Dieter R. Pawelczak

DPas32 - 32 bit Pascal Compiler

The DPas User Manual & Reference Guide

Dieter R. Pawelczak

Fasanenweg 41,
D-85540 Haar,
Germany

For my dear wife Alexandra

About

This is the first BETA release of the *DPas* Compiler. The difference between the previous ALPHA releases is huge, especially concerning the code generation and optimization.

DPas is a Pascal Compiler, that generates directly executable 32-bit Applications without any further Assembling or Linking.

The Compiler structure is a direct code generator, i.e. there are no stages between parsing, scanning and code generation. This makes the source code of the compiler on the one hand very tough, on the other hand, it provides a lot of advantages. The main advantage was, that even with only a very little functionality, the compiler already created executable code. That's, why I already published a preliminary Alpha Version in 1997! This release was just able to understand integer expressions and most standard Pascal commands with a lot of limitations. Nevertheless, with a bit luck, I managed to code the Pro 32 debugger already with this initial release.

Another advantages is, that the compiler requires a minimum of heap memory. Without building a real command tree, the compiler saves a lot of memory. As *DPas* is a DOS product, written with *Turbo Pascal*, this is very important. The compiler needs only memory to store the symbol names. So any source is only limited by the number of symbols.

And last but not least, the compiler is very fast. Again for the same reasons. During the parsing of the source, the compiler already creates code.

In the mean time, the compiler creates highly optimized code. This makes the compiler source code very complex, as each optimization depends on previous read code and following code. Especially hard is, of course, the error tracing. As for each optimization a corresponding test program needs to be written, this is an immense effort. Therefore, *DPas* will remain for quite some time a BETA version.

DPas is an ongoing development. In brief, this is planned for the near future:

- Win32 support
- *DPas* is able to compiler *Pass32*
- *DPas* is able to compile itself

Contents

About 4
Contents 5
Introduction 7

1. Getting Started with DPas 9

1.1 First Example 9
1.2 Compiling with DPas32 10
1.2.1 Compiling Programs and Units 10
1.2.2 Compiling the SYSTEM Unit 11
1.3 The Command Line Arguments 12
1.4 The Standard Program Structure 12
1.4.1 Structure of the Source Code 12
1.4.2 Structure of the Program Code 14
1.5 The Compiler Directives 15
1.5.1 Compiler Control Directives 16
1.5.2 Code Generator Control Directives 17

2. The DPas Language 19

2.1 Types 19
2.1.1 Standard Types 19
2.1.2 Strings 20
2.1.3 Enumeration Types 20
2.1.4 Type substitution 20
2.1.5 Record Types 20
2.1.6 Arrays 21
2.2 Const 21
2.3 Variables 21
2.4 Pascal Constructs 22
2.5 Inline Assembler 22
2.6 Procedure / Function headers 22
2.7 System Unit 23
2.8 RUN TIME LIBRARY 24
2.8.1 CRT32 24
2.8.2 DOS32 24
2.9 Import of Dynamic Link Libraries 24
2.10 Optimization 25
2.10.1 Expressions 25
2.10.2 Multiplication / Division 25
2.10.3 Constant Offsets 25
2.10.4 Assembler Procedures 26

3. Examples 27

3.1 Primes 27

3.2 Direct 27
3.3 Traffic 27
3.4 Passed 27

4. Limits 29

4.1 Symbols 29
4.2 Const 29
4.3 Types 30
4.4 Array Depth 30
4.5 Boolean Expressions 30
4.6 ELSE / END 31
4.7 WITH 31
4.8 ASM 31
4.9 Writeln, ReadLn 31
4.10 Procedure / Function headers 31
4.11 Sets 31
4.12 Error Messages 31

Introduction

What is the *DPas32* Compiler?

The name tells us all: *Pas* stands for Pascal, the number *32* informs us, that the target platform is 32 bit and the prefix *D* comes from my surname. Therefore, *DPas* can not compile *any* Pascal, it can compile *Dieter's Pascal*.

Who uses the *DPas32* Compiler?

Anybody, who wants to work under a fast and stable operating system, i.e. 32 bit DOS. *Pass32* programmers, as *DPas* understands Pass32 DLLs. *Turbo Pascal* programmers, who finally made the jump from 16 to 32 bit.

DPas is a fast 32 bit Pascal Compiler. It is compatible in its syntax to *Turbo Pascal*, but it does not provide the same run time library and it does not provide all features of *Turbo Pascal*. It is a Standard Pascal compiler and does not provide any object oriented programming methods.

DPas supports units in the same manner as *Turbo Pascal* and therefore allows modular programming.

DPas creates highly optimized code, e.g. *DPas* provides optimization of constant expressions, optimization of boolean expressions, optimization of const field expressions, optimization of multiplication / divisions, optimization of processor instructions.

What is *DPas* not?

DPas is not a *Turbo Pascal* clone. It is, of course, possible to port *Turbo Pascal* programs to *DPas* programms, but the effort varies a lot, depending on the *TP* source.

DPas is not a commercial product. It is an ongoing development of 1(!) private person, i.e. me. Therefore, I can not make any guarantees for bug fixes, upgrades, etc. I also can not give support on *DPas*, except, you pay me on the basis of a really high german politician's salary. Nevertheless, I use *DPas* for several projects for myself, so it is in my interest also, to increase the quality of the product

Who should read this manual?

This manual is not a Pascal tutorial, you can find already enough of these in the World Wide Web. This manual gives an introduction on how to use *DPas*. It describes the features of *DPas* as well as its limits (always in comparison to *Turbo Pascal*). In the meantime you learn something about compiler programming and protected mode programming. So, this manual is not only for a *DPas* user, it is also interesting for any protected mode and compiler programmer.

1. Getting Started with DPas

1.1 First Example

Any reader is used to the standard 'Hello, World' - example. Therefore you should not miss it in this manual. The first example prints three times 'Hello, World!' in three different ways:

```
PROGRAM HELLO;
VAR c:STRING = 'HELLO,WO RLD';
    s:STRING;

BEGIN
  writeln('HELLO, WORLD!');
  writeln(c);
  s:='HELLO, WORLD';
  writeln(s);
END.
```

***Example 1:* HELLO.PAS**

The source can be compiled and executed under DOS with:

```
DPas hello
hello
```

This simple application already describes some of the main functions of the compiler:

The generated code is linked together with the Pro32 Dos Extender into a standalone application. In order to perform the `writeln` function, the Compiler links the `SYSTEM` unit to the main program. Try the same example with the `uses` instruction. The `uses` instruction imports and links additional units to the code. In case you import the `CRT32` unit, the standard `readln` and `writeln` are overwritten. The standard `SYSTEM` unit uses DOS functions, whereas the `CRT32` unit access directly the video memory for `writeln` and the keyboard buffer for `readln`. This allows a much faster output, but programs using the `CRT32` unit can not redirect the standard input and output.

```
PROGRAM HELLO2;
USES CRT32;

VAR c:STRING = 'HELLO,WO RLD';
    s:STRING;

BEGIN
  writeln('HELLO, WORLD!');
  writeln(c);
```

```
  s:='HELLO, WORLD';
  writeln(s);
END.
```

***Example 2:* HELLO2.PAS**

When you run the second example, you will notice a lightred output to the screen. Lightred is the default color for the CRT32 unit. You can easily change the value of this color in the CRT32 unit.

1.2 Compiling with DPas32

1.2.1 Compiling Programs and Units

DPas compiles only one program / unit per call. *DPas* does not provide a build feature, nor does it check source code dependencies. This makes compiling a bit more difficult as with *Turbo Pascal* for example. Nevertheless, *DPas* supports modular programming, therefore you just need to re-compile the unit, that actually changed. As long as the interface definition stays compatible, you don't need to re-compile other units, which depend from that modified unit.

In the case of our simple example HELLO2, the CRT32 unit need to be compiled first:

```
DPAS crt32
DPAS hello2
```

If you want to change the default color in the CRT32 unit, you have to recompile the CRT32 unit, and, of course also your application.

Imagine, a second unit: We create a unit HUNIT, that simply writes 3 time 'Hello, World!' and offers a single procedure called WriteHello:

```
UNIT HUNIT;
interface
USES crt32;

PROCEDURE WriteHello;

implementation
VAR s:string = 'Hello, World';
PROCEDURE WriteHello;
begin
  writeln('Hello, World');
  writeln(s);
  s:='Hello, World';
  writeln(s);
```

```
END;
END.
```
***Example 3:* HUNIT.PAS**

Our third example could then look much easier, i.e.:

```
PROGRAM HELLO3
USES hunit;
BEGIN
  WriteHello;
END.
```
***Example 4:* HELLO3.PAS**

Now, we have the following dependencies:

```
HELLO3 --> HUNIT --> CRT32
```

In order to build the application *Example 4* (HELLO3.PAS), you need to compile:

```
DPAS crt32
DPAS hunit
DPAS hello3
```

In case any of these units are already compiled, you don't need to re-compile, of course. If you want to change the color of the hello-message, you can modify `CRT32`. Now, as long as you don't change anything in the procedure interfaces, you **don't** need to re-compile `HUNIT`. You modify the implementation of the Writeln procedure in CRT32, for example, the you re-compile `CRT32` and `HELLO3` and your modifications will have effect.

There is one exception in this rule: the `SYSTEM` unit.

1.2.2 Compiling the SYSTEM Unit

One of the most important features of *DPas* is, that the `SYSTEM` unit comes with complete source code. This makes system specific changes of the compiler very easy.

The system unit is compiled with the compiler switch `/SYSTEM`. The system unit needs to be called `SYSTEM.PAS` for 32 bit DOS-Applications and `WIN32.PAS` for Win32 Applications[1].

Note, the `SYSTEM` unit is the basis for all further units and applications. In case you re-compiled the `SYSTEM` unit, you need to re-compile **all** units, otherwise, your application will terminate with the run-time-error #0, or create an exception.

1.Not supported in DPas Version 1.7

1.3 The Command Line Arguments

DPas expects the following syntax:

```
dpas filename[.ext] [options]
```

As default, *DPas* extends the filename with the '.PAS' extension. The file should be a standard ASCII pascal program, unit or DLL source.

DPas provides the following options:

option	description
-f	skips the linking of the PE-Header / Pro32 dos extender. The result is a binary file, that contains code and data, starting at an offest 0100h.
-w[a]	target is a Win32 GUI application.
-wc[a]	target is a Win32 Console application.
-du	adds debug information of units. The units need to be compiled with the -df option.
-df	creates debug information of the current source. *DPas* creates a .DMP file, which can be read by the Pro32 Debugger in order to display the source code.
-system	allows *DPas* to compiler the system unit.
-mem:xxxx	defines the minimum memory requirement in KBytes.

a. not yet supported by V1.7

1.4 The Standard Program Structure

1.4.1 Structure of the Source Code

A standard *DPas* program begins with the `PROGRAM` instruction. This first keyword `PROGRAM` describes what the compiler should create. Per default, *DPas* creates an application. Therefore the `PROGRAM` instruction is optional.

Instead of `PROGRAM` a *DPas* source could start with the keywords `UNIT` and `DLL`:

PROGRAM	The following source code defines a standalone application. (default)
UNIT	The source describes a UNIT, i.e. a set of identifiers, function and procedures, which can be linked by other UNITs or PROGRAMs.
DLL	This unit does not contain code, it contains only the definition of an external dynamic link library, which is load dynamically during the run time. The unit may contain enum and record type definitions.

Directly after the PROGRAM instruction, units can be imported with the USES command.

The USES command is followed by a comma separated list of module names. The SYSTEM unit is always linked per default and therefore should not appear on this list. The USES command needs to come before any code or data definitions.

Now, typically global types, variables and sometimes even procedure prototypes are defined. In our first example, two strings are defined:

```
VAR c:STRING = 'HELLO,WORLD';
    s:STRING;
```

You already notice a difference to *Turbo Pascal*: global variable definitions can already assign values to the variables. This can either be done with the standard CONST directive, or directly inside a VAR definition. The following examples are valid data definitions:

```
VAR Field:ARRAY[0..24,0..79] OF WORD = $0720;
    Screen:ARRAY[0..24,0..79] OF WORD ABSOLUTE = $B8000;

VAR MaxX,MaxY:INTEGER =  100;
    I,J:BYTE;
    TEST:BOOLEAN;
```

In the first example, a two dimensional ARRAY of WORD is generated. In the opposite to the CONST definitions, you don't have to specify all of the 2000 fields. You can define one initial value, which will be applied for the whole ARRAY.

In the second line, you'll find an ABSOLUTE definition. The specifier ABSOLUTE defines, that the variable is not part of the program's data segment, the variable points to the absolute address as specified. In our case, the field points to $B8000, which is the physical address of the video memory. As both arrays are identically, you probably know, what the first two lines mean: Field can be used as a virtual screen and copied to Screen and vice versa.

The next example shows the integer definition of MaxX and MaxY: both will take the value 100. The next three definitions for I and J will have the init value 0, and FALSE for TEST, which are the default.

A procedure prototype is defined with the PROCEDURE or FUNCTION command:

```
PROCEDURE NAME [ ( Parameter {, Parameter} )] ; [Specifier;]
FUNCTION NAME [ ( Parameter {, Parameter} )] : Type; [Specifier;]
```

A function requires a return type. Note, that *DPas* supports any standard Pascal type plus user defined record and enum types as valid return types. Even pointers are allowed as return types, as the following example shows:

```
function longstr:^string;
begin
  getmem(longstr,1024);
end;
```

Please note, that this is not allowed in Standard Pascal!

The specifier for procedures are:

FORWARD	The procedure is defined as a prototype, the implementation follows later in the source.
ASSEMBLER	The procedure is a plain assembler procedure. As parameters, CPU registers are used.
FAR	The procedure is a far procedure, required to call functions of a Pass32 DLLs.
WIN32[a]	The procedure is a Win32 prototype.
INTERRUPT	The procedure is a protected mode interrupt.

a. in Version 1.7 not yet supported.

The main body of a program begins after the first BEGIN command outside a procedure / function implementation. The main body ends with END followed by a '.'. Note, there can be no further instructions after END.

1.4.2 Structure of the Program Code

a) Global Structure

Every application created with *DPas* has the same structure:

OFFSET	Description
00x00100h[a]	Program Entry Point: jump to the actual start-up code.
00x00105h	reserved
00x00140h	Code and data of the SYSTEM unit.
00x02D40h[b]	Code and data of the first unit in the USES list.
00x*****h	Code and data of the following units.
00******h	Code of the main program.
0*******h	Data of the main program.
********h	Heap memory of the main program.

a. Under the Pro32 Dos-Extender the x stands for 0, under Win32, the x stands for 4.
b. This value depends on the SYSTEM unit version, and, of course on the *DPas* Version, that compiled the SYSTEM unit.

DPas initializes all global data, i.e. all your data declarations will be part of the binary.

For the unit import, *DPas* resolves the dependencies of the units. In our *Example 4* (HELLO3.PAS), the unit CRT32 is used by HUNIT. Therefore, *DPas* will import the units in the following order: SYSTEM, CRT32, HUNIT. In case the user adds CRT32 to the uses section of *Example 4* (HELLO3.PAS), there would be no difference:

```
USES hunit,crt32;       or:
USES crt32,hunit;
```

b) Start-Up Code

The start-up code is created, when *DPas* parses the main body, i.e. the main BEGIN instruction. The start-up code will create a stack frame and then call each main function of each unit. Note, independent on the number of includes of a unit, each main function of a unit is called only once. The main procedure is called last.

Start-up code is only added to the main program and not to units.

1.5 The Compiler Directives

DPas supports similar to *Turbo-Pascal* compiler directives. The main difference is, that combination of multiple directives is not allowed.

1.5.1 Compiler Control Directives

Directive	Description
{$DEFINE Name}	defines a conditional define.
{$IFDEF Name}	The following code is compiled in case Name has been defined.
{$ENDIF}	End of conditional block.
{$ELSE}	Alternative block, that is compiled in case Name has not been defined.
{$UNDEF Name}	undefines a conditional define.

Note, that *DPas* defines always the conditional define:

```
DPAS2
```

To verify whether the source is compiled by *DPas*, you can write:

```
{$IFDEF DPAS2}
{ Now DPAS CODE  }
{$ELSE}
{ Other compiler, e.g. TP }
{$ENDIF}
```

1.5.2 Code Generator Control Directives

Directive	Description
{$M xx,yy,zz}	defines the memory requirements for the program in KByte: xx defines the stack size, yy the minimum and zz the maximum required memory.
{$I+} {$I-}	enables / disables I/O error checking. When enables, after each system call, the value of IOError is checked. When unequal to zero, the program aborts with a run-time error.
{$S+} {$S-}	enables / disables stack size checking. When enables, the program aborts in case the available stack memory is too little.
{$A+} {$A-}	enables / disables alignment. When enabled, each data offset is aligned to 4. Note, that record types then might not represent the actual intended structure.

2. The DPas Language

DPas supports most of standard Pascal and adds some very powerful features, mostly derived from *C*.

2.1 Types

2.1.1 Standard Types

DPas supports the following standard Pascal types:

BYTE	unsigned 8 bit storage
WORD	unsigned 16 bit storage
DWORD	unsigned 32 bit storage
INTEGER	signed 32 bit storage
LONGINT	signed 32 bit storage[a]
POINTER	unsigned 32 bit address storage.
SINGLE	32 bit IEEE float
FLOAT	32 bit IEEE float[b]
DOUBLE	64 bit IEEE float
BOOLEAN	TRUE/FALSE, 8 bit storage
TEXT	32 bit file descriptor
STRING	256 byte long zero terminated string

a. for TP compatibility, may be extended to 64 bit in future.
b. for TP compatibility, there is no FPU emulation.

The ^-operator is a standard operator in *DPas*, which defines a pointer of the specified type. Therefore, the ^-operator can be used with any type in any procedure / function definition.

The following parameter definition

```
PROCEDURE A(P:^STRING);
```

is as valid as a function like:

```
FUNCTION longstr:^STRING;
begin
  getmem(longstr,1024);
end;
```

2.1.2 Strings

Strings are zero terminated in *DPas*. Per default, *DPas* reserves 256 bytes for a string. In case less or more bytes are required, a string can also be defined like:

```
var small_str:string[10]
```

Please note, that there is no range checking for strings.

The difference to Turbo Pascal is, that `string[0]` is not the length, it represents actually the **first** letter of the string.

As strings handled by the system unit and not directly by the compiler, the ABSOLUTE specifier can not be used in combination with strings.

2.1.3 Enumeration Types

Enumeration types are treated like integer constants. Therefore the following definitions are equal:

```
type color=(red,green,blue);
```

is equal to:

```
const red=0;
      green=1;
      blue=2;
type color=byte;
```

2.1.4 Type substitution

Type substitutions are supported, like:

```
type shortptr=^word;
type color=byte;
```

2.1.5 Record Types

Record types are supported. The number of record fields is limited to 100.

```
type pixel=record
        red,green,blue:byte;
        x,y:integer;
     end;
type window=record
        pixels:ARRAY[0..10000] OF pixle;
        x,y:integer;
        title:string;
     end;
```

2.1.6 Arrays

DPas supports a maximum of three dimensional arrays. Array definitions are allowed inside record types, constant types and in standard variable definitions, e.g.:

```
var board:array[0..7,0..7] of char;
const
error_mesg:array[0..1] of string = ('FILE NOT FOUND!', 'ERROR IN FILE!');
```

2.2 Const

DPas allows two kind of constant definitions. Initialization inside the VAR block, or type specified constants inside a CONST block. The difference is, that an initialization inside the VAR block refers to a whole array, or block definition, whereas a specified constant type requires values for each element.

The following example describes the initialization inside the VAR block:

```
VAR Field:ARRAY[0..24,0..79] OF WORD = $0720;
VAR MaxX,MaxY:INTEGER =  100;
```

The whole field is filled with the init value $0720.

The following example shows a standard definition of specified type constants:

```
const p:pixel=(red:127;green:127;blue:127;x:100;y:100);
```

2.3 Variables

DPas provides the specifier ABSOLUTE and IOPORT for global variables. The following exsample demonstrates ABSOLUTE and IOPORT:

```
var screen:screenbuffer absolute $B8000;
var keybcontroller:byte  ioport $60;
```

Note, that IOPORT can only be used with BYTE, WORD, INTEGER / DWORD / LONGINT. Nevertheless, you can still build up record types of these types, e.g.:

```
type VGA_MISCELLANEOUS_REGISTERS=record
         index,read_write:byte;
     end;
var  VGA_MISC:VGA_MISCELLANEOUS_REGISTERS IOPORT $3c2;

VGA_MISC.index:=0;
```

Note, that neither ABSOLUTE nor IOPORT can be used in combination with strings.

2.4 Pascal Constructs

The standard Pascal constructs BEGIN-END, CASE, FOR-TO/DOWNTO-DO, IF-THEN-ELSE, REPEAT-UNTIL, WHILE-DO are supported.

DPas provides the extended pascal constructs ASM-END, BREAK, CONTINUE, DEC and INC.

Additionally, under 32 bit DOS, the following arrays are defined to access memory and I/O ports: MEM, MEMW, MEML and PORT, PORTW, PORTL.

Please note, that these arrays are treated different to standard arrays, as the following example will demonstrate:

```
var kMeml: array [0..-1] of longint absolute = $0;

meml[1]:=5;
kmeml[1]:=5;
```

The first memory access will write the longint number 5 at the address 1, the second will write the longint number 5 at the address 4, because the array element size is 4.

2.5 Inline Assembler

Between ASM and END, the source code can contain assembler instructions. Inside the assembler instructions, global and local variables can be accessed.

The EBP/ESP register need to preserved inside the assembler block.

2.6 Procedure / Function headers

The maximum number of parameters is 20. Additional to standard Pascal, a function can return any type and is not restricted to the standard Pascal types.

Note, that a procedure specifier, i.e. INTERRUPT, FAR, ASSEMBLER, needs to be specified also in any forward definition, therefore also in the interface part of a unit.

An assembler procedure uses CPU registers as parameters. Additionally to the register, the type needs to be defined:

```
[VAR] REG8/REG16/REG32:TYPE;
```

An assembler functions returns its result in EAX (ST(0) for single/double types). In case the type does not fit into EAX (e.g. strings, record-types), in EAX a pointer to the buffer is returned. This allows string functions to return in EAX a pointer to the actual string. See for example the ParamStr function in SYSTEM.PAS.

2.7 System Unit

A lot of standard functions, esp. the handling of TEXT files are supported. Here is an overview of the supported functions. Note, that some functions are handled by the compiler directly.

- WRITELN / READLN
- LENGTH
- Assign
- Erase
- Rename
- Reset - opens TEXT fil e for read
- Rewrite - opens TEXT f ile for write
- Getdir
- Close
- Chdir
- Eof
- halt
- move
- MoveMem
- fillchar
- fillmemchar
- runerror
- dosmemavail
- memavail
- ofs
- zeroofs
- getmem
- freemem
- paramstr
- copy
- pos
- strupcase
- IOResult
- sin
- cos
- arctan
- ln
- sqrt
- Pi
- abs
- trunc
- round
- int
- new
- dispose

2.8 RUN TIME LIBRARY

A quite large run time library comes with *DPas*.

2.8.1 CRT32

- clrscr
- gotoxy
- wherex
- wherey
- color
- delay
- keypressed
- readkey
- getvideomode
- setvideomode
- textcolor
- background

2.8.2 DOS32

- gettime
- loadfile
- filelength
- savefile
- setintvec
- getenv
- exec

2.9 Import of Dynamic Link Libraries

DPas can add dynamic link libraries to programs or units. These libraries are load during the run-time. In order to know these functions, *DPas* requires an interface defintion. The interface definition is similar to a unit, it begins with the `DLL` keyword. The interface may define types, but can not define variables or code.

In case of a Pass32 DLL, the specifier `FAR` and `ASSEMBLER` are required.

2.10 Optimization

2.10.1 Expressions

DPAS optimizes expressions from the left. Therefore you should write all constants lefthanded, e.g.:

```
i:=1024*2*i;
```

instead of

```
i:=1024*i*2;
```

The first results in a single SHL expression, the second in two SHL expressions.

2.10.2 Multiplication / Division

When possible, *DPas* optimizes multiplications. A typical `Putpixel` function like:

```
var screen:ARRAY[0..199,0..319] of byte absolute $A0000;
procedure putpixel(x,y:integer;c:byte);
begin
   screen[y,x]:=c;
end;
```

Is actually optimized to:

```
mov eax,y
lea eax,[4*eax+eax]
shl eax,6
mov esi,x
lea esi,[eax+esi+0a0000]
mov al,c
mov gs:[esi],al
```

Therefore, *DPas* optimizes `mul` instructions by the faster `lea` and/or `shl` instruction.

2.10.3 Constant Offsets

In case an offset for an array expression is constant, *DPas* optimizes the memory access. In case of

```
screen[0,10]:=5;
```

DPas creates the code:

```
mov gs:[0a000a],5
```

2.10.4 Assembler Procedures

DPas uses for all system procedure calls CPU registers for parameter passing. In combination with inline assembler you can also create very fast assembler procedures.

3. Examples

3.1 Primes

The PRIMES example is a benchmark program for calculating prime numbers. It can be compiled with *Turbo Pascal* or *DPas*.

In order to be compatible with both compilers, *DPas* defines the compiler define DPAS2, which can be tested with the directive {$IFDEF DPAS2}.

3.2 Direct

DIRECT draws a worm on the text screen. It uses an ABSOLUTE variable to directly write on the screen. It also defines an array type in order to store and restore the screen contents:

```
type screen:array[0..24,0..79] of word;
var scr:screen absolute  $B8000;
    scrsave:screen;
begin
  scrsave:=scr;
..
  scr:=scrsave;
end.
```

3.3 Traffic

TRAFFIC is again compatible with *TP*. It defines crazy record types to describe a traffic light simulation. The program demonstrates mainly the correct handling of const definitions.

3.4 Passed

PASSED is next to the Pro32 debugger the largest project created with *DPas*. PASSED is an *IDE* for the Pass32 compiler. PASSED comes together with a simple text window manager PROWIN.

4. Limits

As an ongoing development, *DPas* still has a lot of limits. I hope, that this chapter will get obsolete in the near future.

4.1 Symbols

Symbols can not exceed 32 characters. A record field can consist of only 28 characters. *DPas* is still a DOS product. Therefore it is restricted to 640 K memory. *DPas* can handle about 5000 symbols with 560 KBytes free dos memory. This allows to compile Pascal sources between 5000 and 32000 lines depending on your style and the complexity of your code.

As *DPas* support up to 40 units, you can build really huge applications. The source code is not restricted in size, only the number of symbols. Applications can take be up to 64 MByte.

Note, the total limit of symbols is depending from your free DOS memory, additionally, for each symbol type, the following limits are set by *DPas*:

Symbol Type	Limit
Enumeration Type Elememts / Integer Constants	800
Record Types / Array Types	800
(Public) Variables	6500
(Public) Procedures	2000
Lables	6500

4.2 Const

There are only two type of constants supported:

- `INTEGER`, e.g.: `const Max=10;`
- any type specified constant, e.g.
  - `const MaxX:INTEGER=10;`
  - `const Mesg:String='HELLO';`
  - `const p:pixel=(X:100;Y:100);`
  - `const cp:colorpixel=(C:red;P:(X:100;Y:100));`

Not yet supported are string, double constants, like

- `const Hello='Hello';`
- `const Exact_Pi = 3.1415;`

4.3 Types

DPas supports all standard Pascal types.

Note, that a record type can consist of a maximum of 100 record fields.

Arrays can not be combined:

instead of:

```
type field=array [0..10] of array [0..24] of array [0..79] of char;
var bigfield=array [0..10] of field;
bigfield [5][0][0][0]:='A';
bigfield [5,0,0,0]:='A';
```

write:

```
type field=array [0..10,0..24,0..79] of char;
var bigfield=array [0..10] of field;
bigfield [5][0,0,0]:='A';
```

4.4 Array Depth

The array depth is 3, as maximum, a 3 dimensional array field can be defined, e.g.:

```
console: array [0..10,0..24,0..79] of char;
```

You can work around this problem by using array types:

```
type console_type=array [0..10,0..24,0..79] of char;
var console: array [0..10,0..24,0..79] of console_type;
```

Note, that *DPas* expects proper settings of brackets: in this example, console can be accessed like:

```
console[0,24,0][0,o,0]:='B';
```

4.5 Boolean Expressions

For boolean expressions, the compiler needs to learn about the expression type before the comparison symbols like '=','<','>': the compiler can not always resolve record or array types in an expression. When ever possible, a boolean expression should look like:

`CONST = EXPRESSION` and **not** `EXPRESSION = CONST`.

4.6 ELSE / END

The parser sometimes falls over a missing ';' before `ELSE` or `END`. Here, the parser is often inconsequent: sometimes the parser asks for a ';' before `ELSE` or `END`, sometimes the parser reports an error, when there is a ';',

4.7 WITH

`WITH` is **not yet** supported.

4.8 ASM

FPU instructions are **not yet** supported.

Not all x86 instructions are correctly supported.

4.9 Writeln, ReadLn

Output and input of float numbers are not supported, because of section 4.8 on page 31.

`ReadLn` and `WriteLn` does not work with record types or array types, e.g.:

```
readLn(a[0]);
```

4.10 Procedure / Function headers

The maximum number of parameters is 20. Additional to standard Pascal, a function can return any type and is not restricted to the standard Pascal types.

Note, that a procedure specifier, i.e. `INTERRUPT`, `FAR`, `ASSEMBLER`, needs to be specified also in any forward definition, therefore also in the interface part of a unit.

4.11 Sets

DPas does not support `SET` and probably never will.

4.12 Error Messages

Error messages are not very smart yet.