1 Syntax.....29
- 28.2 Description.....29
- 28.3 Example.....29

29 Int.....30
- 29.1 Definition.....30

30 Jmp.....31
- 30.1 Syntax.....31
- 30.2 Description.....31
- 30.3 Example.....31

31 Local.....32
- 31.1 Syntax.....32
- 31.2 Description.....32
- 31.3 Example.....32

32 Loop.....33

33 Offset.....34

34 OnExit.....35
- 34.1 Syntax.....35
- 34.2 Description.....35
- 34.3 Notes.....35
- 34.4 Example.....35

35 Pointer.....37
- 35.1 Definition.....37
- 35.2 Example.....37

36 Private.....39
- 36.1 Syntax.....39
- 36.2 Description.....39
- 36.3 Example.....39

37 Process.....40
- 37.1 Syntax.....40
- 37.2 Description.....40
- 37.3 Local variables as parameters.....40

Table of Contents

37 Process

37.4 Example....41

38 Program....42

38.1 Syntax....42

38.2 Description....42

38.3 Example....42

39 Public....43

39.1 Syntax....43

39.2 Description....43

39.3 Example....43

40 Repeat....44

41 Return....45

41.1 Definition....45

41.2 Example....45

42 Short....46

42.1 Definition....46

43 Step....47

44 String....48

44.1 Definition....48

44.2 Example....48

45 Struct....49

45.1 Definition....49

45.2 Example....49

46 Switch....50

46.1 Syntax....50

46.2 Description....50

46.3 Example....50

47 To....52

48 Type....53

48.1 Datatype declaration....53

48.2 ProcessType....54

49 Until....56

50 Varspace....57

50.1 Definition....57

51 Void....58

51.1 Definition....58

52 While....59

53 Word....60

54 Yield....61

1 Array

1.1 Definition

<datatype> <array name>**[**<upperlimit>**]** [= <values>]

Arrays are datatypes, consisting of a *range* of variables of the *same type*. The range is `0..upperlimit`, meaning there are `upperlimit+1` elements in the array.

The initializing values start at the first (*0th*) element and go up from there (see example).

1.2 Example

1.2.1 Mutliple ints

```
int lottery[9]; // an array of 10 ints
```

Use them like:

```
lottery[0] = 1;
lottery[5] = 35;
lottery[1] = lottery[5];
```

1.2.2 Multiple types

Consider, using Type:

```
Type _point;
    float x;
    float y;
End

_point point[2] = 1,2,// an array of 3 points at positions (1,2), (3,4) and (5,6)
                  3,4,
                  5,6;
```

Use them like:

```
point[0].x = 0;
point[1].y = 54.2;
point[2].x = point[0].x;
point[1].x = point[2].y = 23.2;
```

1.2.3 Multiple structs

See Struct example.

2 Begin

Up to Basic Statements

2.1 Syntax

Template:Syntaxdocbox **Begin**

[<main code>]

[**OnExit**

[<exit code>]

] **End**

2.2 Description

Begin is a reserved word to indicate the start of the code part of a program, process or function. The end is indicated by End. The OnExit statement can be used in between.

2.3 Example

```
Process Main
Begin // Start the main code part of the main process
    proc1(); // create new instance of proc1
End

Process proc1()
Begin // Start the main code part of the process
End

Function int func1()
Begin // Start the main code part of the function
    return 0;
End
```

Used in example: end, process, function

Template:Keywords

3 Break

1. REDIRECT Loops#Manipulating_a_loop

4 Byte

4.1 Definition

BYTE

Bytes are whole numbers ranging from 0 to 2^8-1 (0 to 255). This is because a **byte** uses 8 bits (1 byte) to denote its value. A byte is the smallest datatype directly accessible in nowadays memory.

5 Call

5.1 Syntax

Template:Syntaxdocbox **call** <label> ;

5.2 Description

The **call** command jumps to the given label inside a function or process until it comes across a return statement. When this happens, it jumps back to call statement and resumes after it.

5.3 Example

```
import "mod_say"

Process Main()
Begin

    say(my_function(1));

End

Function my_function(int value)
Private
    int ret;
Begin

    Jmp real_begin;

jumping:
    ret = 300;
    return;

real_begin:
    ret = 100;
    if(value == 1)
        Call jumping;
    end
    ret += 200;
    return ret;

End
```

Used in example: process, function, jmp, **call**, return

The output of this example is `500`, when `value` is 1. This example show 500 because the input value is a one and it causes that goes to the jumping label inserting a 300 and adding a 200 after.

Template:Keywords

6 Case

1. REDIRECT switch

7 Clone

7.1 Syntax

Clone

<sentences>

End

7.2 Description

The **clone** command creates a copy of the actual process which is called a "child process." The original process is then called the "parent process".

Only the child process will run the sentences between the keyword CLONE and the keyword END.

7.3 Example

```
import "mod_key";
import "mod_map";
import "mod_video";
import "mod_proc";

Process Main()
Begin

    squares();

    repeat
        frame;
    until(key(_ESC));

    let_me_alone();

End

Process squares()
Private
    int advance;
Begin

    graph = map_new(5,5,16);
    map_clear(0,graph,rgb(255,0,255));
    advance = 1;

    clone
        graph = map_clone( 0, graph );
        map_clear(0,graph,rgb(255,255,255));
        advance = 2;
    end

    loop
        x += advance;
        frame;
    end
    map_unload(0,graph);

End
```

Used in example: key(), map_new(), map_clear(), rgb(), map_unload() This example shows two squares. One is the child process, that is the white, and the other is the parent process.

Template:Keywords

8 Const

Up to Basic Statements

8.1 Syntax

Template:Syntaxdocbox **Const**

[<constants>]

End

8.2 Description

Const is a reserved word used to initiate the declaration of constants. Terminating the declaration block with an End is needed when the Const statement is not used in conjunction with the main code of the Program.

When declaring constants inside this construct, it's now allowed to explicitly name the type of the constant, i.e. you only have to assign the constant the value you want (see the example).

For a list of predefined constants, see this page.

8.3 Example

```
Const // Declare constants here
    myInt = 4;
    myString = "hello";
End

Process Main()
Begin
End

Const // Declare constants here
End
```

Template:Keywords

9 Continue

1. REDIRECT Loops#Manipulating_a_loop

10 Debug

10.1 Syntax

Debug ;

10.2 Description

Debug is a reserved word used to tell Bennu to go into debug mode, only if the DCB was compiled with debug information (compiler option `-g`). If the module mod_debug was imported as well, the console is immediately invoked and one can begin tracing from the debug statement.

Here's a handy page about debugging a Bennu program.

10.3 Example

```
Function int debug_warning(string warning)
Begin
    say("Warning: " + warning);
    debug;
    return 0;
End
```

Used in example: say(), **debug**

Template:Keywords

11 Declare

11.1 Syntax

Template:Syntaxdocbox **Declare** [**Function** | **Process**] [<returntype>] <name> **(** [<parameters>] **)**

[**Private**

<private variables>
End]
[**Public**
<public variables>
End]

End

11.2 Description

Declare is a reserved word used to declare a process or function before its actual code. This can be useful if the function or process needs to be known before it is actually defined, like when the function returns something other than an int or when the publics of the process need to be accessed before the definition. By default, the returntype of a process or function is an int.

When using this statement, a few things can be defined about the process/function:

- If it's a process or function
- Its returntype
- The parameters of the process or function
- The public variables of the process or function
- The private variables of the process or function

The first three are defined when using the statement Declare, while the last two are defined within the Declare block.

11.3 Example

```
Declare Process example_process()
    Public // Declare public variables for the process example_process
        int public_int;
        string public_string;
    End
/*  The Process definition handles this section
    Private // Declare private variables for the process example_process
        int private_int;
    End
*/
End

Declare Function string example_function( int param_int)
    Private // Declare private variables for the process example_process
        int private_int;
    End
End

Process example_process();
/* The Declare handles this section.
Public
    int public_int;
    string public_string;
*/
Private
    int private_int;
Begin
    Loop
        frame;
    End
End

Function string example_function( int param_int)
Begin
    return "";
End
```

Template:Keywords

12 Default

1. REDIRECT switch

13 Dup

13.1 Syntax

Template:Syntaxdocbox **Dup** [**(** <value> **)**] **;**

13.2 Description

The **dup(value);** return a space data filled with the given value. For example a 10 dup(0) return an array of 10 elements, all with a zero value.

13.3 Example 1

```
import "mod_say";

global
    array[] = 10 dup(0);

begin
    say ( sizeof(array)/sizeof(array[0]) );
end
```

This example prints 10 because the size of array is ten and the size of the first element is one.

13.4 Example 2

```
import "mod_say";

global
    int array[5] = 1 , 4 dup(0);
    int arrayPosition;
begin
    for ( arrayPosition = 0 ; arrayPosition < 5 ; arrayPosition = arrayPosition + 1 )
        say ( array[arrayPosition] );
    end
end
```

This example prints:

1 0 0 0 0

As it duplicates "0" four times in the array data. Template:Keywords

14 Elif

1. REDIRECT if

15 Else

1. REDIRECT if

16 Elseif

1. REDIRECT if

17 Elsif

1. REDIRECT if

18 End

Up to Basic Statements

18.1 Syntax

End

18.2 Description

End is a reserved word used to terminate loads of stuff, such as if-statements, loops, begin-statements, etc...

18.3 Example

```
Process Main()
Begin
    If(something)
       If(something_else)
           Loop
               frame;
           End //ends the loop
       End //ends the second if-statement
    End //ends the first if-statement
End //ends the program block (begin keyword)
```

Used in example: process, if, loop, **end**

Template:Keywords

19 Float

19.1 Definition

FLOAT

Floats are floating point numbers ranging from about -10^38.53 to about 10^38.53. This is achieved by dividing 32 bits (4 bytes) in a certain way, with a certain precision. A float is used for operations in which both very large and small numbers are used, while rounding is not permitted. Unlike ints or shorts, a **float** actually has decimal digits. Their accuracy is about 7 decimal digits.

20 For

1. REDIRECT loops#For ... End

21 Frame

21.1 Syntax

Template:Syntaxdocbox **Frame** [**(** <percentage> **)**] **;**

21.2 Description

The **frame;** command tells the interpreter when a process is done for one game cycle. When the **frame;** is reached, the screen is updated. If there are several processes running, the screen is updated once every process has reached its **frame;** statement.

It is commonly used in loops of processes that should do something like moving around by a certain amount of pixels per game cycle (or per frame).

A possibility is to adjust the amount of cycles to wait. **frame(100);** would wait one cycle (100%), same as **frame;**. However **frame(200);** will wait two cycles (200% means the frame statement provides for 200% frame). So **frame(50);** will wait a half cycle or otherwise said, it will make a loop run twice per frame.

21.3 Example

```
Process Main()
Begin

    square();

    Repeat
        frame;
    Until(key(_ESC))

    exit();

End

Process square()
Begin

    graph = new_map(5,5,16);
    map_clear(0,graph,rgb(255,255,255));

    Loop
        x += 2 * (key(_right)-key(_left));
        frame; //<-vital part
    End

End
```

This example process would give you a square you can move around the screen by 2 pixel before it gets showed again, before the game cycle is over, before the **frame;** happens. If there would be no **frame;** in the loop, it would just run forever and the interpreter would wait forever for the **frame;**, which would result in freezing.

Template:Keywords

22 From

1. REDIRECT loops#From ... End

23 Function

Up to Basic Statements

List of Functions

23.1 Syntax

Template:Syntaxdocbox **Function** <returntype> <name> **(** [<parameters>] **)**
[**Public**

[<public variables>]

End]
[**Private**

[<private variables>]

End]
Begin

[<function code>]

[**OnExit**

[<exit code>]

]
End

23.2 Description

Function is a reserved word used to start the code of a function.

A function is a subroutine to which one or more of the following apply:

- it receives parameters
- it acts on the parameters
- it processes data located elsewhere
- it returns a value

The difference between a function and a process is that the calling process or function waits until the function is completed. When a process or function calls a process, it doesn't wait. This means that, even when the called function contains frame statements, the calling function or process still waits for the function to finish. This is shown in this tutorial.

For a list of functions, see this list of functions.

23.3 Example

```
Function int addInts( int a , int b )
Private // Declare private variables here
Begin // Start the main functioncode
    return a+b;
End // End the main functioncode
```

addInts(3,6); will return 9. One can see that the function does indeed:

- receive parameters.
- act on the parameters.
- return a value.

Template:Keywords

24 Global

Up to Basic Statements

24.1 Syntax

Template:Syntaxdocbox **Global**

[<global variables>]

End

24.2 Description

Global is a reserved word used to initiate the declaration of global variables. Terminating the declaration block with an End is needed when the Global is not used in conjunction with the main code of the Program.

For a list of predefined global variables, see this page.

24.3 Example

```
Global // Declare global variables here
End

Process Main()
Begin
End

Global // Declare global variables here
End
```

Template:Keywords

25 Goto

Equals to Jmp

26 If

Up to Control Flow Statements

26.1 Syntax

Template:Syntaxdocbox **IF (** <condition> **)**

[<code>]

(**ELSEIF (** <condition> **)**

[<code>]) *

[**ELSE**

[<code>]]

END

26.2 Description

If statements are used to control the flow of your program by means of checking conditions.

```
if( <condition1> )
    // code1
elseif( <condition2> )
    // code2
elseif( <condition3> )
    // code3
else
    // code4
end
// code5
```

If at the time the program reaches this **if**-codeblock *condition1* is true, then *code1* will be executed and then *code5*. If *condition1* is false, the program will go to the next **elseif** or and check if that is true or false: if *condition2* is true, *code2* and then *code5* is executed. If *condition2* is false, the program will check the next **elseif** and do the same thing over, until the program reaches the **else** or the **end**. If an **else** is present, the code in the **else**-block will thus be executed when all other conditions are false.

26.3 Example

26.3.1 Execute function

This is a little example of how to make a function perform a certain task depending on a command.

```
Function int execute(String command, int* params)
Begin

    if( command == "SEND" )
        NET_Send(atoi(params[0]),params[1]);
    elseif( command == "RECV" )
        NET_Recv(atoi(params[0]));
    elseif( command == "INIT" )
        NET_Init(atoi(params[0]),atoi(params[1]),atoi(params[2]));
    elseif( command == "QUIT" )
        NET_Quit();
    else
        // error: command unknown
        return -1;
    end

    return 0;

End
```

26.3.2 Movement

Movement with **If**s.

```
Loop
    if(key(_up))    y-=5; end
    if(key(_down))  y+=5; end
    if(key(_left))  x-=5; end
    if(key(_right)) x+=5; end
End
```

Of course, this is faster:

```
Loop
    y += 5 * (key(_down )-key(_up  ));
    x += 5 * (key(_right)-key(_left));
End
```

Template:Keywords

27 Import

27.1 Syntax

import " <filename> **"**

27.2 Description

Imports a Bennu DLL with name *filename* into the program, which allows the usage of additional functionality in a Bennu program. For more information, see the article on DLLs.

27.3 Example

```
import "mod_say"
import "my_dll";

Process Main()
Begin
End
```

Used in example: **import**

Template:Keywords

28 Include

28.1 Syntax

include " <filename> **"**

28.2 Description

When the compiler reaches an **Include** statement, it continues compilation at the included file (usually *.INC) and when it's done resumes compiling from the **Include** statement. In other words, these files contain code that gets inserted at the place of inclusion.

This is very handy for breaking up your code into pieces. The handling of video in one include file, audio in another, game logic in another, etc. This makes code more maintainable and understandable; moreover it makes code reusable. The video handling include file you made for one game can be used for another game (if it was coded in a generic fashion) without spitting through the whole sourcecode of the first game.

Also headers can be used to import DLLs and possibly give a little more functionality to that DLL. For example Network.DLL uses a .INC header file to assure the DLL is only imported once during compilation and provides a little more functionality.

28.3 Example

main.prg

```
// The code in "bar.inc" will be processed first:
include "bar.inc"

import "mod_say"

Process Main()
Private
    int barcode;
Begin
    barcode = bar();
    say(barcode);
End
```

bar.inc

```
import "mod_rand"

Function int bar()
Begin
    return rand(0,10);
End
```

Used in example: **include**, import, write_int(), key()

Template:Keywords

29 Int

29.1 Definition

INT

Ints (short for integer, meaning wholes), are whole numbers ranging from -2^31 to 2^31-1 (-2147483648 to 2147483647). This is because an integer uses 32bits (4 bytes) to denote its value using the Two's complement system.

30 Jmp

30.1 Syntax

Template:Syntaxdocbox **jmp** <label> ;

30.2 Description

The **jmp** command jumps to the given label inside a function. Unlike the call command, there is nothing more to it.

30.3 Example

```
import "mod_say"

Process Main()
Begin

    say(my_function(1));

End

Function my_function(int value)
Private
    int ret;
Begin

    Jmp real_begin;

jumping:
    ret = 300;
    return;

real_begin:
    ret = 100;
    if(value == 1)
        Call jumping;
    end
    ret += 200;
    return ret;

End
```

Used in example: process, function, **jmp**, call, return

The output of this example is `500`, when `value` is 1. This example show 500 because the input value is a one and it causes that goes to the jumping label inserting a 300 and adding a 200 after.

Template:Keywords

31 Local

Up to Basic Statements

31.1 Syntax

Template:Syntaxdocbox **Local**

[<local variables>]

End

31.2 Description

Local is a reserved word used to initiate the declaration of local variables. Terminating the declaration block with an End is needed when the Local is not used in conjunction with the main code of the Program.

For a list of predefined local variables, see this page.

31.3 Example

```
Local // Declare local variables here
End

Process Main()
Begin
End

Local // Declare local variables here
End
```

Template:Keywords

32 Loop

1. REDIRECT Loops#Loop ... End

33 Offset

1. REDIRECT Pointer

34 OnExit

Up to Basic Statements

34.1 Syntax

Template:Syntaxdocbox **Begin**

[<main code>]

[**OnExit**

[<exit code>]

]
End

34.2 Description

The **OnExit** statement can be used between the Begin and End statements in the Process or Function. When the Program, Process or Function is killed, the exit code starts. This can be easily used to free any memory used by the exiting process.

34.3 Notes

Be advised, Frame statements are interpreted as Frame(0) statements, to ensure exit code of the killed instance finishes the same frame as the instance is killed. Code like

```
Loop frame; End
```

will cause Bennu to freeze, as frame(0); doesn't allow switching to another instance.

34.4 Example

34.4.1 Basics

```
import "mod_proc"
import "mod_say"

Process proc1()
Begin // Start the main code part of the process
    say("Proc created");

    // Run indefinitely (or until killed)
    Loop
        frame;
    End
OnExit // Start the exit code of the process
    say("Proc killed!");
End

Function int func1()
Begin // Start the main code part of the function
    say("Func created");
    return 0;
OnExit // Start the exit code of the function
    say("Func killed!");
End

Process Main()
Private
    int p;
Begin // Start the main code part of the main process
    p = proc1(); // create new instance of proc1
    func1();
    say("Main is here");
OnExit // Start the exit code of the main process
    say("Main is killed");
    signal(p,S_KILL);
    say("Main is at the end");
End
```

Used in example: loop, end, process, function, frame, say()

Resulting console messages:

```
Proc created
Func created
Func killed!
Main is here
Main is killed
Main is at the end
Proc killed!
```

34.4.2 Resource cleanup

A good use for this is the following:

```
Process ship()
Begin
    graph = map_new(20,20,8);
    Loop frame; End
OnExit
    map_unload(0,graph);
End
```

Used in example: map_new(), map_unload(), graph, loop, frame, end

Template:Keywords

35 Pointer

35.1 Definition

35.1.1 Statement

Declaration of a pointer:
<datatype> POINTER <pointername>
<datatype> * <pointername>

Assignment of a value to the location pointed to:
POINTER <pointername> = <value>;
* <pointername> = <value>;

35.1.2 Concept

Pointers, are used to point to a location in memory. It uses 32 bits (4 bytes) so it can map 4GB of memory into bytes. **Pointer**s can point to any datatype: ints, shorts, strings or even usermade datatypes. However, using a `struct pointer my_pointer` is pointless, because the compiler has no knowledge of the elements inside the struct pointing to, since it doesn't know which struct is meant, so this is invalid. `MyStruct pointer my_pointer`, where MyStruct is an existing struct, is also not valid, because MyStruct is not a datatype. The only way to have something like a `struct pointer my_pointer` is to use Type as seen in the example.

35.2 Example

```
import "mod_say"

Type _point
    int x;
    int y;
End

Type _person
    string name;
    int age;
End

Global
    _person Person;
End

Process Main()
Private
    int my_int;
    int* my_int_pointer;
    _point myPoint;
    _person* personPointer; // possible, because _person is infact a datatype
    //Person* personPointer; // not possible, because Person is not a datatype
Begin

    my_int_pointer = &my_int;

    my_int = 3;
    say(my_int);
    say(*my_int_pointer);

    *my_int_pointer = 4;
    say(my_int);
    say(*my_int_pointer);

    setXY(&myPoint);
    say(myPoint.x);
    say(myPoint.y);

    personPointer = &Person;
    personPointer.name = "Mies";
    say(Person.name);
    say(personPointer.name);

End

Function int setXY(_point* p)
Begin
    p.x = 3; // this is actually (*p).x = 3, but . can be used like this
    p.y = 5; // this is actually (*p).y = 5, but . can be used like this
    return 0;
```

```
End
```

Used in example: say(), key(), Type, Global, Private, point

The & (offset) operator, when used with pointers, returns a void pointer to a variable. In the example it returns an int pointer to the variable my_int. The * (pointer) operator, when used with pointers, makes it so the pointer variable is not accessed, but the variable it's pointing to. In the example it changes access from my_int_pointer to my_int.

36 Private

Up to Basic Statements

36.1 Syntax

Template:Syntaxdocbox **Private**

[<private variables>]

[**End**]

36.2 Description

Private is a reserved word used to initiate the declaration of private variables. Terminating the declaration block with an End is not necessary, but is possible. Parameters of a function or process will be considered a private variable with the initiated value of the passed argument.

36.3 Example

```
Process My_Process();
Public
Private // Declare private variables here
Begin
End
```

Template:Keywords

37 Process

Up to Basic Statements

37.1 Syntax

Template:Syntaxdocbox **Process** <name> **(** [<parameters>] **)**
[**Public**

[<public variables>]

]
[**Private**

[<private variables>]

]
Begin

[<main code>]

[**OnExit**

[<OnExit code>]

]
End

37.2 Description

Process is a reserved word used to start the code of a process. If *name* is *Main*, that process will be started at the start of the program.

A process is a subroutine to which one or more of the following apply:

- it receives parameters
- it acts on the parameters
- it processes data located elsewhere

In addition to these possibilities, a process *always* has a frame; statement. The difference between a function and a process is a process is treated as a separate thread. This means one can't let a process return a value like a function, as the father process continues its code as well, as soon as the process hits a frame; statement or when the code is done. When that happens, the process 'returns' its ProcessID and continues the code (in the next frame).

When the frame; statement is reached in the code, a number of other local variables are defined or updated not only of the new process, but also of related processes. These are:

- The father variable of the new process.
- The son variable of the father process (updated).
- The bigbro variable of the new process.
- The smallbro variable of the processes called by the father immediately before the new process was called (updated).
- The son and smallbro variables are also defined of the new process, but do not yet carry values.

When there are no more processes alive, the program ends.

37.3 Local variables as parameters

When a process is declared with parameters that are actually local variables, arguments for these parameters will initialise those local variables. This may sound strange, but an example will clear things up.

For example, consider the local variables x, y, z, file and graph. To create a process to move a game sprite around, you can declare it as follows:

```
process Ship (x,y,z,file,graph)
begin

    // move left 1 pixel per frame
    repeat
        x -= 1; // move 1 pixel to the left
        frame; // this process is done for this frame, wait for the next
    until (x<0);

end
```

Calling the process with e.g. `Ship(300,100,5,0,1);` will have the Ship appear at the coordinates (300,100) on Z-Level 5 with the Sprite No.1 in the file number 0. The ship will move left until it leaves the screen. You can change movement by changing the x/y value of the process and animate the ship by changing the graph value.

37.4 Example

```
Process SpaceShip( int x, int y, int angle, int maxspeed, int maxturnspeed)
Public // Declare public variables here
Private // Declare private variables here
    int speed;
Begin // Start the main processcode
    graph = new_map(20,20,8);
    map_clear(0,graph,rgb(0,255,255));
    Loop
        speed+=key(_up)*(speed<maxspeed)-key(_down)*(speed>-maxspeed);
        angle+=(key(_left)-key(_right))*maxturnspeed;
        advance(speed);
        frame;
    End
OnExit // Start the exit code
    unload_map(0,graph);
End // End the main processcode
```

Now one can call this process for example by doing the following.

```
Process Main()
Begin
    SpaceShip(100,100,0,20,5000);
    Repeat
        frame;
    Until(key(_ESC))
    let_me_alone();
End
```

Used in example: new_map(), map_clear(), key(), advance(), unload_map(), let_me_alone(), **Process**, Begin, End, Loop, Repeat, graph, angle

And when the SpaceShip process ends - because the code of it reached the End or something sent an s_kill signal - the OnExit code starts. In this example it will unload the memory used for the created graphic. If there is no OnExit code, the process will just end.

This will make a SpaceShip with a cyan coloured block, able to move around the screen.

Template:Keywords

38 Program

Up to Basic Statements

38.1 Syntax

Program <programname> **;**

38.2 Description

Program is a reserved word used to begin your program. It's not needed to start a program with it.

It should be noted that this is for backwards compatibility only, because it doesn't actually do anything.

38.3 Example

```
Program example; // Name this program "example", which doesn't really matter

Process Main() // This process is started when the program is started
Begin
End
```

When the End of the main code is reached, the program exits, if there are no processes alive anymore, which is logical, as Bennu quits when there are no processes running and Main is a process as well. This process is just like any other process with the addition it gets called when the program starts. This means that you can also call the process using `main()`.

Template:Keywords

39 Public

Up to Basic Statements

39.1 Syntax

Template:Syntaxdocbox **Public**

[<public variables>]

[**End**]

39.2 Description

Public is a reserved word used to initiate the declaration of public variables. Terminating the declaration block with an End is not necessary, but is possible.

39.3 Example

```
Process My_Process();
Public // Declare public variables here
Private
Begin
End
```

Template:Keywords

40 Repeat

1. REDIRECT loops#Repeat ... Until

41 Return

41.1 Definition

Return [<value>];

Return is a reserved word used to return a value in a function. The returned value must be of the datatype specified as the returndatatype (see Function). By default, the returntype of a process or function is an int. When this statement is reached, the function in which it resides will stop execution and return the specified value. If a value was not specified, the ProcessID will be returned.

41.2 Example

```
Function string example_function()
Private
    string s;
Begin
    s = "Some string;
    return s;
End
```

Used in example: Function, Private, Begin, End, **Return**, String

42 Short

42.1 Definition

SHORT or **WORD**

Shorts or **Word**s are whole numbers ranging from 0 to 2^16-1 (0 to 65535). This is because a **short** or **word** uses 16 bits (2 bytes) to denote its value.

43 Step

1. REDIRECT loops#From ... End

44 String

44.1 Definition

STRING

Strings are a sort of character array, combining characters to form text. Because the length of a **string** is dynamic, adding them is done easily. Single and double quotes can be used to create strings.

44.2 Example

```
Program strings;
Private
    String name;
    String surname;
Begin
    name = "Yo";
    surname = "Momma";

    say(name + " " + surname + " has entered.");
    say('User logged on: "' + name + " " + surname + '"');

    Repeat
        frame;
    Until(key(_ESC))

End
```

Used in example: say(), key()

45 Struct

45.1 Definition

Struct <struct name>

[Members]

End

Structs are datatypes, able to contain variables of all datatypes.

To address a member of a struct, use the "." operator: *<structname>.<membername>*. Like all datatypes, one can have a whole range of them, as displayed in the example (also see Array).

There are two ways to fill a struct on declaration:

- Per member
- Afterwards, like Arrays.

See the examples on how to do it.

45.2 Example

Structs can be handy for many aspects of programming.

45.2.1 Grouping of variables

This is for clarity and to avoid colliding variable names.

```
Struct Fileinfo
    String name;
    String path;
End
```

Note that the struct fileinfo is a predefined global variable.

Maybe you want to group some other data, like settings of the screen:

```
Struct Window
    int width = 320;
    int height = 200;
    int depth = 8;
End
```

or (using other initializing syntax):

```
Struct Window
    int width;
    int height;
    int depth;
End = 320,200,8;
```

This example can also be done by defining your own type.

45.2.2 Multiple identical data groups

```
Struct Ship[9]
    int x;
    int y;
    int speed;
    int angle;
End
```

There are 10 'Ship's now. The data can be accessed like:

```
Ship[0].speed++;
Ship[8].angle = 0;
```

46 Switch

Up to Control Flow Statements

46.1 Syntax

Template:Syntaxdocbox **Switch (** <value> **)**

(**Case** <value> **:**

[<code>]
End) *
[**Default:**
[<code>]
End]

End

46.2 Description

A *Switch* is used to control the flow of a program by means of comparing a value to other values and executing the code associated with the correct value.

```
switch ( <value> )
    case <value1>:
        // code1
    end
    case <value2>:
        // code2
    end
    default:
        // code3
    end
end
```

When the **switch** is reached it will compare *value* with the values in the **case**s, going from top to bottom. When a case matches, that code is executed and the **switch** is exited. This is different than the **switch** in C and probably more languages, because there the **switch** is only exited when a break is reached or when the **switch** ends. In Bennu there is no break; for the switch, though.

A value in a **case** can also be a range: `<lowervalue>..<uppervalue>`. Both the *lowervalue* and the *uppervalue* are part of the range.

You can also specify multiple values in a **case**, separated by a comma: `<value1>,<value2>,...<code>. These values can also be ranges.`

46.3 Example

A scoretext function. Notice the **default**: when the points can be 0..100, that code should never be executed. However, an error can occur and blurting an error to the user is not that fancy, so this is a way of showing to the programmer there is an error, but still the user gets some message. In such cases, **default** can be handy. Of course that code could just as easily have been put under the **switch** with the same result, in this case, because every case does a return.

```
Function String scoretext( int points )
Begin

    Switch( points )
        Case 100:
            return "1337 |-|4><0|2!!1";
        End
        Case 90..100:
            return "Awesomely cool, dude!";
        End
        Case 80..90:
            return "You're getting the hang of it!";
        End
        Case 60..80:
            return "Not too shabby, mate.";
        End
        Case 50..60:
            return "Practice some more.";
        End
        Case 30..50:
            return "Dude...weak.";
        End
        Case 1..30:
            return "That's just awful";
        End
        Case 0:
            return "No points? n00b!";
        End
        Default:
            return "I dunno what you did, but...";
        End
    End

End
```

Template:Keywords

47 To

1. REDIRECT loops#From ... End

48 Type

48.1 Datatype declaration

48.1.1 Definition

Type <name>

<variables>

End

Creates a new datatype. It's handled as if it were a struct, so the declared variables are members of the struct.

While it's a convention to use a '_' as first character in the name of a datatype, it's not mandatory.

When used as an argument in a function or process, the parameter is not a copy, but the variable itself, as shown in the first example, and any change made to the parameter is also changed in the argument. It's more elegant to use a pointer though, as it also displayed.

48.1.2 Example

A file with name and path. Note that the assignment `myFile2 = myFile;` makes a copy of *myFile* and throws it into *myFile2*, which is normal. But when it's used as an argument in a function, the parameter is not a copy but the _file itself.

```
Type _file
    String path;
    String name;
End

Process Main()
Private
    _file myFile;
    _file myFile2;
Begin

    myFile.path = "C:\";
    myFile.name = "autoexec.bat";
    say("1: " + myFile.path + myFile.name);

    myFile2 = myFile;
    myFile2.name = "config";
    say("1: " + myFile.path + myFile.name);
    say("2: " + myFile2.path + myFile2.name);

    setName(myFile,"pagefile");
    say("1: " + myFile.path + myFile.name);

    setName2(&myFile2,"pagefile");
    say("2: " + myFile2.path + myFile2.name);

    Repeat
        frame;
    Until(key(_ESC))

End

Function setName(_file f, string name)
Begin
    f.name = name;
End

Function setName2(_file* f, string name)
Begin
    f.name = name; // this is actually (*f).name = name, but . can be used like this
End
```

Used in example: say(), key(), Pointer

This will result in something like:
Template:Image

A point with x and y.

```
// Declare the type _point
```

```
Type _point
    float x;
    float y;
End

// Declare the function distance(), because the function returns a datatype
// other than int, so it needs to be declared before usage.
Declare float distance(_point a,_point b)
End

Process Main()
Private
    _point p1,p2;
Begin

    p1.x = 15.3;
    p1.y = 34.9;
    p2.x = 165.4;
    p2.y = 137.2;

    write(0,0,0,0,"Distance: " + distance(p1,p2));
    drw_line(p1,p2);

    Repeat
        frame;
    Until(key(_ESC))

End

Function float distance(_point a, _point b)
Begin
    return sqrt( (a.x-b.x)*(a.x-b.x) + (a.y-b.y)*(a.y-b.y) );
End

Function int drw_line(_point a, _point b)
Begin
    return draw_line( a.x , a.y , b.x , b.y );
End
```

Used in example: write(), key(), sqrt(), draw_line()

This will result in something like:
Template:Image

48.2 ProcessType

48.2.1 Definition

Type <processname>

Acquires the processTypeID of a processType or function. This can be useful for example with the functions get_id() and signal().

48.2.2 Example

```
Program example;
Private
    proc proc_id; //int could be used too
Begin

    // Start 3 proc's
    proc();
    proc();
    proc();

    // Display all alive proc's
    y = 0;
    while( (proc_id=get_id(type proc)) )
        write(0,0,(y++)*10,0,"proc: " + proc_id);
    end

    // Wait for key ESC
    Repeat
        frame;
    Until(key(_ESC))

End

Process proc()
Begin
    Loop
        frame;
    End
```

```
End
```

Used in example: get_id(), write(), key()

This will result in something like:
Template:Image

49 Until

1. REDIRECT loops#Repeat ... Until

50 Varspace

50.1 Definition

VARSPACE

A **varspace** is a datatype of any datatype. When a function, like sort() or fread(), has a parameter of type **varspace**, it means it needs a variable of any type.

51 Void

51.1 Definition

VOID

Bennu doesn't have **void**s as such. But when we look at for example the function free(), we see that you can pass it a **void pointer**. This means, that you can pass it a pointer of whatever type you want; an **int pointer**, **word pointer** or even a **pointer pointer**. So in this case, **void** means "undefined".

There is another case in which **void**s can occur. This is when a function returns nothing, but this never happens in Bennu.

52 While

1. REDIRECT loops#While ... End

53 Word

1. REDIRECT Short

54 Yield

1. REDIRECT frame