FPL

User's Guide

Latest update: 30 June 1995, for version 12.6

An advanced flexible real-time ASCII text interpreting
multi-file shared library programming language very similar to C.

A General Introduction

FPL was initially programmed to become a programming language in the text editor programmed by FrexxWare called FrexxEd. It was intended to become an easy-to-learn but yet powerful script-language with which you should be able to control most of the happenings in the editor.

Today, FPL is an external shared library based programming language very close to C-interpreting very easily used and implemented by any program and lots of different operating systems (when writing this, it has been successfully compiled and tested under at least 8-9 different OSes).

A.1 About This Manual

This is the FPL LANGUAGE documentation. This deals with about everything that can be said about FPL from the user's view. For information about how to implement fpl.library in your own code, see the FPLlib.guide and fpl.doc files.

Many examples provided are from the FrexxEd environment, meaning that several of the used functions in the examples are FrexxEd specific and not a part of FPL. (FrexxEd is Copyright © 1992-1995 by FrexxWare.)

There should be sufficient information in this manual to allow every user to fully understand and use FPL. If it should prove insufficient, report it!

TECH Notes exist in some parts of the manual, explaining why I solved things the way I did and sometimes also how.

This documentation is written during a long time and during a lot of FPL updates. There might still be some old version thinking in this manual, but I hope it won't destroy your ability to create something good with FPL.

This manual does only handle the language FPL. When FPL is used in real life, it will always be used in a software which is host to FPL. That software will affect FPL a lot and you should read the FPL chapters of that software's documentation closely!

A.2 Contact the Author

For private matters/discussions/questions/ideas, drop me a email/netmail at our own BBS: The Holy Grail (+46-(0)8-6121258, FidoNet 2:201/328, running 28800 bps V34), or reach me at email: `Daniel.Stenberg@sth.frontec.se`!

Problems concerning FPL, the library, bug reports, other FrexxWare products or stuff like that, are dedicated to public message areas making it possible for everyone to share, learn and participate.

If you have any ideas about things you'd like FPL to support, handle or run, don't hesitate to contact me and share your visions. FPL is continuously developing and I need feedback to know in which directions you want it to go.

Swedish users of FPL can take advantage of the FidoNet echo mail area named R20_FPL. Available on backbone.

If there is enough interest shown, I will consider arranging a standard internet mailing list for FPL discussions.

All forthcoming updates and releases will be uploaded as fast as possible to the public ftp site known as AmiNet. (Thanks to Mattias Axelsson for his offering to do this on a regular basis!)

Source code is very much available and if you didn't get it in the package, get in touch!

Snail mail address:

Daniel Stenberg
Ankdammsgatan 36, 4tr
S-17143 SOLNA
Sweden

A.3 General syntax

- Every execution begins at the top of the program and interprets from left to right, downwards.
- All statements must be separated with a semicolon (;).
- FPL is case sensitive. That makes the following two variable names different:

```
hello Hello
```

- There is no maximum length of a script line. In fact, the entire program is indeed as good in one single line as in several lines.
- There is no line orientation in the language at all. Programs can be written in almost any way provided that you follow the syntax rules of the keywords and functions. Whitespaces, comments and new lines can be inserted anywhere to make the code more appealing to you.
- Comments are allowed everywhere and are written exactly as in C and Arexx as well as in C++; starting with a "/*" and ending with a "*/" with no nesting possibilities[1]:

```
/* This is a comment */
```

or starting with a "//" symbol and ending with a new line. Ex:

```
// This is also a comment
```

- Continuation of string lines is written with a backslash (\) character followed by a newline. Ordinary statements don't need any continuation character at all. Just as in C... ;-) Ex:

```
a=b             /* This is a fully working statement */
>
c;


b="hello\
world";         /* Continues a string assign.*/


output          /* Split the name and the arguments... */
("hi");

/* The code below works just fine, but writing much of this */
/* kind might make your code rather confusing! */
  output(\
  "h\
  ello\
   wor\
  ld");
```

(In 'C' you can continue *EVERYTHING* by simply writing a backslash and continue on the next line (thanks to the preprocessor that merges such lines into one for the compiler). FPL cannot perform the same if you're not using FPL together with a preprocessor that replaces

1. Since version 7, FPL can be made to accept nested comments.

such sequences (such as FPP, the FrexxWare Pre Processor).)

❏ All identifier names (labels/functions/variables) is limited to no more than 64 significant characters. More characters can be used, but identifiers with the same 64 first characters are considered identical.

❏ Identifiers can consist of both letters, numbers and underscores ("_") but must not begin with a number. Letters are the 26 characters from a to z and the 26 characters from A to Z.

❏ There is no kind of unconditional jump or goto in FPL. In languages like this (with different local levels), goto is most frequently used in the wrong way and the use of it should anyhow have been very limited (just look at the goto function in C). FPL doesn't need any goto/jump keyword.

❏ Do not, I repeat, do not rely on undocumented features in the language. FPL is constantly being changed and new error checks might be implemented in the next release. The strange feature you may find working in this version, might be a sever error in the next. Code as stated in the manual and your FPL code will have a much bigger change to stay accurate even when the version number ascends.

A.4 Funclibs

This works only when FPL is run under OSes that use shared memory, such as AmigaOS!

Overview

To enable third party programs to add functions to already running FPL sessions, the ‘funclib’ concept was invented. The inspiration source when designing the interface was to make it work like when using shared libraries (known as dynamic linking in some systems).

Using this technique, all FPL programmers can take advantage of functions that is placed in funclibs. By simply opening the funclib all its functions will exist and can get called. Any FPL program can open any funclib.

Funclibs could contain functions for bringing up requesters easier, for compression procedures, for serial port communication or for file handling. The limits are set by the funclib programmer, not by anyone else!

How to?

By opening the desired funclib with the openlib function, all function will be there. There’s nothing more to it than that! After you’ve called the functions you wanted the funclib for, you simply call closelib which concludes access to that funclib.

In some occasions, the host program might have already opened a funclib for you, and then you won’t need to open it and you shouldn’t close it.

A.5 Line Control

I encourage the use of preprocessors together with FPL. FPL interprets the ANSI C standard "#line" instruction as a line number/file name changer.

#line

A line control directive that supplies line numbers for FPL messages. It causes the next source line to be treated as having the specified number.

Syntax

```
# line { decimal_constant [ "file_name" ] }
```

In order for FPL to produce meaningful references to line numbers in preprocessed source, the preprocessor inserts #line directives where necessary (for example, at the beginning at after the end of included text).

A file name specification enclosed in double quotation marks can follow the line number. If a file name is specified, FPL views the next line as part of the specified file. If a file name is not specified, FPL views the next line as part of the current source file.

Note that the keyword 'line' is optional. The directive

```
# line 300
```

is equivalent to

```
# 300
```

Example:

On Amiga, an FPL source could be preprocessed using SAS/C 6.x by entering the following line:

```
"sc PPONLY KEEPLINES <FPL program>"
```

(note that if 'KEEPLINES' isn't specified, SAS/C won't output any #line instructions!)

A.6 Pragmas in FPL

Pragmas are specified by entering

`#pragma <keyword>`

first in a line, and where the <keyword> is a compiler specific instruction. FPL supports the following pragmas:

(NOTE: only use the pragmas if you are very certain of what they are doing, they alter settings done by the FPL implementor.)

Keyword	Description
cache	Make FPL keep this file in memory after execution for faster access (it may very well do it anyway, depending on the FPL setup).
nocache	Opposite to 'cache'. Make FPL to **not** keep this file in memory after usage, even if symbols were exported. The file will be loaded into memory each time it is to be used, and is flushed from memory when not in use.
reread	When executing cached files, the original file might be changed. This pragma will force this file to be reread if it is changed on disk while another version is cached and then executed.
noreread	The opposite to 'reread'. Force FPL to ignore if the file has been changed on disk or not.

A.7 Variables overview

❏ There exist a few kind of variables in FPL:

Type	Description
integer	holds a 32-bit signed numerical value. (maximum: 2147483647, minimum: -2147483648) Declarator: `int' or `long'
short	holds a 16-bit signed numerical value. (maximum: 32767, minimum: -32768) Declarator: `short'
char	holds an 8-bit signed numerical value. (maximum: 127, minimum: -128) Declarator: `char'
string	holds a contiguous sequence of characters. No length limit. Declarator: `string'.

Declare a variable by using the declarator followed by the symbol name and an optional initial assign. Multiple declarations can be done by comma separating them.
Example:

```
int a;
string b;
int a=2;
string foobar="ninja", foo, bar;
```

Variables not assigned when declared, equals zero or zero length strings.

❏ FPL includes NO floating point variables at all. The secret of still making proper calculations is the remainder operator. (Floating point numbers/expressions is expected to appear in a future FPL version.)

❏ Variables can be declared to exist in a few different ways:

Keyword	Description
static	makes the variable to be remembered until the next time this function is invoked.
const	constant variable. After the initialization assign, this cannot be changed.
auto, register	make a variable declared as a global to exist as non-global.
volatile	volatile variables exist in C. They are written back to memory each time they are changed. FPL variables are always written back when changed. Implemented to make C ports easier.
unsigned, signed	not implemented. These words are simply read and ignored.

These keywords can be used together with a declarator keyword in any combination. Examples:

```
static int foobar;
int auto register foobar;
string static foobar;
```

- Create declarator aliases by using the keyword `typedef'. Usage:

```
typedef <declarator> <symbol>;
```

After such a typedef, the alternative declarator can be used wherever the original declarator can be used.

- Scope of FPL Identifiers

The region where an identifier is visible in a program is referred to as the scope of the identifier. The scope of an identifier is determined by where the identifier is declared.

Block scope

The identifier's declaration is located inside a statement block. An identifier with block scope is visible from the point where it is declared to the closing brace (}) that ends the block.

You can nest block visibility. A block nested inside a block can contain declarations that redeclare identifiers declared in the outer block. The new declaration of the identifier applies to the inner block. The original declaration is restored when program control returns to the outer block. An identifier from the outer block is visible inside inner blocks that do not redefine the variable.

File scope

The identifier's declaration appears outside of any block, before the start of the program. An identifier with file scope is visible from the point where it is declared to the end of the source file.

Example:

```
void cool(int);
void stupid(int);

int a=-50;
int b=1000;
cool(a):
exit;

void cool(int a)
{
    stupid(a);
}

void stupid(int a)
{
    {
        int b=0;
    }
    output(b/a);
```

```
}
```

- All FPL variables must be declared before use and they must be declared first in the block. (Blocks are always started with an open brace and ended with return() or a close brace.) (NOTE: The technique is like the one used in common C)
  Ex:

  ```
  int a;
  for(a=0; a<10; a++) {
      int b=0;
      output(b);
  }
  ```

  is correct, but NOT this:

  ```
  char a;
  output(a);
  short b=5; // the line above broke the chain of declarations
  ```

- Variables declared within loop braces will be declared and assigned every loop, this is anyhow not good for execution speed! Ex:

  ```
  int a;
  for(a=0; a<6; a++) {
      int b=a*2;
      output(b, ", ");
  }
  ```

  This example will output() "0, 2, 4, 6, 8, 10 "!

- Multi dimensional variable arrays are supported. To create e.g twenty integers:

  ```
  int hello[20];
  ```

  (These will be accessible by the names hello[n], where n is a number from 0 to 19.)

  or

  ```
  int hello[4][5];
  ```

  (These will be accessible by the names hello[n][m], where n is a number from 0 to 3, and m is a number from 0 to 4.)

- Assign arrays using the {} operators. This example assigns var[3] to 100 and var[4] to 200:

  ```
  var[3]={100, 200};
  ```

  When declaring variables, the array assign always begins with the first member of the array, this line assigns the four members a[0] to a[3]:

  ```
  string a[4]={"hi", "how", "are", "you?"};
  ```

  The value of an array assign expression is always the last member of the assign list. The following example will output the word "world" on the screen:

  ```
  output(a[0]={"hello", "world"});
  ```

  Compound assigns on arrays are also valid! The following line adds 2, 3 and 4 to the variables length[3], length[4] and length[5]:

  ```
  length[3]+={2, 3, 4};
  ```

The value of this expression will be the last addition (length[5]+4).

The string append operator is also array friendly. The following example add strings to the strings names[1] and names[2].(The returning string will be names[2]+=" (stupid)".):

```
names[1]+={" (ill)", " (stupid)"};
```

Of course, this works with array using more than one dimension too:

```
int foo[2][3]={
    {0, 1, 2},
    {2, 3, 4}
};
```

On the fly, things like this can (of course) be used:

```
foo[1]^={{2}, {3, 4}};
```

- ❏ Attempts to create a variable with a name that is already being used by another variable in the same local level will result in an error.
- ❏ FPL supports variable referencing, more familiar to C programmers as pointers to variables. Currently, though, you cannot declare or assign them in any other way than through the parameters of a function call:

```
int foobar(int *barfoo) /* this function receives an integer reference */
{
     *barfoo = 5; /* assign the variable 'barfoo' is referencing! */
     barfoo = 5;  /* ILLEGAL /
}
int ninja;
foobar(&ninja); /* make foobar() assign the 'ninja' variable!
```

The same procedure is indeed possible to perform when referencing strings.

A.8 Strings

- Strings are always written enclosed within quotation marks (“).
- To read the ASCII value of a single character within a string, use square brackets in the form: name[index] where name is the name of the string variable (could of course also be an array member) and index is the column you want to check out. Index below zero or above the length of the source string will result in error. The following example outputs all ASCII codes in a string:

```
string  str = “hello world”;
int  ascii, n;
while ( ascii = str[n++] ) {
     output ( itoa(ascii) + “\n” );
}
```

  This example will output all ASCII values in column 2 in the strings in the array:

```
string  str[3] = {“hello”, “world”, “string”};
int  n;
for ( n = 0; n < 3; n++ ) {
     output ( str[n][2] + “\n” );
}
```

  Reading beyond the end of a string simply returns a zero, while reading a negative column will result in an error.
- Strings not enclosed within quotes are assumed to be variable names.
  Note the difference between:

```
output(“Hello”);
```

  and

```
output(Hello);
```

  (Example 1 outputs the actual string “Hello”, while example 2 outputs the contents of the variable *named* “Hello”.)
- Append strings to a string variable by using the ‘+=’ operator. Ex:

```
string a=”Hello “;
a+=”world”;
output(a);
```

  Outputs the string “Hello world” on the screen.

```
string a = “Hello “;
string b =”world”;
output( a + b );
```

- Special characters within strings should be symbolized with backslash and a character just as in the good old C standard:

Sequence	Description
\a	Alert (bell)

Sequence	Description
\b	Backspace
\f	Form feed (new page)
\n	New-line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\‘	Single quotation mark
\“	Double quotation mark
\?	Question mark
\\	Backslash
\xhh	hex number, where “hh” is a two digit hexadecimal value.
\nnn	octal number, where “nnn” is a three digit octal value.

A.9 Functions

If you are not acquainted to the use/calling of functions from FPL, read the general function usage paragraph on page 21.

In a standard FPL environment there is a lot of functions that the FPL programs can use. They are added to FPL in three different ways:

- ❏ The host program of FPL adds a lot of functions so that the programs really can do anything fun with the software. Such functions are called *external functions* and they can't be described in this document but will be found in the host programs documentation.
- ❏ *Inside functions* that are declared and defined in an FPL program. Any FPL program can create a function that can be called from another function and that is treated and looks just like all the other functions.
- ❏ The *Internal functions* are supplied by FPL itself. Such functions will always be there, no matter what the name or the purpose of the host program is.

A.10 Declare inside functions

This chapter handles FPL function declarations and definitions. A function declaration declares the format and existence of a function prior to its use. A function definition defines a function.

A function in FPL must always be declared (prototyped) or defined before it is used. Otherwise it doesn't know where to find the function and how to interpret its parameters.

Function Declaration (prototyping)

A function declaration establishes the name of the function, the type of result that the function returns and the types of the arguments expected by the function when you call it.

If the function does not return a value, declare it as a function returning void.

If the function is to be accessed from other program than this, it should be declared `export <the rest>'.

Example:

```
int func (int, string);
```

Function Definition

A function definition specifies the name, formal parameters, and body of a function. You must also specify the function's return type.

A function definition contains the following:

- A type specifier, which determines the type of value that the function returns. A function returning no value should be declared to return `void'. A function can have any type specifier.
- A function identifier, which provides the function with a name.
- A list of parameters that the function expects and their types.
- A block statement, which contains data definitions and code.

A function can be called by itself or by any function that appears in the same file as the function definition. If a function has been declared `exported', the function also can be called by functions that appear in other files, otherwise it can only be directly invoked from within the same source file.

The function definition or a declaration for the function must appear before, and in the same file as, a call to the function. All declarations for a given function must be compatible with the function definition. They must have the same return type and the same parameter types.

The following example is a complete definition of the function sum:

```
int sum(int x,int y)
{
    return(x + y);
}
```

The function sum returns int, and receives two values declared as x and y. The function body contains a single statement that returns the sum of x and y.

To indicate that a function accepts no parameters, use the keyword void as the type specifier in place of the parameter. For example:

```
stop(void) { }
```

In the following example, the function f() takes one integer parameter and returns no value, while g() expects no parameters and returns an integer.

```
void f(int); int g(void);
```

Function Body

The body of a function is a block statement. The following function has an empty body:

```
void stub1(void) { }
```

The body of the following function contains a definition for the integer variable big_num and a call to the function printf:

```
void largest(int num1, int num2)
{
    int big_num;
    if (num1 >= num2)
        big_num = num1;
    else
        big_num = num2;

    printf("big_num = %d\n", big_num);
    return 0;
}
```

Block statements are described in chapter *Statements*.

❑ Prototypes **MUST** be first in the program. Before any program starts and outside all braces to be global. Prototyping within the braces of a function makes the function only locally accessible.

A.11 General function use

Here follows the ground rules when using/calling a function in FPL programs.

❏ A function call has three major parts:
1 - The function identifier. The name of the function.
2 - A list of parameters. Most functions demands some kind of input.
3 - A return code from the function call.

In the example:

```
                    retval = foobar ( "hello", 100 );
Reference letter:   A        B      C D        E   F G H
```

Part	Description
A	The variable that received the return code of the function. It must be of the type that the function returns. If the function returns a string, the variable that receives the return code must be a string. A return code from a function can always be ignored. Often it is wise to check for progress, but it is always the choice of the programmer.
B	The function name. We call the "foobar" function in this example.
C	Always write the parameter list within parentheses. They tell FPL that this really is a function. Even functions without in any input parameters must be called with parentheses (but then without any parameters in between)!
D	This is the first parameter to the function. Apparently this function accepts a string type as first parameter. Any string variable or string expression is then a valid parameter.
E	Separate all parameters with a comma ",".
F	This is a second parameter. Apparently this function accepts an integer type as second parameter. Any integer variable or numerical expression is then a valid parameter.
G	Conclude the parameter list with a closing parenthesis. Obviously, this function is happy with two parameters.
H	End of statement is as always indicated with a semicolon ";".

❏ Internal and external functions might have some arguments optional and some functions may even accept parameter lists (an optional amount of parameters of a certain type). All functions must always at least be called with the number of required arguments as declared. Refer to the software docs.

❏ There are four kinds of arguments possible to pass to a function:
1. Strings - constant strings or expressions returning strings.
2. Numeric arguments - are likewise read as numerical expressions including everything true expressions consist of. An integer argument can be sent as any kind of "char", "int" or "short" and can be received by any one of those. FPL is tolerant when speaking about the mix of such.

3. String variable references - pointer to a string variable.
4. Numeric variable references - pointer to an integer variable. Just as with integer arguments, the use of the integers can be any of the three integer types “char”, “short” and “int”/ ”long”.

❏ All arguments that should be sent to and received in the function must be declared by comma separating “int”, “string”, “int *” or “string *”. (Using “char”, “short” or “long” is of course working too.)
“int” - sends an integer result of an expression.
“string” - sends a string result.
“int *” - sends a named integer variable.
“string *” - sends a named string variable.

Sending “int *” or “string *” makes the function able to change the contents of the variables used in the calling function.

All declared arguments are required. Optional arguments are not possible to declare.

C programmers see the obvious inspiration programming language.

A.12 Constants

Specifying constants in FPL can be done in several ways.

Numeric expression constants can be written as:

WHAT	HOW	EXAMPLE
octal number	an octal number with a zero prefix	012
binary number	binary number with a “0b” prefix	0b011010
hexadecimal number	a hexadecimal number with a “0x” prefix	0xDEADBEEF
decimal number	a number	129
ASCII code	a character within apostrophes	‘a’

String constants can also be written as:

WHAT	HOW	EXAMPLE
octal number	“\nnn” where nnn is an 1-3 digit octal number	“\12”
hexadecimal number	“\xhh” where hh is a two digit hexadecimal number	“\xea”
string	anything within quotes	“k i ll^e(rn/injax”

A.13 Statements

These are the statements that FPL includes:

- Block (on page 24)
- break (on page 45)
- continue (on page 48)
- do (on page 50)
- Expression (on page 25)
- for (on page 54)
- if (on page 56)
- Null (on page 25)
- return (on page 61)
- switch (on page 65)
- while (on page 67)

A.14.1 Block Statement

A block statement lets you to group any number of data definitions, declarations, and statements into one statement. All definitions, declarations, and statements enclosed within a single set of braces are treated as a single statement. You can place a block wherever a single statement is allowed.

All definitions and declarations occur at the beginning of a block before statements. Statements must follow the definitions and declarations. A block is treated as a single statement.

If you redefine a data object inside a nested block, the inner object hides the outer object while the inner block is processed. Defining several variables that have the same identifier can make a program difficult to understand and maintain. Therefore, you should limit such redefinitions of identifiers within nested blocks.

If a data object is usable within a block and its identifier is not redefined, all nested blocks can use that data object.

Initialization of an auto or register variable occurs each time the block is run from the beginning. If you transfer control from one block to the middle of another block, initializations are not always performed. You cannot initialize an external variable within a block.

Examples

The following example shows how the values of data objects change in nested blocks:

```
int main(void)
{
    int x = 1;                  /* Initialize x to 1 */
    int y = 3;

    if (y > 0)
    {
        int x = 2;              /* Initialize x to 2 */
        output("second x = ", x, "\n");
```

```
    }
    output("first x = ", x, "\n");
}
```

The example produces the following output:

```
second x = 2
first x = 1
```

Two variables named x are defined in main. The definition of x on line 5 retains storage throughout the execution of main. However, because the definition of x on line 10 occurs within a nested block, line 11 recognizes x as the variable defined on line 10. Line 13 is not part of the nested block. Thus, line 13 recognizes x as the variable defined on line 5.

A.15.2 Null statement

The null statement performs no operation;

SYNTAX

;

EXAMPLE

```
int i;
for(i=0; i<100; function(i++))
    ; /* null statement */
```

A.16.3 Expression statement

Expression statements performs some kind of evaluation of expression(s).

❑ Statements must feature an action or cause an error.

```
int a;        /**************************************/
a++;          /* These are examples of statements    */
a=2;          /* that include some kind of "action". */
go(a);        /**************************************/

2+2;          /**********************************************/
a;            /* These lines are NOT valid stand aloners */
(c>22)*3;     /* since they don't perform any kind of change.*/
a-14*a;       /**********************************************/
```

❑ Expressions containing the operators &&, || or ?: can be only partly evaluated depending on the results of the other parts. Ex, in the expression "(a || b)" a is true, and then FPL won't execute/evaluate b. Likewise "(a && b)" where a is FALSE, will never reach b. Also "a=b?c:d" will reach c only if b is true, and d only if b is false!!

A.17 Keywords

Detailed information is obtained in the keyword reference on page 43.

Summary:

❏ Loop statements are done very similarly as in C, using the ‘for’, ‘while’ or ‘do’ statements. They are used as in the C programming language: (with exception for the wonderful “else” statement in “while”...)

```
do { statement;} while ( expression )
while ( expression ) { statement ; } [ else { statement: } ]
for ( expression1 ; expression2 ; expression3 ) { statement; }
```

❏ A feature (which I’ve always been missing when coding C) has been added, and that’s `` `else' `` after `` `while' ``! If the condition never goes true, the ‘else’ statement will be executed.

❏ `continue` works 100% as in C.

❏ `break` is like break in C, but extended with an optional level argument.

❏ Take care of exceptions just as you do in C using statements like:

```
if ( expression ) { statement ; } [ else  { statement ; } ]
```

❏ Declare variables and functions by using the keywords:
`int`, `char`, `short` and `long` that all creates numerical variables.
`string` declares a string variable.
`resize` changes the size of an already existing array.
`export` tells FPL that the following global declaration is to be exported. Exported global symbols are accessible in any file.

For more variable declaration details, see the chapter ‘variables’ on page 12.

❏ Exit programs or functions by using:
`exit` or `return`

A.18 Hints and Tricks

Since I've created FPL and know about its inner workings, I'll try to sum a few words about what could be useful to think of when coding FPL.

General

Parenthesized expressions will help you from making mistakes in expressions due to lack of operator precedence knowledge.

Do not declare unnecessary variables or variables you don't have to use. Try to reuse already declared ones and stay away from the nasty variable declaration keywords!

Of course FPL is a perfect example of using a C programming preprocessor if you want preprocessor features like include files, macros and such goodies. FPL currently works perfect running preprocessed programs including the preprocessor instruction "#line".

Interpreting speed

FPL Performance philosophy is a structured programmer's nightmare coming true. What is said to be ugly or even dirty coding in C, in fact often is to prefer in FPL since it's interpreting one statement at a time.

Do as much as you can in as few statements as possible is a golden rule. Use as few function calls as possible. Try to append everything possible to the function's argument and call it only once.

Use compound and nested variable assigns! FPL has no compiler, which makes statements like "a=a+2; b=b+a; function(a);" interpret MUCH slower than "function(b+=(a+=2))" even though it might look better and be more readable. (If it was hard to code it should be hard to read, right?)

Avoid long and complicated string handlings. If you must use a string, try putting it in a variable and refer to that variable as much as possible. Especially loops will benefit a *LOT* in speed. Future version of FPL might deal with static strings (non-variable referenced) in a better way.

Bug hunting

Of course even FPL routines will be coded incorrect from time to time.

Comment your code. It's not hard, decreases performance only slightly and makes it so very much easier to change the code. And who knows, perhaps someone else would want to change it in the future!

Use the FPL debugger!

B Numeric Expressions

Numeric expressions are a basic part of FPL programs. Expressions are evaluated on the basis of the operators that the expressions contain and the contexts where the expressions can be used.

An expression can consist of...

B.1 Grouping and Evaluating

Two operator characteristics determine how operands group with operators: precedence and associativity. Precedence provides a priority system for grouping different types of operators with their operands. Associativity provides a left-to-right order for grouping operands to operators that have the same precedence.

You can explicitly state the grouping of operands with operators by using parentheses.

In the expression

```
a + b * c / d
```

the * and / operations are evaluated before + because of precedence. b is multiplied by c before it is divided by d because of associativity.

The following table lists the FPL language operators in their order of precedence. The operators are listed in order of precedence: primary operators have the highest precedence, and the comma operator has the lowest precedence. Operators that appear in the same group have the same precedence.

FPL's operator precedence is 100% ANSI C compatible.

Operator Precedence and Associativity (decreasing order):

Operator Type	**Associativity**	**Operators**
Primary	left to right	`(), function calls`
Unary	right to left	`++ -- - + ! ~ []`
Multiplicative	left to right	`* / %`
Additive	left to right	`+ -`
Bitwise Shift	left to right	`<< >>`
Relational	left to right	`< > <= >=`
Equality	left to right	`== !=`
Bitwise Logical AND	left to right	`&`
Bitwise Exclusive OR	left to right	`^`
Bitwise Inclusive OR	left to right	`|`
Logical AND	left to right	`&&`
Logical OR	left to right	`||`
Conditional	right to left	`? :`
Assignment	right to left	`= += -= *= /= <<= &= ^= |= >>=`
Comma	left to right	`,`

The order of evaluation for the operands of the logical AND (&&) and the logical OR (||) operators is always left-to-right. If the operand on the left side of a && operator evaluates to 0 (zero), the operator on the right side is not evaluated. If the operand on the left side of a || operator evaluates to non-zero, the operator on the right side is not evaluated.

Examples

The parentheses in the following expressions explicitly show how FPL groups operands and operators:

```
total = (4 + (5 * 3));
total = (((8 * 5) / 10) / 3);
total = (10 + (5/3));
```

FPL group operands with operators that are both associative and commutative in a simple left-to-right order. We group the operands and operators of the expression

```
total = price + prov_tax + city_tax;
```

in the following way (as indicated by parentheses):

```
total = ((price + prov_tax) + city_tax);
```

TECH NOTE:

The example above could have been evaluated the following ways too

```
total = ((price + city_tax) + prov_tax);
total = (price + (prov_tax + city_tax));
```

But that's not too good when using C, if you want to use expressions like

```
a = b++ + (b>10?10:20);
```

since you wouldn't know in which order the b variable is read and used... FPL reads and uses it in a left-to-right order.

B.2 Primary Expressions

All primary operators have the same precedence and have left-to-right associativity. See Grouping and Evaluating FPL Expressions on page 29.

B.3.1 Parenthesized Expressions

You can use parentheses to explicitly state how operands group with operators. The following expression does not contain any parentheses used for grouping operands and operators. The parentheses surrounding weight and zip-code form a function call. Notice how FPL groups the operands and operators in this expression:

Expression without Parentheses:

```
-discount * item + handling(weight, zipcode) > 10 * item
```

The following expression is similar, but contains parentheses that change how the operands and operators are grouped:

Expression with Parentheses:

```
(-discount * (item + handling(weight, zipcode))) > (10* item)
```

B.4.2 Function Calls

A function call is a primary expression followed by a parenthesized argument list. The argument list can contain any number of expressions separated by commas, or it can be empty.

For example:

```
stub()
overdue(account, date, amount)
notify(name, (date+5))
report(error, time, date, (num++))
```

The arguments are evaluated, and each parameter is assigned the value of the corresponding argument. Assigning a value to a parameter changes the value within the function, but has no effect on the argument.

B.5 Unary Expression

An unary expression contains one operand of scalar type and an unary operator. All unary operators have the same precedence. As indicated in the following descriptions, the usual arithmetic conversions are performed on the operands of most unary expressions.

❑ Bitwise Negation ~

❑ Decrement --

❑ Increment ++

❑ Logical Negation !

❑ Unary Minus -

❑ Unary Plus +

B.6.1 Increment ++

The ++ (increment) operator adds 1 (one) to the value of the operand. The operand receives the result of the increment operation.

You can place the ++ before or after the operand. If the ++ appears before the operand, the operand is incremented; then the incremented value is used in the expression. If you place the ++ after the operand, the current value of the operand is used in the expression; then the operand is incremented. For example:

```
play = ++play1 + play2++;
```

is equivalent to the following sequence of expressions:

```
play1 = play1 + 1;
play = play1 + play2;
play2 = play2 + 1;
```

In the C language they say “avoid using a variable more than once in an expression where the variable is incremented”. But that’s not necessary in FPL! I’ve created it to work as I wanted it, and so it does...

```
y = x(i) + i++;
```

Does work exactly as you think; first calls the function x(), adds the value of i and finally increases i.

B.7.2 Decrement --

Acts as the ++ operator , but instead of adding it’s subtracting!

B.8.3 Unary plus +

The + (unary plus) operator maintains the value of the operand.

The result of applying the unary plus operator to a signed operand is equivalent to the promoted type of the operand.

B.9.4 Unary minus -

The - (unary minus) operator negates the value of the operand.

The result of applying the unary minus operator to a signed operand is equivalent to the negative promoted type of the operand.

For example, if quality has the value 100, then -quality has the value -100.

B.10.5 Logical negation !

The ! (logical negation) operator determines whether the operand evaluates to 0 (false). If so, the operation yields the value 1 (true). If the expression evaluates to a nonzero value, the operation yields the value 0 (false).

If right is not equal to 0, the following two expressions are equivalent:

```
!right;
right == 0;
```

B.11.6 Bitwise negation ~

The ~ (bitwise negation) operator yields the ones complement of the operand. In the binary representation of the result, every bit has the opposite value of the same bit in the operand.

Suppose x represents the decimal value 5. The 32-bit binary representation of x is:

```
00000000000000000000000000000101
```

The expression ~x yields the following result, represented here as a 32-bit binary number:

```
11111111111111111111111111111010
```

The 32-bit binary representation of ~0 is:

```
11111111111111111111111111111111
```

B.12 Binary Expressions

A binary expression contains two operands separated by one operator.

Not all binary operators have the same precedence .

To ensure correct results, avoid creating expressions that depend on the order in which FPL evaluates the operands.

- Addition +
- Bitwise AND &
- Bitwise Exclusive OR ^
- Bitwise Inclusive OR |
- Bitwise Shift << >>
- Division /
- Equality == !=
- Logical AND &&
- Logical OR ||
- Multiplication *
- Relational < > <= >=
- Remainder %
- Subtraction -

B.13.1 Multiplication *

The * (multiplication) operator yields the product of its operands.

Because multiplication has both associative and commutative properties, the operands will be grouped in a left-to-right order. For example, the expression:

```
sites * number * cost
```

is interpreted in the following way:

```
sites * number) * cost
```

B.14.2 Division /

The / (division) operator yields the quotient of its operands.

If both operands are positive integers and the operation produces a remainder, FPL ignores the remainder. Thus, the expression 7 / 4 yields the value 1 (rather than 1.75 or 2).

The C language does not define how the compiler treats the quotient when either of the operands has a negative value. Thus, -7 / 4 can yield either -1 or -2. However, on all IBM C compilers, -7 / 4 always results in a quotient of -1 and a remainder of -3, and that's the rule FPL has been following.

It ends up in an error message if the second operand (the denominator) evaluates to 0 (zero).

B.15.3 Remainder %

The % (remainder) operator yields the remainder from the division of the left operand by the right operand. For example, the expression 5 % 3 yields 2.

If the right operand evaluates to 0 (zero), it results in an error message. If either operand has a negative value, the result is such that the following expression always yields the value of a if b is not 0 (zero):

```
( a / b ) * b + a % b;
```

B.16.4 Addition +

The + (addition) operator yields the sum of its operands.

B.17.5 Subtraction -

The - (subtraction) operator yields the difference of its operands.

B.18.6 Bitwise left and right shift << >>

The bitwise shift operators move the bit values of a binary object. The left operand specifies the value to be shifted. The right operand specifies the number of positions that the bits in the value are to be shifted.

The << (bitwise left shift) operator indicates the bits are to be shifted to the left. The >> (bitwise right shift) operator indicates the bits are to be shifted to the right.

The right operand should not have a negative value or a value that is greater than the width in bits of the expression being shifted. Bitwise shifts on such values give unpredictable results.

If the right operand has the value 0 (zero), the result is the value of the left operand (after the usual arithmetic conversions).

The << operator fills vacated bits with zeros. For example, if l_op has the value 4019, the bit pattern (in 32-bit format) of l_op is:

```
00000000000000000000111110110011
```

The expression l_op << 3 yields:

```
00000000000000000111110110011000
```

The result is the integral part of the quotient of the left operand divided by the quantity, 2 raised to the power of the right operand. If the left operand has a negative value, the vacated bits of a signed value are filled with a copy of the sign bit of the unshifted value. For example, if l_op has the value -25, the bit pattern (in 32-bit format) of l_op is:

```
11111111111111111111111111100111
```

Vacated bits are filled with ones, and the expression l_op >> 3 yields:

```
11111111111111111111111111111100
```

B.19.7 Relational < > <= >=

The relational operators compare two operands for the validity of a relationship. If the relationship stated by the operator is true, the value of the result is 1 (one). Otherwise, the value of the result is 0 (zero).

FPL has the following relational operators:

Operator	Usage
<	Indicates whether the value of the left operand is less than the value of the right operand.
>	Indicates whether the value of the left operand is greater than the value of the right operand.
<=	Indicates whether the value of the left operand is less than or equal to the value of the right operand.
>=	Indicates whether the value of the left operand is greater than or equal to the value of the right operand.

Relational operators have also left-to-right associativity. Therefore, the expression:

```
a < b <= c
```

is interpreted as:

```
(a < b) <= c
```

If the value of a is less than the value of b, the first relationship is true and yields the value 1 (one). The value 1 (one) is then compared with the value of c.

B.20.8 Equality == !=

The equality operators, like the relational operators, compare two operands for the validity of a relationship. The equality operators, however, have a lower precedence than the relational operators. If the relationship stated by an equality operator is true, the value of the result is 1 (one). Otherwise, the value of the result is 0 (zero).

FPL has the following equality operators:

Operator	Usage
==	Indicates whether the value of the left operand is equal to the value of the right operand.
!=	Indicates whether the value of the left operand is not equal to the value of the right operand.

The following expressions contain examples of equality and relational operators:

```
time < max_time == status < complete
hello != world
```

B.21.9 Bitwise AND &

The & (bitwise AND) operator compares each bit of its first operand to the corresponding bit of the second operand. If both bits are 1's, the corresponding bit of the result is set to 1. Otherwise, it sets the corresponding result bit to 0.

The following example shows the values of a, b, and the result of a & b represented as 32-bit binary numbers:

```
bit pattern of a        00000000000000000000000001011100
bit pattern of b        00000000000000000000000000101110
bit pattern of a & b    00000000000000000000000000001100
```

B.22.10 Bitwise Exclusive OR ^

The ^ (bitwise exclusive OR) operator compares each bit of its first operand to the corresponding bit of the second operand. If both bits are 1's or both bits are 0's, the corresponding bit of the result is set to 0. Otherwise, this operator sets the corresponding result bit to 1.

The following example shows the values of a, b, and the result of a ^ b represented as 32-bit binary numbers:

```
bit pattern of a        00000000000000000000000001011100
bit pattern of b        00000000000000000000000000101110
bit pattern of a^b      00000000000000000000000001110010
```

B.23.11 Bitwise inclusive OR |

The | (bitwise inclusive OR) operator compares the values (in binary format) of each operand and yields a value whose bit pattern shows which bits in either of the operands has the value 1 (one). If both of the bits are 0 (zero), the result of the comparison is 0 (zero); otherwise, the result is 1 (one).

The following example shows the values of a, b, and the result of a | b represented as 32-bit binary numbers:

```
bit pattern of a        00000000000000000000000001011100
bit pattern of b        00000000000000000000000000101110
bit pattern of a | b    00000000000000000000000001111110
```

B.24.12 Logical AND &&

The && (logical AND) operator indicates whether both operands have a nonzero value. If both operands have nonzero values, the result has the value 1 (one). Otherwise, the result has the value 0 (zero).

The following examples show how the language evaluates expressions that contain the logical AND operator:

Expression	Result
`1 && 0`	`0`
`1 && 4`	`1`
`0 && 0`	`0`

Expressions like "a && b" will only evaluate b if a is false!

NOTE: The logical AND (&&) should not be confused with the bitwise AND (&) operator. For example,

`1 && 4 evaluates to 1`

while

`1 & 4 evaluates to 0`

B.25.13 Logical OR ||

The || (logical OR) operator indicates whether either operand has a nonzero value. If either operand has a nonzero value, the result has the value 1 (one). Otherwise, the result has the value 0 (zero).

The following examples show how expressions that contain the logical OR operator are evaluated:

Expression	Result
`1 || 0`	`1`
`1 || 4`	`1`
`0 || 0`	`0`

Expressions like “a || b” will only reach b if a is false.

NOTE: The logical OR (||) should not be confused with the bitwise OR (|) operator. For example,

```
1 || 4 evaluates to 1
```

while

```
1 | 4 evaluates to 5
```

B.26 Conditional Expressions

(operand1?operand2:operand3)

A conditional expression is a compound expression that contains a condition (operand1), an expression to be evaluated if the condition has a nonzero value (operand2), and an expression to be evaluated if the condition has the value 0 (zero) (operand3).

The conditional expression contains one two-part operator. The ? symbol follows the condition, and the : symbol appears between the two action expressions. All expressions between the operators ? and : are treated as one expression.

Examples

The following expression determines which variable has the greater value, y or z, and assigns the greater value to the variable x:

```
x = (y > z) ? y : z;
```

The following is an equivalent expression:

```
if (y > z) {
    x = y;
} else {
    x = z;
}
```

The following expression assigns an integer. If the variable c is less than zero, output receives the value of c. If not, output receives the return code from the search function.

```
c = c<0?c:search("hello");
```

If the last operand of a conditional expression contains an assignment operator, use parentheses to ensure the expression evaluates properly. For example, the = operator has higher precedence than the ?: operator in the following expression:

```
(i == 7) ? j ++ : k = j;
```

This expression generates an error because it is interpreted as if it were parenthesized this way:

```
((i == 7) ? j ++ : k) = j;
```

That is, k is treated as the third operand, not the entire assignment expression k = j. The error arises because a conditional expression is not an lvalue, and the assignment is not valid.

To make the expression evaluate correctly, enclose the last operand in parentheses:

```
(i == 7) ? j ++ : (k = j);
```

B.27 Assignment Expressions

An assignment expression gives a value to the left operand.

The left operand in all assignment expressions must be a variable. The value of the expression is the value of the left operand after the assignment is completed.

The language contains two types of assignment operators, simple assignment and compound assignment operators.

B.28.1 Simple assignment =

The simple assignment operator gives the value of the right operand to the left operand.

The following example assigns in order the value 0 (zero) to strangeness, the value of strangeness to charm, the value of charm to beauty, and the value of beauty to truth:

```
truth = beauty = charm = strangeness = 0;
```

B.29.2 Compound assignment

The compound assignment operators perform an operation on both operands and give the result of that operation to the left operand.

The following table lists the compound assignment operators and shows an expression using each operator:

Operator	**Example**	**Equivalent Expression**
+=	index += 2	index = index + 2
-=	balance -= debit	balance = balance - debit
*=	bonus *= increase	bonus = bonus * increase
/=	time /= hours	time = time / hours
%=	allow %= 1000	allow = allow % 1000
<<=	result <<= num	result = result << num
>>=	form >>= 1	form = form >> 1
&=	mask &= 2	mask = mask & 2
^=	test ^= pre_test	test = test ^ pre_test
|=	flag |= on	flag = flag | on

Although the equivalent expression column shows the left operands (from the example column) evaluated twice, the left operand is evaluated only once.

Note that the expression

```
a *= b + c
```

is equivalent to

```
a = a * (b + c)
```

and NOT

```
a = a * b + c.
```

B.30 Comma Expressions

A comma expression contains two operands separated by a comma operator. Although both operands are evaluated, the value of the right operand is the value of the expression. The left operand is evaluated, possibly producing side effects, and then the value is discarded.

In the following example, if omega has the value 11, the expression increments delta and assigns the value 3 to alpha:

```
alpha = (delta++, omega % 4);
```

Any number of expressions separated by commas can form a single expression. The leftmost expression is evaluated first. The value of the rightmost expression becomes the value of the entire expression. For example, the value of the expression

```
intensity++, shade * increment, rotate(direction);
```

is the value of the expression

```
rotate(direction);
```

Restrictions

You can place comma expressions within lists that contain commas (for example, argument lists and initializer lists). However, because the comma has a special meaning, you must place parentheses around comma expressions in these lists. The comma expression t = 3, t + 2 is contained in the following function call:

```
f(a, (t = 3, t + 2), c);
```

The arguments to the function f are: the value of a, the value 5, and the value of c.

C Keywords

FPL supplies a number of keywords that'll help executing the program. (You might recognize several of them from the C programming language, but please read the descriptions carefully cause some of the keywords behave in a slightly different way in FPL.)

These keywords are e.g looping functions and conditional checks etc. The "arguments" to these keywords are a bit special and tricky, they are therefore not included in this table.

Keyword	Short Description	Page
auto	No global variable.	12
break	Break out of a number of levels of loop braces.	45
case	Used in switch() statements for a specific case.	46
char	Create an 8-bit signed numerical variable.	47
const	Constant variable.	12
continue	Continue the loop.	48
default	Specifies default action in a switch() statement.	49
do	do {statement} while (expression);	50
exit	Stop executing this FPL routine.	52
for	for(statement1;statement2;statement3)	54
if	if(condition) {statement} [else {statement}]	56
int	Creates an 32-bit integer variable.	58
long	Equivalent to `int'.	58
register	Non-global variable.	12
resize	Resize a variable array.	59
return	Return from subroutine.	61
short	Create a 16-bit signed numerical variable.	62
static	Remember local variables between invokes.	12
string	Creates a string variable.	63
switch	Do different actions on different results.	65
typedef	Make declarator aliases.	64
volatile	FPL variables are always volatile.	12
while	while(condition) {statement} [else {statement}]	67

These keywords are reserved for FPL use, but do nothing today:

Keyword	**Description**
double	Not implemented.
enum	Not implemented.
float	Not implemented.
signed	ALL variables are always signed.
struct	Not implemented.
union	Not implemented.
unsigned	Not implemented.

C.1 break

SYNTAX break [expression];

DESCRIPTION

A break statement enables you to end iterative (do, for, while, switch) statements and exit from them at any point other than the logical end.

The break statement ends the loop or switch and moves control to the next statement outside the loop/switch. Within nested statements, break ends only the smallest enclosing do, for, while or switch statement as default.

The possible “else” statement after a while, is *NOT* a loop.

Specifying a following expression, breaks out of a number of iterative statements.

RESTRICTIONS

Place a break statement only in the body of an iterative statement.

EXAMPLES

The following example shows a break statement in the action part of a for statement. If the i*3 is equal to 9, the break statement causes the for statement to end.

```
for (i = 0; i < 5; i++) {
     if (i*3 == 9)
          break;
     length++;
}
```

Break out of a number of loops by giving an argument:

```
for(i = 0; i < 5 ; i++) {
     for(j = 0; j < 5 ; j++) {
          if(i*5+j>18) {
               break 2; // Breaks out of two `for’ loops!
          }
     }
}
```

Break out of a switch() statement:

```
switch(i) {
     case 2:
          break;
     default:
}
```

SEE do, for, while, switch

C.2 case

SYNTAX case expression: statement;

DESCRIPTION

The 'case' statement can only be used within a switch statement. It tells that the following statements should be run if the result of the case expression is the result of the switch expression.

The expression must be followed by a colon ':'.

Break the case statement with a 'break'. Then the execution will continue after the switch() statement.

NOTE

If the expression contains a colon ':', it must be with parentheses or it can confuse the interpreter under certain conditions.

EXAMPLE

The following program show a switch() statement with three cases:

```
switch( Character() ) {
     case '\t':
     case ' ':
          a = 0;
          break;
     case '\n':
          a = -1;
          break;
     default:
          a = 1;
          break;
}
```

SEE ALSO

switch, break, default

C.3 char

SYNTAX char name [= expression];

DESCRIPTION
Declares a char (signed 8-bit) variable (and assign a value to it). Read more about it the chapter discussing variables on page 12. Not assigned variables equals zero (0) after declaration.

INPUTS
string name - The variable name.
char expression - Initial expression.

SEE ALSO
int, short, string

C.4 continue

SYNTAX continue;

DESCRIPTION

A continue statement enables you to stop the current iteration of a loop. Program control is passed from the location in the body of the loop where the continue is found, to the condition part of the loop.

The continue statement ends the execution of the action part of a do, for, or while statement and moves control to the condition part of the statement. If the iterative statement is a for statement, control moves to the third expression in the condition part of the statement, then to the second expression (the test) in the condition part of the statement.

Within nested statements, the continue statement ends only the current iteration of the do, for or while statement immediately enclosing it.

RESTRICTIONS

Place a continue statement only within the body of an iterative statement.

EXAMPLES

The following example shows a continue statement in a for statement. The continue statement causes the system to skip over those elements of the formula that have values less than or equal to 100.

```
output("Try this calculation while <=100.\n");
for (i = 0; i < 10000; i++) {
     if (i*(3-b/c+a) <= 100)
          continue;
     output("The number ", i, " makes it no longer below!");
}
```

SEE ALSO

do, for, while

C.5 default

SYNTAX default: statement;

DESCRIPTION

The ‘default’ statement can only be used within a switch statement. It tells that the following statements is the default action if no case expressions match the switch expression.

The keyword must be followed by a colon ‘:’.

Break the default statement with a ‘break’. Then the execution will continue after the switch() statement.

EXAMPLE

This program executes the ‘default’ statement if ‘x’ does not match “foo” or “bar”:

```
switch(x) {
     case "foo":          /* if x is "foo" */
     case "bar":          /* if x is "bar" */
          /* nanananana */
          break;            /* break out of switch */

     default:
          foobar();         /* if x isn't "foo" or "bar" */
          break;
}
```

SEE ALSO

switch, break, case

C.6 do

SYNTAX do statement; while (expression) ;

DESCRIPTION

A do statement repeatedly executes a statement until a test expression evaluates to 0 (zero). Because of this order of processing, the statement is processed at least once.

The body of the loop is run before the controlling while clause is evaluated. Further processing of the do statement depends on the value of the while clause. If the while clause does not evaluate to 0 (zero), the statement runs again. Otherwise, processing of the statement ends.

A break or return statement can cause the processing of a do statement to end, even when the while clause does not evaluate to 0 (zero).

EXAMPLES

The following statement prompts the system user to enter a 1. If the system user enters a 1, the statement ends execution. If not, the statement displays another prompt.

```
do {
     input("Enter a 1!", reply);
} while (reply != 1);
```

SEE ALSO

break, continue, return

C.7 else

SYNTAX <special>

DESCRIPTION

Used after condition checks to make FPL execute certain parts of the program if the previous check did not evaluate true.

It always belongs to the last `if'/`while' in this block.

EXAMPLES

It can be used in simple one time checks like:

```
if(a!=2)
     break;
else
     output("hello"); /* done only if variable a equals 2 */
```

and even if a while loop never is executed:

```
while(a--)
     output(a);
else
     output("hello");// only executed if the while condition
                     // never was true.
```

Take a look at the two examples below and note the difference!

1. If both "a" and "b" are true, then invoke "build_new_things". if "a" is false, invoke "crash_mostofit":

```
if(a) { // braces are required if we want the `else' to affect this
     if(b)
          build_new_things();
} else
     crash_mostofit();
```

2. If "a" and "b" are true, then invoke "build_new_things". If "a" is true and "b" is false, invoke "crash_mostofit":

```
if(a) // without braces, the else suddenly affects the other if!
     if(b)
          build_new_things();
     else
          crash_mostofit();
```

C.8 exit

SYNTAX exit(return_code);

DESCRIPTION
Stop execution of the current FPL routine.

INPUTS

int/string return_code -	Return code to return to the calling process. See the documentation for the software you're controlling for the exact meaning of these return codes. The return code can be excluded as well as the parenthesis can when returning a value.

RETURNS
A result code to the invoking environment.

SEE ALSO
return

C.9 export

SYNTAX export [global declaration];

DESCRIPTION

Make the following global symbol available to all programs.

EXAMPLES

The following statements make the function “foobar” and the string variable “String” accessiable to all other FPL programs that will succeed this program:

```
export string String;
export int foobar(int);
/* the actual function foobar must reside in the same source file */
```

C.10 for

SYNTAX for ([expression1] ; [expression2] ; [expression3]) statement;

DESCRIPTION

Expression1
: is evaluated only before the statement is processed for the first time. You can use this expression to initialize a variable. If you do not want to evaluate an expression prior to the first iteration of the statement, you can omit this expression.

Expression2
: is evaluated before each iteration of the statement. The expression must evaluate to a scalar type. If it evaluates to 0 (zero), the statement is not processed and control moves to the next statement following the for statement. If expression2 does not evaluate to 0, the statement is processed. If you omit expression2, it is as if the expression had been replaced by a nonzero constant and the for statement is not terminated by failure of this condition.

Expression3
: is evaluated after each iteration of the statement. You can use this expression to increase, decrease, or reinitialize a variable. If you do not want to evaluate an expression after each iteration of the statement, you can omit this expression.

A break or return statement can cause the processing of a for statement to end, even when the second expression does not evaluate to 0 (zero). If you omit expression2, you must use a break or a return statement to stop the for statement from running.

EXAMPLES

The following for statement prints the value of count 20 times. The for statement initially sets the value of count to 1. After each iteration of the statement, count is incremented.

```
for (count = 1; count <= 20; count++) {
     output("count =", count);
}
```

The following sequence of statements accomplishes the same task. Note the use of the while statement instead of the for statement.

```
count = 1;
while (count <= 20) {
     output("count = ", count);
     count++;
}
```

The following for statement does not contain an initialization expression.

```
for (; index > 10; index--) {
     list= var1 + index;
     output("list = ", list, "\n");
}
```

The following for statement continues running until input receives the letter e:

```
for (;;) {
     input("Gimme an e!", letter);
     if (!strcmp(letter, "\n"))
```

```
        continue;
    else if (!strcmp(letter, "e"))
        break;
    output("You entered the letter", letter);
}
```

The following for statement contains multipart initializations and increments. The comma operator makes this construction possible.

```
for (i = 0, j = 50; i < 10; i++, j += 50) {
    output("i = ", i, " and j = ", j);
}
```

The following example shows a nested for statement. The outer statement is run as long as the value of row is less than 5. Each time the outer for statement is processed, the inner for statement sets the initial value of column to zero and the statement of the inner for statement is run three times. The inner statement is run as long as the value of column is less than 3.

```
for (row = 0; row < 5; row++) {
    for (column = 0; column < 3; column++)
        output("column * row =", row * column, "\n");
```

A never-ending loop using `for':

```
for(;;)
    perform_until_death();
```

SEE ALSO

break, continue, do, while

C.11 if

SYNTAX if (expression) statement; [else statement;]

DESCRIPTION

An if statement allows you to conditionally process a statement if the specified test expression evaluates to a nonzero value. You can optionally specify an else clause on the if statement. If the test expression evaluates to 0 (zero), and an else clause exists, the statement in the else clause is run. If the test expression evaluates to a nonzero value, the statement following the expression runs and the else clause is ignored.

When if statements are nested and else clauses are present, a given else is associated with the closest preceding if statement within the same block.

EXAMPLES

The following example causes grade to receive the value A if the value of score is greater than or equal to 90.

```
if (score >= 90)
     grade = A;
```

The following example displays number is positive if the value of number is greater or equal to 0 (zero). Otherwise, the example displays number is negative.

```
if (number >= 0)
     output("number is positive\n");
else
     output("number is negative\n");
```

The following example shows a nested if statement:

```
if (paygrade == 7)
     if (level >= 0 && level <= 8)
          salary *= 2;
     else
          salary *= 3;
else
     salary *= 4;
```

The following example shows an if statement that does not have an else clause.

```
if (gallons > 0) {
     if (miles > gallons)
          mpg = miles/gallons;
} else
     mpg = 0;
```

The following example shows an if statement nested within an else clause. This example tests multiple conditions. The tests are made in order of their appearance. If one test evaluates to a nonzero value, an action statement runs and the entire if statement ends.

```
if (value > 0)
     increase++;
else if (value == 0)
     breaeven++;
else
```

```
++decrease;
```

C.12 int and long

SYNTAX int name [= expression];

or

long name [= expression];

DESCRIPTION

Declare an integer (signed 32-bit) variable and assign a value to it. Read more about it the chapter discussing variables. Not assigned variables equals zero (0) after declaration.

ANSI C specifies that `long' should be a 32-bit number and `int' is more vague. The MC680x0 processor series has - as all decent processors have - 32-bit integers, which make them the exact same data type. FPL is mostly run on such processors making this equivalence natural.

INPUTS

string name	- The variable name.
int/long expression	- Initial expression.

SEE ALSO

string, char, short

C.13 resize

SYNTAX resize name [new_size] ;

DESCRIPTION

A common problem when using arrays is that you don't know the size of it already from the start. This keyword brings a solution to that problem. This one resizes the named array to the new size you specify inside the square brackets. Notice that it's also possible to reduce the size, so be careful not to destroy data you like to reuse.

WARNING: Using multi dimensional arrays and `resize' is a dangerous combination. E.g when changing an array from "[4][5]" to "[5][5]", the old values will not any longer be readable in their old array members. A resize of this kind makes the data move around a bit depending on the internal data array storage. DO NOT USE `resize' IN MULTI DIMENSIONAL ARRAYS IF YOU WANT THE DATA TO STAY INTACT!

INPUTS

name	- The name of the array you'd like to resize.
new_size	- The new size of the array.

EXAMPLES

To make a list of all the names the user enters, you need an array as big as the user pleases. But declaring an enormous size from the start really isn't to recommend. This example shows one way to solve such a problem:

```
int roof=10, num;
string names[roof];
do {
     if(num==roof)
          resize names[roof+=10];
     names[num]=input("Insert name number " num ":");
} while(names[num++]!="");
```

Multi dimensional arrays looses their proper values when resizing, do copy the array to preserve to contents:

```
int a[DIM1][DIM2];       // This is the old array construction
{
     /*   This could with benefit be written into a function to be
          called from several instances */
     int n1, n2;              // counter variables
     int b[NEWDIM1][NEWDIM2]; // The temporary new array outfit!

     for(n1=0; n1<DIM1 && n1<NEWDIM1; n1++)   // temporary store all
          for(n2=0; n2<DIM2 && n2<NEWDIM2; n2++) // vital data in array b
               b[n1][n2]=a[n1][n2];

     resize a[NEWDIM1][NEWDIM2]; // resize a

     for(n1=0; n1<NEWDIM1; n1++)   // restore all data to array a
          for(n2=0; n2<NEWDIM2; n2++)
                a[n1][n2]=b[n1][n2];
```

```
}
```

C.14 return

SYNTAX return [return_code];

DESCRIPTION

A return statement ends the execution of the current function and returns a return code and control to the caller of it. The return code is written exact as in C, within parentheses or not. It can also be totally excluded.

EXAMPLES

We use a function to print the result of the formula, but only if the result is even:

```
int n;
int print(int);
for(n=0; n<10; n++)
     n+=printf(n*n);
exit;

int print(int a)
{
     if(!(a&1))
          output(a);
     return(a&2);  /* Get back to the for loop */
}
```

SEE ALSO

exit, break

C.15 short

SYNTAX short name [= expression];

DESCRIPTION

Declares a short (signed 16-bit) variable (and assign a value to it). Read more about it the chapter discussing variables on page 12. Not assigned variables equals zero (0) after declaration.

INPUTS

string name	- The variable name.
short expression	- Initial expression.

SEE ALSO

int, char, string

C.16 string

SYNTAX string varname [= str_expr];

or

string funcname ([arg1] [, arg2] [...]);

DESCRIPTION

Declare a string variable and assign contents to it or declare a function returning a string.

For function declaration, see the functions chaper on page 19.

INPUTS

string varname	- Name of the variable
string str_expr	- Initial string assign.

or

string funcname	- Name of the function

SEE ALSO

int

C.17 typedef

SYNTAX typedef <declarator> <symbol>;

DESCRIPTION

`typedef' allows you to define your own identifiers that can be used in place of FPL type specifiers such as int, string and char. The data types you define using `typedef' are not new data types; they are synonyms for the primary data types used by FPL.

EXAMPLES

This example creates a synonym for int and then use that to declare three integral variables:

```
typedef int LENGTH;     /* new declarator named `LENGTH' */
LENGTH length, width, height;
```

SEE ALSO

int, string, the variable chapter on page 12

C.18 switch

SYNTAX switch (expression) { [case expression:] [default:expression] }

DESCRIPTION

A switch statement lets you define actions to different results of the switch expression.

The result of the expression written within the parentheses following the switch keyword is matched against the expressions following the ‘ case ‘ keywords within the braces. If no match is found the action specified after ‘ default ‘ will be run. If no ‘default’ is defined either, nothing will happen.

The switch expression may be either a string or numerical expression. The expressions in the following case statements have to be of the same expression type as the initial one, or FPL will report error.

Multiple cases may be specified with the same expression, but only the first one that match the switch expression will be run.

A case statement is aborted by a break statement or the brace ending the switch statement.

NOTE

Unlike C, FPL handles dynamic expressions in case-statements very well!

The case statements will be parsed/scanned from the top to bottom. For speed reasons, you should put the case-statements that most likely occurs at the top and the least likely case-statements at the bottom.

EXAMPLES

In the following program is the result from the equation x+5 controlling what text to assign to ‘output’:

```
int check = x + 5;
string output;
switch( check ) {
     case 5:     /* notice the colon that follows the expression */
          output = "x was zero";
          break;    /* escape from switch! */
     case 4:
          break;    /* do nothing */
     case 7:
          /* fall through! */
     case 8:
          output = "x was two or three";
          break;
     default:
          output = "x was not from -1 to 3";
}   /* breaks out of 'default' */
```

In the following program is the contents of the variable ‘username’ controlling which function to invoke:

```
string username = getname();  /* get user name */
switch(username) {
```

```
     case “superuser”:
          GetSuperUser();
          break;
     case “normal”:
          GetNormalUser();
          break;
     case “unworthy”:
          GetUnworthUser();
          break;
}
```

C.19 while

SYNTAX while (expression) statement ; [else statement;]

DESCRIPTION

A while statement enables you to repeatedly run the body of a loop until the controlling expression is no longer met or evaluates to 0 (zero.)

The expression is evaluated to determine whether or not the body of the loop should be run. If the expression evaluates to 0 (zero), the body of the loop never runs. If not, the body is processed. After the body has run, control passes back to the expression. Further processing depends on the value of the condition.

When coding in ‘C’, I’ve always missed the “while else” function. FPL includes that feature! If the test expression evaluates to false (zero) the FIRST TIME it’s evaluated, and an “else” clause exists, the statement in the “else” clause is run. If the test expression evaluates to a nonzero value, the statement following the expression runs and the “else” clause is ignored.

A break or return statement can cause the processing of a while statement to end, even when the condition does not evaluate to 0 (zero).

NOTE: The else clause is *NOT* a loop. “break” is not a valid move to break out of such.

EXAMPLES

In the following program, item triples each time the value of the expression index++ is less than MAX_INDEX. When index++ evaluates to MAX_INDEX, the while statement ends.

```
while (index < MAX_INDEX) {
     item *= 3;
     output(“item = “, item, “\n”);
     index++;
}
```

The following program first checks if a<0 and if it is, increases a with 100 until it isn’t any longer less than zero. If not, it’ll output a message saying so.

```
while(a<0)
     a+=100;
else
     output(“a was never below 0\n”);
```

SEE ALSO

break, return, do, for, continue

D Functions

These are the internal functions supplied from the start by FPL:

Function	Argument(s)	Short Description	Page
abs	int	Absolute value of integer operand.	69
atoi	str	Convert a string to an integer.	70
debug	int	Controls debug mode.	72
eval	str	Calculate a string.	73
interpret	str	Interpret string as program.	74
itoa	int	Convert an integer to string.	76
itoc	int	Convert an integer to ASCII.	77
joinstr	str, ...	Joins strings!	78
ltostr	n1, n2	Convert an integer to string.	79
sprintf	format, args, ...	Formatted print to a string.	81
strcmp	str1, str2	Compare two strings.	85
stricmp	str1, str2	Compare two strings. CI*	85
strlen	str	String length.	86
strncmp	str1, str2, n	Compare two strings a certain length.	87
strnicmp	str1, str2, n	Compare two strings a certain length. CI*	87
strstr	str1, str2	Find substring str1 in string str2.	88
stristr	str1, str2	Find substring str1 in string str2. CI*	88
strtol	str, base	Convert string to integer.	89
substr	s1, len, col	Get part of string.	90
openlib	lib, ver	Open funclib.[Amiga only!]	80
closelib	lib	Close funclib.[Amiga only!]	71

* = Case insensitive. It doesn’t make any difference to ‘A’ or ‘a’. It does only apply to ASCII characters; that is only a-z and A-Z.

D.1 abs

SYNTAX int abs(Int);

DESCRIPTION
Returns the absolute value of the input integer.

INPUTS
int Int;

RETURNS
see description.

D.2 atoi

SYNTAX int atoi(String);

DESCRIPTION

The atoi keyword returns an integer whose value is represented by the character string pointed to by the String parameter. atoi scans the string up to the first character that is inconsistent. Leading white-space characters are ignored, and an optional sign may precede the digits.

INPUTS

string String;

RETURNS

Upon successful completion, atoi returns the converted value. If no conversion could be performed, 0 (zero) is returned. If the correct value is outside the range of representable values, maximum or minimum value is returned according to the sign of the value.

SEE ALSO

strtol, itoa, ltostr

D.3 closelib

SYNTAX int closelib(Name);

DESCRIPTION

Closes a specified funclib. Funclibs are third party programs that add functions to the FPL session. The funclibs should be stored in FPLLIBS:.

A closelib() concludes access to the functions of the specified funclib. Each call to openlib() should have a matching call to closelib().

For information regarding specified funclibs, refer to the manual of the particular funclib.

INPUTS

string Name - The name of the funclib.

RETURNS

It return 0 on success. Otherwise see ‘openlib’ for closer description, on page 80.

SEE ALSO

openlib and the funclibs section.

D.4 debug

SYNTAX int debug (Enable);

DESCRIPTION

Toggles the “debug mode” state. When FPL runs in the “debug mode”, the debugger is capable of tracing the FPL executions.

If called without parameter, it will return the current “debug mode” status.

NOTE

The FPL debugger is not extensively used, developed or spread. Much more documentation regarding this function and FPL debugging concepts and procedures will appear in future versions of this document.

INPUTS

int Enable - Zero to disable, non-zero to enable.

RETURNS

It returns the “debug mode” state that was when the function was called. That is, if you enable debug mode, it will return the state previous to this function call.

SEE ALSO

FPL debugging chapter

D.5 eval

SYNTAX int eval(String);

DESCRIPTION

The mathematical expression in String is evaluated and returned. strcal handles all variables declared before this point of the execution. An error in the expression will unfortunately return 0 which will look like the result of the expression.

INPUTS

string String - A string including a valid numerical expression.

RETURNS

The result of the mathematical expression in String or 0 if there was any error in the expression.

SEE ALSO

interpret

D.6 interpret

SYNTAX int interpret(String);

DESCRIPTION

Interpret the argument as if it is a continuation of the program. The argument must be a 100% corect FPL program statement. All variables and functions usable in the program at this point, will be usable in the argument-statement too.

Any error in this function will get reported at the position right after the function call.

INPUTS

string String - FPL statement

RETURNS

At this moment: always zero. But do not depend upon that since it might get changed in the future.

EXAMPLE

This example calls a function via the interpret() function:

```
int foobar(int);
interpret("foobar(2);");
exit;

int foobar(int b)
{
     output(b);
}
```

We can create a function that accepts a function name as parameter and then is able to call that named function:

```
int foobar(int);
int foo(int);
int bar(int);
int call(string, int);

call("foobar", 1);
call("foo", 2);
call("bar", 3);
exit;

int call(string command, int parameter)
{
     int ret;      /* declare return code variable */
     string statement=joinstr("ret=", command, "(",
          ltostr(parameter), ");");

     output(statement);     /* output the statement */
     interpret(statement); /* interpret it */
     /*
```

```
         * At this point `ret' has got the return code
         * of the called function.
         */
        return(ret);
    }

    /*
    * Here should the definitions of the functions
    * "foobar", "foo" and "bar" be programmed.
    */
```

SEE ALSO

eval

D.7 itoa

SYNTAX string itoa(Integer);

DESCRIPTION

The itoa function converts the integer given as argument to a string and returns it. The convertion will use base 10, which creates a decimal number.

INPUTS

int Integer - Number to be converted.

RETURNS

Upon successful completion, the function returns the converted string. If no conversion could be performed, the returned string is zero-length.

EXAMPLES

Convert the number the variable ‘num’ holds, to a string:

```
string result= itoa(num);
```

SEE ALSO

ltostr

D.8 itoc

SYNTAX string itoc(Integer);

DESCRIPTION

The itoc function returns the character with the ASCII code given as argument. Any argument number higher than 255 will be ANDed with 255 internally.

INPUTS

int Integer - ASCII code of the desired character.

RETURNS

Upon successful completion, the function returns the character as a string.

EXAMPLES

To get the character with ASCII number 137:

```
string result= itoc( 137 );
```

SEE ALSO

itoa

D.9 joinstr

SYNTAX string joinstr(String, ...);

DESCRIPTION

This function lets you specify any amount of strings, and returns them all as one concatenated single string. *This function is kind of obsolete and should be replaced with use of the string addition feature (read about strings expressions on page 16).*

INPUT

string String - Strings to be merged.

RETURNS

A string holding all joined strings.

EXAMPLES

Merge three strings into one:

```
string a=”one “, c=”two “, b=”three”, d;
d=joinstr(a, b, c); /* ‘d’ now holds “one two three” */
```

It can since FPL version 12 just as well be written like:

```
string a=”one “, c=”two “, b=”three”, d;
d=a + b + c; /* ‘d’ now holds “one two three” */
```

D.10 ltostr

SYNTAX string ltostr(Integer, Base);

DESCRIPTION

The ltostr subroutine returns a string which is the first parameter converted using the second parameter as base.

If the Base parameter is positive and not greater than 36, then it is used as the base for conversion.

INPUTS

int Integer - Number to be converted.
int Base - Specifies the base to use for the conversion.

RETURNS

Upon successful completion, the ltostr subroutine returns the converted string. If no conversion could be performed, the returned string is zero-length.

EXAMPLES

Convert the number 1993 to a hexadecimal string:

```
string result= ltostr(1993, 16);
```

SEE ALSO

strtol

D.11 openlib

SYNTAX int openlib(Name, Version);

DESCRIPTION

Opens a specified funclib. Funclibs are third party programs that add functions to the FPL session. The funclibs should be stored in FPLLIBS:.

An openlib() gains access to the functions of the specified funclib. After finished using the functions of that funclib, the session should be concluded with a call to closelib() . Each call to openlib() should have a matching call to closelib().

For information regarding specified funclibs, refer to the manual of the particular funclib.

INPUTS

string Name - The name of the funclib.
int Version - The lowest acceptable version of the funclib.

RETURNS

It return 0 on success. Otherwise a non-zero number where the numbers have the following meaning:
1 - funclib parameter error
2 - internal funclib error
3 - failed getting a system resource
4 - out of memory error
5 - failed loading the funclib (occurs if you open a non-existant funclib).
6 - the requested version didn’t exist
More error codes are likely to be added within short.

SEE ALSO

closelib and the funclibs section on page 9.

D.12 sprintf

SYNTAX string sprintf (format, arg1, arg2, ...);

DESCRIPTION

This function produces a string of ASCII characters.

The format argument holds a string that contains ordinary characters and conversion specifications that indicate how you want the arguments arg1, arg2, and so on to be printed. The ordinary characters are copied to the result string, but the conversion specifications are replaced with the correctly formatted values of the arguments arg1, arg2, and so on. The first conversion specification is replaced with the formatted value of arg1, the second specification is replaced with the value of arg2, and so on. In some cases, as described below, a conversion specification may process more than one argument.

Each conversion specification must begin with a percent character (%). To place an ordinary percent into the output stream, precede it with another percent in the fmt string. That is, %% will send a single percent character to the output stream. A specification has the following format:

```
%[arg][flags][width][.precision][size]type
```

The brackets ([]) indicate optional fields. Each field is defined as follows:

arg

FPL supports the argument number selection style $<arg>. When an argument number has been given, that specified argument will be used instead of the next one in turn. I.e "%$2d" will produce a formatted integer found as the second argument. You should **not** mix specified arugment numbers with %-codes with unspecified arugment numbers.

flags

controls output justification and the printing of signs, blanks, decimal places, and hexadecimal prefixes.
If any flag characters are used, they must appear after the percent. Valid flags are as follows:

- (minus)

causes the result to be left-justified within the field specified by width or within the default width.

+ (plus)

causes a plus or minus sign to be placed before the result. This flag is used in conjunction with the various numeric conversion types. If it is absent, the sign character is generated only for a negative number.

blank

causes a leading blank for a positive number and a minus sign for a negative number. This flag is similar to the plus. If both the plus and the blank flags are present, the plus takes precedence.

(pound)

causes special formatting. With the o, x, and X types, the pound flag prefixes any nonzero output with 0, 0x, or 0X, respectively.

0 (zero)

pads the field width with leading zeros instead of spaces for the d, i, o, u, x and X conversion types. If the minus flag is also used, the zero flag is ignored. If a precision is specified, the zero flag is ignored for conversion types d, i, o, u, x, and X. Behavior of the zero flag is undefined for the remaining conversion types.

width

specifies the field width, which is the minimum number of characters to be generated for this format item.
The width is a nonnegative number that specifies the minimum field width. If fewer characters are generated by the conversion operation, the result is padded on the left or right (depending on the minus flag described above). A blank is used as the padding character unless width begins with a zero. In that case, zero padding is performed. If the minus flag appears, padding is performed with blanks. width specifies the minimum field width, and it will not cause lengthy output to be truncated. Use the precision specifier for that purpose.
If you do not want to specify the field width as a constant in the format string, you can code it as an asterisk (*), with or without a leading zero. The asterisk indicates that the width value is an integer in the argument list. See the examples for more information on this technique. If the asterisk is followed by a $ sign and a digit 1-9, you can specify which argument that holds the width.

precision

specifies the field precision, which is the required precision of numeric conversions or the maximum number of characters to be copied from a string, depending on the type field.
The meaning of the precision item depends on the field type, as follows:

Type	**Meaning**
c	The precision item is ignored.
b, d, i, o, u, x, X	The precision is the minimum number of digits to appear. If fewer digits are generated, leading zeroes are supplied.
s	The precision is the maximum number of characters to be copied from the string.

As with the width item, you can use an asterisk for the precision to indicate that the value should be picked up from the next argument.

size

Not used by FPL, though all the ANSI C modifiers are still read and ignored for compatibility. (Can be either L for long double, l for large size, or h for small size.)

type

specifies the type of argument conversion to be done. Valid conversion types are as follows:

Type	Convertion
b	specifies binary-integer conversion. The associated argument is taken as an unsigned integer, and it is converted to a string of binary digits. This conversion type is an extension to the ANSI standard.
c	specifies single-character conversion. The associated argument must be an integer. The single character in the right-most byte of the integer is copied to the output.
d	specifies decimal-integer conversion. The associated argument must be an integer, and the result is a string of digits preceded by a sign. If the plus and blank flags are absent, the sign is produced only for a negative integer.
i	specifies decimal-integer conversion. The associated argument must be an integer, and the result is a string of digits preceded by a sign. If the plus and blank flags are absent, the sign is produced only for a negative integer.
n	specifies the argument will be a pointer to an integer into which is written the number of characters written so far by this call to the sprintf function.
o	specifies octal-integer conversion. The associated argument is taken as an unsigned integer, and it is converted to a string of octal digits.
p	specifies pointer conversion. The associated argument is taken as a data pointer, and it is converted to hexadecimal representation.The pointer is printed as 8 hexadecimal digits, with leading zeroes if necessary.
P	specifies pointer conversion. This is the same as the p format, except that uppercase letters are used as hexadecimal digits. This conversion type is an extension to the ANSI standard.
s	specifies string conversion. The associated argument must be a string!
u	specifies unsigned decimal integer conversion. The associated argument is taken as an unsigned integer, and it is converted to a string of decimal digits.
x	specifies hexadecimal-integer conversion. The associated argument is taken as an unsigned integer, and it is converted to a string of hexadecimal digits with lowercase letters.
X	specifies hexadecimal-integer conversion. This is the same as the x format, except that uppercase letters are used as hexadecimal digits.

RETURNS

This function returns the formatted string.

EXAMPLE

```
/* This example prints a message indicating whether */
/* the function argument is positive or negative.   */
/* In the second sprintf, the width and precision   */
/* are 15 and 8, respectively.                      */
string pneg(int value)
{
     string sign;
     if (value < 0) {
          sign = "negative";
     }
     else {
          sign = "not negative";
     }
     return sprintf("The number %d is %s.n",value,sign);
}

Output(pneg(37));
Output(pneg(-18));

/* This example outputs 12 and 8 in reversed order by using the
specified argument number syntax: */

Output( sprintf( " %$2d %$1d ", 12, 8 ) );
```

D.13 strcmp and stricmp

SYNTAX int strcmp (String1, String2);

and

int stricmp (String1, String2);

DESCRIPTION

strcmp lexicographically compares the string in the String1 parameter to the string in the String2 parameter. stricmp() does the same but does a case insensitive compare, which makes uppercase letters equal their lowercase correspondance.

Case insensitive means that the strings “HELLO” and “hello” are treated as identical.

INPUTS

string String1, String2;

RETURNS

strcmp and stricmp return values that are:

Less than 0 (zero) if String1 is less than String2
Equal to 0 (zero) if String1 is equal to String2
Greater than 0 (zero) if String1 is greater than String2.

SEE ALSO

strncmp and strnicmp

D.14 strlen

SYNTAX int strlen(String);

DESCRIPTION
Returns the length of the string String.

INPUTS string String;

RETURNS
The length of the string String.

D.15 strncmp and strnicmp

SYNTAX int strncmp (String1, String2, Num);

and

int strnicmp (String1, String2, Num);

DESCRIPTION
The strncmp and strnicmp subroutines make the same comparisons as strcmp and stricmp, but they compare at most ‘Num’ characters.

INPUTS
string String1, String2;
int Number - Maximum number of characters to compare.

RETURNS
they return values that are:
Less than 0 (zero) if String1 is less than String2
Equal to 0 (zero) if String1 is equal to String2
Greater than 0 (zero) if String1 is greater than String2.

EXAMPLES
Compare the string variable Str1 and check if it matches Str2, but check no more than 12 characters:

```
if(!strncmp(Str1, Str2, 12))
     output(“MATCH!”);
```

This example returns zero (0) in result.

```
result = strncmp(“realization”, “really”, 4);
```

This example returns a value less than zero in result.

```
result = strncmp(“world”, “worldgames”, 100);
```

SEE ALSO
strcmp() and stricmp()

D.16 strstr and stristr

SYNTAX int strstr(String1, String2, Column);

and

int stristr (String1, String2, Column);

DESCRIPTION

The strstr and stristr functions find occurrences of String2 in String1.

The ‘Column’ parameter can be set optionally. It specifies the start searching column!

strstr() requires the substring to match exactly, while the stristr() performs a case insentive search. Case insensitive means that the strings “HELLO” and “hello” are treated as identical.

INPUTS

string String1 - The string to search within.
string String2 - The substring.
int Column - Start search column.

RETURNS

They return in which column they found String2 or -1 if the string was not found. If String2 is a 0 (zero) length string, the functions return 0.

SEE ALSO

strncmp and strnicmp, strcmp and stricmp

D.17 strtol

SYNTAX int strtol(String, Base);

DESCRIPTION

The strtol subroutine returns an integer whose value is represented by the character string pointed to by the String parameter. The strtol subroutine scans the string up to the first character that is inconsistent with the Base parameter. Leading white-space characters are ignored, and an optional sign may precede the digits.

If the Base parameter is positive and not greater than 36, then it is used as the base for conversion. After an optional leading sign, leading zeros are ignored. “0x” or “0X” is ignored if Base is 16 (sixteen).

If the Base parameter is 0 (zero), the string determines the base. Thus, after an optional leading sign, a leading 0 (zero) indicates octal conversion, a leading “0x” or “0X” indicates hexadecimal conversion and a leading “0b” or “0B” indicates a binary conversion. The default is to use decimal conversion.

INPUTS

string String - Character string to be converted.
int Base - Specifies the base to use for the conversion.

RETURNS

Upon successful completion, the strtol subroutine return the converted value. If no conversion could be performed, 0 (zero) is returned. If the correct value is outside the range of representable values, maximum or minimum value is returned according to the sign of the value.

EXAMPLES

Convert the string holding the hexadecimal “0xdeadbeef” to decimal:

```
string String = “0xdeadbeef”
int num = strtol(String, 16);
```

SEE ALSO

ltostr

D.18 substr

SYNTAX string substr(Source, Column, Length);

DESCRIPTION

The substr function returns a substring to a destination string variable. Giving a length of -1 will make substr get the rest of the string from the column specified.

- If the substring is told to keep on beyond the end of the Source, the substring will end where the Source string does.

- If the column parameter is below zero or larger than the length of the source string, a zero length string is returned.

- The first column of the source is column 0 (zero).

INPUTS

string Source - The source string from which the substring will be copied from.
int Column - Start column of the substring.
int Length - The Length of the substring.

RETURNS

The substring!

EXAMPLES

Extract a substring of 3 characters from column 4 of the source string:

```
string Dest;
Dest=substr("One Two Three", 4, 3);
```

Get the right part of a line split by the character "|":

```
string Dest;
string Line="Names     |Addresses";
int pos;
pos=strstr(Line, "|");
if(pos<0)                    /* if we can't find a "|" */
     exit;
pos++;                   /* We don't want to include the "|"
                         Character in our substring! */
Dest=substr(Line,        /* The source string */
     pos,                /* where from */
     -1);                /* To the end of the string. */
```

It's OK to try getting more than you will get. This example get the rest of the line too, but maximum 10000 characters...:

```
Dest=substr(Line, pos, 10000);
```

SEE ALSO

strstr and stristr

E Examples

1. Ask the user for his age and output the number of days old he is:

```
int a;                          /* Create an integer called "a"  */
a=num_input("Age in years:");   /* Get age of user in variable a */
output("Makes " + itoa(a*365) + " days!\n"); /* output text */
```

2. Ask the user for his name and output it a hundred times with spaces between:

```
string name;                        /* Create a string variable */
name=str_input("What's your name?"); /* Input name              */
int i;                              /* Create an integer        */
for (i=0; i<100; i++)               /* Count 100 times          */
     output(name " ");              /* Output the name          */
```

3. Replace all form feed characters with the proper number of newlines to suit the page length given by the user (for guys using printers which by some reason don't handle form feeds):

```
int len, i, page;                   // Create a number of ints
len=num_input("Page length?");      // Get page length
while(search_for("\f", "S")) {      // Search for form feed
     page=len-line()%len;           // Calc number of newlines
     for(i=0, i<page, i++)          // Count to number above
          output("\n");             // Output a newline
}                                   // End of while
status("Done!");                    // Text in status line.
```

4. Make a nice little table with temperatures in Celsius and Fahrenheit. Count from 0 to 100 Celsius with ten degrees steps:

```
int c, f;                       /* Declare two variables */
while(c<=100) {                 /* while c less than or equal to 100 */
     output(c" degrees Celsius = ");/* output Celsius */
     f=c*18/10+32;              /* The formula. Made to work */
                                /* without floating points */
     output(itoa(f) + " degrees Fahrenheit! \n"); /* output result  */
     c+=10;                     /* Add ten to c. */
}                               /* loop */
```

5. Create a routine that make a division with an innumerable number of decimals:

```
int a, b, n;                    /* Declare all variables needed     */
a=355;                          /* Set the operands of the division */
b=14;
output(itoa(a) + " / " +
     itoa(b) + " = ");          /* View the evaluation to the user  */
output(a/b);                    /* Calculate...                     */
if(a%b) {                       /* If there remains anything...     */
     output(".");               /* Display a dot.                   */
     for(n=0;n<100;n++) {       /* Only 100 decimals this time!     */
          a=a%b*10;             /* Evaluate next left operator      */
          output(a/b);          /* View the next figure             */
     }
```

```
}
output("\n");                     /* Make an extra newline             */
```

6. Perform the same task as above. This is an example program showing some simple moves to make the code run faster:

```
int a=355, b=14, n;
string s=itoa(a) + " / " + itoa(b) + " = " + itoa(a/b);
if(a%b)
     for(s+=".";n++<100;)
          s+=((a=a%b*10)/b);
output(s + "\n");
```

7. Try out the recursive possibilities of FPL:

```
int b;
int label(int &);
label(b);
output(b"\n");
exit;

int label(int &b)
{
     if(++b<5) {
          while(1) {{{{{{label(b);break;}}}}}}
     }
     return;
}
```

F Error Messages

DIVISION BY ZERO
Division by zero is not a permitted mathematical move (note that this may also appear with the remainder operator “%”).
See division on page 34 or remainder on page 35.

FILE
Something about the specified file is wrong. Check the file name again.

IDENTIFIER NOT FOUND
The variable or function is not found!

IDENTIFIER ALREADY USED
This name is already used by an identifier. You cannot use it too!

ILLEGAL ARRAY
You cannot create an array with less than one element, access elements with higher number than you declared it to or access negative elements. When using the resize keyword the variable name must already be an array.
See the variables chapter on page 12 or resize on page 59.

ILLEGAL ASSIGN
FPL does not support that type of compound assignment .

ILLEGAL BREAK
Break can ONLY be used within some kind of loop (while, for or do) or switch, and this wasn’t such a statement or you tried to break out of an illegal amount of levels!
See break on page 45.

ILLEGAL CASE
The ‘case’ keyword can only be used within a switch() statement!
See case on page 46 and switch on page 65.

ILLEGAL CONDITION OPERATOR
Conditional operators should be used like a ? b : c. If any of the ? or : characters are missing, this might become the error message!
See conditional operators on page 39.

ILLEGAL CONTINUE
Continue can ONLY be used within some kind of loop (while, for or do), and this wasn’t such a statement.
See continue on page 48.

ILLEGAL DECLARATION
The declaration was not placed in the beginning of a block, or a function isn’t declared the way it was prototyped.

ILLEGAL DEFAULT
The ‘default’ keyword can only be used within a switch() statement!
See default on page 49.

ILLEGAL PROTOTYPE
Something in the just parsed function prototype was not using correct syntax and/or keywords.

ILLEGAL REFERENCE
A variable reference was used the wrong way!

ILLEGAL RESIZE
Resizing of an array did not follow correct syntax!
See resize on page 59.

ILLEGAL STATEMENT
The statement was not allowed at this point. It might be a numerical expresion where a string expresion was expected or vice versa. It might also be a statement that cannot be placed here.

ILLEGAL STRING INDEX
Trying to read a negative column off a string will most likely cause this.

ILLEGAL VARIABLE TYPE
You mixed string and integer variables/ expressions illegally or you tried to send to wrong kind of variable to a function.

INTERNAL ERROR
The library's internal data/structures has not been handled properly by the coder of the software you're using. Report this immediately to him/her. (See the docs of the software you're controlling with FPL.) This is not the FPL programmer's fault.

INSIDE FUNCTION NOT FOUND
A function that has been prototyped to be an `inside' function was not found! Notice that the function *MUST* be prototyped and declared the same way. A function prototyped as `int' but declared as `void', will **NOT** be found by the FPL interpreter!

INCOMPLETE STATEMENT
The statement just parsed did not include a required action such as a variable change or a function call.

MISSING APOSTROPHE
The program most likely contains a character within single quotes ('), where the following quote is missing.

MISSING ARGUMENT
The function / keyword you tried to invoke, requires more arguments!

MISSING BRACE
Some places just require braces. I.e block statements that have an initial brace or Array assigns that must end with a close brace.

MISSING BRACKET
Array references and other need an open bracket ([) and a close bracket (]).

MISSING COLON
There should be a colon (:) here, but it's not!

MISSING OPERAND
Making an expression that starts with an operator most likely causes this error.

MISSING PARENTHESES
The statement just interpreted had a lack of parenthesis.

MISSING SEMICOLON
All statements must be separated with semicolons. This wasn't!

MISSING WHILE
There is a missing ‘while’ keyword after the do-while statement!

OUT OF MEMORY
The system has run out of memory. This isn’t your fault, and there is nothing you can do about it but decrease memory usage. Note that if **any** internal FPL allocation call fails, the program stops with this error code.

OUT OF STACK
The stack size hit the upper limit. The program is too recursive!

PARAMETER IS OUT OF RANGE
The parameter(s) specified must be within a certain range, which you clearly did not manage. (Refer to the docs of the action you just tried.)

PROGRAM STOPPED
The program was stopped by a force outside the library. This is not a real error message but more like information.

READ ONLY VIOLATION
You tried changing the contents of a read-only variable.

SYNTAX ERROR
Severe error in your writing. Rethink and try again. All functions, labels or variable identifiers MUST start with a letter .

TOO MANY PARAMETERS
Either a function is called with too many parameters, or there is a missing paranthesis after the parameters.

UNBALANCED COMMENT
There is no corresponding end of comment.

UNEXPECTED END OF PROGRAM
The program ended where it wasn’t supposed to! Probable cause is a lack of closing comment sign or lack of a closing paren of some kind.

UNEXPECTED INTEGER STATEMENT
The interpreter expected a string expression, but read an integer one.

UNEXPECTED STRING STATEMENT
The interpreter expected an integer expression, but read a string one.

UNKNOWN
The programmer of the host software sent a strange error code to FPL which there is no corresponding error message to!

UNMATCHED BRACE
This brace has no corresponding one.

Symbols

Numerics

A

B

C

D

E