Pépé le Manual

Version 1.03

by Philippe Guillot

pguillot@gmail.com

Pépé le compiler is a freeware standard Pascal compiler for the **Android** platform, that runs directly on the handheld. Pépé generates true native ARM executable code from standard Pascal source program.

Console applications can be generated on board in the train, while waiting the bus, while fishing, on the beach, or what ever you are, either on a tablet or on a smartphone, provided it runs with the **Android** operating system and with an ARM processor.

The name of the story. This compiler has first been developed for the PalmOS platform, generating on board true 68 000 native code. The name of this application was PP for « Palm Pascal » and to be pronounced « Pépé » with the french accent.

When Palm devices migrated to the ARM processor, a new version of this compiler, generating true ARM native code was developed.

Pépé le compiler is the successor of these two compilers, and the name suggests that it still allows to program Pascal in the palm of the hand.

And voilà.

Pépé la reference

Installation. To install Pépé le compiler just copy the "Pepe.apk" file on the external memory of the device, and launch it by using the "File" application.

User manual. When the application is launched, a list of files appears. Files recognized by Pépé le compiler are :

- Pascal source files, with the extension « .pas »
- Executable files, with the extension « .exe »

The menu allows to

- create a new source file.
- Choose the default memory location internal or external (see further). By default, the external location is set.

A short click on a source file compiles it and builds the executable.

A short click on an executable file runs it.

A long click displays a context menu. This context menu depends on the type of the file.

On a source file, the context menu allows to :

- **edit** : launch the editor in order to edit the file
- **build** : compiles the file and builds the executable. This is equivalent to a short click on the file name.

On an executable file, the context menu allows to :

- **inspect** : launch a disassembler to show the generated ARM instructions.
- **run** : execute the file. This is equivalent to a short click on the file name.

The source files may be stored either, by default in the external memory (sdcard) or in the internal memory.

The external memory is in the directory `/mnt/sdcard/Pepe/`.

The internal memory is in the directory `/data/data/pp.compiler/`, which is the internal storage location of the Pépé application.

The executable files are always stored in the internal memory, as this is the only location that may be mapped as executable zone.

The programs are executed in a console that allows all the features of the Standard Pascal.

Le language characteristic

The language syntax is mostly compliant to the Standard Pascal ISO 7185:1990 level 1. Documentation on this standard may be found at moorecad.com/standardpascal

In particular, it supports:

- non local goto,
- procedure and function parameters,
- standard syntax for new and dispose for pointer to record with variant.

In addition, the language supports some Turbo Pascal extensions such as strings, @ operator, bitwise operators, etc.

The program name in the program header defines the file name of the created executable file.

```
program Minimal;
begin
end.
```

The compilation creates an executable file "Minimal.exe" in the internal storage location.

Non standard and implementation defined features

In this section, are presented the main non standard features implemented for convenience.

There is no run time overflow control nor range check.

```
type days=(mon,tue,wed,thu,fri,sat,sun);
var d : day;
begin
   d:=sun;
   d:=succ(d); // does not generate error
end;
```

Order of the declaration. In the declaration section of a block, the constants, variables, types, procedures and functions can be declared in any order.

Forward domain type for pointer types must be declared in the same type declaration block.

The only standard constraint on the order of the declaration is for the label. The labels must be declared first, before any other declaration.

Constant precomputation. Most constant expression computable at compilation time are allowed anywhere a constant is expected.

```
const
      n=5;
      nn=2*n;
type
      table : array[0..nn-1] of n..3*n;
```

Predefined function such as sin, cos, exp, sqrt, log are not precomputed in constant expression.

Strings. Strings of character can either be defined through the standard definition `packed array[1..n] of char` or with the non standard keyword **string**.

`string` is equivalent to a `packed array[1..255] of char`.

`string[n]` is equivalent to a `packed array[1..n] of char`. The maximum allowed value of n is 255.

The first character of a string s, is accessed by s[1].

Strings are implemented with a terminal zero. The assignment of a string copy the source until the terminal zero only.

The operator + is applicable to string and means the concatenation.

The comparison operators <, >, <=, >=, =, <> apply the lexicographic order.

The predefined function length returns the actual length of a string, which is the number of characters until the terminal zero.

The string type is compatible with the char type and with any other string type of any length.

The string type is allowed as the return type of a function.

Hexadecimal notation. The character '$' starts hexadecimal representation of integer.

```
const mask = $0000ffff;
```

Comments. The C++ style comments is supported. The characters following // are ignored until the end of the line.

```
// this is a comment, ignored by the compiler.
```

Usual Pascal style comments can be nested. When several comments are open inside a comment, they all must be closed.

```
(* … (* … *)  these characters are ignored until the final *)
```

Identifiers. The underscore character '_' is allowed inside an identifier.

```
var my_name : string;
```

Pointer type. A predefined type 'pointer' is assignment compatible with all other pointer types.

Address operator. The address operator @, applied to a variable, a function or a procedure, returns a pointer type equal to the address of the variable.

```
var
      p : pointer;
      a : integer;
...
p:=@a;
```

Remark : When this operator is applied to a var formal parameter of a procedure or a

function, it returns the value passed to the function, which is the address of the variable.

Inline attribute. This attribute, following a function or procedure declaration, allows to insert code in place where the function is called.

When an inline procedure or function is called, the compiler just insert the code, without any prolog and epilog.

```
function even(x:integer):boolean; inline($e2000001,$e2200001);
// and r0,r0,#1
// eor r0,r0,#1
```

Typeless var parameter. Typeless formal parameter are allowed in the header of a procedure or a function. In this case, any variable is compatible with the actual parameter. The value passed to the procedure or function is the adress of the actual parameter.

```
procedure memmove(var dest, src; numbytes : integer);
type bytearray=array[0..1000]of 0..255;
var psrc, pdest : ^bytearray;
    i : integer;
begin
  psrc:=@src;
  pdest:@dest;
  for i:=0 to numbytes-1 do pdest^[i]:=psrc^[i];
end;

var a,b : any_type;

begin
  memmove(a,b,sizeof(any_type));
  ...
```

Const keyword in a formal parameter list. The `const` keyword is allowed in a formal parameter list. It means that this parameter cannot be modified inside the procedure, and allows some code optimization.

If the parameter a scalar less that 4 bytes long, then the value of the actual parameter is passed to the called procedure.

If the parameter is a string, a set or is more than 4 bytes long, then the address of the actual parameter is passed to the called procedure.

Unsigned integer. A type declared as a subrange of the integer type and with a non negative lower value is considered as an unsigned integer, for example:

```
type byte = 0..255;

type shortint = 0..65535;
```

The value of the division is not the same when they are applied to signed and to unsigned parameters.

- The signed operator is applied if one of the two parameters is signed.
- The unsigned operator is applied if the two parameters are unsigned.

Shift operators. Additional keywords `shl` and `shr` are the usual shift operator. They have the same precedence as the multiplication.

Then the first parameter is signed, then, when applying the `shr` operator, the sign bit is propagated, such that `x shr 1` is the same as `x div 2`.

Boolean operators applied to integer type. The Boolean operators **and**, **or** and **xor** can be applied to integer type. In this case, they perform bitwise operations.

Xor operator. Additional keyword **xor** performs the exclusive or with the same precedence as or.
Applied to integer type, the **xor** operator performs bitwise exclusive or.
Applied to set type, the xor operator performs the symmetric difference of the set.

Sizeof function. A predefined function `sizeof` is applicable to a variable identifier or a type identifier. It returns a constant equal to the number of bytes of the variable or the type.

Ordinal value of the enumerated constants. By default, the constants defined in an enumerated type are the integers starting at zero and incremented by one on each enumerated constant.

Any integer value may be assigned to these constant, for example :

```
type DayOfWeek = (mon = 1, tue, wed, fri, sat, sun);
```

Any integer type value is supported. The values must be declared in increasing order.

The size of an enumerated may be 1, 2 or 4 bytes, depending on the value of the greatest constant.

```
type enum = (a=0, b=$ffff, c=$10000); // is 4 byte long
```

Functions invoked as procedures. If a function has a side effect and if the value is not required, then it may be invoked like a procedure.

```
function double(var x : integer):integer;
begin
  x:=2*x;
 double:=x:
end;
var x : integer;
begin
  x:=2;
  double(x); // the value is ignored
..
```

Use of the keyword nil as actual parameter. The keyword nil is allowed as actual parameter congruent to a var formal parameter, or a const string parameter. In this case, the value of nil is passed to the procedure. This allows to define system functions with a pointer parameter and to use the ability of passing nil if the corresponding parameter is not required for assignment.

Procedural types. Standard Pascal allows procedural parameter. In addition, a procedural type exists.

```
type proctype = procedure(var i : integer);
type functype = function(var c : char): real;
```

The value or a variable declared of this type is the address of the code to execute to run this function or procedure. The pointer type is assignment compatible to procedural types.

Default case statement. In a case statement, no error is generated if some case labels are missing from the selector type.

Moreover, an optional **else** statement is allowed for other labels

```
case n of // n is an integer
  0 : x:=1;
  1 : x:=-1;
  else x:=0;
end;
```

Inline assembler. To get faster and smaller code, it is possible to define a procedure or a function in the ARM assembly language, with the keyword **asm** after the procedure header.

```
function Swap32(x:integer):integer;
asm
   eor  r1, r0, r0, ror 16
   bic  r1, r1, $00ff0000
   mov  r0, r0, ror 8
   eor  r0, r0, r1, lsr 8
   mov  pc, lr                 // return from subroutine
end;
```

Special use of the keyword nil. The keyword **nil** is allowed as actual parameter congruent to a **var** formal parameter or a **const string** parameter. In this case, the value of **nil** is passed to the procedure.

```
function zero(var x : integer);
begin
  if @x<>nil then x:=0;
end;

zero(nil);  // does noting
```

This ability is useful for system call that take a pointer parameter and assign a result only if this pointer is not nil. These system call can be defined in Pascal style with a var parameter. The value **nil** can be passed if the parameter is not required.

Le Pépé language reference

Symbols, key words and lexical tokens

Characters :

- letters `'a'...'z'`.
  The Pascal language is not case sensitive. Capital letter have the same significance as low case letters.
- Digits `'0'..'9'`;
- hexadecimal digits : `'0'..'9', 'a'..'f'`;
- special symbols:

```
+ - * / = < > [ ] . , ( ) : ; ^ @ { } ' $
```

Alternative token `@` is not a replacement for `^` and standard alternative tokens `(.` and `.)` for [and] are not supported.

- double characters :

```
<=    >=    :=    ..    (*    *)
```

- key words : they are reserved word that cannot be redefined.

```
and     array    asm      begin    case       const
div     do       else     end      for        function
goto    if       in       label    mod        nil
not     of       or       packed   procedure  program
record  repeat   set      shl      shr        string
then    to       type     until    var        while
var     xor
```

- predefined symbols : these symbols are predefined, but are not keyword nor identifiers.
  - Directives

```
inline    forward
```

  - Predefined identifiers

```
abs       arctan    boolean   char      chr       close
cos       dispose   eof       eoln      exp       false
get       input     integer   length    ln        maxint
new       odd       ord       output    pointer   pred
put       read      readln    real      reset     rewrite
round     sin       sizeof    sqr       sqrt      succ
text      true      trunc     write     writeln
```

Identifiers

An identifier starts with a letter followed by a sequence of letters, digits, and underscore '_', that is different form any key word. They serve to identify constant, variables, types,

procedures and functions.
There is no limit for the length of an identifier. It is only bounded by the available memory.

$$identifier = letter \{ letter \mid digit \mid \_ \}$$

Examples: `Writeln, print_array, String64`

Labels

A label is a sequence of at most 4 digits. They are used together with the `goto` instruction.

Examples: `99, 666, 9999, 0`

Numbers

A number is either a signed integer or a signed real.
A signed integer is represented by an optional sign '+' or '-' followed by either a sequence of digits.
The sequence of digits may be either a sequence of decimal digits, or the character '$' followed by a sequence of hexadecimal digits.

Examples: `15, -1000, $aaaa, -$ffffffff, +2011`

An integer is internally implemented as a 32 bits word. The range of signed integer is the interval $[-2^{31}, +2^{31}-1] = [-2147483648, +2147482647]$. It allows about 16 digits precision in computations. Results are rounded to the nearest value.

A signed real is represented by an optional sign, a mantissa and an optional exponent. The mantissa is an integer part, followed by an optional fractional part.

Syntax for reals:

decimal_number ::= [+ | -] *digit_sequence* [**.** *digit_sequence*] [**e** [+ | -]*digit_sequence*]
hexadecimal_number ::= **$** *hexadecimal_digit_sequence*
digit_sequence ::= *digit* [*digit*]

Examples : `13.25e+6, 3.1415926, 1e-30`

A real is internally implemented in a IEEE 754 double precision floating point format. The type size is 8 bytes (64 bits).

`sizeof(real)` returns 8.

The minimum positive value is 4.9406564584124654 x 10^{-324} (minimum subnormal positive double).

The maximum positive value is 1.7976931348623157 x 10^{308} (maximum double).

Strings

Litteral strings are delimited by quotes. To insert a quote in a string, it must be repeated once.

```
const My_mame = 'My name is ''Philippe Guillot''';
// My name is 'Philippe Guillot'
```

A string must be defined in a single line.

Any editable ascii character is allowed in a string.

Constant string expressions are precomputed and can define a constant. This allows to insert non editable characters in a string, and to define long string constants on several lines :

```
const
     // how inserting non editable character in a string ?
   two_lines_string = 'first line' + chr(10) + 'second_line';
     // how defining a string on several lines ?
   string_defined_on_two_lines=
       'Philippe'+
       'Guillot';
```

Comments

Section of program between `(*` and `*)`, and between `{` and `}`, are not interpreted by the compiler.

NaPP supports also the C++ style comments. Portions of code after `//` are ignored until the end of the line.

The `(* ... *)` and `{ ... }` can be nested.

```
(* This start a comment
   this is ignored. Any ascii chàractér cañ be üsed
   (* This starts a new comment of level 2
    *) This only closes the comment opened in the previous line
   This text is still ignored
*) // this closes the main comment.
   x:=x+1;  // this line is now interpreted by the compiler
            // provided it is not embedded in a comment
```

Blocks

A block is a set delimited by a beginning and an end. It contains declarations and instructions. It may define a procedure, a function or a program.

All the identifiers and labels declared in a block are local to this block.

Syntax

block ::= *declaration_part* *statement_part*

declaration_part ::= *label_declaration_part*
{ (*constant_definition_part* | *type_definition_part* | *variable_definition_part* | *function_definition_part* | *procedure_definition_part*) }

statement_part ::= *compound_statement*

Labels

label_declaration_part ::= [**label** *label* { **,** *label* } **;**]

A label is a sequence of 4 decimal digits. The declared label are those used in the statement part.

```
label 6, 666, 1234, 99;
```

Constants

constant_declaration ::= [**const** *constant_definition* **;** { *constant_definition* **;** }]

A constant definition introduces an identifier to denote a value. It defines the constants usable in the block.

constant_definition ::= *identifier* *expression*

The expression must define constant value which can be evaluated at compilation time.

```
const message = 'hello world !';
      n = 22;
      m = 1 shl n;
```

Types

type_declaration ::= [**type** *type_definition* **;** { *type_definition* **;** }]

The type determines the values that a variable can take. A type definition introduces an identifier to denote a type.

type_definition ::== *identifier* **=** *type_denoter*

type_denoter ::= *type_identifier* | *simple_type* | *pointer_type* | *structured_type* | *string_type*

To be continued...

Predefined functions and procedures

Abs

Description Returns the absolute value of the given numerical parameter.

Comment This function accepts both real and integer parameters, and it returns a value of the same type.

Arctan

Description Returns the angle in radian whose tangent is given.

Declaration
```
function arctan(x:real):real;
```

Chr

Description Returns the character that correspond to a given integer value. Only the 8 less significant bits are taken into account. The others are ignored.

Declaration
```
Function chr(n:integer):char;
```

Example `Chr(10)` returns the *newline* character, that break a string on an new line.

Close

Description This procedure close an open file or text.

Comment
```
close(f);
```
The parameter f is any file or text variable.
This procedure has no effect on `input` and `output` predefined text variables.

Clrscr

Description Clears the screen and set the cursor at position (1,1)

Declaration
```
Procedure clrscr;
```

Comment This procedure is equivalent to `rewrite(Output);`

Cos

Description Returns the cosine of the given angle in radian.

Declaration
```
function cos(x:real):real;
```

Dispose

Description This procedure free the memory that has previously been allocated for the pointer parameter .

Syntax

WhereY

Description Returns the y position of the cursor, in the range 1..ScreenHeight

Déclaration

```
function wherey:integer;
```

Eof

Description

Comment

Eoln

Description

Comment

exp

Description Returns e $(\approx 2{,}71\cdots)$ raise to the power x.

Declaration

```
function exp(x:real):real;
```

Get

Description

Comment

GotoXY

Description Set cursor position on the screen

Declaration

```
procedure gotoxy(x,y:integer);
```

Comment The origin is located at (1,1) on the upper left corner of the screen. If the coordinates are out of the screen bound, they are forced either to the minimum value, equal to 1, or to the maximum value that depends on the current screen dimensions.

Halt

Description Brusquely leaves the program with a given return value, and returns to the console.

Declaration
```
procedure halt(output_code:integer);
```

Length

Description Returns the length of a string

Déclaration
```
function length(const s : string): integer;
```

Comment This function returns the actual number of characters of a string until the terminal zero.

Ln

Description Returns the natural logarithm in base e

Declaration
```
function ln(x:real):real;
```

New

Description

Syntax

WhereY

Description Returns the y position of the cursor, in the range 1..ScreenHeight

Déclaration
```
function wherey:integer;
```

Odd

Description This function returns `true` if the integer parameter is odd, and `false` if it is even.

Declaration
```
function odd(x:integer):boolean;
```

Ord

Description This function is applicable to any scalar type. It returns the integer equal to the scalar value.

Comment If `n` is an integer, then `ord(n)` returns `n`
If `c` is a character, then `ord(c)` returns the ascii value of `c`
`Ord(false)` returns `0`
`Ord(true)` returns `1`

Pred

Description This function returns the predecessor of the parameter. It takes any ordinal type and returns a value of the same type.

Comment This function just decrement its parameter. If the parameter is an enumerated type with assigned values, then the returned value may not be a valid value.

```
type enum=(a=1, b=5, c=10);
var x : enum;
begin
  x:=pred(b); // the ordinal value assigned to x is 4 !
end;
```

Example

```
x:=pred(x);
```

Put

Description

Comment

Read

Readln

Description

Comment

Reset

Description

Comment

Rewrite

Description

Comment

Round

Description This function returns the nearest integer of the real parameter.

Declaration
```
function round(x:real):integer;
```

ScreenHeight

Description Returns the current height of the screen

Declaration
```
function screenheight:integer;
```

Comment The value returned by this function may change while the program runs, depending on the orientation of the device.

ScreenWidth

Description Returns the current width of the screen

Declaration
```
function screenwidth:integer;
```

Comment The value returned by this function may change while the program runs, depending on the orientation of the device.

Sin

Description Returns the sine of the given angle in radian.

Declaration
```
function sin(x:real):real;
```

Sizeof

Description Returns a constant integer equal to number of bytes of a variable or a type.

Comment This function is applicable to any variable and any type identifier.

Example
```
const n = sizeof(integer);
```

Sqr

Description Returns the square of the actual parameter.

Comment This function is applicable to both integer and real type. The returned value is of the same type as the parameter.
If the parameter is a constant, then the value is precomputed at compilation time.

Example
```
const n = sqr(sizeof(integer));
```

Sqrt

Description Returns the square root of the parameter.

Declaration

```
function sqrt(x:real):real;
```

Succ

Description

Comment

Trunc

Description This function returns the greatest integer lower than the real parameter.

Declaration

```
function trunc(x:real):integer;
```

WhereX

Description Returns the x position of the cursor, in the range 1..ScreenXWidth

Declaration

```
function wherex:integer;
```

WhereY

Description Returns the y position of the cursor, in the range 1..ScreenHeight

Declaration

```
function wherey:integer;
```

Write

Description

Comment

Writeln

Description

Comment

Les compiler directives

{$i *file_name*} This directive tells the compiler to include the file file_name in the compiled text.

{$define *name*} Defines the symbol name.

{$undef *name*} If the symbol name was previously defined, this directive undefines it.

**{$ifdef *name*}** Starts a conditional compilation. If the symbol name is not defined, this will skip the compilation of the text that follows it, up to the first {$else} or {$endif} directive.

If the symbol name is defined, then the compilation continues up to the {$else} directive.

**{$else}** This directives switches conditional compilation. If the text delimited by the previous {$if name} directive was ignored, then the text that follows {$else}, up to the {$endif} directive is normally compiled, and vice versa.

{$endif} This directive ends a conditional compilation sequence.

{$echo *message*} Displays a message on the console during compilation.

How to...

Function Readkey

This function waits and returns a character hit on the keyboard. It is predefined in Turbo pascal, and may be written as follows in Standard Pascal :

```
function readkey:char;
var c : char;
begin
  c:=input^;
  get(input);
  readkey:=c;
end;
```

System calls

System calls can be performed either with ARM assembler defined function, or inline defined function.

```
mov r7,#SYSCALL_NUMBER
swi 0
```

The values of the syscall numbers are in the file "linux-syscall.h" provided in the native development kit (NDK) of the android system.

Description and declarations of the system functions may be found for example in http://www.linux.die.net/man/

Example : function times

This system function returns a number of clock ticks from an arbitrary origin and assign values of the user time, system time, children user time, and children system time.

The number of ticks per second is 100. This value is given by the macro "#define HZ 100" in the header file "param.h" of the Android NDK. The *syscall number* of this function is 43.

a) assembler

```
Program syscall;

Type tms=record
  utime, stime, cutime, cstime : integer;
end;

function times(var t : tms):integer;
asm
  mov r7,#43
  swi 0
  mov pc,lr
end;

begin
  writeln(times(nil));
end.
```

b) as an inline function

```
Program syscall;

Type tms=record
  utime, stime, cutime, cstime : integer;
end;

function times(var t : tms):integer; inline ($e3a07000+43, $ef000000);

begin
  writeln(times(nil));
end.
```

Pépé le assembler reference

In an assembler defined function or procedure, the comments are the same as in Pascal section, with `{...}`, `(*...*)` or `//` delimiters.

The processor registers

The ARM processor has 16 general 32 bit registers, r0 to r15. Some of them are dedicated for special use.

- r0, r1, r2, r3 are used to pass parameters to procedures and functions.
- r0 and r1 are used to return values from functions.
- r4,... r11 are supposed to keep the same value after a procedure call.
- r9 is the base address of shared libraries. It is uses for system calls.
- r10 contains the base address of global variables.
- r11 in the frame register for local variables and parameters
- sp (stack pointer) is a synonym for r13
- lr (link register) is a synonym for r14. It contains the return address of procedures.
- pc (program counter) is a synonym for r15. It contains the address of instructions to execute.

In addition, the *current program status register* (CPSR) contains contains the following flags:

31	30	29	28	27	26	...	8	7	6	5	4	3	2	1	0
N	Z	C	V	Q	-	...	-	I	F	T	M_4	M_3	M_2	M_1	M_0

C	carry
Z	set if the result is Zero
N	set if the result is Negative
V	is set if an oVerflow has occurred in a signed operation
Q	overflow in DSP enhanced instructions
T	set while processor runs in THUMB mode
I	disable IRQ when set
F	disable FIQ when set
M_4...M_0	defines mode: • 10000 : User • 10001 : FIQ • 10010 : IRQ • 10011 : Supervisor • 10111 : Abort • 11011 : Undefined • 11111 : System

Constant definitions

These directives define constant in the code.

- **dcb**: *define byte constant* (8 bits). The operand is a list of either integer constant, or litteral strings delimited by simple quotes.
  Example:

```
 dcb   'Hello World',0
dcb   1,2,4,4,5,$ff
```

  Each section defined by a dcb directive is word aligned and eventually completed by zeroes. The strings are not completed by a zero. The terminal zero must be defined explicitely.
- **dch**: *define half word constant* (16 bits).
  The operand is a list of 16 bit integer constant.
  Example:

```
dch   1000, 1001, 1002
```

  Each section defined by a `dch` directive is word aligned and eventually completed by zeroes.
- **dcd**: *define word constant* (32 bits).
  The operand is a list of 32 bit integer constants.
  Example:

```
dcd   $ffffffff,$73616861
```

- **dcf**: *define float constant* (64 bits IEEE 754 double precision).
  The operand is a list of real valued constant. If an integer is specified, then it is automatically converted into IEEE 754 64 bit double precision format.
  Example:

```
dcf   3.1415,  0.5,   1
```

Local labels

A label is a sequence of letters, digit and underscore that starts with the **@** character, for example.

```
@loop_1
```

It defines the value of destination based on the value of the PC register.

- In branch instructions

```
@loop
    ...
    bne @loop
```

- in PC relative data transferts:

```
 ldr r0,@size
  ...
@size
   dcd 5000
```

The value of the offset is limited to +/- 4092 bytes for word or unsigned byte data transfer, and to +/- 252 bytes for signed and half word data transfer. If the offset is beyond these limits then an error is generated.

- in PC relative address definition:

```
@my_string
    dcb   'Hello world!',0
        ...
    mov  r0,@my_string
    bl   PrintString
```

The assembler automatically convert this instruction into the suitable `add r0,PC,#offset` or `sub r0,pc,#offset`, depending on the sign of the offset. Any offset is not implementable in a single instruction. If the value of the offset cannot be defined in a single instruction, an error is generated.

The scope of a local label is limited to the current assembly bloc.

Condition code.

Each instruction is executed conditionally to the state of the code condition register and according to a suffix appended to the mnemonic. The remarkable property of the ARM processor is that this conditional execution is applied to any instruction, not only branch instructions as usual in most processors.

Suffix	Flag	Meaning
eq	Z=1	equal
ne	Z=0	not equal
cs	C=1	carry set
hs	C=1	unsigned higher or same
cc	C=0	carry clear
lo	C=1	unsigned lower
mi	N=1	negative
pl	N=0	equal
vs	V=1	overflow
vc	V=1	no overflow
hi	C=1 and Z=0	unsigned higher
ls	C=0 or Z=1	unsigned lower or same
ge	N=V	signed greater than or equal
lt	N<>V	signed less than
gt	Z=0 and (N=V)	signed greater than
le	Z=1 or (N<>V)	signed less than or equal
al	-	always

Example:

```
function abs(n:integer):integer
asm
   cmps   r0,#0      // compare r0 with 0
   rsbmi  r0,r0,#0   // if r0 is negative, then negate it
   mov    pc,lr      // return from subroutine
end;
```

Branch

- Branch and Exchange This instruction copies the content of a register RDest into the program counter PC ignoring the least significant bit b0. If this bit equals 1, then, it switch to THUMB mode, else, it remains in ARM mode. As the program may be called from THUMB mode, the BX instruction must be used at the end of a program to return to the OS.

  Syntax: (**bx** | **blx**)[cond] RDest
  Rdest is the register that contains the destination address.
  Bx stands for branch.
  Blx stands for branch with link, i.e. the address after this instruction is loaded into the link register.
  Example:

```
bx    r0    // jump to r0 and swicth to mode defined by bit 0
blx   r12   // save return address in lr and jump to r12
```

- Branch This instruction increment the counter program by the signed 24 bit immediate value contained in the instruction.
  - Syntax: B{cond} @label | function identifier | procedure identifier
  - Example:

```
   b   Error  // branch to the procedure Error defined above
@here
   ...
   bne @here  // branch to @here if z flag equals 0
   b   @there // unconditionnal branch to there
   ...
@there
```

- Branch with link Same as Branch, but load the link register LR with the content of the program counter that follows immediately this instruction. This is used for call to subroutines. The subroutine may be local, if used with a local label, or external, when used with the identifier of a defined function or procedure.
  - Syntax : BL{cond} @label | function identifier | procedure identifier
  - Example:

```
   bl    @convert  // branch to local subroutine convert
     ...
@convert
     blne  Pascal_proc // conditionnal call to an external Pascal
procedure
```

Data processing.

The general syntax is

opcode{*cond*}{**s**} *register*{,*register*},*operand*

where

operand ::= *register*{ **,** *shift*} | #*expr*

shift ::= *shiftname*
register | *shiftname* **#***expression*| **rrx**

shiftname ::= **asl** | **lsl** | **lsr** | **asr**| **ror**

expression is either a constant integer valued expression computable at compilation time. It is coded as a 8 bit value combined with an even rotation. If the value does not fit in this constraint, then an error is generated.

or a label, only for **mov** instruction. In this case, the value is the offset relative to the current position of PC.

opcode can be

and:	bitwise logical and	*dest* := *op1* and *op2*
eor:	bitwize logical exculsive or.	*dest* := *op1* xor *op2*
sub:	arithmetic substraction.	*dest* := *op1* - *op2*
rsb:	reversed substraction.	*dest* := *op2* - *op1*
add:	arithmetic addition.	*dest* := *op1* + *op2*
adc:	addition with carry.	*dest* := *op1* + *op2* + C
sbc:	substraction with carry	*dest* := *op1* - *op2* + C - 1
rsc:	reversed substraction with carry	*dest* := *op2* - *op1* + C - 1
tst:	bit test	*op1* and *op2*
teq:	equality test	*op1* eor *op2*
cmp:	comparison	*op1* - *op2*
cmn:	compare negate	*op1* + *op2*
orr:	bitwize logical or.	*dest* := *op1* or *op2*
mov:	move	*dest* := *op2* (*op1* ignored)
bic:	bit clear	*dest* := *op1* and not *op2*
mvn:	move bitwise complement	*dest* := not *op2* (*op1* ignored)

Shiftname :

lsl: logical shift left

asl: synonym of lsl

lsr: logical shift right

asr: arithmetic shift right, with sign propagation

ror: rotate right

rrx: rotate right one bit with extent, same as **ror,#1**

It the indicator **s** is specified, then the code condition is updated with the result of the operation.
For test instruction `cmp`, `teq`, `tst` and `cmn`, the condition code is always set, whatever the indicator **s** is specified or not.

Examples.

```
add   r3,r0,r1      // add the content of r0 and r1 and write the result in r3
sbc   r0,r0,#0      // negate r0
mov   pc,lr         // return from subroutine
cmp   r0,r2 lsl 2   // compare r0 with 4 times r2
movs   r1,r2 lsl r3 // shift right r2 by the amount of r3,
                    //and write the result in r1 and update the cc register
movne  r0,@string   // if z=0 then assign the address of @string to r0
                    // implemented as addne (or subne) r0,pc,#offset
```

Count leading zeros

This instruction returns in the destination register the number of zero bits before the first one bit of the source register.

Syntax: **clz**[cond] Rdest,Rsrc
Rdes is the destination register.
Rsrc is the source register.

Example:

```
clz   r1,r0         // count the number of zero digits of r0
mov   r0,r0,lsl r1  // normalise r0 by placing 1 to the most significant bit
```

Program status register transfert

These instructions copy the program status register.

1. Copy program status register to a register R_n:

   Syntax: **mrs**[cond] Rdest **,** (**cpsr** | **cpsr_all** | **spsr** | **spsr_all**)

   Rdest is the destination register.

   cpsr or **cpsr_all** (synonyms) :
   transfer from the current program status register.
   spsr or **spsr_all** (synonyms):
   transfer from the saved program status register (exception mode).

2. Copy program status register from a register R_n

   Syntax: **msr**[cond] (**cpsr** | **cpsr_all** | **spsr** | **spsr_all** | **cpsr_flg** | **spsr_flg**) **,** Rsrc

   Rsrc is the source register.
   cpsr or **cpsr_all** (synonymous)
   transfer all bits to the current program status register.
   spsr or **spsr_all** (synonymous)
   transfer all bits the saved program status register (exception mode).

cpsr_flg
transfer only flag bits N, Z, C, V to the current program status register.

spsr_flg
transfer only flag bits N, Z, C, V to the saved program status register (exception mode).

Multiplication

This instruction multiplies the content of two registers with a result on 32 bits.

Syntax:
mul[cond][s] Rdest,Ra,Rb
mla[cond][s] Rdest,Ra,Rb,Rc

[s]
set condition code if present.

Rdest
is the destination register.

Ra, Rb and Rc
are operands.

mul	multiplication	*Rdest* := *Ra* * *Rb*
mla	multiplication and accumulate	*Rdest* := *Ra* * *Rb* +*Rc*

Long multiplication

This instruction multiplies the content of two registers with a result on 64 bits.

Syntax: (**umull** | **smull** | **smul** |**smlal**) [cond][s] Rdest_lo,Rdest_hi,Ra,Rb

[s] set condition code if present.

Rdest_lo and Rdest_hi
are the the destination registers, low and high part respectively.

Ra and Rb are operands.

umull	unsigned multiplication	(*Rdest_lo,Rdest_hi*) := *Ra* * *Rb*
smull	signed multiplication	(*Rdest_lo,Rdest_hi*) := *Ra* * *Rb*
umlal	multiplication and accumulate	(*Rdest_lo,Rdest_hi*) := *Ra* * *Rb* + (*Rb* ,*Ra*)
smlal	signed multiplication and accumulate	(*Rdest_lo,Rdest_hi*) := *Ra* * *Rb* + (*Rb* ,*Ra*)

```
type
  Int64=record  // 64 bit integer type
    lo,hi : integer;
  end;

//  64 bit multiplication a * b --> r
```

```
procedure Mul64(a,b:Int64;var r:Int64);
asm  // a=(r0,r1)  b=(r2,r3) and address of r is in stack at [r11]
   stmfd   sp!,(r4,r5)   // push them
   umull   r4,r5,r0,r2   // lo_a * lo_b on 64 bits (r4,r5)
   mla     r5,r0,r3,r5   // accumulate lo_a * hi_b in r5
   mla     r5,r1,r2,r5   // accumulate lo_b * hi_a in r5
   ldr     r12,[r11]     // address of r
   stmia   r12,(r4,r5)   // save result there
   ldmfd   sp!,(r4,r5)   // pop them
   mov     pc,lr         // return
end;
```

Data transfert to and from a register

ldr instruction loads a memory location into a single destination register.
str store the content of a single register into a memory location.
The memory location is defined by an address which is defined by the value of a base register plus an offset. The offset is either an immediate value contained in the instruction itself, or the content of an offset register.
The address may be

- pre-indexed: the final address is computed before the transfert.
- post-indexed: the final address is computed after the transfert, the value of the base register being updated with the final address.

In the pre-indexed mode, the base register may be updated too. This is indicated by a **!** in the instruction.

Syntax:

ldr | **str**{cond}{**b** | **h** | **sb** | **sh**}{**t**} *register* , *address*

opcode suffix means

s for sign extension transfert. This is significant for load operation.
b for byte transfert;
h for half word transfert;

address can be

- a local label @label; the final address is an offset from PC.
- a pre-indexed address:
  - **[** *register* **]** ; the address is given by the value of the register.
  - **[** *register* ,**#** *expression* **]** {**!**} ; the address is ths sum of the register value and of the expression.
  - **[** *register* **,** { **+** | **-** } *register* {**,** *shift* } **]** { **!** } ; the address is the sum of the base register value and of the offset register value eventually shifted.

  The **!** indicates that the base register is updated with the address value.
- a post-indexed address
  - [register], #expression
  - [register], {+|-} register {,shift}

Examples

```
@data
      dcd    123456
      ldr    r0,@data  // load r0 with 123456

function StrLength(const s:string):integer;
asm
      add    r2,r0,#1       // memorise starting position
@loop
      ldrb    r1,[r0],1     // load in r1 byte at address r0 and increment
r0
      teqs    r1,#0         // test loaded byte
      bne    @loop          // repeat until terminal zero
      sub    r0,r0,r2       // compute length
      mov    pc,lr          // return from subroutine
end;
```

Multiple data transfert

These instructions transfer data from/to a memory location defined by a base register to/from several registers defined by a bitmap in the instruction code.

Syntax:

< **ldm** | **stm** >{cond} < **fd** | **sd** | **fa** | **ea** | **ia** | **ib** | **da** | **db** > , *register* { **!** }, *register_list*

the suffixes means:

fd: full descending stack, i.e. post-increment load and pre-decrement store.
ed: empty descending stack, i.e. pre-increment load and post-decrement store.
fa: full ascending stack, i.e. post-decrement load and pre-increment store.
ea: empty ascending stack, i.e. pre-decrement load and post increment store.
ia: increment after.
ib: increment before.
da: decrement after.
db: decrement before.

where *register_list* is a list of registers delimited by parenthesis (because braces are comment delimiters in Pascal).

Examples

```
stmfd    sp!,(r0,lr)    // push r0 and lr in a full descending stack
...
ldmfd    sp!,(r0,pc)    // pop r0 and return from subroutine
```

Data swap

This instrucion swap a word or byte quantity between a register and the memory location defined by a base register.

Syntax: **swp**[cond][b] Rdst,Rsrc,'['Rbase']'

[b] if present, it specifies a byte swap, else, it is a word swap.
Rdst is the destination register
Rsrc is the source register
Rbase is the base register

This instruction write the value of the source register to the memory location and load the previous content of the memory location in the destination register. These two operation cannot be interrupted.

```
swp  r1,r2,[r3]  // load r1 with the word addressed by r3
                    // and store r2 at r3
```

Software interrupt

Syntax: **swi**[cond] *expression*
expression is any 24 bit integer value for a comment field ignored by the processor.

This instruction is used to call the operating system. It branches in supervisor mode to the interruption vector 8. In general, the expression is interpreted by the interrupt to perform a given system call.

Breakpoint

Syntax: **bkpt** expression
expression is any 16 bit integer value for a comment field ignored by the processor
This instruction is always executed. It is used to insert a software break point. It branches in abort mode to the interruption vector 12.