Pascal implementation

The P6 Compiler

Scott A. Franco

[1 Overview of Pascal-P6 9](#__RefHeading___Toc15532_2478585429)

[1.1 Licensing information 9](#__RefHeading___Toc15534_2478585429)

[1.2 Repositories for Pascal-P6 9](#__RefHeading___Toc15536_2478585429)

[1.3 Introduction: history of Pascal-P6 10](#__RefHeading___Toc15538_2478585429)

[2 Terminology used in this document 11](#__RefHeading___Toc15540_2478585429)

[3 Pascal-P6 vs. Pascal-P5 11](#__RefHeading___Toc15542_2478585429)

[4 Pascal-P6 as a strict ISO 7185 compiler 11](#__RefHeading___Toc15544_2478585429)

[5 Pascal-P6 vs. FPC (Free pascal) and GPC (Gnu Pascal), Borland, UCSD or other dialects 12](#__RefHeading___Toc15546_2478585429)

[6 The Pascal-P6 source language 13](#__RefHeading___Toc15548_2478585429)

[6.1 Features of Pascaline NOT implemented in Pascal-P6 13](#__RefHeading___Toc15550_2478585429)

[6.2 Pascal-P6 implementation details 13](#__RefHeading___Toc15552_2478585429)

[6.2.1 Header files 14](#__RefHeading___Toc15554_2478585429)

[6.2.2 Alternative header value input 14](#__RefHeading___Toc15556_2478585429)

[6.2.3 Character escapes 15](#__RefHeading___Toc15558_2478585429)

[6.2.4 Character sets 15](#__RefHeading___Toc15560_2478585429)

[6.2.5 Modular structure 15](#__RefHeading___Toc15562_2478585429)

[6.2.6 Implementation of Annexes G-O 15](#__RefHeading___Toc15564_2478585429)

[6.2.6.1 Enabling externals 16](#__RefHeading___Toc15566_2478585429)

[6.2.6.2 Using externals 16](#__RefHeading___Toc15568_2478585429)

[7 Using Pascal-P6 17](#__RefHeading___Toc15570_2478585429)

[7.1 Configuring P6 17](#__RefHeading___Toc15572_2478585429)

[7.2 Compiling and running Pascal programs with P6 18](#__RefHeading___Toc15574_2478585429)

[7.2.1 Compiling with multiple modules 18](#__RefHeading___Toc15576_2478585429)

[7.2.2 Compiling on different run configurations 18](#__RefHeading___Toc15578_2478585429)

[7.3 Compiler options 18](#__RefHeading___Toc1840015_3010434754)

[7.3.1 Option descriptions 21](#__RefHeading___Toc15580_2478585429)

[7.4 Other operations 23](#__RefHeading___Toc15582_2478585429)

[7.5 Reliance on Unix commands in the P6 toolset 23](#__RefHeading___Toc15584_2478585429)

[7.6 The “flip” command and line endings 24](#__RefHeading___Toc15586_2478585429)

[8 Pascal-P6 debugger mode 24](#__RefHeading___Toc15588_2478585429)

[8.1 Sample program for debug 26](#__RefHeading___Toc15590_2478585429)

[8.2 Debug mode invocation 27](#__RefHeading___Toc15592_2478585429)

[8.3 Debug mode expressions 28](#__RefHeading___Toc15594_2478585429)

[8.4 Executing multiple commands 29](#__RefHeading___Toc15596_2478585429)

[8.5 Debug mode Command descriptions 29](#__RefHeading___Toc15598_2478585429)

[8.5.1 h or help Print help menu 29](#__RefHeading___Toc15600_2478585429)

[8.5.2 l [m] [s[ e|:l] List source lines 30](#__RefHeading___Toc15602_2478585429)

[8.5.3 li [s[ e|:l] List machine instructions 32](#__RefHeading___Toc15604_2478585429)

[8.5.4 lc [s[ e|:l] List source and machine lines coordinated 33](#__RefHeading___Toc15606_2478585429)

[8.5.5 p v Print expression 34](#__RefHeading___Toc15608_2478585429)

[8.5.6 d [s[ e|:l] Dump memory 35](#__RefHeading___Toc15610_2478585429)

[8.5.7 e a v[ v]... Enter byte values to memory address 36](#__RefHeading___Toc15612_2478585429)

[8.5.8 st d v Set program variable 36](#__RefHeading___Toc15614_2478585429)

[8.5.9 pg Print all globals 37](#__RefHeading___Toc15616_2478585429)

[8.5.10 pl [n] print locals for current/number of enclosing blocks 37](#__RefHeading___Toc15618_2478585429)

[8.5.11 pp [n] print parameters for current/number of enclosing blocks 38](#__RefHeading___Toc15620_2478585429)

[8.5.12 ds Dump storage parameters 38](#__RefHeading___Toc15622_2478585429)

[8.5.13 dd [n] Dump display frames 39](#__RefHeading___Toc15624_2478585429)

[8.5.14 df [n] Dump frames formatted (call trace) 39](#__RefHeading___Toc15626_2478585429)

[8.5.15 b [m] a Place breakpoint at source line number/routine 40](#__RefHeading___Toc15628_2478585429)

[8.5.16 tp [m] a Place tracepoint at source line number/routine 40](#__RefHeading___Toc15630_2478585429)

[8.5.17 bi a Place breakpoint at instruction 41](#__RefHeading___Toc15632_2478585429)

[8.5.18 tpi a Place tracepoint at instruction 42](#__RefHeading___Toc15634_2478585429)

[8.5.19 c [a] Clear breakpoint/all breakpoints 42](#__RefHeading___Toc15636_2478585429)

[8.5.20 lb List active breakpoints 43](#__RefHeading___Toc15638_2478585429)

[8.5.21 w a Watch variable 43](#__RefHeading___Toc15640_2478585429)

[8.5.22 lw List watch table 44](#__RefHeading___Toc15642_2478585429)

[8.5.23 cw [n] Clear watch table entry/all watch entries 44](#__RefHeading___Toc15644_2478585429)

[8.5.24 lia List instruction analyzer buffer 44](#__RefHeading___Toc15646_2478585429)

[8.5.25 lsa List source analyzer buffer 45](#__RefHeading___Toc15648_2478585429)

[8.5.26 s [n] Step next source line execution 45](#__RefHeading___Toc15650_2478585429)

[8.5.27 ss [n] Step next source line execution silently 47](#__RefHeading___Toc15652_2478585429)

[8.5.28 si [n] Step instructions 47](#__RefHeading___Toc15654_2478585429)

[8.5.29 sis [n] Step instructions silently 48](#__RefHeading___Toc15656_2478585429)

[8.5.30 hs Report heap space 48](#__RefHeading___Toc15658_2478585429)

[8.5.31 pc [a] Set/print pc contents 49](#__RefHeading___Toc15660_2478585429)

[8.5.32 sp [a] Set/print sp contents 49](#__RefHeading___Toc15662_2478585429)

[8.5.33 mp [a] Set/print mp contents 49](#__RefHeading___Toc15664_2478585429)

[8.5.34 np [a] Set/print np contents 49](#__RefHeading___Toc15666_2478585429)

[8.5.35 ti Turn instruction tracing on 49](#__RefHeading___Toc15668_2478585429)

[8.5.36 nti Turn instruction tracing off 50](#__RefHeading___Toc15670_2478585429)

[8.5.37 tr Turn system routine tracing on 50](#__RefHeading___Toc15672_2478585429)

[8.5.38 ntr Turn system routine tracing off 51](#__RefHeading___Toc15674_2478585429)

[8.5.39 ts Turn source line tracing on 51](#__RefHeading___Toc15676_2478585429)

[8.5.40 nts Turn source line tracing off 52](#__RefHeading___Toc15678_2478585429)

[8.5.41 spf Turn on source level profiling 52](#__RefHeading___Toc15680_2478585429)

[8.5.42 nspf Turn off source level profiling 53](#__RefHeading___Toc15682_2478585429)

[8.5.43 an Turn on analyzer mode 54](#__RefHeading___Toc15684_2478585429)

[8.5.44 nan Turn off analyzer mode 54](#__RefHeading___Toc15686_2478585429)

[8.5.45 r Run program from current pc 54](#__RefHeading___Toc15688_2478585429)

[8.5.46 ps Print current registers and instruction 54](#__RefHeading___Toc15690_2478585429)

[8.5.47 q Quit interpreter 55](#__RefHeading___Toc15692_2478585429)

[9 Different generation and run options: pint, pmach and cmach 55](#__RefHeading___Toc15694_2478585429)

[9.1 Running the different machines 55](#__RefHeading___Toc15696_2478585429)

[9.2 Difference between different byte machines 56](#__RefHeading___Toc15698_2478585429)

[9.3 Setting machine options on pmach and cmach 56](#__RefHeading___Toc15700_2478585429)

[9.4 Packaged application mode 57](#__RefHeading___Toc15702_2478585429)

[9.5 Using packaged mode to port to a machine not supported by a Pascal compiler 58](#__RefHeading___Toc15704_2478585429)

[10 Portability of intermediate decks 58](#__RefHeading___Toc15706_2478585429)

[11 Portability of hex/binary decks 59](#__RefHeading___Toc15708_2478585429)

[12 Pascal-P6 implementation language 59](#__RefHeading___Toc15710_2478585429)

[12.1 Language extension routines 59](#__RefHeading___Toc15712_2478585429)

[12.1.1 procedure assigntext(var f: text; var fn: filnam); 60](#__RefHeading___Toc15714_2478585429)

[12.1.2 procedure assignbin(var f: bytfil; var fn: filnam); 60](#__RefHeading___Toc15716_2478585429)

[12.1.3 procedure closetext(var f: text); 60](#__RefHeading___Toc15718_2478585429)

[12.1.4 procedure closebin(var f: bytfil); 60](#__RefHeading___Toc15720_2478585429)

[12.1.5 function lengthbin(var f: bytfil): integer; 60](#__RefHeading___Toc15722_2478585429)

[12.1.6 function locationbin(var f: bytfil): integer; 60](#__RefHeading___Toc15724_2478585429)

[12.1.7 procedure positionbin(var f: bytfil; p: integer); 60](#__RefHeading___Toc15726_2478585429)

[12.1.8 procedure updatebin(var f: bytfil); 60](#__RefHeading___Toc15728_2478585429)

[12.1.9 procedure appendtext(var f: text); 60](#__RefHeading___Toc15730_2478585429)

[12.1.10 procedure appendbin(var f: bytfil); 60](#__RefHeading___Toc15732_2478585429)

[12.1.11 function existsfile(var fn: filnam): boolean; 60](#__RefHeading___Toc15734_2478585429)

[12.1.12 procedure deletefile(var fn: filnam); 60](#__RefHeading___Toc15736_2478585429)

[12.1.13 procedure changefile(var fnd, fns: filnam); 60](#__RefHeading___Toc15738_2478585429)

[12.1.14 procedure getcommandline(var cb: cmdbuf; var l: cmdnum); 60](#__RefHeading___Toc15740_2478585429)

[13 Building the Pascal-P6 system 61](#__RefHeading___Toc15742_2478585429)

[13.1 Compiling and running P6 with an existing ISO 7185 compiler 61](#__RefHeading___Toc15744_2478585429)

[13.2 Evaluating an existing Pascal compiler using P6 62](#__RefHeading___Toc15746_2478585429)

[13.3 Notes on using existing compilers 63](#__RefHeading___Toc15748_2478585429)

[13.3.1 GPC 63](#__RefHeading___Toc15750_2478585429)

[13.3.1.1 GPC for mingw and Windows 64](#__RefHeading___Toc15752_2478585429)

[13.3.1.2 GPC for Linux 64](#__RefHeading___Toc15754_2478585429)

[pgen – AMD64 code generator 66](#__RefHeading___Toc15756_2478585429)

[Pgen, specifically pgen\_gcc\_amd64, is a code generator for 64 bit AMD/Intel CPUs. It is also dependent on gcc/gas (the GNU C compiler and GNU assembler for AMD64 cpus). pgen is executed as: 66](#__RefHeading___Toc15758_2478585429)

[> pgen <input file> <output file> 66](#__RefHeading___Toc15760_2478585429)

[Pgen translates Pascal-P6 intermediate files to gas assembly files, which are then assembled, along with C language support files, into target programs. An example run is: 66](#__RefHeading___Toc15762_2478585429)

[> pcom hello.pas hello.p6 66](#__RefHeading___Toc15764_2478585429)

[> pgen hello.p6 hello.s 66](#__RefHeading___Toc15766_2478585429)

[> gcc source/psystem.c source/psystem\_asm.s main.s hello.s -o hello -lm 66](#__RefHeading___Toc15768_2478585429)

[This sequence is given in the script file p6\_pgen, which has the form: 66](#__RefHeading___Toc15770_2478585429)

[> p6\_pgen <input file> [<input file>]… 66](#__RefHeading___Toc15772_2478585429)

[The components of a pgen deck are: 66](#__RefHeading___Toc15774_2478585429)

[psystem.c The C runtime support library for Pascal-P6 66](#__RefHeading___Toc15776_2478585429)

[psystem\_asm.s The assembly language runtime support library for Pascal-P6 66](#__RefHeading___Toc15778_2478585429)

[main.s The startup card for Pascal-P6, defines the location of “main”. 66](#__RefHeading___Toc15780_2478585429)

[hello.s The assembly code generated by pgen. 66](#__RefHeading___Toc15782_2478585429)

[Hello The output executable file. 66](#__RefHeading___Toc15784_2478585429)

[-lm Links the C math library. 66](#__RefHeading___Toc15786_2478585429)

[13.4 Calling convention 66](#__RefHeading___Toc15788_2478585429)

[Pascal-P6 uses a modified version of the System V AMD64 ABI register based calling convention. In this convention the following register assignments are used: 66](#__RefHeading___Toc15790_2478585429)

[Integer/pointer arguments 1-6 RDI, RSI, RDX, RCX, R8, R9 66](#__RefHeading___Toc15792_2478585429)

[Floating point arguments 1-16 XMM0-XMM15 66](#__RefHeading___Toc15794_2478585429)

[Return value RAX,RDX 66](#__RefHeading___Toc15796_2478585429)

[All parameters are also passed on the stack. The register arguments are copies of the stacked arguments. The caller is responsible for cleanup. 66](#__RefHeading___Toc15798_2478585429)

[The Pascal-P6 calling convention is different from the System V AMD64 ABI register based calling convention in that the first parameter at top of stack is the last parameter of the procedure or function, that is, parameters on the stack are pushed from left to right, instead of the C based calling convention of parameters stacked from right to left, so that the first parameter on stack is the first parameter of the function. The reason for this is that C supports so called “variadic functions”, where the parameters can be any number. Pascal does not. 66](#__RefHeading___Toc15800_2478585429)

[Besides floating point, integers pointers, Pascal-P6 supports so called “wide pointers”, which are arrays with templates or lengths following. If passed as a parameter, such pointers take two registers in a row, such as: 67](#__RefHeading___Toc15802_2478585429)

[Would take regsisters: 67](#__RefHeading___Toc15804_2478585429)

[rdi Address of array y. 67](#__RefHeading___Toc15806_2478585429)

[rsi Length of array y. 67](#__RefHeading___Toc15808_2478585429)

[If a wide pointer is returned from a function as a result, it would occupy the registers: 67](#__RefHeading___Toc15810_2478585429)

[Rax Address of array returned. 67](#__RefHeading___Toc15812_2478585429)

[Rdx Length of array returned. 67](#__RefHeading___Toc15814_2478585429)

[13.5 Calling C from Pascaline 67](#__RefHeading___Toc15816_2478585429)

[When using from 1 to 6 parameters, the calling conventions of C and Pascaline are compatible. The parameters and return values will line up correctly: 67](#__RefHeading___Toc15818_2478585429)

[Pascal 67](#__RefHeading___Toc15820_2478585429)

[procedure x(a, b: integer; r: real); 67](#__RefHeading___Toc15822_2478585429)

[C 67](#__RefHeading___Toc15824_2478585429)

[void x(int a, int b, double r); 67](#__RefHeading___Toc15826_2478585429)

[Pascal-P6 always treats arrays as if they were passed as containers. That is: 67](#__RefHeading___Toc15828_2478585429)

[Type s = array [1..10] of char; 67](#__RefHeading___Toc15830_2478585429)

[Procedure x(a: s); 67](#__RefHeading___Toc15832_2478585429)

[Is caller equivalent to: 67](#__RefHeading___Toc15834_2478585429)

[Procedure x(a: string); 67](#__RefHeading___Toc15836_2478585429)

[Meaning that the C external function must be ready for a length argument: 67](#__RefHeading___Toc15838_2478585429)

[Void x(char a[], int l); 67](#__RefHeading___Toc15840_2478585429)

[This means the C external function always knows what the length of the array parameter is. This can aid greatly in forming calls from Pascaline to C. 67](#__RefHeading___Toc15842_2478585429)

[Pascal-P6 does this for any container argument. If the container array has more than one dimension, it will be passed with a template: 68](#__RefHeading___Toc15844_2478585429)

[Pascal 68](#__RefHeading___Toc15846_2478585429)

[Type a = array of array of integer; 68](#__RefHeading___Toc15848_2478585429)

[Procedure x(b: a); 68](#__RefHeading___Toc15850_2478585429)

[C 68](#__RefHeading___Toc15852_2478585429)

[int template = { 10, 20 }; 68](#__RefHeading___Toc15854_2478585429)

[Void x(integer b[][], int t[]); 68](#__RefHeading___Toc15856_2478585429)

[The C function will get the template as a list of integers, one per “order” or number of dimensions in the array. Thus in this example, there are two dimensions and two sizes in the template list for the multidimensional array. 68](#__RefHeading___Toc15858_2478585429)

[No value parameters for structured parameters 68](#__RefHeading___Toc15860_2478585429)

[Pascal-P6 does not pass value or copy parameters for structured parameters. In the procedure 68](#__RefHeading___Toc15862_2478585429)

[Pascal 68](#__RefHeading___Toc15864_2478585429)

[type a = array [1..10] of char; 68](#__RefHeading___Toc15866_2478585429)

[Procedure x(b: a); 68](#__RefHeading___Toc15868_2478585429)

[C 68](#__RefHeading___Toc15870_2478585429)

[void x(char\* b); 68](#__RefHeading___Toc15872_2478585429)

[In Pascal-P6 procedures/functions, this value parameter is done by having the callee make a local copy of the parameter. Thus it is up to the C routine to perform this function. 68](#__RefHeading___Toc15874_2478585429)

[13.5.1 Stack inversion 68](#__RefHeading___Toc15876_2478585429)

[13.5.2 Translation of strings 68](#__RefHeading___Toc15878_2478585429)

[13.5.3 Pointers an VAR parameters 69](#__RefHeading___Toc15880_2478585429)

[13.6 Pascal-P6 module stacking sequence 69](#__RefHeading___Toc15882_2478585429)

[14 Files in the Pascal-P6 package 70](#__RefHeading___Toc15884_2478585429)

[14.1 Directory: bin 70](#__RefHeading___Toc15886_2478585429)

[14.2 Directory: doc 72](#__RefHeading___Toc15888_2478585429)

[14.3 Directory: gpc 72](#__RefHeading___Toc15890_2478585429)

[14.4 Directory: gpc/linux\_X86 73](#__RefHeading___Toc15892_2478585429)

[14.5 Directory: mac\_X86 73](#__RefHeading___Toc15894_2478585429)

[14.6 Directory: gpc/standard\_tests 73](#__RefHeading___Toc15896_2478585429)

[14.7 Directory: gpc/windows\_X86 73](#__RefHeading___Toc15898_2478585429)

[14.8 Directory: ip\_pascal 73](#__RefHeading___Toc15900_2478585429)

[14.9 Directory: ip\_pascal/standard\_tests 73](#__RefHeading___Toc15902_2478585429)

[14.10 Directory: ip\_pascal/windows\_X86 73](#__RefHeading___Toc15904_2478585429)

[14.11 Subdirectory: sample\_programs 74](#__RefHeading___Toc15906_2478585429)

[14.12 Directory: standard\_tests 74](#__RefHeading___Toc15908_2478585429)

[15 The intermediate language 75](#__RefHeading___Toc15910_2478585429)

[15.1 Format of intermediate 75](#__RefHeading___Toc15912_2478585429)

[15.2 Comments 76](#__RefHeading___Toc15914_2478585429)

[15.3 Source lines 77](#__RefHeading___Toc15916_2478585429)

[15.3.1 Label 77](#__RefHeading___Toc15918_2478585429)

[15.3.2 Source line marker 77](#__RefHeading___Toc15920_2478585429)

[15.3.3 Options 77](#__RefHeading___Toc15922_2478585429)

[15.3.4 Global space count 77](#__RefHeading___Toc15924_2478585429)

[15.3.5 Block start and end 78](#__RefHeading___Toc15926_2478585429)

[15.3.6 Symbols 78](#__RefHeading___Toc15928_2478585429)

[15.3.7 Faults 81](#__RefHeading___Toc15930_2478585429)

[15.3.8 Logical variant table 81](#__RefHeading___Toc15932_2478585429)

[15.3.9 Template tables 81](#__RefHeading___Toc15934_2478585429)

[15.3.10 Constant tables 82](#__RefHeading___Toc15936_2478585429)

[15.4 Templates 83](#__RefHeading___Toc12854_1780621904)

[15.5 Variant record tables 84](#__RefHeading___Toc15938_2478585429)

[15.6 Undefined accesses 85](#__RefHeading___Toc15940_2478585429)

[15.7 Code strips 85](#__RefHeading___Toc15942_2478585429)

[15.8 Exception frames 85](#__RefHeading___Toc15944_2478585429)

[15.9 Operator overload calling frames 85](#__RefHeading___Toc15946_2478585429)

[15.10 Intermediate instruction set 86](#__RefHeading___Toc15948_2478585429)

[15.11 System calls 110](#__RefHeading___Toc15950_2478585429)

[16 Testing P6 125](#__RefHeading___Toc15952_2478585429)

[16.1 Running tests 125](#__RefHeading___Toc15954_2478585429)

[16.1.1 testprog 125](#__RefHeading___Toc15956_2478585429)

[16.1.2 Other tests 126](#__RefHeading___Toc15958_2478585429)

[16.1.3 Regression test 126](#__RefHeading___Toc15960_2478585429)

[16.2 Test types 126](#__RefHeading___Toc15962_2478585429)

[16.3 The Pascal acceptance test 127](#__RefHeading___Toc15964_2478585429)

[16.4 The Pascal rejection test 127](#__RefHeading___Toc15966_2478585429)

[16.4.1 List of tests 128](#__RefHeading___Toc15968_2478585429)

[16.4.1.1 Class 1: Syntatic errors 128](#__RefHeading___Toc15970_2478585429)

[16.4.1.2 Class 2: Semantic errors 134](#__RefHeading___Toc15972_2478585429)

[16.4.1.3 Class 3: Advanced error checking 137](#__RefHeading___Toc15974_2478585429)

[16.4.1.4 Class 4: Field checks 137](#__RefHeading___Toc15976_2478585429)

[16.4.2 Running the PRT and interpreting the results 137](#__RefHeading___Toc15978_2478585429)

[16.4.2.1 List of tests with no compile or runtime error. 138](#__RefHeading___Toc15980_2478585429)

[16.4.2.2 List of differences between compiler output and “gold” standard outputs. 138](#__RefHeading___Toc15982_2478585429)

[16.4.2.3 List of differences between runtime output and “gold” standard outputs. 138](#__RefHeading___Toc15984_2478585429)

[16.4.2.4 Collected compiler listings and runtime output of all tests. 138](#__RefHeading___Toc15986_2478585429)

[16.4.3 Overall interpretation of PRT results 138](#__RefHeading___Toc15988_2478585429)

[16.5 Sample program tests 139](#__RefHeading___Toc15990_2478585429)

[16.6 Previous Pascal-P versions test 139](#__RefHeading___Toc15992_2478585429)

[16.6.1 Compile and run Pascal-P2 139](#__RefHeading___Toc15994_2478585429)

[16.6.2 Compile and run Pascal-P4 140](#__RefHeading___Toc15996_2478585429)

[16.7 Self compile 140](#__RefHeading___Toc15998_2478585429)

[16.7.1 pcom 140](#__RefHeading___Toc16000_2478585429)

[16.7.1.1 Changes required 141](#__RefHeading___Toc16002_2478585429)

[16.7.2 pint 141](#__RefHeading___Toc16004_2478585429)

# Overview of Pascal-P6

This section contains background material on Pascal-P6. If you want to get started using Pascal-P6 now, skip to “Using Pascal-P6”

## Licensing information

Pascal-P6 is covered by the BSD license:

Copyright (c) 1996, 2018, Scott A. Franco

All rights reserved.

Redistribution **and** use in source **and** binary forms, with **or** without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, **this** list of conditions **and** the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, **this** list of conditions **and** the following disclaimer in the documentation **and**/**or** other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS

INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The views **and** conclusions contained in the software **and** documentation are those of the authors **and** should **not** be interpreted as representing official policies, either expressed **or** implied, of the Pascal-P6 project.

Note that the original project from which this code is based, Pascal-p4, was public domain. The BSD license covers only the changes/and or additions made to the original source code. To determine what is and what is not covered by public domain vs. the BSD license, compare the original Pascal-P4 sources to Pascal-P6.

The easiest way to obtain the public domain portion of Pascal-P6 is to simply obtain the Pascal-P4 source code, which is freely available online.

## Repositories for Pascal-P6

Pascal-P6 is managed using git version control system. It is stored at the following repositories:

<https://sourceforge.net/projects/pascal-p6/>

<https://github.com/samiam95124/Pascal-P6>

Please submit any bug tickets to:

https://sourceforge.net/p/pascal-p6/tickets/

## Introduction: history of Pascal-P6

The Pascal-P series compilers were the original proving compilers for the language Pascal. Created in 1973, Pascal-P was part of a “porting kit” designed to enable the quick implementation of a Pascal language compiler on new machines. It was released by Niklaus Wirth’s students at ETH in Zurich.

The implementation and description of the language Pascal in terms of itself and in terms of a “pseudo machine” were important factors in the propagation of the language Pascal. From the early version of Pascal-P came the CDC 6000 full compiler at Zurich, several independent compilers including an IBM-360 compiler and a PDP-11 compiler, and the UCSD “bytecode” interpreter.

The original article for the Pascal-P compiler is “The\_Pascal\_P\_Compiler\_implementation\_notes.pdf” in the doc area of the Pascal-P6 project directory tree.

In the name “Pascal-P” the “P” stood for “portable”, and this was what Pascal-P was designed to do. It also stood for an example and reference implementation of Pascal, although Wirth later issued a paper, together with Tony Hoare for the “Axiomatic definition of Pascal”, which was also aimed at exactly specifying the semantics of Pascal.

As the importance of Pascal-P grew, the authors adopted a version number system and working methodology for the system. A new, cleaner and more portable version of the system was created in 1974 with the name Pascal-P2, and left the multiple early versions of the system as termed Pascal-P1.

From the Pascal-P2 revision of the compiler comes many of the original Pascal compilers, including UCSD. In 1976, Wirth’s group made one last series of improvements and termed the results Pascal-P3 and Pascal-P4. Pascal-P3 was a redesigned compiler, but used the same pseudo machine instruction set as P2, and thus could be bootstrapped from an existing P2 implementation. P4 featured a new pseudo instruction set, and thus was a fully redesigned compiler.

Pascal-P was always an imcomplete implementation of the Pascal language (a subset), and was designed to be so. After it was created, the ISO 7185 standard for Pascal was issued, and today Pascal-P4 exists and is still usable with minor changes to bring it into ISO 7185 compliance for its source (not the language it compiles).

However, Pascal-P4 has it’s legacy problem of being a subset compiler of the full language. Further, it is only usable for programs that avoid its weaknesses, such as string storage. Keep in mind that Pascal-P was never designed to be a general purpose system, but rather to compile itself on a new machine – and then rapidly be improved to become a full compiler.

In 2008 I set out to improve the P4 code to accept the full language Pascal as stated by the ISO 7185 language standard. The name of the result was obvious: Pascal-P5, although I certainly beg forgiveness of any of the original Wirth group members who might be offended by my implication of extending and improving their work. Think of it this way: I too have been a lifelong student of Niklaus Wirth’s, abet a long distance one.

Pascal-P5 exists as a separate project and is an interpreter only. It can form the basis for any ISO 7185 based project. The project intent was to only add the minimum code to the Pascal-P4 implementation to bring it up to ISO 7185 status. It also introduced a large suite of tests to verify compliance with the ISO 7185 Pascal standard.

The purpose of Pascal-P6 was to create a working compiler implementation from Pascal-P5. Thus Pascal-P5 serves as a working, interpretive model of ISO 7185 Pascal, and Pascal-P6 serves as an actual machine implementation of it.

The first task of the Pascal-P6 implementation was to create a series of extensions to ISO 7185 Pascal. All actual machine implementations of Pascal, including the original CDC 6600 implementation, include a series of extensions. With Pascal-P6, I wanted to introduce a series of extensions that would both bring the compiler to a usable state as practical compiler and also bring it up to the “state of the art” (at least as of 2018).

I realize that introducing a set of new features to Pascal is going to be controversial. I wanted the extensions to be more than just a laundry list of features added by a compiler implementor (as has happened with Pascal so many times in the past). Thus I collected the features implemented by Pascal-P6 as a formal language in its own right, as the language “Pascaline”. There is a formal specification accompanying Pascal-P6 (“pascaline.docx”) as well as a rationale (“Pascaline\_rationale.docx”), a discussion of both the language, its features, and its implementation.

# Terminology used in this document

Pascal-P6 is a 45 year old compiler. In honor of that, I frequently use the vernacular of the day:

|  |  |  |
| --- | --- | --- |
| Card | As in “option card” | A computer punch card. These could carry a line of text or binary data, typically 80 characters (bytes) or more per card. |
| deck | As in “object deck” | A stack of computer cards constituting an entire source or object program or data. |

# Pascal-P6 vs. Pascal-P5

P6 processes the entire ISO 7185 Pascal language as a proper subset of Pascaline. At the ISO 7185 Pascal level, Pascal-P6 and Pascal-P5 are completely compatible. However, there are reasons you should use Pascal-P6, even if you are only using strictly the ISO 7185 Pascal language:

* Pascal-P6 provides many new features, including debug mode, enhanced standards checking, and new output modes such as the faster cmach mode and eventually target machine output generators.
* Pascal-P6 fixes some bugs/issues with Pascal-P5. Just as Pascal-P4 fixed some Pascal-P2 issues, and Pascal-P5 fixed some Pascal-P4 issues, not all fixes of Pascal-P5 problems in Pascal-P6 made it back to Pascal-P5. There is considerable work involved with backwards porting all such fixes.

Thus unless you have a good reason, you should be using Pascal-P6, the latest in the series. See also the comments on strict ISO 7185 useage in the next section.

# Pascal-P6 as a strict ISO 7185 compiler

Pascal-P6 can easily be used as a strict ISO 7185 Pascal compiler. Using the s+ flag, or “strict” flag, the source language for Pascal-P6 and Pascal-P5 is identical. It is, however, important to understand the exact meaning of the s+ flag.

I use the term “strict” for the s+ flag, and not “standard”, because the P6 compiler is fully compatible with the ISO 7185 Pascal language regardless of if the flag is off or on[[1]](#footnote-2). In pascal-P6 the option has the following meanings when ON:

* All extended keywords from Pascaline are turned OFF, meaning they can be used as program identifiers.
* So called “force sequences” or “\” character special interpretations in strings is turned off, meaning the “\” character has no special meaning in strings.

Both of these are in compliance with the ISO 7185 standard. An implementation is allowed to define new keywords in ISO 7185 Pascal. Also the interpretation of characters in ISO 7185 Pascal is up to the implementation (it does not even specify ASCII).

I recommend that even if you are developing strict ISO 7185 Pascal source code that you leave the s flag OFF (the default). Why? If the flag is off, that allows you to use keywords that are reserved in Pascaline, or use the force character “\” in your source. This will cause problems later if you then decide to use Pascaline extensions. In fact, the best way to develop truly portable ISO 7185 Pascal programs is to avoid using other ISO 7185 Pascal implementations’ keywords as well. You might want to examine the keyword list for FPC (Free Pascal) and GPC (Gnu Pascal) as well to make sure.

The use of the force character (“\”) is trickier. If you put it in your ISO 7185 programs with the s flag OFF (s-), then you will need to specify “\\” to get just the force character. But if you then compile it on a system that does not understand force sequences, it will evaluate to “\\” (double “\”). The best method is to avoid the force character whenever possible. Fortunately, most ISO 7185 Pascal compilers today do understand force sequences.

When using Pascal-P6 as a ISO 7185 compiler, I recommend that you occasionally check your compiles by turning on the s+ switch. This will make sure that you have not inadvertently used Pascaline constructs in your source.

# Pascal-P6 vs. FPC (Free pascal) and GPC (Gnu Pascal), Borland, UCSD or other dialects

Pascal-P6 does not accept the common dialects[[2]](#footnote-3) of Borland or UCSD languages[[3]](#footnote-4). There are good implementations of these dialects, both by Borland themselves and by the FPC group. My recommendation is if you want to use that source language, use the best compiler for it, namely the FPC or Borland compilers. There is no plan for a Pascal-P series compiler that is also Borland dialect compatible (perhaps you could make one?), and further, the type protection model of the two are diametrically opposed.

FPC recently completed implementation of an ISO 7185 compliance switch. This makes it a good backup compiler for ISO 7185 Pascal compliant applications. See the FPC web site for more details.

Both UCSD, variants of (like Apple Pascal) and GPC are falling out of use and support or have fallen out of use and support.

See also then next section for comments on source language.

# The Pascal-P6 source language

Pascal-P6 understands the Pascaline source language as documented in “pascaline.docx”, available in the doc or documents directory of Pascal-P6 project. Pascaline is fully compatible with ISO 7185. There are, however, several things you will find only in this documentation and not the Pascaline specific documentation:

* Features of Pascaline that are not (yet) implemented in Pascal-P6.
* Local extensions in Pascal-P6 that are not also in general Pascaline.
* Concrete details of features in Pascaline specific to Pascal-P6.

These will be detailed.

## Features of Pascaline NOT implemented in Pascal-P6

The following features of Pascaline are not yet implemented in Pascal. The expectation is all of them will be implemented shortly. Also keep in mind that Pascal-P6 is still a project in development, and carries a version number < 1.0.

|  |  |
| --- | --- |
| **Pascaline manual section** | **Description** |
| **6.35** | Matrix mathematics |
| **6.36** | Properties |
| **6.38** | Channels |
| **6.39** | Classes |

Note: outside of 6.18 Fixed types, and 6.25 Overloading of procedures and functions, Pascal-P6 implements all of the IP Pascal source language.

In addition to the list, Pascal-P6 also does not implement:

* Parallel modules.
* Most of the annexes, including Unicode, string library, system library, etc.

Additionally, I plan to implement a full unsigned cardinal type. Presently, it is just an alias of integer, as the Pascaline standard allows for.

Please see “pascaline.docx” in the doc file section for more details on the Pascaline language.

## Pascal-P6 implementation details

There are no particular limits of integer sizes, reals or sets built into Pascal-P6. As a hosted compiler, it picks up many or even most of its characteristics from the host compiler.

At this writing, there is a 32 bit and a 64 bit implementation of Pascal-P6. The following characteristics exist:

|  |  |
| --- | --- |
| **Type** | **Size/limit** |
| **integer** | maxint = 2147483647 |
| **linteger** | Alias of integer |
| **cardinal** | Alias of integer |
| **lcardinal** | Alias of integer |
| **set** | 256 elements |
| **real** | 64 bit real |
| **sreal** | Alias of real |
| **lreal** | Alias of real |

The maximum size of any input line, identifier, or string constant is 250 characters.

The base character set is ISO 8859-1, or 8 bit characters. The compiler has no reliance on any character in the range 128 to 255 and will accept any character in that range for input or output. In the characters 0 to 128,

### Header files

All of the Pascaline header files are implemented, input, output, error, list and command. In the interpreters pint, pmach, and cmach, output, error and list all go to the standard output file. As detailed in Annex C of the Pascaline standard, undefined program header parameters must be declared for type. Any type of valid Pascaline file is acceptable, including complex structured types.

The filenames corresponding to the names of the header file parameters are read from the command line:

program print(myfile, output);

var myfile: text;

…

executed with a command line of:

print document.txt

Will assign file myfile to the name on disk “document.txt”.

The command file reads the command line passed to the program on startup. There is only one line in the file, and eof() immediately follows eoln() in the file. When the command file is read as well as other parameters appear in the program header, then the command file will be positioned after all other header file names or entries have been read from the command line.

### Alternative header value input

Pascal-P6 can have integer or real parameters in the header file, in addition to just files:

program test(myint, myreal);

var myint: integer;

myreal: real;

Pascal-P6 automatically reads these values off of the command line and into the specified variables. If a they are mixed values on the header, they will be read in turn off the command line as specified in Annex C of the Pascaline standard.

### Character escapes

Pascal-P6 implements Pascaline Annex E: character escapes in total.

### Character sets

Pascal-P6 implements Annex D.1 “ISO 8859-1 Character Set Encodings”. However, it only relies on the parts of ISO 8859 that are common to all pages in the standard. Thus, Pascal-P6 will accept, and will generate, any character in any page in the set of ISO 8859 code pages.

Note that Pascal-P6 is dependent on ISO 8859 because it can generate force control characters specific to the ISO 8859 standard.

### Modular structure

Pascal-P6 implements modules by outputting an intermediate file for each module, then the intermediates are concatenated and loaded into pint as a single file. The intermediate contains sufficient information for pint to understand the module structure, and it links the modules together.

Modules in Pascal-P6 are implemented by the following method:

1. Each module calls its own initializer block, then calls the next module in the stacking order. When the module it calls returns, that module then calls the deinitializer for the module, and returns to the module that called it. Program modules only have a single initializer block. When the block is done, the program module returns.
2. Each module is stacked up on the last. The interpreter or runtime system arranges to call the first module in the stacking order, then when it returns, the entire program exits.
3. Its up to the user to stack the modules properly on the command line. If a module is called via a function or procedure, and accesses global data from that module, the program will fault with uninitialized access error. Module overrides are an exception. Each virtual function or procedure in the module is initialized to point to an error routine when the module is started, and attempts to override an uninitialized virtual vector will generate an error.
4. The program module MUST be the last module in the stacking order.
5. A process module is like a program module, but it starts a new processor thread to run its initialize block, then calls the next module in the stacking order. From the time the thread is branched off to run the process block, the main thread and the thread running the process module run in parallel.

Note: overloads for external modules are not implemented at this time.

### Implementation of Annexes G-O

The pint interpreter has the ability to link to and run procedures and functions from the standard Pascaline libraries. These include:

|  |  |
| --- | --- |
| **Name of module** | **Function** |
| **services** | Provides basic operating system services, such as directory list, environment variables, etc. |
| **sound** | Access to sound system, including midi, wave input and output, etc. |
| **terminal** | Access to a text based terminal. |
| **graph** | Access to a graphics terminal. |
| **network** | Access to networking. |

The Pascaline libraries, originally written in Pascaline, were rewritten in C (for several reasons), and appear as the independent library called “Petit-ami”. They can be accessed by any language.

Note that not all Pascaline libraries appear in Petit-ami. For example the strings module is still written in Pascaline, and makes no sense for other languages outside of Pascaline.

#### Enabling externals

External linking to Petit-ami is enabled via the EXTERNALS define when compiling Pascal-P6. When this definition is active, all modules with the names in the Pascaline external module table above will be redirected to external defintions.

#### Using externals

The externals in Pascal-P6 are compiled with the system as object files that are linked into the system.Further, there is a block of code in pint that translates each call to the C language, in the file:

externals\_<system>.inc

Where <system> is a specific compiler implementation. The existing implementations are:

|  |  |
| --- | --- |
| **File** | **Selected by define** |
| **externals\_gnu\_pascal.inc** | GNU\_PASCAL |
| **externals\_iso7185\_pascal.inc** | ISO7185\_PASCAL |
| **externals\_pascaline.inc** | PASCALINE |

Note that the ISO 7185 file essentially selects “no externals”, since ISO 7185 Pascal does not have that capability.

Programs that use the externals will simply use or join them the same way as any other module:

program dir(output);

uses services;

var fp: filptr;

begin

list(‘\*.pas’, fp);

while fp <> nil do begin

writeln(fp^.name);

fp := fp^.next

end

end.

To allow the program to see the definitions, the module used must have a local Pascaline file:

services.pas

However, it is empty and has no procedures or functions in it.

The module used should not be included (as normal modules are) in the input deck for pint (usually created for you by the P6 script). It will not harm anything, but it will do nothing and will take up space. All the functions in the module will be redirected to the externals handling in pint.pas.

The joins statement can be equally used to access externals. The above program using this method is:

program dir(output);

joins services;

var fp: services.filptr;

begin

services.list(‘\*.pas’, fp);

while fp <> nil do begin

writeln(fp^.name);

fp := fp^.next

end

end.

# Using Pascal-P6

## Configuring P6

P6 has a simple configuration script to set up the binary, script files and compiler in use for the system, that uses the proper defaults for your system:

[Windows]

> setpath

> configure

> make

[Linux/Mac]

$ ./setpath

$ ./configure

$ make

You can avoid “setpath” by placing the ./bin directory on your path.

The configure script attempts to automatically determine the environment you are running under, choose the correct compiler, bit width of your computer, etc. You can override this by using the options for configure:

|  |  |
| --- | --- |
| **Option** | **Meaning** |
| **--gpc** | Selects the GPC compiler. |
| **--ip\_pascal** | Selects the IP Pascal compiler. |
| **--fpc** | Selects the FPC compiler. |
| **--32** | Selects 32 bit mode. |
| **--64** | Selects 64 bit mode. |
| **--help** | Prints a help menu. |

The configure script will take the preconfigured versions of the p6 binaries, the script files and other files and install them for the specified compiler.

Although the directory will contain working copies of the binaries and scripts, there is no guarantee which version it contains. Always run the configure script to setup the particular system you are using.

## Compiling and running Pascal programs with P6

To simply compile a run a program, use the P6 batch file:

C:\> p6 hello

When a pascal program is run this way, it gets it's input from the terminal (you), and prints its results there. The p6 script accommodates the compiler that was used to build the system, and therefore you don’t need to know the exact command format of the executable.

### Compiling with multiple modules

The P6 script can accept multiple modules:

C:\> p6 mylib1 mylib2 test

The script will compile and load each of the modules in turn, and run them as a group. The program module should always be last in the module stacking order. See “Modular structure”.

### Compiling on different run configurations

There are several types of run arrangements for P6. The current options are:

|  |  |
| --- | --- |
| **Option** | **Meaning** |
| **--pmach** | Run on the pmach machine. |
| **--cmach** | Run on the cmach machine. |
| **--package** | Run via the packaging option (using cmach). |

Note that the result of running in these different configurations will not produce different results, and this feature is mostly used for testing the compiler.

## Compiler options

P6 uses a "compiler comment" to indicate options to the compiler, of the form:

(\*$option+/-,...\*)

This option can appear anywhere a normal comment can. The first character of the comment MUST be "$". This is followed by any number if option switches separated by ",". If the option end with "+", it means to turn it on. If the option ends with "-", it means turn it off.

Example:

(\*$l-\*)

Turns the listing of the source code OFF.

The following options are available:

|  |  |  |
| --- | --- | --- |
| **Option** | **Meaning** | **Default** |
| **t+/-** | Print/don't print internal tables after each routine is compiled. | OFF |
| **l+/-** | List/don't list the source program during compilation. | ON |
| **d+/-** | Add extra code to check array bounds, subranges, etc. | ON |
| **c+/-** | Output/don't output intermediate code. | ON |
| **v+/-** | Check variant records. | ON |
| **r+/+** | Perform reference checking. | ON |
| **u+/-** | Perform undiscriminated variant checking. | ON |
| **s+/-** | Restrict input language to ISO 7185 Pascal. | OFF |
| **x+/-** | Dump lexical information during the run. | OFF |
| **b+/-** | Print goto labels used at the end of the compile. | OFF |
| **y+/-** | Dump display information at the end of the compile (symbols). | OFF |
| **i+/-** | Check VAR block violations. | OFF |
| **g+/-** | Dump goto and other location labels at intermediate assembly time. | OFF |
| **h+/-** | Add source line sets to code. This tells pint where the source lines are in the code. | ON |
| **n+/-** | Obey heap recycle requests. If false, no space is recycled. | ON |
| **o+/-** | Check for arithmetic overflow. | ON |
| **p+/-** | Check for reuse of freed entry in heap space. | OFF |
| **m+/-** | Break returned entries in heap into occupied and free blocks. | OFF |
| **q+/-** | Check accesses to undefined memory. | ON |
| **w+/-** | Enter debug mode (pint only). | OFF |
| **r+/-** | Perform reference checks. | ON |
| **a+/-** | Enter debugger on fault | OFF |
| **f+/-** | Perform source level debugging. | OFF |
| **e+/-** | Output binary deck from pint instead of running the assembled results (pint only). | OFF |

It is possible to cause problems in the code if the options are turned on or off arbitrarily in the middle of the program. It is recommended that you place option sets at the top of the program only unless you are sure what you are doing.

P6 provides a lot of switches for developers to use to debug the compiler. A good rule is: if you don’t know what a switch does, don’t use it. P6 is a compiler developer’s compiler. This means that everything in the system is available to change, including things that make no sense for end users.

Note that a lot of the debug switches previously available in Pascal-P5 have been moved to the debug mode in P6. See “Pascal-P6 debugger mode”.

### Option descriptions

t+/- Print/don't print internal tables after each routine is compiled.

Prints the complete set of identifier and types declared in each block at the end of the block.

l+/- List/don't list the source program during compilation.

Enables the compiler to output a source listing.

d+/- Add extra code to check array bounds, subranges, etc.

Enables several debug checks in the code.

c+/- Output/don't output intermediate code.

Supresses the creation of the intermediate file contents. Note that the output intermediate file is always created, but it’s contents will be empty.

v+/- Check variant records.

Perform active variant checks. Checks if the variant in a variant record is active when accessed.

r+/+ Perform reference checking.

Checks if identifiers used in the program have been referenced by the code.

u+/- Perform Undiscriminated variant checking.

Forces checking on undiscriminated variants. These are variant records where the tagfield is not allocated, meaning there is no tagfield to check before accessing a given variant. The u flag forces the tagfield to be allocated as an anonymous member of the record, and will assign it the correct value any time a write access is done within a variant. On read access to the variant, the anonymous tag field value is checked, and faulted if not correct. This option actually generates more code than just specifying the tag field exists and setting it properly. However, the overhead of this check can be simply removed by removing the option. See also the v option.

s+/- Restrict input language to ISO 7185 Pascal.

Produces errors on any Pascaline level language features. Disables all Pascaline extended keywords. Removes the ability to use force sequences in strings (“\”). Note that this is ***not a ISO 7185 compliance flag***. Pascaline is completely compliant with ISO 7185 Pascal regardless of the state of the s option. It simply restricts the source language to ISO 7185 Pascal only.

x+/- Dump lexical information during the run.

Outputs the lexical tolkens scanned from the input source.

b+/- Print goto labels used at the end of the compile.

Prints all goto labels used in the compile. For debugging purposes.

y+/- Dump display information at the end of the compile (symbols).

Performs a dump of all identifiers in the display at the end of the compile run. For debugging purposes.

i+/- Check VAR block violations.

Checks various faults caused by having a parameter VAR access outstanding. This includes changing a variant while there is an outstanding reference, or disposing a dynamic variable, etc. This is an expensive check, and thus normally set OFF. It should be enabled and used for advanced checking of a running program.

g+/- Dump goto and other location labels at intermediate assembly time.

Dumps the interpreter label tables at the end of intermediate assembly.

h+/- Add source line sets to code. This tells pint where the source lines are in the code.

Adds source set instructions in the output byte code. These are used to produce error diagnostics indicating what source line was executing. Note that debug mode does not use these instructions for listing, only for breakpoints on given source lines.

n+/- Obey heap recycle requests. If false, no space is recycled.

This flag is used to turn off all recycling of dynamic variables. Any variables recycled will simply be left occupied, effectively quarantining them. This is used to debug recycle issue. Note that there are other options that fault on attempts to access variables that have previously been disposed, such as the p and m flags.

o+/- Check for arithmetic overflow.

Enables checking for various overflow conditions.

p+/- Check for reuse of freed entry in heap space.

When variables are freed, the space returned is broken into a flagged entry that generates faults on use, followed by free space. This effectively enforces use after recycle checks but with low waste of freed blocks.

m+/- Modifies the p flag to flag the remaining space as occupied.

The use of p option should check all attempts to use a previously freed block. However there are cases where, especially when mixed protection checks are enabled, that it is better not to reuse the recycled space at all. This is effectively a mixture of the n and p flags, and thus combines stoppage of recycling with flagging of reuse after dispose checks.

q+/- Check accesses to undefined memory.

Checks if an undefined location is accessed in the code. The interpreter keeps a large bitmap of memory and checks read/write accesses. Note that some undefined checks such as pointer undefined cannot be turned off. Note also that this check is only performed in the interpreter.

w+/- Enter debug mode (pint only).

Causes pint to enter debug mode before starting the program. See also “Pascal-P6 debugger mode”.

r+/- Perform reference checks.

Checks if identifiers declared in the current block have been referenced, and outputs a warning if not.

a+/- Enter debugger on fault

Enters debug mode if any fault occurs. See also the w option.

f+/- Perform source level debugging.

Loads the source code file(s) for the current code, and cross indexes the source code lines. Debug mode then changes to source mode. See also “Pascal-P6 debugger mode”.

e+/- Output binary deck from pint instead of running the assembled results (pint only).

Changes the way pint works. After the intermediate is assembled to byte code into the internal store, the entire byte code program is output in a hex file format. Then pint stops without running the resulting program. This flag is used to prepare binaries for pmach or cmach. Note that the output file, which normally contains output from the running program, now will contain the hex encoded binary, and thus should be treated differently. See also “Error: Reference source not found”.

## Other operations

Within the P6 toolset, you will find a series of scripts to perform common operations using P6. This includes building the compiler and interpreter using an existing ISO 7185 compatible compiler, and also testing P6.

The scripts used in P6 are designed to be independent of what operating system you are running on. The P6 system as been successfully run on the following systems:

* Windows
* Ubuntu linux
* Mac OS X

To enable this to work, there are two kinds of scripts available, one for DOS/Windows command shells, and another for Unix/Bash. These two script files live side by side, because the DOS/Windows scripts use a .bat extension, and Bash scripts use no extentions. Thus, when a script command is specified here, the particular type of script file is selected automatically.

The only exception to this rule is that Unix users commonly do not place the current directory in the path. This means to execute a script file in the current directory, you need to specify the current directory in front of the script. For example:

~/P6$ ./P6 hello

## Reliance on Unix commands in the P6 toolset

Most of the scripts in this package, even the DOS/Windows scripts, rely on Unix commands like cp, sed, diff, chmod and others. I needed a reasonable set of support tools that were command line callable, and these are all both standard and reasonable.

For Windows, the Mingw toolset is available:

http://www.mingw.org/

Mingw uses GNU programs that are compiled as native Windows .exe files without special .dll files

Where possible, I have tried to use DOS/Windows commands. The scripts are available in both DOS/Windows and bash versions. I could have just required the use of bash, which is part of the cygwin toolkit, but my aim is not to force Windows users into a Unix environment.

Using the Mingw toolkit, it is possible to use the bash scripts by simply executing bash under Windows.

## The “flip” command and line endings

Every effort was made to make the Pascal-P6 compile and evaluate system idenpendent of what system it is running on, from Windows command shell, to Linux with Bash shell. One common thing I have found is that several utilities don’t appreciate seeing a line ending outside of their “native” line ending, such as CRLF for Windows, and LF for linux. Examples include “diff” (find file differences) and Bash.

Therefore many of the scripts try to remove the line ending conserations, either by ignoring such line endings, or by converting all of the required files to the particular line ending in use.

The key to this is the “flip” utility. After searching for several line ending converters, “flip” was found on the most number of systems, as well as being one of the most clear and reliable utilities (it translates in both directions, it tolerates any mode of line ending as input, will not corrupt binaries, etc.).

Unfortunately, even flip was not found on some systems. The simpliest way to fix this was to include the flip.c program with the distribution, then let you compile to form a binary on your system to replace the utility.

To make the flip utility, you run:

$ make\_flip

Then flip will exist in the root directory.

# Pascal-P6 debugger mode

Debug mode in Pascal-P6 gives you the ability to step through, set break points, dump variable values and other facilities to enable debugging of issues with Pascaline code, both at the machine level and also at source level.

There are two levels you can debug a Pascal-P6 program at, the interpreter (byte code) level using pint and machine code level using gdb. With optimizing compilers, parts of the code can be reordered, changed or even completely removed. It is difficult, and sometimes impossible to track source lines positions in the code. In addition, gdb was not designed for the Pascal language, so the ability to print values at machine code time can be poor.

For this reason I believe you will find debugging at the interpreter level a valuable tool. Using debug mode, you can:

* Trace through the code at source level.
* Print the contents of any variable in Pascal formatted form.
* Print what blocks are currently running.
* Watch changes in variables.
* Place breakpoints at source lines.
* Run profiles on the code.

And many other operations.

Debug mode is invoked in two different ways. You can enter debug mode before the program is run with the $w option, or you can enter debug mode only after a fault occurs with the $a option.

The commands possible in debug mode are:

Commands:

|  |  |  |
| --- | --- | --- |
| **Command** | **Parameters** | **Description** |
| **h|help** |  | Help |
| **l** | [m] [s[ e|:l] | List source lines |
| **lc** | [s[ e|:l] | List source and machine lines coordinated |
| **li** | [s[ e|:l] | List machine instructions |
| **p** | v | Print expression |
| **d** | [s[ e|:l] | Dump memory |
| **e** | a v[ v]... | Enter byte values to memory address |
| **st** | d v | Set program variable |
| **pg** |  | Print all globals |
| **pl** | [n] | print locals for current/number of enclosing blocks |
| **pp** | [n] | print parameters for current/number of enclosing blocks |
| **ds** |  | Dump storage parameters |
| **dd** | [n] | Dump display frames |
| **df** | [n] | Dump frames formatted (call trace) |
| **b** | [m] a | Place breakpoint at source line number/routine |
| **tp** | [m] a | Place tracepoint at source line number/routine |
| **bi** | a | Place breakpoint at instruction |
| **tpi** | a | Place tracepoint at instruction |
| **c** | [a] | Clear breakpoint/all breakpoints |
| **lb** |  | List active breakpoints |
| **w** | a | Watch variable |
| **lw** |  | List watch table |
| **cw** | [n] | Clear watch table entry/all watch entries |
| **lia** |  | List instruction analyzer buffer |
| **lsa** |  | List source analyzer buffer |
| **s** | [n] | Step next source line execution |
| **ss** | [n] | Step next source line execution silently |
| **si** | [n] | Step instructions |
| **sis** | [n] | Step instructions silently |
| **hs** |  | Report heap space |
| **pc** | [a] | Set/print pc contents |
| **sp** | [a] | Set/print sp contents |
| **mp** | [a] | Set/print mp contents |
| **np** | [a] | Set/print np contents |
| **ti** |  | Turn instruction tracing on |
| **nti** |  | Turn instruction tracing off |
| **tr** |  | Turn system routine tracing on |
| **ntr** |  | Turn system routine tracing off |
| **ts** |  | Turn source line tracing on |
| **nts** |  | Turn source line tracing off |
| **spf** |  | Turn on source level profiling |
| **nspf** |  | Turn off source level profiling |
| **An** |  | Turn on analyzer mode |
| **Nan** |  | Turn off analyzer mode |
| **R** |  | Run program from current pc |
| **Ps** |  | Print current registers and instruction |
| **Q** |  | Quit interpreter |

## Sample program for debug

The following program will be used for reference in the text. The sample run is here:

C:\projects\PASCAL\pascal-p6>p6 test

Compiling test...

P6 Pascal compiler vs. 0.1.x

1 -32 program test(output);

2 -32

3 -32 var a: array 10 of integer;

4 -32 i: integer;

5 -32

6 -32 function y(add: integer): integer;

7 -36

8 -36 var i: integer;

9 -40

10 -40 begin

11 8

12 8 i := 10;

13 10

14 10 result add+i

15 11

16 11 end;

17 15

18 15 begin

19 17

20 17 for i := 1 to 10 do

21 27 a[i] := y(i);

22 46 for i := 10 downto 1 do writeln('Value: ', a[i]);

23 83

24 83 end.

Errors in program: 0

P6 Pascal interpreter vs. 0.1.x

Assembling/loading program

Running program

Value: 20

Value: 19

Value: 18

Value: 17

Value: 16

Value: 15

Value: 14

Value: 13

Value: 12

Value: 11

program complete

## Debug mode invocation

You invoke debug mode with a $w (debug always) or $a (debug on fault) option specified in the source. The modifier for debug mode is $f (debug source level).

To invoke the debug mode with source debugging on, we add the option comment “${w,f}” to the top of the program:

{$w,f}

program test(output);

...

As run, it executes:

C:\projects\PASCAL\pascal-p6>p6 test

Compiling test...

P6 Pascal compiler vs. 0.1.x

1 -32 {$w,f}

2 -32 program test(output);

...

Errors in program: 0

P6 Pascal interpreter vs. 0.1.x

Assembling/loading program

Running program

=== break ===

P6 debug mode

pc: 000000 sp: 1000000 mp: 1000000 np: FFFFFFFF

b \* 000000: 14 lnp\* 00000230

debug>

Note that the program is started at location 0, which has startup machine instructions inserted by pint. Debug mode defaults to machine level mode when it detects that the location being executed does not lie within a source module. To reach the first source program statements, we execute:

debug> s

1: 0: b {$w,f}

2: 1: \* program test(output);

3: 0:

debug>

“s” for “step source statement”, which tells the debugger to execute until the next source line is reached, which in this case is the first source line of the program.

## Debug mode expressions

Where a debug mode command takes a value, it can usually take an expression. Expressions follow (loosely) Pascal expression rules, from lowest to highest priority:

a = b, a <> b, a <= b, a >= b, a < b, a > b a in b

a + b, a – b, a or b, a xor b

a \* b, a / b, a div b, a and b

not a, (a), [set]

Operands can be integer, real, string or set. Constants are:

123 Number

$ac0 hex number

&654 octal number

%10101 binary number

‘hello’’s’ string

* 1. (real)

Variables are any standard Pascaline variable reference. Qualident notation can be used.

test.y.i

Refers to the identifier i in the function y of the module test. Unlike Pascaline, qualidents can be used to refer to any given nested symbol.

Qualidents need only be used for symbols that are not local to the current execution context. So while executing inside function y:

i

y.i

and

test.y.i

All refer to the local variable i of the function y in module test.

All of the Pascaline variable operators function in debug:

a[i] Reference to element I of array a.

r.f Reference to field f of record r.

p^ Reference to the object p points to.

## Executing multiple commands

Any number of commands can be executed on a single command line by separating the commands with “;” (semicolon):

debug> p 1;p 2

1

2

## Debug mode Command descriptions

* + 1. h or help Print help menu

The help or h command simply prints the command menu:

debug> help

Commands:

h|help Help (this command)

l [m] [s[ e|:l] List source lines

lc [s[ e|:l] List source and machine lines coordinated

li [s[ e|:l] List machine instructions

p v Print expression

d [s[ e|:l] Dump memory

e a v[ v]... Enter byte values to memory address

st d v Set program variable

pg Print all globals

pl [n] print locals for current/number of enclosing blocks

pp [n] print parameters for current/number of enclosing blocks

ds Dump storage parameters

dd [n] Dump display frames

df [n] Dump frames formatted (call trace)

b [m] a Place breakpoint at source line number/routine

tp [m] a Place tracepoint at source line number/routine

bi a Place breakpoint at instruction

tpi a Place tracepoint at instruction

c [a] Clear breakpoint/all breakpoints

lb List active breakpoints

w a Watch variable

lw List watch table

cw [n] Clear watch table entry/all watch entries

lia List instruction analyzer buffer

lsa List source analyzer buffer

s [n] Step next source line execution

ss [n] Step next source line execution silently

si [n] Step instructions

sis [n] Step instructions silently

hs Report heap space

pc [a] Set/print pc contents

sp [a] Set/print sp contents

mp [a] Set/print mp contents

np [a] Set/print np contents

ti Turn instruction tracing on

nti Turn instruction tracing off

tr Turn system routine tracing on

ntr Turn system routine tracing off

ts Turn source line tracing on

nts Turn source line tracing off

spf Turn on source level profiling

nspf Turn off source level profiling

an Turn on analyzer mode

nan Turn off analyzer mode

r Run program from current pc

ps Print current registers and instruction

q Quit interpreter

* + 1. l [m] [s[ e|:l] List source lines

Lists source lines in the program:

debug> l test

1: 0: b {$w,f}

2: 1: \* program test(output);

3: 0:

4: 0: var a: array 10 of integer;

5: 0: i: integer;

6: 0: t: (one, two, three);

7: 0:

8: 0: function y(add: integer): integer;

9: 0:

10: 0: var i: integer;

11: 0:

12: 0: begin

13: 0:

14: 0: i := 10;

15: 0:

16: 0: result add+i

17: 0:

18: 0: end;

19: 0:

20: 0: begin

21: 0:

22: 0: for i := 1 to 10 do

23: 0: a[i] := y(i);

24: 0: for i := 10 downto 1 do writeln('Value: ', a[i]);

25: 0:

26: 0: end.

The columns to the left:

2: 1: b\* program test(output);

Have the following meaning. The first column is the source line number. The second number is the profile number, it is a count of the number of times that line has been executed. Next, if the line is marked with a “b”, it means there is a breakpoint on the line. Next, if the column contains a “\*” it means that this is the currently executing line.

If the l command is followed by a module, then the source is printed from that module. Thus:

l test

is the same command while executing in test as:

l

Even if there are not multiple modules in the current program, this can be useful:

Running program

=== break ===

P6 debug mode

pc: 000000 sp: 1000000 mp: 1000000 np: FFFFFFFF

b \* 000000: 14 lnp\* 000002A0

debug> l

\*\*\* Module must be active

debug> l test

1: 0: b\* {$w,f}

2: 0: program test(output);

3: 0:

4: 0: var a: array 10 of integer;

5: 0: i: integer;

Recall when we started the program it began in system code, outside of any module. But we simply specified a particular module, test, and then we can list its’ source.

The list command can specify a starting source line to list:

debug> l 20

20: 0: begin

21: 0:

22: 0: for i := 1 to 10 do

23: 0: a[i] := y(i);

24: 0: for i := 10 downto 1 do writeln('Value: ', a[i]);

25: 0:

26: 0: end.

You can also specify an end line:

debug> l 20 22

20: 0: begin

21: 0:

22: 0: for i := 1 to 10 do

Or alternatively, you can use a length, or number of lines to list:

debug> l 20:4

20: 0: begin

21: 0:

22: 0: for i := 1 to 10 do

23: 0: a[i] := y(i);

* + 1. li [s[ e|:l] List machine instructions

The list instructions command is the same as source list, but lists machine instructions:

debug> li

Addr Op Ins P Q

----------------------------------

b 000000: 14 lnp\* 000002A0

000005: 15 cal 0000000B

00000A: 3A stp\*

00000B: AE mrkl\* 00000001

000010: AE mrkl\* 00000002

Note that you cannot specify which module to list from.

Just as for list source, you can specify start, start and end, or start and length:

Addr Op Ins P Q

----------------------------------

b 000000: 14 lnp\* 000002A0

000005: 15 cal 0000000B

00000A: 3A stp\*

debug> li 0:10

Addr Op Ins P Q

----------------------------------

b 000000: 14 lnp\* 000002A0

000005: 15 cal 0000000B

The numbers used for the command are the addresses in program store. Note the length is not “number of instructions” but rather a swath of addresses. When specifying a length, the net end address is e-1, so:

li 0:10

is equivalent to:

li 0 10-1

The meaning of the fields in the instruction listing are:

1. Addr Address of the instruction.
2. Op Opcode (byte code) of the instruction.
3. Ins Memnonic of the instruction.
4. P P or routine nesting index.
5. Q 0 or more operands of the instruction.
   * 1. lc [s[ e|:l] List source and machine lines coordinated

The lc command combines the l (list source) and li (list instructions) commands. For each source line printed, the machine code generated by that source line is also printed:

debug> lc 8:10

8: 0: function y(add: integer): integer;

000037: AE mrkl\* 00000008

00003C: AE mrkl\* 00000009

9: 0:

00003C: AE mrkl\* 00000009

000041: AE mrkl\* 0000000A

10: 0: var i: integer;

000041: AE mrkl\* 0000000A

000046: 0D ents FFFFFFD8

00004B: AD ente FFFFFFF8

000050: AE mrkl\* 0000000B

11: 0:

000050: AE mrkl\* 0000000B

000055: AE mrkl\* 0000000C

12: 0: begin

000055: AE mrkl\* 0000000C

00005A: AE mrkl\* 0000000D

13: 0:

00005A: AE mrkl\* 0000000D

00005F: AE mrkl\* 0000000E

14: 0: i := 10;

00005F: AE mrkl\* 0000000E

000064: 7B ldci 0000000A

000069: 02 stri 00,FFFFFFD8

00006F: AE mrkl\* 0000000F

15: 0:

00006F: AE mrkl\* 0000000F

000074: AE mrkl\* 00000010

16: 0: result add+i

000074: AE mrkl\* 00000010

000079: 00 lodi 00,FFFFFFDC

00007F: 00 lodi 00,FFFFFFD8

000085: 1C adi

000086: 02 stri 00,FFFFFFFC

00008C: AE mrkl\* 00000011

17: 0:

00008C: AE mrkl\* 00000011

000091: AE mrkl\* 00000012

This can be very useful if you are interested in what machine code goes with what source line.

The numbers used with the command are source line numbers.

* + 1. p v Print expression

Prints a Pascaline/debug mode expression.

The expression can be any of the Pascaline-like expressions shown in “Debug mode expressions”. Examples (at the source location line 15):

14: 1:

15: 1: b\* result add+i

16: 0:

debug> p i

10

debug> p test.i

1

debug> p i+42

52

debug> p a

array 11, 12, 13, 14, 15, 16, 17, 18, 19, 20 end

Note in the function y(), there is a I that is local, and another that is global that we access via test.i.

The array a that is specified is printed in total. Any structured type that is printed is printed in whole with the exception of files. An example complex structure is:

var r: record a: (one, two, three);

b: integer;

c: array 5 of integer

end;

debug> p r

record

two(2),

42,

array -9, -8, -7, -6, -5 end

end

The p command prints structured values using the same syntax as the Pascaline “fixed” declaration.

The p command also obeys Pascal-P6 radix conversions:

debug> p 42$

2A

debug> p 42&

52

debug> p 42%

101010

debug> p $42$

42

Note that “$42$” means convert hex 42 into hex 42, i.e., no-op.

If the value printed is undefined, a “\*” will be printed:

debug> b test 23

debug> r

=== break ===

22: 10: a[i] := y(i);

23: 1: b\* for i := 10 downto 1 do writeln('Value: ', a[i]);

24: 0:

debug> p i

\*

We broke in after the for loop. Why is I undefined after the for loop? Its actually in the ISO 7185 specification: the loop variable on a for loop is specified as undefined after the for loop.

Note that when the print command prints an encoded value such as an enumerated type or character, it prints the decimal value in parenthses:

var c: char;

debug> p c

'c'(99)

* + 1. d [s[ e|:l] Dump memory

Dumps byte machine memory.

debug> d 0:50

000000: 14 98 02 00 00 15 0B 00 00 00 3A AE 01 00 00 00

000010: AE 02 00 00 00 0B 00 0C 00 92 00 00 00 16 AE 03

000020: 00 00 00 AE 04 00 00 00 AE 05 00 00 00 AE 06 00

000030: 00 00

Dump memory defaults to all of memory, 0 to memory top. As with source line lists, you can specify a start address, a start and end address, or a start address and a length.

* + 1. e a v[ v]... Enter byte values to memory address

Enters values to byte machine memory. The first value is the address to enter to. This is followed by 1 to n values to be placed into memory in turn:

debug> d 0:20

000000: 14 98 02 00 00 15 0B 00 00 00 3A AE 01 00 00 00

000010: AE 02 00 00

debug> e 0 1 2 3

debug> d 0:20

000000: 01 02 03 00 00 15 0B 00 00 00 3A AE 01 00 00 00

000010: AE 02 00 00

Note that there is no restriction on where you enter bytes into memory. You can just as easily rewrite the program code in memory. Enter is a sharp tool.

* + 1. st d v Set program variable

Sets a variable in the current context. The first parameter is the variable to set. The second is the value to set the variable to. The variable to set can be any variable reference:

=== break ===

22: 10: a[i] := y(i);

23: 1: b\* for i := 10 downto 1 do writeln('Value: ', a[i]);

24: 0:

debug> p a

array 11, 12, 13, 14, 15, 16, 17, 18, 19, 20 end

debug> st a[3] 42

debug> p a

array 11, 12, 42, 14, 15, 16, 17, 18, 19, 20 end

Set program variable is fairly flexible:

var s: packed array 10 of char;

debug> p s

'hi there '

debug> st s 'lo there '

debug> p s

'lo there '

var st: set of char;

debug> p st

['a'..'d','z']

debug> st st ['0'..'9']

debug> p st

['0'..'9']

* + 1. pg Print all globals

Prints the globals for the current program.

debug> pg

Globals:

i \*

a array 11, 12, 13, 14, 15, 16, 17, 18, 19, 20 end

* + 1. pl [n] print locals for current/number of enclosing blocks

Prints the locals for the current block or enclosing blocks:

=== break ===

14: 2:

15: 2: b\* result add+i

16: 1:

debug> pl

Locals for block: y

i 10

add 2

If a number of blocks argument appears, then that number of enclosing blocks are printed:

debug> pl 2

Locals for block: x

q Undefined

c 'c'(99)

Locals for block: y

i 10

add 1

Parameters are not considered part of the locals.

* + 1. pp [n] print parameters for current/number of enclosing blocks

Prints parameters for the current or surrounding blocks:

debug> pp

Locals for block: x

c 'c'(99)

debug> pp 2

Locals for block: x

c 'c'(99)

Locals for block: y

add 1

* + 1. ds Dump storage parameters

ds prints the storage areas of the byte machine:

debug> ds

Storage areas occupied

Program 000000-00026B (620)

Constants 00026C-0002AB (64)

Globals 0002AC-00032F (132)

Stack/Heap 000330-FFFFFF (16776400)

The program area is where the bytecode program lives. Constants is the location of the constants deck. Globals contains all of the program globals. Finally, the stack/heap contains the rest of the byte machine space, with the heap growing up, and the stack growing down.

* + 1. dd [n] Dump display frames

Dumps the contents of all active display frames. This is mainly for debugging the interpreter. It shows all of the active stack marks and their contents.

debug> b test 24

debug> dd

No displays active

debug> r

=== break ===

23: 10: a[i] := y(i);

24: 1: b\* for i := 10 downto 1 do writeln('Value: ', a[i]);

25: 0:

debug> dd

Mark @FFFFFC

sl: 00FFFFF0: 01000000

dl: 00FFFFEC: 01000000

ep: 00FFFFE8: 00000005

sb: 00FFFFE4: 00FFFFD4

et: 00FFFFE0: 00FFFFC4

ra: 00FFFFDC: 0000001D

Mark @1000000

sl: 00FFFFF4: 00000000

dl: 00FFFFF0: 01000000

ep: 00FFFFEC: 01000000

sb: 00FFFFE8: 00000005

et: 00FFFFE4: 00FFFFD4

ra: 00FFFFE0: 00FFFFC4

The number, if present, will specify how many stack frames to dump. The default is to dump all active stack frames.

* + 1. df [n] Dump frames formatted (call trace)

The df or dump formatted command is like the dd command, but tells you what block was executing by name, what address within the block it was executing, and what the start and end address of the block of local variables is for that frame.

debug> b test 24

debug> r

=== break ===

23: 10: a[i] := y(i);

24: 1: b\* for i := 10 downto 1 do writeln('Value: ', a[i]);

25: 0:

debug> df

test: addr: 00000143 locals/stack: 00FFFFC4-00FFFFDC (24)

Note that the program/module frame has very few locals, since most of its variables are global.

* + 1. b [m] a Place breakpoint at source line number/routine

The b or break command places a source line breakpoint. If the program is run, it will stop at that breakpoint.

debug> b test 23

debug> r

=== break ===

22: 1: for i := 1 to 10 do

23: 1: b\* a[i] := y(i);

24: 0: for i := 10 downto 1 do writeln('Value: ', a[i]);

debug>

When the break occurs, the line the execution stopped on is printed, along with the line above and the line below, if they are available (the currently executing line could be at the start or end of the program).

There is a limit of 10 breakpoints outstanding. The module containing the source line can optionally be specified, otherwise the default is the current module executing.

* + 1. tp [m] a Place tracepoint at source line number/routine

The tp or tracepoint command sets a tracepoint at the given source address. Specifying the module containing the source line is optional.

Tracepoints are like breakpoints, but they do not stop after printing the currently executing line. Thus they will effectively “trace” the program execution through that point in the code.

debug> tp 23

\*\*\* Module must be active

debug> tp test 23

debug> r

22: 1: for i := 1 to 10 do

23: 1: t\* a[i] := y(i);

24: 0: for i := 10 downto 1 do writeln('Value: ', a[i]);

22: 1: for i := 1 to 10 do

23: 2: t\* a[i] := y(i);

24: 0: for i := 10 downto 1 do writeln('Value: ', a[i]);

22: 1: for i := 1 to 10 do

23: 3: t\* a[i] := y(i);

24: 0: for i := 10 downto 1 do writeln('Value: ', a[i]);

...

Just as for source breakpoints, the module can be specified or left to the current default.

* + 1. bi a Place breakpoint at instruction

Places a machine instruction level breakpoint. These breakpoints can be placed anywhere in the program, and thus no module is required.

debug> lc test

1: 0: b\* {$w,f}

2: 0: program test(output);

3: 0:

00001E: AE mrkl\* 00000003

000023: AE mrkl\* 00000004

4: 0: var a: array 10 of integer;

000023: AE mrkl\* 00000004

000028: AE mrkl\* 00000005

5: 0: i: integer;

...

000138: 05 lao 00000298

00013D: BD inv

00013E: AE mrkl\* 00000018

24: 0: for i := 10 downto 1 do writeln('Value: ', a[i]);

00013E: AE mrkl\* 00000018

000143: 7B ldci 0000000A

...

25: 0:

0001EC: AE mrkl\* 00000019

0001F1: AE mrkl\* 0000001A

26: 0: end.

0001F1: AE mrkl\* 0000001A

0001F6: 0E retp

debug> bi $13e

debug> r

=== break ===

23: 10: a[i] := y(i);

24: 1: \* for i := 10 downto 1 do writeln('Value: ', a[i]);

25: 0:

debug>

Note that the break instruction print will be in source mode unless you change to machine mode.

Note also that there is only one breakpoint table, and it is shared by all source and machine level breakpoints and tracepoints.

* + 1. tpi a Place tracepoint at instruction

Places an instruction level tracepoint.

* + 1. c [a] Clear breakpoint/all breakpoints

Clears either a single breakpoint or tracepoint, or all of them from the table. The number to clear corresponds to the entry number in the table.

debug> lb

Breakpoints:

No Src Addr Trc/brk

=======================

1: 1: 000000 b

2:\*\*\*\*: 00013E b

debug> c 2

debug> lb

Breakpoints:

No Src Addr Trc/brk

=======================

1: 1: 000000 b

* + 1. lb List active breakpoints

Lists the breakpoints in the table.

debug> b test 15

debug> tp test 18

debug> bi 5

debug> tpi 10

debug> lb

Breakpoints:

No Src Addr Trc/brk

=======================

1: 1: 000000 b

2: 15: 000079 b

3: 18: 000096 t

4:\*\*\*\*: 000005 b

5:\*\*\*\*: 00000A t

The first entry in the table is the logical entry number. This is what you use to clear the breakpoint. The second entry is the source line number. This will be all “\*\*\*\*” if there is no source line associated with it (it is a machine level breakpoint). This is followed by the address of the breakpoint, and then a “b” (for breakpoint) or “t” for tracepoint.

* + 1. w a Watch variable

Set a variable watch. A variable watch prints a diagnostic every time the indicated variable is changed. Any Pascaline variable in the current scope can be watched.

debug> b test 22

debug> r

=== break ===

21: 1:

22: 1: b\* for i := 1 to 10 do

23: 0: a[i] := y(i);

debug> w i

debug> r

Watch variable: @000000D1: i@00000298: \* -> 1

Watch variable: @0000012E: i@00000298: 1 -> 2

Watch variable: @0000012E: i@00000298: 2 -> 3

Watch variable: @0000012E: i@00000298: 3 -> 4

...

The watch message line tells you what program instruction modified the variable, followed by what the variable was that was modified, and its’ address, then what the value of the variable was, and what it was changed to.

Only simple variables can be watched, not structured. If you want to watch a variable within a structured type, indicate what element you wish to watch:

debug> b test 22

debug> r

=== break ===

21: 1:

22: 1: b\* for i := 1 to 10 do

23: 0: a[i] := y(i);

debug> w a[5]

debug> r

Watch variable: @00000112: a@00000280: \* -> 15

When any block ends, any watches to variables within that block are automatically evicted.

* + 1. lw List watch table

Lists the watch table:

debug> lw

Watch table:

1: 00000270

This gives the logical number of the watch entry, followed by the address the watch was placed on.

* + 1. cw [n] Clear watch table entry/all watch entries

Clears either a specific watch entry by logical number, or clears the entire watch table.

* + 1. lia List instruction analyzer buffer

Lists the contents of the instruction analyzer buffer. The instruction analyzer buffer is a queue that continually stores the address of the executing instructions and evicting the oldest addresses.

debug> an

debug> b test 22

debug> r

=== break ===

21: 1:

22: 1: b\* for i := 1 to 10 do

23: 0: a[i] := y(i);

debug> lia

last instructions executed:

b \* 0000B5: 7B ldci 00000001

0000B0: AE mrkl\* 00000016

0000AB: AE mrkl\* 00000015

0000A6: AE mrkl\* 00000014

0000A1: AE mrkl\* 00000013

00009C: AD ente FFFFFFF0

000097: 0D ents FFFFFFD8

000017: 0C cup 00,00000097

000015: 0B mst 00

000010: AE mrkl\* 00000002

debug>

Because running instruction analysis slows down instruction execution a bit, it must be specifically enabled. This is done with the an (analysis on) command.

* + 1. lsa List source analyzer buffer

Lists the contents of the source analyzer buffer. The source analyzer buffer is a queue that continually stores the address of the executing source lines and evicting the oldest addresses.

debug> b test 24

debug> an

debug> r

=== break ===

23: 10: a[i] := y(i);

24: 1: b\* for i := 10 downto 1 do writeln('Value: ', a[i]);

25: 0:

debug> lsa

last source lines executed:

23: 10: a[i] := y(i);

24: 1: b\* for i := 10 downto 1 do writeln('Value: ', a[i]);

18: 10: end;

17: 10:

16: 10: result add+i

15: 10:

14: 10: i := 10;

13: 10:

12: 10: begin

11: 10:

* + 1. s [n] Step next source line execution

Steps source line execution. The current source line is executed and stopped at the following instruction.

=== break ===

P6 debug mode

pc: 000000 sp: 1000000 mp: 1000000 np: FFFFFFFF

b \* 000000: 14 lnp\* 000002A0

debug> s

1: 0: b {$w,f}

2: 1: \* program test(output);

3: 0:

debug> s

1: 0: b {$w,f}

2: 1: \* program test(output);

3: 0:

debug> s

1: 0: b {$w,f}

2: 1: \* program test(output);

3: 0:

debug> s

19: 1:

20: 1: \* begin

21: 0:

debug> s

21: 1:

22: 1: \* for i := 1 to 10 do

23: 0: a[i] := y(i);

debug> s

21: 1:

22: 1: \* for i := 1 to 10 do

23: 0: a[i] := y(i);

As stated, a side use of the s command is to go from the startup code to the first source line.

If a number appears on the s command, then that many steps are done:

=== break ===

P6 debug mode

pc: 000000 sp: 1000000 mp: 1000000 np: FFFFFFFF

b \* 000000: 14 lnp\* 000002A0

debug> s 5

1: 0: b {$w,f}

2: 1: \* program test(output);

3: 0:

1: 0: b {$w,f}

2: 1: \* program test(output);

3: 0:

1: 0: b {$w,f}

2: 1: \* program test(output);

3: 0:

19: 1:

20: 1: \* begin

21: 0:

21: 1:

22: 1: \* for i := 1 to 10 do

23: 0: a[i] := y(i);

* + 1. ss [n] Step next source line execution silently

The ss (step silent) command is the same as the s (step) command except that it does not print.

=== break ===

P6 debug mode

pc: 000000 sp: 1000000 mp: 1000000 np: FFFFFFFF

b \* 000000: 14 lnp\* 000002A0

debug> ss 5

debug> ps

21: 1:

22: 1: \* for i := 1 to 10 do

23: 0: a[i] := y(i);

* + 1. si [n] Step instructions

Steps a single machine level instruction, or optionally the number of instructions specified.

=== break ===

P6 debug mode

pc: 000000 sp: 1000000 mp: 1000000 np: FFFFFFFF

b \* 000000: 14 lnp\* 000002A0

debug> si

pc: 000005 sp: 1000000 mp: 1000000 np: 0002A0

\* 000005: 15 cal 0000000B

debug> si

pc: 00000B sp: FFFFFC mp: 1000000 np: 0002A0

\* 00000B: AE mrkl\* 00000001

* + 1. sis [n] Step instructions silently

Steps a single machine level instruction silently, or optionally the number of instructions specified.

=== break ===

P6 debug mode

pc: 000000 sp: 1000000 mp: 1000000 np: FFFFFFFF

b \* 000000: 14 lnp\* 000002A0

debug> sis

debug> sis

debug> sis

debug> ps

pc: 000010 sp: FFFFFC mp: 1000000 np: 0002A0

\* 000010: AE mrkl\* 00000002

* + 1. hs Report heap space

Prints the space allocated on the heap. This is mainly used for diagnostic purposes.

debug> b test 23

debug> r

=== break ===

22: 1: new(p);

23: 1: b\* for i := 1 to 10 do

24: 0: a[i] := y(i);

debug> hs

Heap space breakdown

addr: 0002B0: 8: alloc

The first field is the address of the heap entry. The second is the length. The last indicates the entry is allocated.

* + 1. pc [a] Set/print pc contents

Set or print the contents of the pc register. If a value exists, the pc register is set. Otherwise it is printed.

debug> pc

pc: 000000C6

debug> pc 0

debug> pc

pc: 00000000

* + 1. sp [a] Set/print sp contents

Set or print the contents of the pc register. If a value exists, the pc register is set. Otherwise it is printed.

debug> sp

sp: 00FFFFD4

debug> sp $a40

debug> sp

sp: 00000A40

* + 1. mp [a] Set/print mp contents

Set or print the contents of the pc register. If a value exists, the pc register is set. Otherwise it is printed.

debug> mp

mp: 00FFFFFC

debug> mp $562

debug> mp

mp: 00000562

* + 1. np [a] Set/print np contents

Set or print the contents of the pc register. If a value exists, the pc register is set. Otherwise it is printed.

debug> np

np: 000002B8

debug> np $555

debug> np

np: 00000555

* + 1. ti Turn instruction tracing on

Turns on instruction tracing. When instruction tracing is on, a single line for each bytecode machine instruction is printed.

debug> b 20

\*\*\* Module must be active

debug> b test 20

debug> ti

debug> r

b \* 000000/1000000: 14 lnp\* 00000298

\* 000005/1000000: 15 cal 0000000B

\* 00000B/FFFFFC: AE mrkl\* 00000001

\* 000010/FFFFFC: AE mrkl\* 00000002

\* 000015/FFFFFC: 0B mst 00

\* 000017/FFFFDC: 0C cup 00,00000092

\* 000092/FFFFDC: 0D ents FFFFFFD8

\* 000097/FFFFD4: AD ente FFFFFFF0

\* 00009C/FFFFD4: AE mrkl\* 00000012

\* 0000A1/FFFFD4: AE mrkl\* 00000013

\* 0000A6/FFFFD4: AE mrkl\* 00000014

\* 0000AB/FFFFD4: AE mrkl\* 00000015

b \* 0000B0/FFFFD4: 13 brk

=== break ===

19: 1: begin

20: 1: b\*

21: 1: for i := 1 to 10 do

The instruction trace lines consist of the following:

1. A column with space or “b” to indicate that instruction has a breakpoint set.
2. A column with space or “\*” to indicate that the point of execution is on this instruction. Note that since the instruction is always active when tracing, this is always set.
3. The value of the pc register.
4. The value of the sp register.
5. The bytecode instruction value.
6. The memnonic for the instruction.
7. The p value for the instruction.
8. One or more parameters for the instruction.
   * 1. nti Turn instruction tracing off

Turns off instruction tracing.

* + 1. tr Turn system routine tracing on

Turns on system routine tracing. In this mode, a message line is produced each time a system routine is called.

debug> tr

debug> r

400/16777156-> 6

Value:

437/16777160-> 8

20

439/16777168-> 5

400/16777156-> 6

Value:

437/16777160-> 8

19

439/16777168-> 5

400/16777156-> 6

Value:

437/16777160-> 8

18

439/16777168-> 5

400/16777156-> 6

Value:

437/16777160-> 8

17

439/16777168-> 5

Note the routine messages here are intermixed with the program output.

The first value printed is the pc register. The second value is the sp register. The last value is the call number.

* + 1. ntr Turn system routine tracing off

Turns system routine tracing off.

* + 1. ts Turn source line tracing on

Turns source line tracing on. Source line tracing prints a trace each time a source line is executed.

debug> ts

debug> r

\*\*\* No active module

1: 0: b {$w,f}

2: 1: \* program test(output);

3: 0:

17: 0: end;

18: 1: \*

19: 0: begin

18: 1:

19: 1: \* begin

20: 0:

19: 1: begin

20: 1: \*

21: 0: for i := 1 to 10 do

20: 1:

21: 1: \* for i := 1 to 10 do

22: 0: a[i] := y(i);

21: 1: for i := 1 to 10 do

22: 1: \* a[i] := y(i);

23: 0: for i := 10 downto 1 do writeln('Value: ', a[i]);

For each source line executed, the line before, the current line executing, and the next line are printed.

* + 1. nts Turn source line tracing off

Turns off source line tracing.

* + 1. spf Turn on source level profiling

Turns on source level profiling. Source level profiling is a count kept for each source line that is incremented each time that line is executed.

debug> r

Value: 20

Value: 19

Value: 18

Value: 17

Value: 16

Value: 15

Value: 14

Value: 13

Value: 12

Value: 11

\*\*\* Stop instruction hit

=== break ===

\*\*\* No active module

debug> l test

1: 0: b {$w,f}

2: 1: program test(output);

3: 0:

4: 0: var a: array 10 of integer;

5: 0: i: integer;

6: 0:

7: 0: function y(add: integer): integer;

8: 0:

9: 0: var i: integer;

10: 10:

11: 10: begin

12: 10:

13: 10: i := 10;

14: 10:

15: 10: result add+i

16: 10:

17: 10: end;

18: 1:

19: 1: begin

20: 1:

21: 1: for i := 1 to 10 do

22: 10: a[i] := y(i);

23: 1: for i := 10 downto 1 do writeln('Value: ', a[i]);

24: 1:

25: 1: \* end.

The source line profile count is the second number on the source listing after the line number.

Source line profiling is not time intensive, and defaults to on.

* + 1. nspf Turn off source level profiling

Turns source line profiling off.

debug> nspf

debug> r

Value: 20

Value: 19

Value: 18

Value: 17

Value: 16

Value: 15

Value: 14

Value: 13

Value: 12

Value: 11

\*\*\* Stop instruction hit

=== break ===

pc: 00000A sp: 1000000 mp: 1000000 np: 000298

\* 00000A: 3A stp\*

debug> l test

1: b {$w,f}

2: program test(output);

3:

4: var a: array 10 of integer;

5: i: integer;

6:

7: function y(add: integer): integer;

8:

9: var i: integer;

10:

11: begin

12:

13: i := 10;

14:

15: result add+i

16:

17: end;

18:

19: begin

20:

21: for i := 1 to 10 do

22: a[i] := y(i);

23: for i := 10 downto 1 do writeln('Value: ', a[i]);

24:

25: \* end.

The listing will have the source line profiling column removed.

* + 1. an Turn on analyzer mode

Turns on analyzer mode. Both source and instruction analyzer collection is turned on. Note that analyzer mode does take a small amount of execution time, and so is normally set off.

* + 1. nan Turn off analyzer mode

Turns off analyzer mode.

* + 1. r Run program from current pc

The byte code interpreter runs from the current pc until a stop or breakpoint is hit.

* + 1. ps Print current registers and instruction

Prints the current machine execution status. The format of the printout will depend on what mode the debugger is in, either source level or machine level debugging.

debug> b test 22

debug> r

=== break ===

21: 1: for i := 1 to 10 do

22: 1: b\* a[i] := y(i);

23: 0: for i := 10 downto 1 do writeln('Value: ', a[i]);

debug> ps

21: 1: for i := 1 to 10 do

22: 1: b\* a[i] := y(i);

23: 0: for i := 10 downto 1 do writeln('Value: ', a[i]);

For source level. Note that the status is printed each time the program breaks, automatically.

debug> b test 22

debug> r

=== break ===

pc: 0000E7 sp: FFFFD4 mp: FFFFFC np: 000298

b \* 0000E7: 05 lao 0000026C

For machine level. The $f option was

How can source level breakpoints be set even when not in source debugging mode (The “b test 22” above)? Debug mode understands line numbers even if it does not load the source text for the program.

* + 1. q Quit interpreter

Exits the pint interpreter.

# Different generation and run options: pint, pmach and cmach

Pint, the traditional interpreter for the Pascal-P series, can generate a binary deck in Pascal-P6 to be run by a separate byte code machine using the $e option. See “Compiler options”. A separate bye code machine does not have the overhead of assembly or debugging of the intermediate. Thus it can be smaller and (in some cases) faster than pint. There are two alternate byte code machines available:

pmach

This is the byte machine core from pint, and directly tracks pint. Because it comes from the same code base, it has identical run time. pmach tracks pint, meaning that updates to the pint byte code machine are copied to pmach.

cmach

This is the byte machine core from pint, but translated into the C language according to ISO 9899:1990 or ANSI/ISO C (no C99 constructs are used). The purpose of cmach is as a porting tool for Pascal-P6. Any compiler that obeys the original ANSI C language can be used to implement a Pascal-P6 system using this module. It is also can be nearly an order of magnitude faster than the mainline pint/pmach machine (with the proper options set). Further, cmach features a “packaged mode”, which enables the generation of executables from a Pascal-P6 source deck.

## Running the different machines

Most of the program run scripts feature flags to optionally run programs on the different machines:

--pmach Run on the pmach machine.

--cmach Run on the cmach machine.

Example:

p6 –pmach test

To run program test on the pmach machine.

## Difference between different byte machines

The differences between the number of source code lines and run times for different machines are detailed (at current writing, and using the “prime.pas” test in the sample\_programs directory):

|  |  |  |
| --- | --- | --- |
| **Machine** | **Source lines** | **Run time for “prime.pas”** |
| **pint** | 6970 | 10.628 s |
| **pmach** | 3154 | 10.946 s |
| **cmach** | 3094 | 1.171 s |

Although technically the pmach and cmach run times “should” be similar, because they are run through the same compiler backend (GPC and GCC share a backend), there are clearly practical differences between the machines.

## Setting machine options on pmach and cmach

Because both pmach and cmach do not have access to the intermediate, they use the defaults for several run time options that are detailed in “Compiler options”. For both pmach and cmach, these options must be set in the source and then the modules recompiled.

For cmach, the options can be set via compiler flags as follows:

Option name Corresponding Pascal-P6 option Function

WRDSIZ32 - Sets 32 bit word size.

WRDSIZ64 - Sets 64 bit word size.

DOCHKOVF $o Check arithmetic overflow.

DOSRCLIN $h Add source line sets to code.

DORECYCL $n Obey heap space recycle requests.

DOCHKRPT $p Check reuse after dispose attempts.

DONORECPAR $m Do not recycle second part of disposed entries.

DOCHKDEF $q Check undefined variable accesses.

ISO7185 $s Check strict adherence to the ISO 7185 Pascal standard.

GPC - Obey GPC header file conventions, or use Pascaline header file conventions.

Where “-“ means “does not apply”. pint and pmach do not need their word size defined externally.

Note that all of the options are true/false except for WRDSIZ32 and WRDSIZ64, which are defined/not defined options. This means they must be set to 0 (false) or 1 (true).

Note that the GPC flag causes cmach to use the GPC convention that the input file prd is input from the files “prd” and “prr” (literal source names). This makes the resulting binaries compatible with the equivalent binaries compiled by GPC. This is used mainly for testing, since it can create pint and pcom binaries that are fully compatible with their GPC compiled executables.

## Packaged application mode

cmach has the ability to create “packaged” applications. This means the given application byte code is united with the cmach interpreter and generated as an executable. The resulting program looks and behaves just as a fully machine compiled application would, but does not execute any faster than the interpreted version. A packaged application may be smaller than the equivalent machine compiled application, because byte machine code is often more dense than target machine code. This needs to be determined on a case by case basis.

An application does not typically get the “interpretation bonus” until it is large enough to overcome the overhead of needing to include its’ interpreter in the program, as well as the need to carry a full set of library routines, instead of just the minimal routines needed.

“running as a bytecode machine” may be a good idea on very small machines such as 16 bit or even 8 bit microprocessors[[4]](#footnote-5). For these types of machines, the key is finding a good, compact C compiler for cmach.c, or even recoding the interpreter in native assembly language. This was done, for example, with UCSD Pascal.

Packaged applications can be generated by the pg script:

pg test

Will generate a test or test.exe executable.

Using the $e option, pint generates a byte instruction code file in hex format. The program genobj takes this byte code and converts it to a static array in the C language, then it is compiled together with cmach.c to make a final program.

The pint generated hex file format

The generated bytecode from pint using the $e format appears as:

:100000000000000000141C250000150B0000003AAE010000005E

:100000000000000010AE02000000AE03000000AE04000000AEC1

:10000000000000002005000000AE06000000AE07000000AE0824

:100000000000000030000000AE09000000AE0A000000AE0B0028

…

:1000000000000004A065726174696F6E7301000000FF1F000084

:00000000000000000000

It is based on a modified form of “Intel hex” format, see:

<https://en.wikipedia.org/wiki/Intel_HEX>

However, note it is not exactly compatible with Intel Hex format. It does not use a record type field.

The fields present in each line are:

1. Start code, the character “:”.
2. Byte count, two hex digits or 8 bits, indicating the length of the data in the record. If the byte count is zero, this is the last record in the file, and terminates the file.
3. Address, 16 hex digits or 64 bits in big endian or most significant byte to least significant byte order. This carries the address the data in the record is to be loaded to. It is all zeros on the last or “stop” record.
4. Data, 2 hex digits or 8 bits for each of the data record bytes specified by the byte count.
5. checksum, 2 hex digits or 8 bits. This is the sum of all the data bytes in the record, modulo 256, meaning that any overflow fro m the 8 bit sum is discarded. The checksum starts with a value of 0. It is all zeros on the last or “stop” record, since there are no data bytes in that record.

For further details on the hex bytecode format, see the program genobj.pas.

## Using packaged mode to port to a machine not supported by a Pascal compiler

cmach can be used to form a working set of Pascal-P6 programs on a machine that only supports an ANSI C compiler. To perform this port, you will need to package at least the set of applications:

pcom Compiler.

pint Assembler/interpreter.

genobj Byte code to C source code generator.

Note that although you can use cmach itself to run the programs pcom generates, and not pint, you need pint to perform the assembly and link step to generate input decks to cmach.

A typical target for an unsupported machine is Mac OS X, which has been unsupported since the LLVM compiler convertion (due to lack of support for GCC).

# Portability of intermediate decks

Pascal-P6 incorporates a lot of machine specific parameters into the intermediate, using the MPB or “machine parameter block” system. Indeed, this is how the system was able to port to a variety of different machines over the 45 years since it was first written[[5]](#footnote-6).

However, modern microprocessors have very similar designs, and such things as byte addressability, object alignment and other concerns are standard across machines. Pascal-P6 does not even have endian dependencies[[6]](#footnote-7). Thus, intermediate decks are typically only dependent on their machine word size. Thus at this writing, there are:

|  |  |
| --- | --- |
| **16 bit intermediate** | **Covers 8 and 16 bit microprocessors.** |
| **32 bit intermediate** | Covers 32 bit microprocessors. |
| **64 bit intermediate** | Covers 64 bit microprocessors. |

# Portability of hex/binary decks

The portability of hex/binary decks emitted by pint using the $e option are similar to intermediate decks, but, because they have encoded machine values in them, they add endian concerns to the decks. At the present time, there is no option to generate a binary with different endian mode than the host.

The hex/binary decks are:

|  |  |  |
| --- | --- | --- |
| **Word size** | **Endiam mode** | **Description** |
| **16 bit hex/binary** | Little endian | Covers 8 and 16 bit microprocessors. |
| **16 bit hex/binary** | Big endian | Covers 8 and 16 bit microprocessors. |
| **32 bit hex/binary** | Little endian | Covers 32 bit microprocessors. |
| **32 bit hex/binary** | Big endian | Covers 32 bit microprocessors. |
| **64 bit intermediate** | Little endian | Covers 64 bit microprocessors. |
| **64 bit intermediate** | Big endian | Covers 64 bit microprocessors. |

# Pascal-P6 implementation language

Pascal-P6 accepts the language Pascaline. However, Pascal-P6 is itself written in ISO 7185 Pascal. This means that Pascal-P6 can be run on any host compiler that complies with the ISO 7185 Pascal language, but then itself implements the full Pascaline language. Thus it can be used to “bootstrap” any ISO 7185 Pascal installation into the Pascaline language.

To do this, it utilizes two basic “tricks” to get beyond the limitations in ISO 7185 Pascal. First, Pascal-P6 uses the Unix/Linux cpp preprocessor to inable things like file includes and conditional compilation. cpp is run via a script pascpp or pascpp.bat. It takes a \*.pas pascal input source and creates a file by the name \*.mpp.pas, or the processed output file, then it uses the ISO 7185 Pascal host compiler to compile that file.

To make cpp work on a non-C based language, the options:

cpp -P -nostdinc -traditional-cpp

Are used. This makes cpp run as close to as possible to not relying on any interpretation of the source language.

If you are going to make changes and compile the compiler, you should expect that if you get errors, you need to look in the \*.mpp.pas files, not the original \*.pas files. The reason is because while C compilers can compensate for the line and character location differences of error reporting in the source file, Pascal compilers (esp. Pascal-P6) cannot do that. Thus you must examine the cpp output files to make sense of the errors encountered.

## Language extension routines

Although it may seem strange that a limited compiler as an ISO 7185 Pascal source compiler can compile for a language much more powerful than its source language, in fact it is not difficult and is done commonly. The biggest difference is in fundamental operating system features covered by the language. Thus, access to named files, seeking within those files, and access to the command line passed to the program are common functions that are beyond the ISO 7185 Pascal language. These are handled by so called “language extension routines”. These are short routines with standard ISO 7185 call sequences that implement the extended feature. In fact they are (at current time):

* + 1. procedure assigntext(var f: text; var fn: filnam);

Assign name to text file.

* + 1. procedure assignbin(var f: bytfil; var fn: filnam);

Assign name to binary file.

* + 1. procedure closetext(var f: text);

Close text file.

* + 1. procedure closebin(var f: bytfil);

Close binary file.

* + 1. function lengthbin(var f: bytfil): integer;

Find length binary file.

* + 1. function locationbin(var f: bytfil): integer;

Find location binary file.

* + 1. procedure positionbin(var f: bytfil; p: integer);

Position binary file.

* + 1. procedure updatebin(var f: bytfil);

Update binary file.

* + 1. procedure appendtext(var f: text);

Append text file.

* + 1. procedure appendbin(var f: bytfil);

Append binary file.

* + 1. function existsfile(var fn: filnam): boolean;

Find file exists by name.

* + 1. procedure deletefile(var fn: filnam);

Delete file by name.

* + 1. procedure changefile(var fnd, fns: filnam);

Change filename.

* + 1. procedure getcommandline(var cb: cmdbuf; var l: cmdnum);

Get the shell command line.

For each different host compiler, a complete library implementing these calls in the particular host language exists. The following are the host support library files at this writing:

extend\_gnu\_pascal.inc

Implements specific extension routines for GPC (Gnu Pascal) as the host compiler.

extend\_iso7185\_pascal.inc

This file is a dummy, and simply gives an error when any of the routines are called. This allows Pascal-P6 to be compiled with a strict ISO 7185 compiler (like Pascal-P5), and still function, but without the extended capabilities.

extend\_pascaline.inc

Implements the extension routines using Pascaline extensions. This file is used when there is a native Pascaline implementation to simply self define the routines.

# Building the Pascal-P6 system

## Compiling and running P6 with an existing ISO 7185 compiler

You do not need to compile P6 unless you are using an alternative compiler or installation. The current P6 has been compiled and run with the following compilers and operating systems:

|  |  |
| --- | --- |
| **Compiler** | **Installations** |
| **IP Pascal** | Windows |
| **GPC** | Windows, Ubuntu, Mac OSx |

First, you must have a ISO 7185 Pascal compiler available. There are several such compilers, see:

http://www.standardpascal.org/compiler.html

You will probally need to compile pcom.pas and pint.pas with the ISO 7185 Pascal compatibility mode option on for your compiler. See your compilers’ documentation for details.

If you are using a compiler or version of a compiler that is not tested to ISO 7185 standards, you will want to make sure that it is ISO 7185 compliant. See “Testing P6” section 16 for details on how to test an existing compiler to ISO 7185 standards.

You can use the script configure to select which existing compiler to run:

configure ip\_pascal

Or

configure gpc

To compile pcom and pint, the components of the P6 compiler, use the make file:

make

The Makefile is specific to the compiler you are using. If your host compiler is not one of the standard compilers, you need to create a new make file specific to your compiler. You also need to create compiler specific versions of the following script files.

p6/p6.bat The single program compile and run batch file.

compile/compile.bat To compile a file with all inputs and outputs specified.

run/run.bat To run (interpret) the intermediate file with all inputs and outputs specified.

The good news is that all of the other scripts in the bin directory call the above scripts to do their work, so these are the only scripts you have to modify.

The reason you need to change these files is because pcom.pas uses the header file "prr" to output intermediate code, and pint.pas uses "prd" for input and "prd" for output. You need to find out how to connect these files in the program header to external named files.

For example, in IP Pascal, header files that don't bear a standard system name (like "input" and "output") are simply assigned in order from the command line. Thus, P6.bat is simply:

pcom %1.p6 < %1.pas

pint %1.p6 %1.out

Where %1 is the first parameter from the command line.

P6.bat lets the input and output from the running program go to the user terminal. Compile.bat and run.bat both specify all of the input, output, prd and prr files. The reason the second files are needed is so that the advanced automated tests can be run using batch files that aren't dependent on what compiler you are using.

If your compiler does nothing with header files at all, you will probally have to change the handling of the prd and prr files to get them connected to external files. To do this, search pcom and pint for "!!!" (three exclamation marks). This will appear in comments just before the declaration, reset and rewrite of these files.

## Evaluating an existing Pascal compiler using P6

If you plan to compile and run P6 using your compiler, you should evaluate your compiler's ISO 7185 Pascal compliance. Of course, simply compiling pcom.pas and pint.pas is one way to achieve that. But since this package gives you the ability to fully evaluate your compiler, I would suggest you use it.

First, you need to determine if your compiler has a ISO 7185 Pascal compliance option and turn it on if needed. I say "if needed", because some compilers actually change their behavior with the option enabled, and thus it is not possible to compile and run standard Pascal programs unless the option is on (a very unfortunate property of a Pascal implementation).

Within ISO 7185 Pascal, there are two characteristics of an implementation that could cause P6 to not compile, even if the implementation otherwise completely complies with the standard:

1. Conflict with extended keywords.
2. Character formats.

The first concerns an implementation that defines a new keyword conflicting with an identifier used in P6. For example, if your compiler has an extended keyword "variant", this would cause pcom.pas not to compile: it uses that as an identifier. Ideally, the ISO 7185 Pascal option should turn off such extended

keywords, but you may have to invoke another option to do this. Such extended keywords are allowed by the ISO 7185 Pascal standard.

The second is simply that the character set in use is not specified by the ISO 7185 Pascal standard. This is rarely an issue now, because virtually all implementations are based on either ISO 8859-1 (or ASCII), or are based on a character set that contains ISO 8859-1 as a base standard (both ISO 8859 and Unicode do this).

It is also possible that an implementation may define special character formats. For example, the commonly implemented character force sequences are a special format:

'this is a string\n'

This is valid, since ISO 7185 Pascal does not specify the exact format of strings.

Fortunately, P6 does not contain nor need force sequences, so this will not cause problems.

Besides compiling pcom.pas and pint.pas, I strongly recommend you run and check at least ISO7185pat.pas. This is a fairly comprehensive test of ISO 7185 Pascal compliance.

If you wish to run the entire compliance test on your compiler, you simply need to change or create a version of compile.bat and run.bat for your implementation that operate with your compiler. Then you can run the regression test ***without*** the self compile features (cpcoms and cpints).

Finally, building P6, and then running it through a full regression is itself a good final test of ISO 7185 compliance. It does not substitute for direct testing of your compiler. P6 could well run correctly even if your compiler is not fully ISO 7185 Pascal compliant!

For further details concerning the ISO 7185 tests, see “Testing P6”.

## Notes on using existing compilers

### GPC

GPC (GNU Pascal Compiler) is used in the following version:

GNU Pascal version 20070904, based on gcc-4.1.3 20080704 (prerelease) (Ubuntu 2.1-4.1.2-27ubuntu2).

Copyright (C) 1987-2006 Free Software Foundation, Inc.

I have had several difficulties with other versions of GPC, which give errors on standard ISO 7185 source, or crash, or other difficulties. The GPC developers announced they were halting development on GPC in the gpc mailing list. Please see their web page:

[http://www.gnu-pascal.de](http://www.gnu-pascal.de/)

For any further information.

The main difficulty with GPC vis-a-vie P6 is that testing of the GPC compiler for ISO 7185 compatability was not regularly done on GPC releases. Thus, otherwise working GPC releases were not able to compile and run standard ISO 7185 source code.

Because of this, I can only recommend the above version of GPC be used, which compiles and runs P6 error free.

In addition, please be aware that I have not run the GPC compiler, including the above version, through a current ISO 7185 compliance test such as appears here. My only concern is that GPC be able to complile and run P6, and that the resulting P6 runs the compliance tests. I leave it for others to run full compliance for GPC itself.

#### GPC for mingw and Windows

Mingw (Minimal GNU for Windows) is a different port of the GNU catalog for windows that runs directly on windows. That is, each binary is statically linked with its support library, and it is designed to work with windows directly.

Mingw does not come natively with GPC installed (or much else). I recommend you also pick up the MSYS package for mingw, which is a series of GNU programs that are compiled to run in the windows environment using Mingw.

To get the mingw distribution of GPC, follow the steps:

1. Go to the website:

<http://www.gnu-pascal.de/binary/mingw32/>

And download and install:

[gpc-20070904-with-gcc.i386-pc-mingw32.tar.gz](http://gnu-pascal.de/contrib/chief/win32/cygwin/gpc-20070904-with-gcc.i686-pc-cygwin.tar.gz)

(4.4mb, gpc-20070904, based on gcc-3.4.5, with gcc-3.4.5 support files)

1. After installing this package in an appropriate directory, say c:\gpc, modify your path to include c:\gpc\usr\bin directory in the path.

To reiterate the steps that follow:

> configure gpc Configure for GPC compiler.

> make Build the P6 binaries.

> regress Run the regression suites to check the P6 compiler.

This solution was tried and works for the 32 bit version of GPC under windows. I was unable to locate a 64 bit version of GPC under Mingw/Windows.

#### GPC for Linux

A prebuilt package of 64 bit GPC for linux can be found here:

<http://rpm.pbone.net/index.php3/stat/4/idpl/33822565/dir/scientific_linux_other/com/gpc-20070904-185.1.x86_64.rpm.html>

Download and install that.

Now add the following line to your startup file, usually .bash\_aliases:

LIBRARY\_PATH="/usr/lib/x86\_64-linux-gnu:/usr/lib/gcc/x86\_64-linux-gnu/7:/usr/lib/x86\_64-linux-gnu"  
export LIBRARY\_PATH

This was done with gcc version 7 in mind. You will need to substitute the current version for the "7" above for other versions.

I recommend you match the GPC bit width to the version of Linux you are using. In this case, it is a 64 bit compiler for 64 bit Linux. It is possible to set up a cross compile, but that is more (perhaps much more) difficult.

Run the usual:

. setpath  
./configure  
make

# pgen – AMD64 code generator

Pgen, specifically pgen\_gcc\_amd64, is a code generator for 64 bit AMD/Intel CPUs. It is also dependent on gcc/gas (the GNU C compiler and GNU assembler for AMD64 cpus[[7]](#footnote-8)). pgen is executed as:

> pgen <input file> <output file>

Pgen translates Pascal-P6 intermediate files to gas assembly files, which are then assembled, along with C language support files, into target programs. An example run is:

> pcom hello.pas hello.p6

> pgen hello.p6 hello.s

> gcc source/psystem.c source/psystem\_asm.s main.s hello.s -o hello -lm

This sequence is given in the script file p6\_pgen, which has the form:

> p6\_pgen <input file> [<input file>]…

The components of a pgen deck are:

psystem.c The C runtime support library for Pascal-P6

psystem\_asm.s The assembly language runtime support library for Pascal-P6

main.s The startup card for Pascal-P6, defines the location of “main”.

hello.s The assembly code generated by pgen.

hello The output executable file.

-lm Links the C math library.

## Calling convention

Pascal-P6 uses a modified version of the System V AMD64 ABI register based calling convention. In this convention the following register assignments are used:

Integer/pointer arguments 1-6 RDI, RSI, RDX, RCX, R8, R9

Floating point arguments 1-16 XMM0-XMM7

Return value RAX,RDX

All parameters are also passed on the stack. The register arguments are copies of the stacked arguments. The caller is responsible for cleanup.

The Pascal-P6 calling convention is different from the System V AMD64 ABI register based calling convention in that the first parameter at top of stack is the last parameter of the procedure or function, that is, parameters on the stack are pushed from left to right, instead of the C based calling convention of parameters stacked from right to left, so that the first parameter on stack is the first parameter of the function. The reason for this is that C supports so called “variadic functions”, where the parameters can be any number. Pascal does not.

Besides floating point, integers and simple pointers, Pascal-P6 supports so called “wide pointers”, which are arrays with templates or lengths following. If passed as a parameter, such pointers take two registers in a row, such as:

type a = array of integer;

procedure x(y: a);

Would take regsisters:

rdi Address of array y.

rsi Length of array y.

If a wide pointer is returned from a function as a result, it would occupy the registers:

type a = array of integer;

b = ^a;

function x: b;

rax Address of array returned.

rdx Length of array returned.

Pascal-P6 aligns the stack to 64 bits/16 bytes.

## Calling C from Pascaline

When using from 1 to 6 parameters, the calling conventions of C and Pascaline are compatible. The parameters and return values will line up correctly:

Pascal

procedure x(a, b: integer; r: real);

C

void x(int a, int b, double r);

Pascal-P6 always treats arrays as if they were passed as containers. That is:

type s = array [1..10] of char;

procedure x(a: s);

Is caller equivalent to:

procedure x(a: string);

Meaning that the C external function must be ready for a length argument:

void x(char a[], int l);

This means the C external function always knows what the length of the array parameter is. This can aid greatly in forming calls from Pascaline to C.

Pascal-P6 does this for any container argument. If the container array has more than one dimension, it will be passed with a template:

Pascal

Type a = array of array of integer;

Procedure x(b: a);

C

int template = { 10, 20 };

Void x(integer b[][], int t[]);

The C function will get the template as a list of integers, one per “order” or number of dimensions in the array. Thus in this example, there are two dimensions and two sizes in the template list for the multidimensional array.

### No value parameters for structured parameters

Pascal-P6 does not pass value or copy parameters for structured parameters. In the procedure

Pascal

type a = array [1..10] of char;

Procedure x(b: a);

C

void x(char\* b);

In Pascal-P6 procedures/functions, this value parameter is done by having the callee make a local copy of the parameter. Thus it is up to the C routine to perform this function.

### Stack inversion

If the procedure or function called from Pascal to C has more than 6 parameters, the C function must get parameters past 6 off of the stack. For this, you must understand that the stack is both inverted from the usual C stack ordering, as well as the fact that ***all*** parameters appear on stack, including those found in registers. It is up to the callee to compensate for this.

### Translation of strings

In C, string data is often zero terminated. This tends to be deprecated these days, especially in system calls, because the convention can cause overflow problems. Pascaline always uses the convention of a string address with a length following. If what is wanted is a zero terminated string, a simple translation can be used:

void wrapmyfunction(char s[], int l)

{

char buffer[100];

char\* p;

p = buffer;

while (l--) do \*p++ = \*s++;

\*s = 0;

myfunction(buffer);

}

### Pointers an VAR parameters

Pascaline and C pointers are equivalent in that they both are addresses of a base type. However VAR parameters are not equivalent to pointer types as in C, nor can any object be pointed to at will as in C. Pascaline is a type secure language, and not all C methods can be used in Pascaline. This means that C interfaces have to be designed for more general use, and not use methods specific to C. Alternatively, wrappers can be used for translation.

### Module name coining

Pascal-P6 uses what is called (in Pascal-P6 terminology) “coining” to produce module names. Each name in an external module has it’s filename prepended along with a ‘\_’ character:

mymodule with symbol myfunction

becomes

mymodule\_myfunction

Thus to link external modules you must either create global functions with the filename coined as shown, or create a wrapper module that translates the names to that of the existing functions in C. Such a wrapper can also be used for translations as required, for example convertion to and from C “zero terminated” strings.

If the (fairly small) overhead of C function entry and exit sequences is not desired, it is possible to create an assembly file that just creates name aliases for functions that don’t need wrappers.

## Calling Pascaline from C

Calling from C to Pascaline works, but only to top level routines in Pascal. You must obey the module coining convention as above. If a routine that is nested is called from C, the result is a crash. In general, callbacks are a C convention that is better avoided in nested procedure/function languages such as Pascal.

## Pascal-P6 module stacking sequence

Pascal-P6 uses a different method of module startup and shutdown than C. First of all, a series of Pascaline modules are entered at the bottom or starting address:

main mod1 mod2 program

Since the program is started at the bottom and not the “main” function as in C, a “shim” is provided as main.s, which specifies the starting location of the program. Each module in Pascaline starts by performing its initialization, then calling the next module above it. So in the example mod1 calls mod2 and finally mod2 calls program. The program module executes the program function, and when it terminates, it returns to mod2, which executes its finalisation block, then returns to mod1, etc. Thus:

1. All “startup” blocks are executed in series before the program starts.

2. All “finalization” blocks are executed in reverse series after the program ends.

The modules should always be stacked in dependence order. That is, if mod2 uses mod1, it must appear in the sequence ***after*** mod1. Since the program uses both mod1 and mod2, it must appear after them both. This means that a module or program that uses another module, must not access the data for that module unless that module has had a chance to initialize itself.

Non-Pascal-P6 modules ***must not*** be inserted in the Pascal-P6 module chain. This will cause a crash, since Pascal-P6 modules will attempt to execute the non-Pascal-P6 module in the chain. The place for non-Pascal-P6 modules is before or after the entire Pascal-P6 module chain.

# Files in the Pascal-P6 package

Note: for script files, both a DOS/Windows (X.bat) and bash script (X) are provided. Their function is identical, one is for use with the DOS/Windows command shell, the other for bash shell.

configure.bat

configure Sets the current compiler to use to create P6 binaries.

## Directory: bin

compile

compile.bat Batch mode compile for P6. It takes all input and output from supplied files, and is used by all of the other testing scripts below. You will need to change this to fit your particular Pascal implementation.

\*\*\* You will need to change this to fit your particular Pascal system \*\*\*

It uses input and output from the terminal, so is a good way to run arbitrary programs.

cpcom

cpcom.bat Script file to compile the compiler side of P6 to binary on your computer.

\*\*\* You will need to change this to fit your particular Pascal system \*\*\*

cpcoms

cpcoms.bat Self compile, run and check the pcom.pas file. This batch file compiles com.pas, then runs it on the interpreter and self compiles it, and checks the intermediate files match.

cpint

cpint.bat Script file to compile the interpreter side of P6 to binary on your computer.

\*\*\* You will need to change this to fit your particular Pascal system \*\*\*

cpints

cpints.bat Self compile, run and check the pint.pas file. This batch file compiles pint.pas and iso7185pat.pas, then runs pint on itself and then runs iso7185pat.pas, and checks the result file.

diffnole

diffnole.bat Runs a diff, but ignoring line endings (DOS/Windows vs. Unix).

doseol

doseol.bat Fixes the line endings on text files to match the DOS/Windows convention, CRLF.

fixeol

fixeol.bat Arranges the line endings on bash scripts to be Unix, and those of the DOS/Windows scripts to be DOS/Windows line endings. This is required because the editors on the respective systems insert their own line endings according to system, and this can cause problems when they are run on a different system.

flip.c C program to replace the local version of “flip”, the Unix line ending fixup tool. It is provided in source form here because not all Unix installations have it (for example MAC OS X didn’t have it). This allows you to compile it yourself for your target system.

make\_flip

make\_flip.bat A script to compile deoln and ueoln and create a flip script for Unix. This is used to replace the “flip” program if required.

P6

P6.bat A batch file that compiles and runs a single Pascal program. You will need to change this to fit your particular Pascal implementation. It uses input and output from the terminal, so it is a good way to run arbitrary programs.

\*\*\* You will need to change this to fit your particular Pascal system \*\*\*

It uses input and output from the terminal, so is a good way to run arbitrary programs.

pcom

pcom.exe The IP Pascal compiled pcom binary for Windows/Unix. See comments in 7.2 “Compiling and running Pascal programs with P” for how to use this. All of the supplied batch files are customized for this version.

pcom.pas The compiler source in Pascal.

pint.exe The IP Pascal compiled pint binary for Windows. See comments in 7.2 “Compiling and running Pascal programs with P” for how to use this. All of the supplied batch files are customized for this version.

pint.pas The interpreter source in Pascal

prtprt.bat Batch file, takes all of the rejection test error files and concatenates them to standard\_tests/iso7185prt.lst.

readme.txt Brief introduction to the project, it points to this document now.

regress

regress.bat The regression test simply runs all of the possible tests through P6. It is usually run after a new compile of P6, or any changes made to P6.

run

run.bat Batch mode run for P6. It takes all input and output from supplied files, and is used by all of the other testing scripts below. You will need to change this to fit your particular Pascal implementation.

\*\*\* You will need to change this to fit your particular Pascal system \*\*\*

It uses input and output from the terminal, so is a good way to run arbitrary programs.

testpascals

testpascals.bat Runs a compile and check on pascals.pas, the Pascal subset interpreter created by Niklaus Wirth.

testprog

testprog.bat An automated testing batch file. Runs a given program with the input file, delivering an output file, then compares to a reference file.

Testprog is used to test the following program files for P6: hello, roman, match, startrek, basics and iso7185pat.

the\_P6\_compiler.doc

the\_P6\_compiler.html

unixeol

unixeol.bat Fixes the line endings on text files to match the Unix convention, LF.

ZipP6.bat Creates a zipfile for the entire P6 project. This is used to create the releases available on the P6 web site.

## Directory: doc

iso7185rules.html A description of the ISO 7185 Pascal language.

news.txt Contains various information about the current release.

the\_P6\_compiler.docx This document in word 2007 form.

The\_Programming\_Language\_Pascal\_1973.pdf

Niklaus Wirth's description of the Pascal language, the last version to come from ETH. This is the equivalent of the "Report", from "Pascal user's manual and report [Jensen and Wirth].

todo.txt Contain a list of "to do" items in P6.

## Directory: gpc

This directory contains scripts specifically modified for GPC.

compile

compile.bat The GPC specific version of the compile script.

cpcom

cpcom.bat The GPC specific version of the compile compiler script.

cpint

cpint.bat The GPC specific version of the compile interpreter script.

P6

P6.bat The GPC specific version of the P6 script.

run

run.bat The GPC specific version of the run script.

## Directory: gpc/linux\_X86

pcom

pint Contains binaries compiled by GPC for Linux/Ubuntu

## Directory: mac\_X86

A placeholder for Mac OS X binaries.

## Directory: gpc/standard\_tests

iso7185pat.cmp Contains the compare file for iso7185pat for gpc.

iso7185pats.cmp Contains the compare file for iso7185pats for gpc.

## Directory: gpc/windows\_X86

pcom.exe

pint.exe Contains binaries compiled by GPC for Windows.

## Directory: ip\_pascal

This directory contains scripts specifically modified for IP Pascal.

compile

compile.bat The IP Pascal specific version of the compile script.

cpcom

cpcom.bat The IP Pascal specific version of the compile compiler script.

cpint

cpint.bat The IP Pascal specific version of the compile interpreter script.

P6

P6.bat The IP Pascal specific version of the P6 script.

run.bat

run The IP Pascal specific version of the run script.

## Directory: ip\_pascal/standard\_tests

iso7185pat.cmp Contains the compare file for iso7185pat for IP Pascal.

iso7185pats.cmp Contains the compare file for iso7185pats for IP Pascal.

## Directory: ip\_pascal/windows\_X86

pcom.exe

pint.exe Contains binaries compiled by IP Pascal for Windows

## Subdirectory: sample\_programs

basics.pas A tiny basic interpreter in Pascal.

basics.inp Input test file for basics. In fact, it is a basic verion of "match" above.

basics.cmp Compare file for basics.

hello.pas One of several test programs used to prove the P6 system. This is the standard "hello, world" program.

hello.inp Input to hello for automated testing.

hello.cmp Hello compare file for automated testing.

match.bas The basic version of the match game.

match.pas A game, place "match" a number game.

match.cmp Compare file for match automated testing.

match.inp Input file for match automated testing.

pascals.pas Niklaus Wirth’s Pascal-s subset interpreter..

pascals.cmp Compare file for pascals automated testing.

pascals.inp Input file for pascals automated testing.

roman.pas A slightly more complex test program, prints roman numerals. From Niklaus Wirth's "User Manual and Report".

roman.inp Input file for roman automated testing.

roman.cmp Compare file for roman automated testing.

startrek.pas The startrek game.

startrek.inp Startrek input file.

startrek.cmp Startrek compare file.

## Directory: standard\_tests

iso7185pat.cmp Contains the output from the PAT file with the IP Pascal compiled P6 executables above.

iso7185pat.inp The input file for the Pascal acceptance test.

iso7185pat.pas The Pascal Acceptance Test. This is a single Pascal source that tests how well a given Pascal implementation obeys ISO 7185 Pascal. It can be used on P6 or any other Pascal implementation.

iso7185pats.cmp Contains the output from the PAT file resulting from the cpints run. This is slightly different than the normal run.

iso7185prt.bat Compiles the Pascal rejection tests.

iso7185prtXXXX.cmp Comparision file for pascal rejection test. XXXX is a four digit number. See the rejection tests text for information.

Iso7185prtXXXX.inp Contains the input file for the Pascal rejection test XXXX.

Iso7185prtXXXX.pas Contains the source file for the Pascal rejection test XXXX.

# The intermediate language

The intermediate language for Pascal-P6 is in the form of an assembly language file for the abstract machine called the P-Machine. Each instruction in the intermediate specifies the semantics of operations in the language.

The intermediate is input by both the interpreter, pint, and the compiler, pgen\_x\_x. If the source program consists of more than one module (program or module), then the intermediate code from each module run is concatenated in definition order. That is, each module that uses definitions from another module must follow that module in the concatenated desk. Then the resulting combined deck is input to pint. pint will handle linking the modules together.The deck must end with a “program” module.

Modules in Pascal-P6 “stack”, meaning that the first module initializes itself and calls the next one after it. This continues until the program module is executed, then control flow goes backwards down the stack again executing finalization sections until the first module is reached.

If the compiler pgen\_x\_x is used, only one module at a time is input. Linking modules together is done after the output of the compiler is given as input to the linker. The module “stacking” mechanisim of pint is simulated by using the constructor and destructor attributes of gcc. See the section on pgen for more information.

## Format of intermediate

The intermediate has the format:

<main code>

<start code>

<further prd input>

The main section contains all of the generated code, except for a startup section:

mst 0

cup 0 l 3

stp

The main section is assembled past the startup code, then the assembly location is restarted and the startup section is assembled, placed under the main code.

After both the main code and start code sections, the contents of the prd file are not read. The interpreted code can keep reading from the prd file at this point. This means that input for the prd file as used by the interpreted program can be concatenated to the intermediate file. This feature is used for the self compile and run.

## Intermediate line format

For each line in the intermediate, the first character indicates:

|  |  |  |
| --- | --- | --- |
| **Character** | **Parameter(s)** | **Description** |
| **!** | <arbitrary characters> | Indicates a comment, the rest of the line is discarded. |
| **l** | <number> | A label. Used to establish jump locations in the code. |
| **q** |  | Marks the end of the main code or startup code section. |
| **(blank)** |  | An intermediate instruction. |
| **:** | <number> | A source line marker. |
| **o** | <options> | Passes the settings of all options to the backend. |
| **g** | <number> | Passes the total size of globals to the back end. |
| **v** | l<label> <size>  <logical variant numbers> | Passes a “logical variant table” to the backend. Size indicates the number of logical variant numbers that will appear. |
| **f** | f <number> | Number of source code errors found by pcom. |
| **b** | b <type> | Block start. |
| **e** | e <type> | Block end. |
| **s** | s <name> <offset> <digest> | Symbol. |
| **t** | <label> <number of dimentions> <dimension1> [<dimentionn>]… | Fixed array template. |
| **n** | <label> <length> | Constant table begin. |
| **c** | <type> <value> | Constant table entry. |
| **x** |  | Constant table end. |

### Comments

A comment appears as:

!<any text>

Comments are used to generate any descriptive text in the intermediate. They are also used to output a marker every 10 intermediate instructions, of where the “logical program count” or index of instructions, is currently located in the intermediate with a marker of the form:

i n

Where n is the logical program count.

### Label

Label lines are of the format:

l m.n[= val]

The first part of the label is the module to which the label belongs, which is a full name [‘a’..’z’, ‘A’..’Z’, ‘0’..’9’]. The label number is either a numeric [‘0’..’9’] or starts with with an alphanumeric label leader [‘a’..’z’, ‘A’..’Z’, ‘0’..’9’]. If the label is numeric, it is a “near” label and only refers to the current module being compiled. If the label is alphanumeric, it is a “far” label, and can either be in the current module or an external module. If “=” follows, the instruction address of the label appears, otherwise it is set to the current instruction being processed (the pc).

The logical label number is a value from 0 to n. The value can be anything, and it is used both to define parameters as well as addresses. When used in an instruction, the assembler is capable of processing forward references to labels not yet defined in the assembly.

### End of section

Marks the end of a startup or main code section:

q

### Intermediate code

Intermediate code lines are introduced by a blank character in the first collumn.

### Source lines

A source line marker appears as:

: n [<source line>]

The number given is the source line number, 1 to N, in the source being compiled. The compiler also has an option to pass the entire source line, so that the intermediate is annotated with the original source. If the source lines are included, it appears after the line number and a space.

### Options

Options appear as:

o a+b-c+…z+

Options are a series of pairs of single characters and either ‘+’ or ‘-’. The possible options are listed in Compiler options 7.3.

### Global space count

The global space contains the total space in globals used by the current block. It is used to merge multiple, separately compiled program blocks for simulation. It is not used in compilation. It is of the form:

g n

Where n is the total size of globals in the current block.

### Logical variant table

When processing tag field values for variant records, Pascal-P6 converts tag field values to and from “logical variant numbers”, which are sequental numbers from 1 to n that denote variants in tables. For example:

type

select = 1..3;

r = record case s: select of 1, 2: (c: char); 3: (i: integer) end;

The reason for this is that variants are unique, but tag values are not. More than one tag value can correpond to a given variant. Each variant record definition generates a logical variant to tag value table, and each table is a line in the intermediate. These lines are of the form:

v <label> <length> <value>[<value]…

The label gives the address of the lookup table. The length gives the number of entries in the table. The values give the equivalent logical variant values for each tagfield value. An example table and the source code that produces it is:

var r: record case q: s of

1, 2: (c: char);

3: (i: integer);

end;

v l test.7 4 0 1 1 2

This table is four entries long, and translates tag field value 0 to 0 (not used), tagfield values 1 and 2 to logical variant 1, and tagfield value 3 to logical variant 2.

The table is always 0 to n. If the tagfield values only cover a subset of that range, those entries are left empty (0). The logical variants themselves are numbered 1 to n.

### Source errors

The faults or errors line gives the number of source file errors that pcom generated when parsing the program. It is of the form:

f <errors>

Where errors gives the total number of errors encontered. The purpose of the fault line is to give pint/pgen the option to reject the intermediate if the source contains errors.

Indicates how many source errors were found in the compile:

f 42

Used to determine if the program should be run.

### Block start and end

The block start and end marker show where program, module, procedure and function sections begin and end. They are of the form:

b t <name>

e t

A block begin contains the type of block, and the name of the block. The type of block is:

p Program block.

m Module block.

r Procedure.

f Function.

Blocks can be nested to any level, but program and module blocks do not nest in other blocks.

### Symbols

Symbols carry the names of source constants, variables, procedures, etc. They are of the form:

s <name> <storage> <offset> <type digest>

The name is the source name. The storage type is one of:

g Global.

p Parameter.

l Local.

The offset is the offset address of the symbol, ie, the start of globals, parameters or locals. The type digest carries the type of the symbol and is used in the debugger. This is referred to in Pascal-P6 as “type spagetti”, and is a recursive description of the symbol’s type. The base types are denoted by the sequences:

i Integer

b boolean

c Char

n Real

x(id[,id]…) Enumerated type.

x(min,max)<type> Subrange

Enumerated types are types like (one, two, three), and each id appears. So an example is:

For the type:

(one, two, three)

The digest form is:

x(one,two,three)

Subrange is similar:

For the type:

1..10

The digest form is:

x(1,10)i

Note that it is terminated by it’s base type. Note also that pint/pgen can tell if an enumerated type is meant or a subrange because its either a label or a number. In a digest nothing is symbolic. If the source was:

const one = 1;

const two = 2;

And the declaration was:

one..two

The digest would be:

x(1,2)i

This applies to characters as well. If the source was:

set of '0'..'9'

The digest of the set’s base type would be:

x(48,57)c

In other words, a subrange starting with the ASCII code for ‘0’ (in decimal), and ending with the ASCII code for ‘9’, and the subrange base type is char.

The type digest is terminated by one of the above sequences, ie., these are the ultimate base types. The structured types then form collections of those types:

p <typ> Pointer

s <typ> Set

a <inx> <base> Array

v <base> Variable length array

r (<fld>[,<fld>]) Record

f <typ> File

e Exception

Each of the indicator characters is unique. For example reads are denoted by ‘n’ because ‘r’ is a record and ‘f’ is a file. Each of the structured type indicators is followed by its base type or more complex descriptions of it’s substructure. For example:

fi

is “file of integer” with no space. Similarly:

sx(1,9)i

Means:

set of 1..9

Arrays have both an index type and a base type. For example:

array [1..10] of integer

Appears in digest form:

ax(1,10)i

Variable arrays have no index type, since it is always integer.

Records are the most complex types of digests. ‘r’ is followed by a list of record fields in paranthesis. Each field is of the form:

<name>:<offset>:<type>

The name is the field name. The offset is the net offset from the start of the record in bytes. The type is the field type. Thus:

record a: integer; b: char end

Would be in digest form:

r(a:0:i,b:8:c)

If a variant appears in the record, it has the special notation:

<tag name>:<offset>:type(<tag constant>(<field>)[<tag constant>(<field>)]…

So the record:

r: record a: integer; b: char

case c: boolean of

true: (d: real);

false: (e: boolean);

end;

In digest form:

r(a:0:i,b:8:c,c:9:b(0(e:12:b)1(d:12:n)))

In short, type digests can get pretty complex. They are not meant for humans to read.

### Template tables

Templates are sent to pint/pgen by a line of the form:

t <label> <number of dimentions> <dimension1> [<dimentionn>]…

The label is the address of the template. Following that is the number of dimensions in the table, then all of the dimentions in the table.

See the section 15.3 “Templates” for the format of templates.

### Constant tables

Constant tables are a means for fixed types to be represented in the intermediate. Any number of tables can appear in the intermediate. Each table has a start and end marker, and any number of constant entries can appear within the table.

The format of a start marker is:

n <label> <length>

The label gives the address the constant is placed at. The length gives the total size, in bytes, of the constant data.

The constants are introduced by:

c <type> <value>

One line appears for each constant. The type of the constants are:

i Integer

r Real

p Set (powerset)

s String

c Char

b Boolean

x Byte

For all except string, the value is a simple integer constant. Strings appear in single quotes, like ‘sandy’. Fixed constants can be structured, but that is not marked in the intermediate. The constant table is just a collection of values.

The constant section is terminated by a line:

x

An example of a fixed type and the constant table it generates is:

fixed a: record a: integer; b: char end = record 1, 'c' end;

n l test.12 9

c i 1

c c 99

x

Here the constant table length is 9, for an 8 byte integer and a 1 byte character. This is followed by the two values, and the end marker.

## Templates

Pascal-P6 implements variable length arrays with so called “step” templates. There are several intermediate instructions dealing with step templates. The basis of a step template is a “tagged pointer” of the form:

<base pointer><length or template pointer>

The base pointer is type address. The length is type integer. The template pointer is type address. If the array that the pointer indexes is a single dimension or vector, the tag is a length. If the index is 2 or more, then the tag is a pointer to a template table. The template table is a list of array dimensions in major to minor order. So the template for a complex array is:

array 10, 20, 5 of …

10

20

5

What makes the system “step” templates is that they advance as array slices are taken. Thus:

a: array 10, 20, 5 of ...;

a[6]

Yields a tagged pointer:

<base address of a>

<pointer to template at 20>

As an array reference is advanced from major to minor index, it goes through the template table, thus 10, 20 and 5. When it gets to 5, it performs what is called “stepping off”. Because the template is at its end, it picks up the final dimension from the template, and the tagged pointer becomes:

<base address of a>

5

All single dimension arrays are what are called “simple” variable arrays, that is, their references consist of a base pointer/length pair. Complex arrays that use templates are converted to simple arrays if the slice taken yeilds a single dimention array.

All variable dimension arrays are dynamically allocated, although pint/pgen often hides that fact by using stack allocation, which is automatically freed when the routine exits. They can also be returned from functions on the stack.

The location of the template also changes with the type of allocation. For a dynamically allocated array, the template appears at the start of the dynamic allocation, and the array data appears after that. For global or stack allocated containers, the address of the variable appears before the template, followed by the template. The data for the array is either allocated dynamically for global arrays, or on stack for procedure/function arrays.

## Variant record tables

Pascal-P6 can verify that variant records allocated dynamically obey rules such as not accessing inactive variants, not accessing variants that have references, etc. To do this Pascal-P6 must know what tags were used to allocate a variant record. This table must be created at runtime when the variant record is allocated.

In Pascal-P6 this is done by allocating the table alongside the record data itself. When the record is allocated, space is added for the table and the record itself is placed ***after*** the table. The base address of the record is returned to the client program of the record address only. pint/pgen finds the table by looking in “negative space” below the variant record. Since the runtime does not know the size of the table, the length of the table is stored above the table, and pint/pgen can simply look one machine word below the variant record to determine both the size of the table as well as how to find the start of the table.

The format of the tag list is

record address ->

Number of tags

Tag N

…

Tag 3

Tag 2

Tag 1

Note: pint adds a bias of the size of an address+1 to the number of tags. This is done to make sure that an allocation that is accessed after it has been freed is not mistaken for a tag count. See the workings of pint for more information.

## Undefined accesses

Pascal-P6 can optionally keep track of accesses to undefined variables. This requires a database that is 1/8 the size of memory, one bit for each byte in memory. For this to work, the database must cover all of global variables, stack and heap.

For the most part, this option is transparent to both the program and to P-machine instructions. However, there are a few instructions that directly manipulate this database. These handle cases such as returning record variants that are no longer active to undefined status, marking the index variable of a “for” loop as undefined after the for loop has ended, etc.

## Code strips

When Pascal-P6 compiles parameterized variable declaractions, there is a catch 22. It must generate code for simple expressions, but it does not have the information it needs to start the procedure or function the declaration is a part of. The compiler handles this by generating “code strips”, or small sections of code to process the declaration expression(s). Then these are called by the procedure or function startup code, using a simple form of call and return that does no framing, and has no parameters or return value. Code strips are called and returned from by special instructions.

## Exception frames

Exception frames are a series of linked frames placed on the stack that set up a chain of exception handlers. They are machine dependent, but pint uses the format (in stack top to bottom order):

vector The current exception variable as thrown.

expmrk The mark address of the exception handler.

expstk The stack address of the exception handler.

expadr The address of the current/topmost exception handler.

What the exception frame does is give sufficient information to recover from an error, reset the stack and mark, then execute the exception handler. The exception handler matches a series of exception variables to the onstack exception address, including a “wild card” or any exception handler, then if not found, climbs the chain of exception handlers until one is found.

## Operator overload calling frames

When Pascal-P6 performs an operator overload, it does not know until the expression is parsed what type of call is going to be peformed, built in or overload. Thus at the time of the overload, it has not set up a calling frame for the overload. It gets around this by setting up a calling frame using the mst instruction, then uses the cpp instruction to “hoist” or copy each of the parameters to the overload function from their place over the mark down to the bottom of the stack. After the call is made, the result is placed back above the frame by a cpr instruction.

## Intermediate instruction set

What follows is a complete listing of all instructions, parameters, and numeric equivalences in the interpreter instruction set. The following instruction endings are common and indicate the type of the operation:

i Integer (2/4/8 bytes)

x Byte (1 byte)

r Real (8 bytes)

s Set (32 bytes)

b Boolean (1 byte)

a Address (2/4/8 bytes)

m Memory (or memory block)

c Character (1 byte)

Note that all boolean, character and byte values are extended to 4 byte integers when loaded to the stack. Reals and sets are loaded as whole entities, 8 and 32 bytes respectively.

Note that there will often not be a type indicator if there is only one type operated on by the instruction.

The operation codes are from 0 to 255 or $00 to $ff, fitting in one unsigned byte. The format of the opcodes and operands is:

Op [p] [q [q1]]

The p parameter, if it is exists, is always an unsigned byte. The q parameter, if it exists, is either one or N bytes long, where N is the word size of the machine. If q1 exists, q is always N bytes long. q1 is always N bytes long. These will be refered to below as q8 and qn (8 bit and B bits). The endian nature of words will depend on the compiler used for P6.

This table shows the byte and bit length of words in Pascal-P6:

Word size N (bytes) Bit size B

2 16

4 32

8 64

Each instruction is listed first in the form it appears in the intermediate assembly form. The second is the format the opcode will take in interpreter store. The stack contents are listed in order on the stack, so the first parameter is the top of stack, the second parameter is the second on stack, etc. Both the contents of the stack before and after the operation are listed (stack in and stack out).

This list of instructions is in alphabetical order.

Instruction Opcode Stack in Stack out

abi 40 integer integer

abr 41 real real

Find absolute value of integer or real on top of stack.

Instruction Opcode Stack in Stack out

adi 28 integer integer integer

adr 29 real real real

Add the top two stack integer or real values and leave result on stack.

Instruction Opcode Stack in Stack out

and 43 boolean boolean boolean

Find logical ‘and’ of the top two Booleans on stack, and replace with result.

Instruction Opcode Stack in Stack out

apc lvl size 210 qn qn sadr stmp dadr dtmp

Assign pointer complex. Expects the number of levels in the array and the size of the base element in the instruction. The stack has the source address at top, followed by the source template address, followed by the destination address, followed by the destination template. Calculates the size of the complex array using the templates, then moves that many bytes from the source address to the destination address.

Instruction Opcode Stack in Stack out

aps size 178 addr int addr int

Assign simple pointer data. The size indicates the size of the base element. The tagged pointer data at stack top is assigned to the tagged pointer data at second on stack. Both tagged pointers are removed.

Instruction Opcode Stack in Stack out

bge eadr 207 eadr sadr madr vec

Begin exception frame. Establishes a new exception frame. Expects an exception handler address in the instruction, then saves a new exception frame consisting of the previous exception handler address, stack address, mark address, and a dummy exception vector, then sets up the current exception variables.

Instruction Opcode Stack in Stack out

brk 19

Break execution. Used by pint only. Stops execution and enters the debugger, if enabled.

Instruction Opcode Stack in Stack out

cal adr 21 qn raddr

Call initalizer code strip. The current address is pushed onto the stack, and the address of the specified location called.

Instruction Opcode Stack in Stack out

ccs lvl siz 223 dadr tadr dadr tadr

Copy complex container to stack. Expects the number of levels and the base size in the instruction. On stack are the data address and the template address/length. Finds the total size of the array and copies it onto the stack. Replaces the original data address and template address with the on stack version. This instruction is used while passing a complex container as a value parameter.

Instruction Opcode Stack in Stack out

chka low high 95 qn address address

chkb low high 98 qn boolean boolean

chkc low high 99 qn char char

chki low high 26 qn integer integer

chkx low high 199 qn integer integer

chks low high 97 qn

Bounds check. The q parameter contains the address of a pair of low and high bounds values, low followed by high, each one integer in size. The bounds pair is placed in the constant area during assembly, and the address placed into the instruction.

The value on top of the stack is verified to lie in the range low..high. If not, a runtime error results. The value is left on the stack.

If the instruction was chka, the low value will contain the code:

0 Pointer is not being dereferenced

1 Pointer is being dereferenced

Pointer values are considered valid if between 0..maxaddr, where maxaddr is the maximum address in the interpreter. If the dereference code is present, it indicates a check if the pointer is nil. This instruction is only applied to dynamic variable addresses, and thus other checks are possible.

Set values are checked to see if any elements outside the elements from low..high are set. If so, a runtime error results.

Instruction Opcode Stack in Stack out

chr 60 integer character

Find character from integer. Finds the character value of an integer on stack top. At present, this is a no-op.

Instruction Opcode Stack in Stack out

cip p 113 mp address

Call indirect procedure/function. The top of stack has the address of a mp/address pair pushed by lpa. The dl of the current mark is replaced by the mp, and the address replaces the current pc. The mp/ad address is removed from stack. The size of the parameters appears as p, which shows where to find the mark.

Instruction Opcode Stack in Stack out

cjp lval hval adr 8 qn qn qn val

Compare and jump. Expects a value on the stack to compare with. The instruction contains the low value, high value, and address to jump to. The value is checked to lie within the range from the low value to high value, inclusive, and the jump is taken if so. The value is removed from the stack.

Instruction Opcode Stack in Stack out

cke 188 boolean tagcst

Terminate active variant check, started by cks. Expects the “running boolean” value on stack top, followed by the current value of the tagfield. If the Boolean value is false, meaning that none of the constants matched the tagfield, a runtime error results.

Both the Boolean value and the tagfield value are removed from the stack.

Instruction Opcode Stack in Stack out

ckla low high 190 address address

Bounds check. This instruction is identical to the instruction chka, but indicates a pointer to a tagged record. At present this requires no special action over the normal processing of chka.

ckla an chka check if the pointer is dereferencing an allocation that has been freed. In the current implementation this check works on both tag listed and normal dynamic variables. Because this is dependent on the way the allocator is structured, ckla exists in case tag list pointers need special handling.

Instruction Opcode Stack in Stack out

cks 187 tagval boolean tagval

Start active variant check. The current value of the tag is on stack top. Pushes a false binary value onto the stack. This value is ‘or’ed with all of the following variant checks.

Instruction Opcode Stack in Stack out

ckvb val 179 qn boolean tagval boolean tagval

ckvc val 180 qn boolean tagval boolean tagval

ckvi val 175 qn boolean tagval boolean tagval

ckvx val 203 qn boolean tagval boolean tagval

Checks a tagfield for active variants. The “running Boolean” is expected at stack top, followed by the current tagfield value. The tagfield is compared to the constant val and the Boolean equality ‘or’ed into the running Boolean. Both are left on stack.

Instruction Opcode Stack in Stack out

cpc lvl 177 addr len addr len addr len addr len

Compare complex templates. Expects two tagged pointers on stack, which is an address followed by a template pointer. The number of levels in the templates are in the instruction. The two templates are compared, and an error generated if they don’t match. The two tagged pointers remain on stack.

Instruction Opcode Stack in Stack out

cpp poff psiz 239 qn qn par

Copy procedure parameter. Used to prepare operator overload parameters. Expects the offset to the parameter location and the parameter size in the instruction. The parameter is located above the mst frame and copied down into the stack. The offset is from the stack pointer plus the mark size. After the instruction, the parameter is left on stack. The size will depend on the parameter type, rounded up to the stack alignment.

Instruction Opcode Stack in Stack out

cpr rsiz roff 240 qn qn fres

Copy result. Expects the size of the function result and the offset to its location in the frame in the instruction. Used to copy back the result of an overloaded function to its frame. At the end of instruction, the original parameters are discarded and only the function result remains.

Instruction Opcode Stack in Stack out

cps 176 addr len addr len addr len addr len

Compare simple templates. Expects two tagged pointers on stack, which is an address followed by a length. The two lengths are compared, and an error generated if they don’t match. The two tagged pointers remain on stack.

Instruction Opcode Stack in Stack out

csp rout 15 q8 Per call Per call

Call system routine. The routine number rout indicates the routine to call, from 0 to 255 (or $00 to $ff). Note that the main difference between system and user procedures is that system calls have no framing information.

Instruction Opcode Stack in Stack out

cta off lvl lvn 191 qn qn qn newtagval tagaddr newtagval tagaddr

Check tag assignment for dynamically allocated record. Expects the value to assign to the tagfield on stack top, followed by the address of the tagfield as second on stack. Dynamic records containing tagfields are allocated by a special procedure that creates a tagfield constant list with the constants that were used to allocate the record with variants. The cta instruction looks at the list to see if the given tagfield appears in that list, indicating that the tagfield is fixed and cannot be changed. If such a list entry exists and does not match the tagfield assignment, a runtime error results. The instruction contains the offset from the start of the record to the tagfield, at what nesting level the tagfield exists, and the location of the logical variant table. If uses the tagfield offset to find the start of record and look for the list, and the nesting level to find the exact tag constant, if it exists. Leaves both values on stack.

Instruction Opcode Stack in Stack out

cup p l addr 12 p8 qn

Call user procedure. The instruction contains the number of bytes in the parameter section, and the address of the procedure to call. The mp or mark pointer is set to the location of the stack mark record established by the mst instruction, then the return address is placed into the mark record. The specified location is then jumped to.

The assembly code represents the address as a forward referenced label. This is defined later in a label statement and back referenced by the assembler.

Instruction Opcode Stack in Stack out

cuv p l addr 27 p8 qn

Call user virtual procedure/function vector. The instruction contains the number of bytes in the parameter section, and the address of the vector to the procedure to call. The mp or mark pointer is set to the location of the stack mark record established by the mst instruction, then the return address is placed into the mark record. The vector to the specified location is then jumped to.

Instruction Opcode Stack in Stack out

cvbi off size lvn 100 qn qn qn tagval tagaddr tagval tagaddr

cvbx off size lvn 100 qn qn qn tagval tagaddr tagval tagaddr

cvbb off size lvn 100 qn qn qn tagval tagaddr tagval tagaddr

cvbc off size lvn 100 qn qn qn tagval tagaddr tagval tagaddr

Check change to tagfield in variable referenced variant. Expects a new setting for a tagfield at stack top, and the address of the tagfield below that. If the tagfield was defined, and the new tag value is different than the old tag value, the variant area controlled by the tagfield is checked if it overlaps an outstanding variable reference. If it does, an error results, since ISO 7185 specifies variants cannot be changed with an outstanding variable reference. The off instruction field constains the offset from the tagfield to the base of the variant. The size instruction field contains the size of the variant. The logical variant table contains a label of the tagfield value to logical variant lookup table for the variant record in use. The offset is applied to the address of the tagfield, and the variant is checked for variable reference. Note that the size of the variant is the maximum size of all possible subvariants. The stack is left the same way it was found.

The stack is unchanged during this operation.

Instruction Opcode Stack in Stack out

cxc lvl siz 212 inx dadr tadr dadr tadr

Complex container index. Expects the size of the base type in the instruction. The index is at top of stack, followed by the address of the array, then the address of the template. The index is checked to lie within the array data, then multiplied by the base size and all levels of array, from the present level down to the last, then added to the base address of the array to find the element. The template address is moved to the next level and placed on stack, followed by the element address at stack top. The index, array address and template address are removed, and replaced by the new element address and template address. Note that this instruction is only used to advance from one complex container to an enclosed one. The cxs instruction is always used on the last level, because it picks up the last template entry and carries the length with the pointer instead of the template address (this is a so called “pick up template”).

Instruction Opcode Stack in Stack out

cxs siz 211 qn inx adr len adr

Simple container index. Expects the size of the base type in the instruction. The index is at top of stack, followed by the address of the array, then the length. The index is checked to lie within the array data, then multiplied by the base size and added to the array address to find the element address. The index, array address and len are removed from stack and replaced by the element address.

Instruction Opcode Stack in Stack out

decb cnt 103 qn boolean boolean

decc cnt 104 qn character character

deci cnt 57 qn integer integer

decx cnt 202 qn integer integer

The unsigned cnt parameter is subtracted from the value at stack top.

Instruction Opcode Stack in Stack out

dif 45 set set set

Find set difference, or s1-s2. The members of the set on stack top are removed from the set that is second on the stack, and the top set on stack is removed.

Instruction Opcode Stack in Stack out

dmp cnt 117 qn value

The unsigned cnt parameter is subtracted from the current stack pointer. The effect is to “dump” or remove the topmost cnt data from the stack.

Instruction Opcode Stack in Stack out

dupa 182 address address address

dupb 185 boolean boolean boolean

dupc 186 character character character

dupi 181 integer integer integer

dupr 183 real real real

dups 184 set set set

Duplicate stack top. The top value on the stack is copied to a new stack top value according to type.

Instruction Opcode Stack in Stack out

dvi 53 integer integer integer

dvr 54 real real real

Divide. The value second on stack is divided by the value first on stack. The division is done in integer or real according to type.

Instruction Opcode Stack in Stack out

ede 208 eadr sadr madr vec

End exception frame. Expects an exception handler address, stack address, mark address, and vector variable on stack. Removes all and continues. This instruction is used after the exception has been successfully handled.

Instruction Opcode Stack in Stack out

ente l size 173 qn

Sets the maximum extent of stack use. The size constant in the instruction gives a compiler calculated size that is the maximum extent of stack that will be used by this routine. If that overlaps the bottom of the heap, then a runtime error results. The ep or extreme stack pointer is updated, and also stored in the stack mark record for use in the ipj instruction to jump between procedures.

The assembly code represents the size as a forward referenced label. This is defined later in a label statement and back referenced by the assembler.

Instruction Opcode Stack in Stack out

ents l size 13 qn

Enter routine and allocate local variable space. The size field in the instruction contains the size of data for the routine being entered, including stack mark, parameters, and locals. The mp or mark pointer is offset by size, the area between the sp or stack pointer and the new allocation is cleared to zero and the sp set after that. The bottom of the stack field in the stack mark record is set from that, which allows the ipj instruction to find the proper setting of the stack for interprocedural jumps.

Checks if the resulting locals allocation would overrun the heap, and thus require more memory that is available. A runtime error results if not.

The assembly code represents the size as a forward referenced label. This is defined later in a label statement and back referenced by the assembler.

Instruction Opcode Stack in Stack out

equa 17 address address boolean

equb 139 boolean Boolean boolean

equc 141 character character boolean

equi 137 integer integer boolean

equm size 142 qn address address boolean

equr 138 real real boolean

equs 140 set set boolean

Find equal. The top of stack and second on stack values are compared, and a Boolean result of the comparision replaces them both. The compare is done according to type. Note that types a, b, c and i are treated equally since values are normalized to 32 bit integer on stack.

Instruction Opcode Stack in Stack out

equv 215 len sadr len sadr boolean

Compare two vectorized strings. Expects the length of the first vector on stack top, followed by the first vector address, then the length of the second vector, then the address of the second vector. Vectorized strings are those that are passed or indexed as variable length arrays. Finds the strings are equal, and returns true if so, otherwise false. Removes the two lengths and two addresses, and replaces them with the truth value.

Instruction Opcode Stack in Stack out

eext 242

Execute external. This is a pint internal instruction. It is used to execute an externally defined routine. A table in pint is filled with eext instructions, and the address of the instruction determines which external routine is executed.

Instruction Opcode Stack in Stack out

fjp addr 24 qn boolean

Jump false. Expects a Boolean value atop the stack. If the value is false, or zero, execution continues at the instruction constant addr. Removes the boolean from stack.

Instruction Opcode Stack in Stack out

flo 34 integer real real real

flt 33 integer real

Convert stack integer to floating point. flt converts the top of stack to floating point from integer. flo converts the second on stack to floating point, but it assumes that the top of stack is real as well.

Instruction Opcode Stack in Stack out

geqb 151 boolean boolean boolean

geqc 153 character character boolean

geqi 149 integer integer boolean

geqm size 154 qn address address boolean

geqr 150 real real boolean

geqs 152 set set boolean

Find greater than or equal. The top of stack and second on stack values are compared for second on stack greater than or equal to the top of stack, and a Boolean result of the comparision replaces them both. The compare is done according to type. Note that types a, b, c and i are treated equally since values are normalized to 32 bit integer on stack.

Instruction Opcode Stack in Stack out

geqv 220 len sadr len sadr boolean

Compare two vectorized strings. Expects the length of the second vector on stack top, followed by the second vector address, then the length of the first vector, then the address of the first vector. Vectorized strings are those that are passed or indexed as variable length arrays. Finds if the second string is greater than or equal to the first string, and returns true if so, otherwise false. Removes the two lengths and two addresses, and replaces them with the truth value.

Instruction Opcode Stack in Stack out

grtb 157 boolean boolean boolean

grtc 159 character character boolean

grti 155 integer integer boolean

grtm size 160 qn address address boolean

grtr 156 real real boolean

grts 158 set set boolean

Find greater than. The top of stack and second on stack values are compared for second on stack greater than the top of stack, and a Boolean result of the comparision replaces them both. The compare is done according to type. Note that types a, b, c and i are treated equally since values are normalized to 32 bit integer on stack.

Instruction Opcode Stack in Stack out

gtrv 218 len sadr len sadr boolean

Compare two vectorized strings. Expects the length of the second vector on stack top, followed by the second vector address, then the length of the first vector, then the address of the first vector. Vectorized strings are those that are passed or indexed as variable length arrays. Finds if the second string is greater than the first string, and returns true if so, otherwise false. Removes the two lengths and two addresses, and replaces them with the truth value.

Instruction Opcode Stack in Stack out

inca cnt 90 qn address address

incb cnt 93 qn boolean boolean

incc cnt 94 qn character character

inci cnt 10 qn integer integer

incx cnt 201 qn integer integer

Increment top of stack. The unsigned constant cnt is added to the value on stack top. Note that type a is not subject to arithmetic checks, but the other types are.

Instruction Opcode Stack in Stack out

inda off 85 qn address address

indb off 88 qn address boolean

indc off 89 qn address character

indi off 9 qn address integer

indx off 198 qn address integer

indr off 86 qn address real

inds off 87 qn address set

Load indirect to stack. Expects the address of an operand in memory on the stack top. The constant off is added to the address, then the operand fetched from memory and that replaces the address on stack top.

Instruction Opcode Stack in Stack out

inn 48 set value boolean

Set inclusion. Expects a set on stack top, and the value of a set element below that. Tests for inclusion of the value in the set, then replaces both of them with the Boolean result of that test.

Instruction Opcode Stack in Stack out

int 46 set set set

Set intersection. Finds the intersection of the two sets on the stack and replaces them both with the resulting set.

Instruction Opcode Stack in Stack out

inv 189 address

Invalidate address. Expects an address on stack, then flags that location as undefined if undefined checking is enabled. This instruction is used to return variables to the undefined state. Removes the address from stack.

Instruction Opcode Stack in Stack out

ior 44 boolean boolean boolean

Boolean inclusive ‘or ‘. Expects to find two Boolean values on stack. Replaces them with the inclusive ‘or’ of the values.

Instruction Opcode Stack in Stack out

ipj p l addr 112 p8 qn

Interprocedure jump. The instruction contains the relative frame number p, and the address to jump to within the target procedure addr. The frames above the target procedure frame are discarded, and the target frame is loaded. Execution then proceeds at the given address addr.

The assembly code represents the address addr as a forward referenced label. This is defined later in a label statement and back referenced by the assembler.

Instruction Opcode Stack in Stack out

ivti off size lvt 192 qn qn newtagval tagaddr newtagval tagaddr

ivtx off size lvt 101 qn qn newtagval tagaddr newtagval tagaddr

ivtb off size lvt 102 qn qn newtagval tagaddr newtagval tagaddr

ivtc off size lvt 111 qn qn newtagval tagaddr newtagval tagaddr

Invalidate tagged variant. Expects a new setting for a tagfield at stack top, and the address of the tagfield below that. If the tagfield was defined, and the new tag value is different than the old tag value, the variant area controlled by the tagfield is set as undefined. The off instruction field constains the offset from the tagfield to the base of the variant. The size instruction field contains the size of the variant. The logical variant table contains a label of the tagfield value to logical variant lookup table for the variant record in use. The offset is applied to the address of the tagfield, and the entire variant is set undefined. Note that the size of the variant is the maximum size of all possible subvariants. The stack is left the same way it was found.

The stack is unchanged during this operation.

Instruction Opcode Stack in Stack out

ixa size 16 qn index address address

Scale array access. Expects an index value at stack top, and the address of an array below that. The size instruction field contains the size of the base element of the array. The index is multiplied by the size, then added to the address of the base of the array, then that element address replaces both the index and the base address on stack.

Note that this operation does not remove the array index offset, ie., array [1..10] does not have 1 removed here.

Instruction Opcode Stack in Stack out

lao off 5 qn address

Load address with offset. Loads the address in the globals area with offset off. This instruction is used to index globals. The address is placed on stack top.

Instruction Opcode Stack in Stack out

lca len ‘str’ 56 qn address

Load string address. P6 accepts any length of string between quotes, but stores the string as the given length. The string is padded as required with spaces. Accepts a “quote image” in the string.The string is placed in the constants area, and the instruction gets the address of that. Loads the string constant address off to the top of the stack.

Instruction Opcode Stack in Stack out

lcp 135 addr addr integer

Load complex pointer. Expects the address of a complex pointer on stack. Loads the address/length or address/template address pair onto the stack with the address at stack top, followed by the length or template.

Instruction Opcode Stack in Stack out

lda p addr 4 p8 qn address

Load local address. Expects the relative frame number of the target procedure in p, and the offset address of the local in addr. The frame is found and the address calculated as an offset from the locals there and placed on the stack.

Instruction Opcode Stack in Stack out

ldcb bool 126 q8 boolean

ldcc char 127 q8 character

ldci int 123 qn integer

Load constant. Loads a constant from within the instruction according to the type, Boolean, character or integer.

Instruction Opcode Stack in Stack out

ldcn 125 nilcst

Load nil value. Loads the value of nil to the stack top.

Instruction Opcode Stack in Stack out

ldcr real 124 qn real

Load constant real. The address in the instruction gives the address of the real, which is loaded to stack top. The real given is loaded into the constants area on assembly, and the instruction contains the address of that.

Instruction Opcode Stack in Stack out

ldcs ( set ) 7 qn set

Load constant set. The address in the instruction gives the address of the set, which is loaded to stack top. This is used to load sets from the constant area. The set is specified as a series of values in the set between parenthsis. This set is loaded into the constant area on assembly, and the instruction contains the address of that.

Instruction Opcode Stack in Stack out

ldoa off 65 qn address

ldob off 68 qn boolean

ldoc off 69 a32 character

ldoi off 1 qn integer

ldox off 194 qn integer

ldor off 66 qn real

ldos off 67 qn set

Load from offset. Loads a global value from the globals area. The off constant gives the offset of the variable in the stack area. The value is loaded to the top of stack according to type.

Instruction Opcode Stack in Stack out

ldp 225 adr adr len

Load complex pointer. Expects the address of the complex pointer as a address/length pair. Loads the complex pointer with the address first on stack, followed by the length. The original address is removed.

Instruction Opcode Stack in Stack out

leqb 163 boolean boolean boolean

leqc 165 character character boolean

leqi 161 integer integer boolean

leqm size 166 qn address address boolean

leqr 162 real real boolean

leqs 164 set set boolean

Find less than or equal. The top of stack and second on stack values are compared for second on stack less than or equal to the top of stack, and a Boolean result of the comparision replaces them both. The compare is done according to type. Note that types a, b, c and i are treated equally since values are normalized to 32 bit integer on stack.

Instruction Opcode Stack in Stack out

leqv 219 len sadr len sadr boolean

Compare two vectorized strings. Expects the length of the second vector on stack top, followed by the second vector address, then the length of the first vector, then the address of the first vector. Vectorized strings are those that are passed or indexed as variable length arrays. Finds if the second string is less than or equal to the first string, and returns true if so, otherwise false. Removes the two lengths and two addresses, and replaces them with the truth value.

Instruction Opcode Stack in Stack out

lesb 169 boolean boolean boolean

lesc 171 character character boolean

lesi 167 integer integer boolean

lesm size 172 qn address address boolean

lesr 168 real real boolean

less 170 set set boolean

Find less than. The top of stack and second on stack values are compared for second on stack less than the top of stack, and a Boolean result of the comparision replaces them both. The compare is done according to type. Note that types a, b, c and i are treated equally since values are normalized to 32 bit integer on stack.

Instruction Opcode Stack in Stack out

lesv 217 len sadr len sadr boolean

Compare two vectorized strings. Expects the length of the second vector on stack top, followed by the second vector address, then the length of the first vector, then the address of the first vector. Vectorized strings are those that are passed or indexed as variable length arrays. Finds if the second string is less than the first string, and returns true if so, otherwise false. Removes the two lengths and two addresses, and replaces them with the truth value.

Instruction Opcode Stack in Stack out

lft tadr 213 qn adr adr tadr

Load complex fixed container. Expects the address of a container on stack, and the address of the template for the container in the instruction. The template address is placed on stack over the container address. This instruction is used to pick up a statically declared array and convert it to a complex container.

Instruction Opcode Stack in Stack out

lip p addr 120 p8 qn mp pfaddr

load procedure function address. Expects the relative mark count p, and the address of an existing mark/function address pair in the instruction. Loads a mark/address pair for a procedure or function parameter onto the stack from a previously calculated pair in memory. Used to pass a procedure or function parameter that was passed to the current function to another procedure or function. See the cip instruction for further information.

Instruction Opcode Stack in Stack out

lnp gadr 20

This is an internal setup instruction for pint. It contains the address of the globals, which are then cleared to zero, and other setup operations are performed.

Instruction Opcode Stack in Stack out

loda p off 105 Address

lodb p off 108 boolean

lodc p off 109 character

lodi p off 0 integer

lodx p off 193 integer

lodr p off 106 real

lods p off 107 set

Load local value. Expects the display offset p and the offset address in the local procedure frame off in the instruction. The value is loaded according to type.

Instruction Opcode Stack in Stack out

lpa p l addr 114 p8 qn mp pfaddr

Load procedure address. The current mark pointer is loaded onto the stack, followed by the target procedure or function address. This puts enough information on the stack to call it with the callers environment.

The assembly code represents the address addr as a forward referenced label. This is defined later in a label statement and back referenced by the assembler.

Instruction Opcode Stack in Stack out

lsa off 241 adr

Load stack address. Expects an offset to the current stack in the instruction. The offset is added to the current stack and the resulting address pushed onto stack. Used in overloaded operators to index parameters on the stack.

Instruction Opcode Stack in Stack out

ltcb off 231 qn boolean

ltcc off 232 qn char

ltci off 228 qn integer

ltcr off 229 qn real

ltcs off 230 qn sadr

ltcx off 233 qn integer

Load from constants area. Expects an offset address to the constants area in the instruction. The constant is loaded by type and placed on stack.

Instruction Opcode Stack in Stack out

lto off 234 qn adr

Load constant address. Expects an offset address to the constants area in the instruction. An address to the constant is loaded to stack.

Instruction Opcode Stack in Stack out

max lvl 214 lvl dadr tadr integer

Find maximum dimension of array. Expects a level in the instruction, a level argument on stack, a data address followed by a template address or length. The length of the array at the level indicated is found and replaces the level, data address and template address on stack.

Instruction Opcode Stack in Stack out

mod 49 integer integer integer

Find modulo. Finds the modulo of the second on stack by the top of stack. The result replaces both operands. This is always an integer operator.

Instruction Opcode Stack in Stack out

mov size 55 qn srcaddr destaddr

Move bytes. The instruction contains the number of bytes to move. The top of stack contains the source address, and the second on stack contains the destination address. The specified number of bytes are moved. Both addresses are removed from the stack.

Instruction Opcode Stack in Stack out

mpi 51 integer integer integer

mpr 52 real real real

Multiply. The first and second on stack are multiplied together, and the result replaces them both.

Instruction Opcode Stack in Stack out

mrkl line 174

Mark source line. This instruction exists only within the interpreter, and is not specified in the intermediate. The instruction parameter line contains the number of the source line that was read in at this point in the intermediate generation. The interpreter can use this to provide various source level debug services.

Instruction Opcode Stack in Stack out

mse 209 vec madr, sadr, eadr

Handle next exception frame. Expects the exception vector, mark address, stack address and exception address on stack. The current exception frame is discarded and the previous frame restored. The exception is then “thrown” to the new frame/next frame in stacking seqeunce, which then matches the exception. If there are no more frames in the frame stack, the exception goes to the unhandled exception handler. This instruction is used when the current exception handler does not match any condition, direct match with vector, wildcard match, or else.

Instruction Opcode Stack in Stack out

mst p q 11 p8 qn

Mark stack. This instruction is used before parameters are loaded in expectation of a user procedure call. The p parameter in the instruction contains the level of the calling procedure minus the level of the called procedure and is used to establish the static links in the mark. The q parameter is the size of the function result. A stack mark record is placed onto the stack an the sl or static link is placed, the dl or dynamic link, and the ep or extended stack pointer.

Instruction Opcode Stack in Stack out

neqa 18 address address boolean

neqb 145 boolean Boolean boolean

neqc 147 character character boolean

neqi 143 integer integer boolean

neqm size 148 address address boolean

neqr 144 real real boolean

neqs 146 set set boolean

Find not equal. The top of stack and second on stack values are compared for second on stack not equal to the top of stack, and a Boolean result of the comparision replaces them both. The compare is done according to type. Note that types a, b, c and i are treated equally since values are normalized to 32 bit integer on stack.

Instruction Opcode Stack in Stack out

neqv 216 len sadr len adr boolean

Compare two vectorized strings. Expects the length of the second vector on stack top, followed by the second vector address, then the length of the first vector, then the address of the first vector. Vectorized strings are those that are passed or indexed as variable length arrays. Finds if the second string is not equal to the first string, and returns true if so, otherwise false. Removes the two lengths and two addresses, and replaces them with the truth value.

Instruction Opcode Stack in Stack out

ngi 36 integer integer

ngr 37 real real

Negate. The operand atop the stack is negated according to type.

Instruction Opcode Stack in Stack out

notb 42 boolean boolean

noti 205 integer integer

Logical ‘not’. The Boolean or integer value atop the stack is inverted.

Instruction Opcode Stack in Stack out

odd 50 integer boolean

Find odd/even. The lowest bit of the integer on stack is masked to find the even/odd status of the integer as a Boolean value.

Instruction Opcode Stack in Stack out

ordb 134 boolean integer

ordc 136 character integer

ordi 59 integer integer

ordx 200 integer integer

Find ordinal value of Boolean, character or integer. This is a no-op, because P6 always converts values to 32 bit integers on load to stack.

Instruction Opcode Stack in Stack out

pck sizep sizeu 63 qn qn inx parr uprr

Convert unpacked array to packed array. Because P6 does not support packing, this is effectively a copy operation. The instruction contains the size of the packed array sizep and the size of the unpacked array sizeu. The stack contains the address of the packed array at top, the starting index of the unpacked array under that, and the unpacked array address as third on stack. The number of elements in the packed array are moved from the unpacked array at the starting index to the packed array. All parameters are removed from the stack.

Instruction Opcode Stack in Stack out

ret 22 adr

Return from initializer code strip. Returns to the caller. This instruction is used to return from an initializer code strip, which is a short section of code used to set up parameterized variables. The return address is removed from stack.

Instruction Opcode Stack in Stack out

reta 132 address

retb 131 boolean

retc 130 character

reti 128 integer

retx 204 integer

retp 14

retr 129 real

Return from procedure or function with result. Returns from the current procedure or function. The pc, sp, ep and mp pointers are restored from the saved data in the active stack mark record. For retp or return from procedure, this is all that is done. For the others, a return value is processed according to type. The value of the result is loaded from its place in the stack mark record, and expanded to 32 bits if Boolean or character.

Instruction Opcode Stack in Stack out

rgs 110 high low set

Build range set on stack. Expects a high value on stack top, and a low value below that. A set is constructed that has all of the members from the low value to the high value in it, then that set is placed as stack top.

The range builder instruction, along with sgs or build singleton set, is used to construct complex sets by creating sets of each individual ranges of values, then adding the resulting sets together on stack.

Instruction Opcode Stack in Stack out

rnd 62 real integer

Round real value to integer. Converts the real on top of the stack to integer by rounding.

Instruction Opcode Stack in Stack out

sbi 30 integer integer integer

sbr 31 real real real

Subtract. Subtracts the top of stack value from the second on stack value according to type.

Instruction Opcode Stack in Stack out

scp 224 dadr tadr adr

Store complex pointer. Expects an array data address on stack top, followed by a template address or length, then the desination address. The complex pointer is stored at the given address, with the array data address first, followed by the template or length. All parameters are removed from stack.

Instruction Opcode Stack in Stack out

sgs 32 integer set

Construct singleton set. Expects a value on stack. Constructs a set with that value as the single member, then that set replaces the value on stack.

The singleton set instruction, along with rgs or range set builder, is used to construct complex sets by creating sets with individual ranges of values, then adding the resulting sets together on stack.

Instruction Opcode Stack in Stack out

spc 222 addr addr addr integer

Simplify complex pointer. Expects a complex pointer at stack top, with the base address at top, followed by a template pointer. The value at the template pointer replaces the template pointer, thus simplifying the pointer. This is used when a complex pointer is at the lowest index of the template.

Instruction Opcode Stack in Stack out

sqi 38 integer integer

sqr 39 real real

Find square of value. Expects a value on stack top, and finds the square of that value according to type. This replaces the value on stack top.

Instruction Opcode Stack in Stack out

sroa off 75 qn address

srob off 78 qn boolean

sroc off 79 qn character

sroi off 3 qn integer

sror off 76 qn real

sros off 77 qn set

srox off 196 qn integer

Store global value. The instruction contains the offset of a variable in the stack bottom, where the globals for the program exist. The variable is stored from the stack according to type, and the value removed from the stack.

Instruction Opcode Stack in Stack out

stoa 80 address address

stob 83 boolean address

stoc 84 character address

stoi 6 integer address

stor 81 real address

stos 82 set address

stox 197 integer address

Store value to address. Expects a value according to instruction type atop the stack, and an address below that. The value is stored to the address, and both are removed from the stack.

Instruction Opcode Stack in Stack out

stp 58

Stop. Executing this instruction causes the interpreter to exit.

Instruction Opcode Stack in Stack out

stra p off 70 address

strb p off 73 boolean

strc p off 74 character

stri p off 2 integer

strx p off 195 integer

strr p off 71 real

strs p off 72 set

Store local. Instruction parameter p contains the relative count of the procedure frame to access. Instruction parameter off contains the offset address within the frame. The value at stack top is stored to the local variable according to type, and the value removed from the stack.

Instruction Opcode Stack in Stack out

suv radr vadr 91 qn qn

Set user virtual procedure/function vector. The first parameter of the instruction contains the new routine address (procedure/function) for the virtual routine. The second address contains the vector. The new override procedure is placed atop the existing virtual routine vector. The original vector is usually stored locally. The stack is unchanged.

Instruction Opcode Stack in Stack out

swp size 118 value address address value

Swap first and second stack operands. The instruction contains the size of the second value on stack. The top value is assumed to be 32 bit. The top and second values are swapped. This instruction is used primarily to chain writes together to the same file access pointer.

Instruction Opcode Stack in Stack out

tjp l addr 119 boolean

Jump true. Expects a Boolean value on stack top. If the Boolean is true, the jump is taken to address addr. The Boolean is removed from the stack.

The assembly code represents the address addr as a forward referenced label. This is defined later in a label statement and back referenced by the assembler.

Instruction Opcode Stack in Stack out

trc 35 real integer

Truncate to integer. The real on stack top is converted to integer by truncating the fractional part. The integer replaces the real on stack top.

Instruction Opcode Stack in Stack out

ujc 61 qn

Output case error. This instruction is used in case statement tables to output case value errors. See xjp for the case table format. It is designed to be the same length as ujp, which is used to build the case table. The 32 bit address in the struction is a dummy, and contains zero. Always outputs a bad case select error. Inserted into the table where the value for the case would be invalid.

Instruction Opcode Stack in Stack out

ujp l addr 23

Unconditional jump. The instruction contained address addr is jumped to.

The assembly code represents the address addr as a forward referenced label. This is defined later in a label statement and back referenced by the assembler.

Instruction Opcode Stack in Stack out

uni 47 set set set

Find set union. Expects two sets on stack. The union of the sets is found, and that replaces them both.

Instruction Opcode Stack in Stack out

upk sizep sizeu 64 qn qn sindex upackarr packarr

Unpack a packed array. Pascal-P6 does not implement packing, so this instruction is implemented as a copy operation. Expects the size of the packed array sizep and the size of the unpacked array sizeu in the instruction. The stack contains the starting index of the unpacked array at top, followed by the address of the unpacked array, and the address of the packed array as third on stack. The entire contents of the packed array are transferred into the unpacked array, which can be larger than the packed array.

Instruction Opcode Stack in Stack out

vbe 92 qn addresss

Variable reference block end. Ends a variable reference.

Instruction Opcode Stack in Stack out

vbs len 92 qn addresss

Variable reference block start. Establishes a variable reference to a block. Expects a base address for the referenced block on stack. The instruction contains the length of the block. Used to enforce the rule that tag values cannot be changed in a variable referenced block. The variable reference will exist until terminated by a vbe instruction. For each vbs instruction, there must be a matching vbe instruction.

Instruction Opcode Stack in Stack out

vdd 227 adr

Vector dispose array. Expects the address of a dynamically allocated array on stack. Disposes of the array. The address is removed from stack. This instruction is used to deallocate dynamically allocated arrays.

Instruction Opcode Stack in Stack out

vdp 221 adr

Vector dispose array. Expects the address of a dynamically allocated array on stack. Disposes of the array. The address is removed from stack. This instruction is used to deallocate global parameterized arrays.

Instruction Opcode Stack in Stack out

vin lvl siz 226 qn qn adr siz [siz]...

Vector initialize dynamic. Allocate container array. Expects the address of the container dynamic variable on stack top, followed by one or more sizes of array dimensions. The instruction contains the level or number of dimensions, and the size contains the base element size. The total size of the array is calculated along with a template size, with one element for each dimension. The array is then allocated, and the template for it filled in at the start of the array. Dynamically created container arrays have their templates placed with the array at the start. All elements are removed from the stack.

Instruction Opcode Stack in Stack out

vip lvl siz 133 qn qn adr siz [siz]...

Vector initialize pointer global. Expects the address of the container dynamic variable on stack top, followed by one or more sizes of array dimensions. The instruction contains the level or number of dimensions, and the size contains the base element size. The total size of the array is calculated along with a template size, with one element for each dimension. The template is placed in global memory after the array address. The array is then allocated dynamically, and the address of that placed before the template. All elements are removed from the stack.

Instruction Opcode Stack in Stack out

vis lvl siz 122 qn qn adr siz [siz]...

Vector initialize stack. Expects the address of the container dynamic variable on stack top, followed by one or more sizes of array dimensions. The instruction contains the level or number of dimensions, and the size contains the base element size. The total size of the array is calculated along with a template size, with one element for each dimension. The template is placed in local memory after the array address. All elements are removed from the stack, then the array is then allocated on stack, and the address of that placed before the template.

Instruction Opcode Stack in Stack out

xjp l addr 25 qn index

Table jump. Expects an address of a jump table within the instruction, and a jump table index on the stack. The jump table constists of a series of jump instructions using the ujp instruction. The index is multiplied by the length of the ujp instruction, which is 5 bytes long, and the pc set to that address. The effect is to jump via a table of entries from 0 to n. The index is removed from stack.

This instruction is used to implement the case statement.

The assembly code represents the address addr as a forward referenced label. This is defined later in a label statement and back referenced by the assembler.

Instruction Opcode Stack in Stack out

xor 206 int int int

‘xor’s integers or booleans. Expects two integers on stack or two booleans, which are treated the same. The integers are ‘xor’ed and the result replaces them on stack. If either integer is negative, an error results.

## System calls

The P6 interpreter partitions the execution work done in the stack machine into directly executed instructions and system calls. The system call instructions consist of I/O related calls and dynamic storage allocation. These are normally things that are handled by the runtime support system.

The difference between a function executed in an instruction and a function executed in a system call is minimal. A system call uses parameters stacked onto the interpreter operand stack in reverse just as the intermediate does.

The difference between a function executed in a user based procedure or function and a function executed in a system call is that the system call requires no framing or locals.

System calls are executed by number via the csp instruction.

The following system call descriptions show the intermediate menmomic of the call, the instruction number used as a parameter to the csp instruction, and the operands as they exist on the stack before the call. The stack operands are listed from top of stack first at left, to the bottommost operand at the right.

Each system call can be a procedure or a function. If it is a function, it leaves its result on the stack.

System Call Number Stack in Stack out

aefb 79 len sadr filadr

Assign external filename to binary file. Expects the length of the filename at stack top, followed by the filename address, then the file address. The filename passed is the name of the variable for the file. The standard Pascaline behavior is to read the filename off the command line. However, other behaviors are possible, such as using the variable name for the filename. The compiler garantees that the assignments are carried out in order. The length, the filename address and the file address are all removed from stack.

System Call Number Stack in Stack out

aeft 78 len sadr filadr

Assign external filename to text file. Expects the length of the filename at stack top, followed by the filename address, then the file address. The filename passed is the name of the variable for the file. The standard Pascaline behavior is to read the filename off the command line. However, other behaviors are possible, such as using the variable name for the filename. The compiler garantees that the assignments are carried out in order. The length, the filename address and the file address are all removed from stack.

System Call Number Stack in Stack out

appb 58 filadr

Append binary file. Expects a file address on stack. The file is set to write mode and positioned to the end. The file is removed from the stack.

System Call Number Stack in Stack out

appt 50 filadr

Append text file. Expects a file address on stack. The file is set to write mode and positioned to the end. The file is removed from the stack.

System Call Number Stack in Stack out

assb 56 len addr addr

Assign file name binary. Associates the file name with a binary file. The length of the filename is expected on stack top, followed by the address of the filename string, then the address of the binary file itself. All stack parameters are removed.

System Call Number Stack in Stack out

asst 46 len sadr filadr

Assign file name text. Associates the file name with a text file. The length of the filename is expected on stack top, followed by the address of the filename string, then the address of the text file itself. All stack parameters are removed.

System Call Number Stack in Stack out

ast 60 integer

Assert. Expects a truth value on stack. If the value is true, the execution continues. If not, exection stops. The integer value is removed.

System Call Number Stack in Stack out

asts 61 len str int

Assert with message. Expects a length, followed by the string length, and a truth value under that. The value is checked for true, and if false, the program stops with an error message given by the string. Otherwise execution continues. The length, string address, and the integer are all removed.

System Call Number Stack in Stack out

atn 19 real real

Find arctangent. Expects a real on stack. Finds the arctangent, and that replaces the stack top.

System Call Number Stack in Stack out

chg 52 osadr oslen nsadr nslen

Change the name of a file. Expects the existing name string on stack top, the length of that below it, then the new string, and the new string length. The filename is changed. All strings and lengths are removed.

System Call Number Stack in Stack out

clsb 57 filadr

Close binary file. Expects the file address on stack top. Closes the file, and removes it if it is a temp file. Removes the file address from stack.

System Call Number Stack in Stack out

clst 47 filadr

Close text file. Expects the file address on stack top. Closes the file, and removes it if it is a temp file. Removes the file address from stack.

System Call Number Stack in Stack out

cos 15 real real

Find cosine. Expects a real on stack. Finds the cosine, and that replaces the stack top.

System Call Number Stack in Stack out

del 51 sadr len

Delete file by name. Expects the filename string at stack top, followed by the length. Removes both string and length.

System Call Number Stack in Stack out

dsl 40 size tagcnt tagcst… addr

Dispose of dynamic tagged record. This is a special form of dispose for a dynamically allocated record with tagged variants. The top of the stack contains the size of the record, after any fixed tag constants specified are taken into account. Below that, the number of tagfield constants that were specified exists. Below that, a list of all the constants that were used to specify the allocation. This list is from leftmost in sourcecode order and deepest in the stack to rightmost in source and topmost in stack. Following that, the address of the dynamic record.

The allocation of the record, done by the system call nwl, contains a matching list of tagfield constants allocated “behind” the allocated pointer, with the number of tag constants last in the list so that it can be found just below the pointer. These lists represent the tagfield list used to allocate the record, in the dynamic itself, and the tagfield list used to dispose of it, on stack. These lists are compared for both number and value, and a runtime error results if not equal. The total allocation, record plus taglist, is then disposed of.

System Call Number Stack in Stack out

dsp 26 size addr

Dispose of dynamic variable. The top of stack contains the variable size, and the address under that is the address of the dynamic record. The variable is deallocated. Both operands are removed from stack.

System Call Number Stack in Stack out

efb 42 fileaddr boolean

Find eof of binary file. Expects the address of a file variable on stack. The file is tested for eof() true, and the Boolean result replaces the address on stack.

System Call Number Stack in Stack out

eln 7 fileaddr boolean

Test for eoln of text file. Expects the address of a file variable on stack. The file is tested for eoln() true, and the Boolean result replaces the address on stack.

System Call Number Stack in Stack out

eof 41 fileaddr boolean

Find eof of text file. Expects the address of a file variable on stack. The file is tested for eof() true, and the Boolean result replaces the address on stack.

System Call Number Stack in Stack out

exp 16 real real

Find exponential. Expects a real on stack. Finds the exponential, and that replaces the stack top.

System Call Number Stack in Stack out

exs 55 len str boolean

Find if file exists. Expects the length of the filename at stack top, followed by the string address. Finds and returns true if the file exists, otherwise false. Both length and string are removed.

System Call Number Stack in Stack out

fbv 43 fileaddr fileaddr

File buffer validate text. Expects the address of a file variable on stack. Ensures the file buffer variable is loaded. If the file is a read file, or the file is in read mode, the file buffer variable is loaded by reading the file. The file address pointer is left on stack. This call is used to insure the file buffer contains data from the file, if it exists, when the file buffer variable is referenced. It is part of the “lazy I/O” file scheme.

System Call Number Stack in Stack out

fvb 44 size fileaddr fileaddr

File buffer validate binary. Expects the base element size of the file on stack, followed by the address of a file variable on stack. Ensures the file buffer variable is loaded. If the file is a read file, or the file is in read mode, the file buffer variable is loaded by reading the file. The element length tells the call how many bytes to read. The file address pointer is left on stack, but the size is purged. This call is used to insure the file buffer contains data from the file, if it exists, when the file buffer variable is referenced. It is part of the “lazy I/O” file scheme.

System Call Number Stack in Stack out

gbf 35 size fileaddr

Get file buffer binary. Expects the base element size of the file on stack, followed by the address of a file variable on stack. If the file buffer variable is indicated as full, then it is marked empty, thus discarding the data there. If it is full, new data is read over the file buffer variable. Both the file address pointer and the size are discarded from stack. This operation is part of the “lazy I/O scheme”.

System Call Number Stack in Stack out

get 0 fileaddr

Get file buffer text. Expects the address of a file variable on stack. Reads the next element of the file into the file variable buffer. The file pointer is discarded.

System Call Number Stack in Stack out

hlt 59

Halt program. Simply halts the running program.

System Call Number Stack in Stack out

len 53 filadr

Find length in elements of binary file. Expects the file address on stack. Find the number of base elements in the file. Removes the file.

System Call Number Stack in Stack out

loc 54 filadr integer

Find location of read/write in a binary file. Expects the file address on stack. Find the location in the file and places on stack. Removes the file.

System Call Number Stack in Stack out

log 17 real real

Find logarithm. Expects a real on stack. Finds the logarithm, and that replaces the stack top.

System Call Number Stack in Stack out

new 4 addr size

Allocate dynamic variable. Expects the size of variable in bytes to allocate on stack top, followed by the address of the pointer variable. Space with the given size is allocated, and the address placed into the pointer variable. Both operands are removed from stack.

System Call Number Stack in Stack out

nwl 39 size tagcnt tagcst… addr

Allocate dynamic tagged record. This is a special form of new() for a dynamically allocated record with tagged variants. The top of the stack contains the size of the record, after any fixed tag constants specified are taken into account. Below that, the number of tagfield constants that were specified exists. Below that, a list of all the constants that were used to specify the allocation. This list is from leftmost in sourcecode order and deepest in the stack to rightmost in source and topmost in stack. Following that, the address of the pointer variable.

The space required for the tagfield constant list and length is added to the total space required, and that space is allocated. The tagfield constant list, with the following length, is copied to the start of the allocated space, and the pointer variable is set just after that. Thus the tagfield constant list exists “behind” the pointer or in “negative space”, so that it can be found and referenced for various purposes. These are:

1. To check that the tagfield constant list is equal between new() and dispose() operations.
2. To check if a tagfield used to set the total space allocated for the variable is assigned with a different value than originally used in the new() call.

The size, tagfield list and length, and the address of the pointer variable are discarded.

System Call Number Stack in Stack out

pag 21 fileaddr

Page text file. Expects the address of a file variable on stack. A page request is sent to the file, and the file address is discarded.

System Call Number Stack in Stack out

pbf 36 size fileaddr

Put file buffer binary. Expects the base element size of the file on stack, followed by the address of a file variable on stack. If the file buffer variable is empty, a runtime error results. Otherwise, the contents of the file buffer variable are written out to the file. The size and file address are discarded.

System Call Number Stack in Stack out

pos 36 loc filadr

Position file binary. Expects the location of the file element on stack, followed by the address of a file variable on stack. The file access point is moved to the specified location if it exists. The location and file address are discarded.

System Call Number Stack in Stack out

put 1 fileaddr

Put file buffer text. Expects the address of a file variable on stack. If the file buffer is empty, a runtime error results. Otherwise, writes the contents of the file buffer variable to the file. The file address is discarded.

System Call Number Stack in Stack out

rbf 32 len varaddr fileaddr fileaddr

Read binary file. Expects a file variable address on stack, the address to place read data above that, and the length of the base file element on stack top. If the buffer for the file is has data, then the data is read from that, otherwise the data is read from the file. The number of bytes in the element length are read.

Purges the target variable address and the element length from the stack, but leaves the file variable address. This is because a group of single reads can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

rcb 38 max min vaddr fileaddr fileaddr

Read character from text file with range check. Expects the file variable address on stack, the variable address to read to above that, then the minimum value min and the maximum value max at stack top. Reads a single character from the file, verifies that it lies in the specified range, and places it to the variable. If the character value lies out of range, a runtime error results.

Purges the target variable address from the stack, but leaves the file variable address. This is because a group of single reads can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

rcbf 74 fld max min vadr filadr filadr

Read character from text file with range check and field. Expects the file variable address on stack, the variable address to read to above that, then the minimum value min and the maximum value max, then the field at stack top. Reads a single character from the file, verifies that it lies in the specified range, and places it to the variable. If the character value lies out of range, a runtime error results. Only reads from characters within the field.

Purges the target variable address from the stack, but leaves the file variable address. This is because a group of single reads can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

rdc 13 varaddr fileaddr fileaddr

Read character from text file. Expects the file variable address on stack, and the variable address to read to above that. Reads a single character from the file to the variable.

Purges the target variable address from the stack, but leaves the file variable address. This is because a group of single reads can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

rdcf 75 field varadr filadr filadr

Read character from text file with field. Expects the file variable address on stack, and the variable address to read to, then the field above that. Reads a single character from the file to the variable.

Purges the target variable address from the stack, but leaves the file variable address. This is because a group of single reads can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

rdi 11 varaddr fileaddr fileaddr

Read integer from text file. Expects the file variable address on stack, and the variable address to read to above that. Reads a single integer from the file to the variable.

Purges the target variable address from the stack, but leaves the file variable address. This is because a group of single reads can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

rdie 80 len sadr vadr

Read external integer. Expects the length of the header name, followed by the header variable name address, followed by the address of the integer to read. Reads the header integer to the variable. The standard Pascaline behavior is to read the integer from the command line. The name of the header variable is passed to enable other behaviors. Removes the length, header name string, and variable address.

System Call Number Stack in Stack out

rdif 72 fld varaddr fileaddr fileaddr

Read integer from text file with field. Expects the file variable address on stack, and the variable address to read to, then the field above that. Reads a single integer from the file to the variable. Only the characters from the field are read.

Purges the target variable address from the stack, but leaves the file variable address. This is because a group of single reads can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

rdr 12 varaddr fileaddr fileaddr

Read real from text file. Expects the file variable address on stack, and the variable address to read to above that. Reads a single real from the file to the variable.

Purges the target variable address from the stack, but leaves the file variable address. This is because a group of single reads can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

rdre 81 len sadr vadr

Read external real. Expects the length of the header name, followed by the header variable name address, followed by the address of the real to read. Reads the header integer to the variable. The standard Pascaline behavior is to read the integer from the command line. The name of the header variable is passed to enable other behaviors. Removes the length, header name string, and variable address.

System Call Number Stack in Stack out

rdrf 73 fld varadr filadr fileaddr

Read real from text file with field. Expects the file variable address on stack, and the variable address to read to, then the field above that. Reads a single real from the file to the variable. Only the characters in the field are read.

Purges the target variable address from the stack, but leaves the file variable address. This is because a group of single reads can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

rds 70 len sadr filadr

Read string. Expects the length of the string on stack top, followed by the string address, then the address of the file to read. Only the number of characters in the string are read. The length, string address and file are removed.

System Call Number Stack in Stack out

rdsf 76 len fld sadr filadr

Read string with field. Expects the length of the string on stack top, followed by the field, and then the string address, then the address of the file to read. If the field is larger than the string, then trailing blanks are expected. If the field is shorter than the string, then only the number of characters in the field are read, and the rest of the string is left uninitialized. The length, field, string address and file are removed.

System Call Number Stack in Stack out

rdsp 77 len sadr filadr

Read padded string. Expects the length of the string at stack top, followed by the base address of the string, then the file address under that. Reads characters into the string until either the string is full, or eoln occurs. If the string is full, and more characters exist on the line, an error results. If eoln occurs before the string is filled, then the string is padded out until the end with blanks. The length, string and file address are removed.

System Call Number Stack in Stack out

rib 37 max min vaddr fileaddr fileaddr

Read an integer from text file with range check. Expects the file variable address on stack, the variable address to read to above that, then the minimum value min and the maximum value max at stack top. Reads a single integer from the file, verifies that it lies in the specified range, and places it to the variable. If the character value lies out of range, a runtime error results.

Purges the target variable address from the stack, but leaves the file variable address. This is because a group of single reads can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

ribf 72 fld max min vadr filadr filadr

Read an integer from text file with range check and field. Expects the file variable address on stack, the variable address to read to above that, then the minimum value min and the maximum value max, and the field at stack top. Reads a single integer from the file, verifies that it lies in the specified range, and places it to the variable. If the character value lies out of range, a runtime error results. The read length is limited by the field.

Purges the target variable address from the stack, but leaves the file variable address. This is because a group of single reads can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

rln 3 fileaddr fileaddr

Read next line from text file. Expects the file variable address on stack. The text file is read until an eoln() or eof() is encountered.

Leaves the file variable address. This is because a group of single reads can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

rsb 33 fileaddr

Reset file binary. Expects the file variable address on stack. The binary file is reset(). The file variable address is purged.

System Call Number Stack in Stack out

rsf 22 fileaddr

Reset file text. Expects the file variable address on stack. The text file is reset(). The file variable address is purged.

System Call Number Stack in Stack out

rwb 34 fileaddr

Rewrite file binary. Expects the file variable address on stack. The binary file is rewritten. The file variable address is purged.

System Call Number Stack in Stack out

rwf 23 fileaddr

Rewrite file text. Expects the file variable address on stack. The text file is rewritten. The file variable address is purged.

System Call Number Stack in Stack out

sin 14 real real

Find sine. Expects a real on stack. Finds the sine, and that replaces the stack top.

System Call Number Stack in Stack out

sqt 18 real real

Find square root. Expects a real on stack. Finds the square root, and that replaces the stack top.

System Call Number Stack in Stack out

thw 2 evadr

Throw exception. The exception variable is at stack top. The exception stack is unwound by disposing of the stack to the last exception handler, then the exception variable is replaced on stack. The exception statement then does one or more matches for the exception variable, and throws another exception level if not found.

System Call Number Stack in Stack out

upd 49 filadr

Update file. Expects a file address on stack. The file is changed to write mode and positioned at the start. The file is removed from the stack.

System Call Number Stack in Stack out

wbb 31 boolean fileaddr fileaddr

Write Boolean to binary file. Expects the file variable address on stack, and the Boolean to write to above that. Writes a single boolean to the file from the variable.

Purges the boolean from the stack, but leaves the file variable address. This is because a group of single writes can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

wbc 30 char fileaddr fileaddr

Write character to binary file. Expects the file variable address on stack, and the character to write to above that. Writes a single character to the file from the variable.

Purges the character from the stack, but leaves the file variable address. This is because a group of single writes can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

wbf 27 size varaddr fileaddr fileaddr

Write binary variable to binary file. Expects the file variable address on stack, the address of the variable to write above that, and the size in bytes of the variable at stack top. Writes all the bytes of the variable to the file single boolean to the file from the variable.

Purges the size and the variable address from the stack, but leaves the file variable address. This is because a group of single writes can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

wbi 28 integer fileaddr fileaddr

Write integer to binary file. Expects the file variable address on stack, and an integer to write to above that. Writes a single integer to the file from the variable.

Purges the integer from the stack, but leaves the file variable address. This is because a group of single writes can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

wbr 29 real fileaddr fileaddr

Write a real to binary file. Expects the file variable address on stack, and a real to write to above that. Writes a single real to the file from the variable.

Purges the real from the stack, but leaves the file variable address. This is because a group of single writes can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

wbx 45 integer fileaddr fileaddr

Write byte to binary file. Expects the file variable address on stack, and an integer to write to above that. Writes a single byte to the file from the variable.

Purges the integer from the stack, but leaves the file variable address. This is because a group of single writes can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

wiz 66 width integer filadr filadr

Write integer to text file in decimal form (base 10) with leading zeros. Expects the file variable address on stack, a integer to write to above that., and a field width at stack top. Writes the integer in the given width, using zeros to fill out the field.

Purges the width and the integer from the stack, but leaves the file variable address. This is because a group of single writes can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

wizb 69

Write integer to text file in binary form (base 2) with leading zeros. Expects the file variable address on stack, a integer to write to above that., and a field width at stack top. Writes the integer in the given width, using zeros to fill out the field.

Purges the width and the integer from the stack, but leaves the file variable address. This is because a group of single writes can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

wizh 67

Write integer to text file in hexadecimal form (base 16) with leading zeros. Expects the file variable address on stack, a integer to write to above that., and a field width at stack top. Writes the integer in the given width, using zeros to fill out the field.

Purges the width and the integer from the stack, but leaves the file variable address. This is because a group of single writes can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

wizo 68

Write integer to text file in octal form (base 8) with leading zeros. Expects the file variable address on stack, a integer to write to above that., and a field width at stack top. Writes the integer in the given width, using zeros to fill out the field.

Purges the width and the integer from the stack, but leaves the file variable address. This is because a group of single writes can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

wln 5 fileaddr fileaddr

Write next line to text file. Expects the file variable address on stack. A new line is written to the text file.

Leaves the file variable address. This is because a group of single reads can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

wrb 24 width boolean fileaddr fileaddr

Write boolean to text file. Expects the file variable address on stack, a boolean to write to above that., and a field width at stack top. Writes the Boolean in the given width.

Purges the width and the boolean from the stack, but leaves the file variable address. This is because a group of single writes can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

wrc 10 width char fileaddr fileaddr

Write character to text file. Expects the file variable address on stack, a character to write to above that., and a field width at stack top. Writes the character in the given width.

Purges the width and the character from the stack, but leaves the file variable address. This is because a group of single writes can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

wrf 25 frac width real fileaddr fileaddr

Write real to text file in fixed point notation. Expects the file variable address on stack, a real to write to above that., a field width above that, and a fraction at stack top. Writes the real in the given width.

Purges the fraction, width and the real from the stack, but leaves the file variable address. This is because a group of single writes can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

wri 8 width integer fileaddr fileaddr

Write integer to text file in decimal form (base 10). Expects the file variable address on stack, a integer to write to above that., and a field width at stack top. Writes the integer in the given width.

Purges the width and the integer from the stack, but leaves the file variable address. This is because a group of single writes can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

wrib 64 width integer fileaddr fileaddr

Write integer to text file in binary form (base 2). Expects the file variable address on stack, a integer to write to above that., and a field width at stack top. Writes the integer in the given width.

Purges the width and the integer from the stack, but leaves the file variable address. This is because a group of single writes can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

wrih 62

Write integer to text file in hexadecimal form (base 16). Expects the file variable address on stack, a integer to write to above that., and a field width at stack top. Writes the integer in the given width.

Purges the width and the integer from the stack, but leaves the file variable address. This is because a group of single writes can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

wrio 63

Write integer to text file in octal form (base 2). Expects the file variable address on stack, a integer to write to above that., and a field width at stack top. Writes the integer in the given width.

Purges the width and the integer from the stack, but leaves the file variable address. This is because a group of single writes can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

wrr 9 width real fileaddr fileaddr

Write real to text file in floating point notation. Expects the file variable address on stack, a real to write to above that., and a field width at stack top. Writes the real in the given width.

Purges the width and the real from the stack, but leaves the file variable address. This is because a group of single writes can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

wrs 6 width saddr len fileaddr fileaddr

Write string to text file. Expects the file variable address on stack, the length of the string above that, address of the string to write to above that, and the field width above that.

Purges the length, width and the string address from the stack, but leaves the file variable address. This is because a group of single writes can use the same file reference, so it is left on stack until specifically removed.

System Call Number Stack in Stack out

wrsp 65 sadr len filadr

Write padded string to text file. Expects the string address first on stack, followed by the length, then the file address. The padded length of the string is determined, which is the position of the last non-space character in the string. Then the characters to the left of that position, inclusive, are output to the file. Removes the string address, length and file address.

# Testing P6

In the original implementation of Pascal and the Pascal-P porting kit, the implementation of tests on the system were largely undefined. Today, most programmers realize that any program exists as a collection of the code, documentation, and finally the tests for the program to prove it correct. If any one of these elements is missing, much as a three legged stool, the program will fall over. In reality, an “undocumented” program that is popular gets documented by its users or by third parties, such as independent book writers. Tests can be carried out ad-hoc, or essentially fall to the users (their happiness with this situation being quite another matter!).

Thus, as important to P6 as its code or documents are its tests.This was widely recognized with original Pascal, and an extensive series of tests were created with the advent of the ISO 7185 standard, as documented in “Pascal compiler validation” [Brian Wichmann & Z. J. Chechanowicz]. This was an excellent series of tests that showed close relationship to the standard. Unfortunately, like the “model compiler”, the rights to this test series, which was initially openly distributed, were closely held, and the project is essentially dead today.

Accordingly, I have created a new series of tests for P6, and now that is distributed with P6 itself, and continues to be improved. The PAT or Pascal Acceptance Test is completely automated. Further, because the original ISO 7185 validation tests were distributed in the “Pascal User’s Group” free of restrictions, I have used them to check on the completeness of the PAT. Thus, even if the original ISO 7185 validation tests cannot legally be distributed, the PAT is tracable back to these tests and is a reliable substitute for them.

## Running tests

### testprog

The main test script for testing is:

testprog <Pascal source file>

testprog.bat <Pascal source file>

testprog is a “one stop” test resource for most programs. It expects the following files to exist under the given primary filename:

program.pas The Pascal source file.

program.inp The input file for the running program.

program.cmp The reference file for the expected output.

testprog compiles and runs the target program, and checks its output against the reference file. Several files are produced during the process:

program.P6 Contains the intermediate code for the program as compiled by pcom.

Program.err Contains the output from pcom. This includes the status line indicating the number of errors. It also typically contains a listing of the program as compiled, which contains line numbers and other information. Note that if the pcom compile run produces errors, testprog will stop.

Program.lst Contains the output from the program when run. This is all output to the standard “ output” file.

Program.out This is all of the output to the “prr” special file. This is only used by P6 for special purposes, such as to output the intermediate code, and most programs will be empty.

Program.dif This is the output of the diff command between program.lst and program.cmp. It should be empy if the program produced the output expected.

Not all of the files for testprog need to have contents. For example, a program that does not do input does not need to have a program.inp file. An example of this would be the “hello” program. However, testprog expects all of the files to exist.

To determine if the test ran correctly, the output of first the program.err file should be checked for zero errors, then the resulting program.dif file is checked for zero length. All of these results are announced during the test:

C:\projects\PASCAL\P6\_ext>testprog sample\_programs\roman

Compile and run sample\_programs\roman

Compile fails, examine the sample\_programs\roman.err file

For a program that has a compile error.

C:\projects\PASCAL\P6\_ext>testprog sample\_programs\roman

Compile and run sample\_programs\roman

03/18/2012 10:44 AM 76 roman.dif

For a program that does not match it’s expected output.

If a large series of test programs are compiled and run with testprog, it may be more efficient to examine the output files to determine the result, first the program.err file, then the program.dif file. The rejection test is a good example of such a large series of files.

### Other tests

Not all test programs work well with the testprog script. Examples are “pascals”, and the self compile. For these programs a special test script is provided.

### Regression test

All of the “positive” tests are wrapped up in a single script, “regress.bat”. This script runs all of the positive tests in order of successive complexity. Whenever a change is made to P6, the regression test is run to verify that the resulting compiler is still valid.

## Test types

The Pascal-P6 tests are divided into two major catagories, referred here as “positive” and “negative” tests. The positive tests are designed to test what the compiler should accept as a valid program. The negative tests are designed to test what the compiler should reject as invalid programs.

Thus, they are termed here as the “Pascal Acceptance” test, and the “Pascal rejection test”. The two test types are fundamentally different. For example, all of the acceptance tests can appear in a single program, since the entire program should compile and run as a valid program. However, the Pascal rejection test cannot practically be represented as a single program because it would (or should) consist of a repeated series of errors. The compiler could conceivably recover from the each error sufficiently to encounter and properly flag the next error, but this would not be reliable, and would vary from compiler to compiler. Further, such a test would test error recovery as well as simple error detection, which are two separate issues, and should be covered by two separate tests.

## The Pascal acceptance test

The Pascal acceptance test consists of program constructs that are correct for ISO 7185 Pascal. It should both compile and run under a ISO 7185 Pascal compatible compiler. The acceptance test is present in a single source file:

iso7185pat.pas

## The Pascal rejection test

The Pascal rejection test is arranged as a series of short tests because each test is designed to fail. Thus, ideally only one failure point at a time is executed.

The test script, runprt, is designed so that each result is checked for if an error occurred or not, with the lack of an error being a fail. However, there are at least two characteristics of a test result that go to quality, and thus imply that the result should be hand checked at least once:

1. The indicated error may or may not be appropriate for the error that was caused (for example, indicating a missing ':' when the problem was a missing ';').
2. The error may generate many collateral errors, or even refuse to compile the remainder of the program. This includes generating recycle balance errors.

The error signature of the compile can be compared to the previous runs, but a miscompare does not automatically fail the compiler, since error handling may have changed. The typical stragey is to run a new compiler version through the PRT test, and hand-examine only the error differences. If the changes in the compiler are minor, a compare go/no go will serve as a regression test.

Here are descriptions of each of the tests.

Class 1: Syntax graph visitation. These tests are for various points on the syntax graph, typically addition or elidation of a critical symbol. These tests are not meant to be exhaustive, since there are an infinite number of possible syntax constructions. Rather, it is designed to be representative.

The syntax visitation is according to the Pascal Users's manual and Report" 4th edition, Appendix D, Syntax diagrams. The tests run in reverse, starting with "program", and working backwards. Thus, the syntax checks are performed top to bottom.

Class 2: Semantic tests. These tests are for the meaning of the program. Examples include using an undefined variable, using a type in the incorrect context, executing a case statement where there is no case matching the selector, etc.

Note that the tests are not designed to be a complete set of failure modes for the compiler, nor is that possible, since the set of possible failure modes is infinite. Instead, this test is designed to be representative", or contain a series of known failure modes that test the quality of error checking and recovery in the compiler.

Note that the semantic errors are the same as those enumerated in the ISO 7185 standard in Appendix D, offset in number by the section 1700. Not all errors in this section have equivalent test cases. It is possible for an error defined by the standard to be without expression as a program. The description of the error is as was written in the ISO 7185 standard.

The ISO 7158 listed errors are a mix of compile time and runtime errors. On some errors, either mode is possible, on others, not. For these cases, we have tried to break the test program into two sections, one that could fail at compile time, and others that do not. These are labeled with an appended A, B, C, etc.

There can be other reasons to divide a test case into two programs.

With semantic errors, there is always the posibility that the code could be changed by an agressive compiler to eliminate or nulify the test case. There is no requirement in ISO 7185 that the compiler render into code a statement that clearly has no effect in the final program, just to achieve an error result. Its really not possible to defeat this kind of optimization entirely. Printing to output is one possibility, but even then there is nothing to prevent the compiler from assuming it knows what the output will be and changing it. This is perhaps a subject for further research.

### List of tests

#### Class 1: Syntatic errors

Program

iso7185prt0001 Missing semicolon after program statement

iso7185prt0002 missing "program" word-symbol

iso7185prt0003 missing program name word-symbol

iso7185prt0004 Moved

iso7185prt0005 Moved

iso7185prt0006 missing period

iso7185prt0007 Extra semicolon

iso7185prt0008 Missing header parameters

iso7185prt0009 Opening paren for program header only

iso7185prt0010 Closing paren for program header only

iso7185prt0011 Improperly terminated header list

iso7185prt0012 Consecutive semicolons

Block

iso7185prt0013 Missing number for label

iso7185prt0014 Missing semicolon after label

iso7185prt0015 Non-numeric label

iso7185prt0016 Unterminated label list

iso7185prt0017 Unstarted label list

iso7185prt0018 Missing constant

iso7185prt0019 Missing constant right side

iso7185prt0020 Missing "=" in const

iso7185prt0021 Incomplete second in const

iso7185prt0022 Missing ident in const

iso7185prt0023 Missing semicolon in const

iso7185prt0024 Reverse order between label and const

iso7185prt0025 Missing type

iso7185prt0026 Missing type right side

iso7185prt0027 Missing "=" in type

iso7185prt0028 Missing ident in type

iso7185prt0029 Incomplete second in type

iso7185prt0030 Missing semicolon in type

iso7185prt0031 Reverse order between const and type

iso7185prt0032 Missing var

iso7185prt0033 Missing var right side

iso7185prt0034 Missing var ident list prime

iso7185prt0035 Missing var ident list follow

iso7185prt0036 Missing ":" in var

iso7185prt0037 Missing ident list in var

iso7185prt0038 Incomplete second in var

iso7185prt0039 Missing semicolon in var

iso7185prt0040 Reverse order between type and var

iso7185prt0041 Missing "procedure" or "function"

iso7185prt0042 Missing ident

iso7185prt0043 Missing semicolon

iso7185prt0044 Consecutive semicolons start

iso7185prt0045 Missing block

iso7185prt0046 Missing final semicolon

iso7185prt0047 Misspelled directive

iso7185prt0048 Bad directive

iso7185prt0049 Misspelled procedure

iso7185prt0050 Misspelled function

iso7185prt0051 Bad procedure/function

iso7185prt0052 Missing ":" on return type for function

iso7185prt0053 Missing type id on return type for function

iso7185prt0054 Reverse order between var and procedure

iso7185prt0055 Reverse order between var and function

iso7185prt0056 missing begin word-symbol

iso7185prt0057 missing end word-symbol

Statement

iso7185prt0100 Missing label ident

iso7185prt0101 Missing label ":"

iso7185prt0102 Missing assignment left side

iso7185prt0103 Missing assignment ":="

iso7185prt0104 Missing procedure identifier

iso7185prt0105 Missing "begin" on statement block

iso7185prt0106 Misspelled "begin" on statement block

iso7185prt0107 Missing "end" on statement block

iso7185prt0108 Mispelled "end" on statement block

iso7185prt0109 Missing "if" on conditional

iso7185prt0110 Misspelled "if" on conditional

iso7185prt0111 Missing expression on conditional

iso7185prt0112 Missing "then" on conditional

iso7185prt0113 Misspelled "then" on conditional

iso7185prt0114 Misspelled "else" on conditional

iso7185prt0115 Missing "case" on case statement

iso7185prt0116 Misspelled "case" on case statement

iso7185prt0117 Missing expression on case statement

iso7185prt0118 Missing "of" on case statement

iso7185prt0119 Misspelled "of" on case statement

iso7185prt0120 Missing constant on case stament list

iso7185prt0121 Missing 2nd constant on case statement list

iso7185prt0122 Missing ":" before statement on case statement

iso7185prt0123 Missing ";" between statements on case statement

iso7185prt0124 Missing "end" on case statement

iso7185prt0125 Misspelled "end" on case statement

iso7185prt0126 Missing "while" on while statement

iso7185prt0127 Mispelled "while" on while statement

iso7185prt0128 Missing expression on while statement

iso7185prt0129 Missing "do" on while statement

iso7185prt0130 Missing "repeat" on repeat statement

iso7185prt0131 Misspelled "repeat" on repeat statement

iso7185prt0132 Missing "until" on repeat statement

iso7185prt0133 Misspelled "until" on repeat statement

iso7185prt0134 Missing expression on repeat statement

iso7185prt0135 Missing "for" on for statement

iso7185prt0136 Misspelled "for" on for statement

iso7185prt0137 Missing variable ident on for statement

iso7185prt0138 Misspelled variable ident on for statement

iso7185prt0139 Missing ":=" on for statement

iso7185prt0140 Missing start expression on for statement

iso7185prt0141 Missing "to"/"downto" on for statement

iso7185prt0142 Misspelled "to" on for statement

iso7185prt0143 Misspelled "downto" on for statement

iso7185prt0144 Missing end expression on for statement

iso7185prt0145 Missing "do" on for statement

iso7185prt0146 Misspelled "do" on for statement

iso7185prt0147 Missing "with" on with statement

iso7185prt0148 Misspelled "with" on with statement

iso7185prt0149 Missing first variable in with statement list

iso7185prt0150 Missing second variable in with statement list

iso7185prt0151 Missing "goto" in goto statement

iso7185prt0152 Misspelled "goto" in goto statement

iso7185prt0153 Missing unsigned integer in goto statement

iso7185prt0154 Missing 1st constant on case statement list

iso7185prt0155 Missing only variable in with statement list

iso7185prt0156 Missing ',' between case constants

iso7185prt0157 Missing ',' between variables in with statement

Field list

iso7185prt0200 Missing field ident

iso7185prt0201 Missing first field ident

iso7185prt0202 Missing second field ident

iso7185prt0203 Missing ':' between ident and type

iso7185prt0204 Missing type

iso7185prt0205 Missing ';' between successive fields

iso7185prt0206 Misspelled 'case' to variant

iso7185prt0207 Missing identifier for variant

iso7185prt0208 Missing type identifier with field identifier

iso7185prt0209 Missing type identifier without field identifier

iso7185prt0210 Missing 'of' on variant

iso7185prt0211 Misspelled 'of' on variant

iso7185prt0212 Missing case constant on variant

iso7185prt0213 Missing first constant on variant

iso7185prt0214 Missing second constant on variant

iso7185prt0215 Missing ':' on variant case

iso7185prt0216 Missing '(' on field list for variant

iso7185prt0217 Missing ')' on field list for variant

iso7185prt0218 Missing ';' between successive variant cases

iso7185prt0219 Attempt to define multiple variant sections

iso7185prt0220 Standard field specification in variant

iso7185prt0221 Missing ',' between first and second field idents

iso7185prt0222 Missing ',' between first and second field idents in variant

Procedure or Function Heading

iso7185prt0300 Missing 'procedure'

iso7185prt0301 Misspelled 'procedure'

iso7185prt0302 Missing procedure identifier

iso7185prt0303 Missing 'function'

iso7185prt0304 Misspelled 'function'

iso7185prt0305 Missing function identifier

iso7185prt0306 Missing type ident after ':' for function

Ordinal Type

iso7185prt0400 Missing '(' on enumeration

iso7185prt0401 Missing identifier on enumeration

iso7185prt0402 Missing 1st identifier on enumeration

iso7185prt0403 Missing 2nd identifier on enumeration

iso7185prt0404 Missing ')' on enumeration

iso7185prt0405 Missing 1st constant on subrange

iso7185prt0406 Missing '..' on subrange

iso7185prt0407 Missing 2nd constant on subrange

iso7185prt0408 Missing ',' between identifiers on enumeration

Type

iso7185prt0500 Missing type identifer after '^'

iso7185prt0501 Misspelled 'packed'

iso7185prt0502 Missing 'array'

iso7185prt0503 Missing '[' on array

iso7185prt0504 Missing index in array

iso7185prt0505 Missing first index in array

iso7185prt0506 Missing second index in array

iso7185prt0507 Missing ']' on array

iso7185prt0508 Missing index specification in array

iso7185prt0509 Missing 'of' on array

iso7185prt0510 Misspelled 'of' on array

iso7185prt0511 Missing type on array

iso7185prt0512 Missing 'file' or 'set' on file or set type

iso7185prt0513 Misspelled 'file' on file type

iso7185prt0514 Missing 'of' on file type

iso7185prt0515 Missing type on file type

iso7185prt0516 Misspelled 'set' on set type

iso7185prt0517 Missing 'of' on set type

iso7185prt0518 Missing type on set type

iso7185prt0519 Missing 'record' on field list

iso7185prt0520 Misspelled 'record' on field list

iso7185prt0521 Missing 'end' on field list

iso7185prt0522 Misspelled 'end' on field list

Formal Parameter List

iso7185prt0600 Missing parameter identifier

iso7185prt0601 Missing first parameter identifier

iso7185prt0602 Missing second parameter identifier

iso7185prt0603 Missing ',' between parameter identifiers

iso7185prt0604 Missing ':' on parameter specification

iso7185prt0605 Missing type identifier on parameter specification

iso7185prt0606 Missing parameter specification after 'var'

iso7185prt0607 Misspelled 'var'

iso7185prt0608 Missing ';' between parameter specifications

Expression

iso7185prt0700 Missing operator

iso7185prt0701 Missing first operand to '='

iso7185prt0702 Missing second operand to '='

iso7185prt0703 Missing first operand to '>'

iso7185prt0704 Missing second operand to '>'

iso7185prt0705 Missing first operand to '<'

iso7185prt0706 Missing second operand to '<'

iso7185prt0707 Missing first operand to '<>'

iso7185prt0708 Missing second operand to '<>'

iso7185prt0709 Missing first operand to '<='

iso7185prt0710 Missing second operand to '<='

iso7185prt0711 Missing first operand to '>='

iso7185prt0712 Missing second operand to '>='

iso7185prt0713 Missing second operand to 'in'

iso7185prt0714 Alternate '><'

iso7185prt0715 Alternate '=<'

iso7185prt0716 Alternate '=>'

iso7185prt0717 Missing first operand to 'in'

Actual Parameter List

iso7185prt0800 Empty list (parens only)

iso7185prt0801 Missing leading '(' in list

iso7185prt0802 Missing ',' in parameter list

iso7185prt0803 Missing first parameter in parameter list

iso7185prt0804 Missing second parameter in parameter list

iso7185prt0805 Missing ')' in list

Write Parameter List

iso7185prt0900 Empty list (parens only)

iso7185prt0901 Missing leading '(' in list

iso7185prt0902 Missing ',' in parameter list

iso7185prt0903 Missing first parameter in parameter list

iso7185prt0904 Missing second parameter in parameter list

iso7185prt0905 Field with missing value

iso7185prt0906 Fraction with missing value

iso7185prt0907 Field and fraction with missing field

iso7185prt0908 Missing ')' in list

Factor

iso7185prt1000 Missing leading '(' for subexpression

iso7185prt1001 Missing subexpression in '()'

iso7185prt1002 Misspelled 'not'

iso7185prt1003 'not' missing expression

iso7185prt1004 Missing '[' on set constant

iso7185prt1006 Missing first expression in range

iso7185prt1007 Missing second expression in range

iso7185prt1008 Missing '..' or ',' in set constant list

iso7185prt1009 Missing first expression in ',' delimited set constant list

iso7185prt1010 Missing second expression in ',' delimited set constant list

Term

iso7185prt1100 Missing first operand to '\*'

iso7185prt1101 Missing second operand to '\*'

iso7185prt1102 Missing first operand to '/'

iso7185prt1103 Missing second operand to '/'

iso7185prt1104 Missing first operand to 'div'

iso7185prt1105 Missing second operand to 'div'

iso7185prt1106 Missing first operand to 'mod'

iso7185prt1107 Missing second operand to 'mod'

iso7185prt1108 Missing first operand to 'and'

iso7185prt1109 Missing second operand to 'and'

Simple Expression

iso7185prt1200 '+' with missing term

iso7185prt1201 '-' with missing term

iso7185prt1203 Missing second operand to '+'

iso7185prt1205 Missing second operand to '-'

iso7185prt1206 Missing first operand to 'or'

iso7185prt1207 Missing second operand to 'or'

Unsigned Constant

iso7185prt1300 Misspelled 'nil'

Variable

iso7185prt1400 Missing variable or field identifier

iso7185prt1401 Missing '[' in index list

iso7185prt1402 Missing expression in index list

iso7185prt1403 Missing first expression in index list

iso7185prt1404 Missing second expression in index list

iso7185prt1405 Missing ',' in index list

iso7185prt1406 Missing ']' in index list

iso7185prt1407 Missing field identifier after '.'

Unsigned number

iso7185prt1500 Missing leading digit before '.'

iso7185prt1501 Missing digit after '.'

iso7185prt1502 Missing digit before and after '.'

iso7185prt1503 Mispelled 'e' in exponent

iso7185prt1504 Missing 'e' in exponent

iso7185prt1505 Missing digits in exponent

iso7185prt1506 Missing digits in exponent after '+'

iso7185prt1507 Missing digits in exponent after '-'

iso7185prt1508 Missing number before exponent

Character String

iso7185prt1600 String extends beyond eol (no end quote)

#### Class 2: Semantic errors

iso7185prt1701 For an indexed-variable closest-containing a single index-expression, it is an error if the value of the index-expression is not assignment-compatible with the index-type of the array-type.

iso7185prt1702 It is an error unless a variant is active for the entirety of each reference and access to each component of the variant.

iso7185prt1703 It is an error if the pointer-variable of an identified-variable denotes a nil-value.

iso7185prt1704 It is an error if the pointer-variable of an identified-variable is undefined.

iso7185prt1705 It is an error to remove from its pointer-type the identifying-value of an identified-variable when a reference to the identified-variable exists.

iso7185prt1706 It is an error to alter the value of a file-variable f when a reference to the buffer-variable f^ exists.

iso7185prt1707 It is an error if the value of each corresponding actual value parameter is not assignment compatible with the type possessed by the formal-parameter.

iso7185prt1708 For a value parameter, it is an error if the actual-parameter is an expression of a set-type whose value is not assignment-compatible with the type possessed by the formal-parameter.

iso7185prt1709 It is an error if the file mode is not Generation immediately prior to any use of put, write, writeln or page.

iso7185prt1710 It is an error if the file is undefined immediately prior to any use of put, write, writeln or page.

iso7185prt1711 It is an error if end-of-file is not true immediately prior to any use of put, write, writeln or page.

iso7185prt1712 It is an error if the buffer-variable is undefined immediately prior to any use of put.

iso7185prt1713 It is an error if the file is undefined immediately prior to any use of reset.

iso7185prt1714 It is an error if the file mode is not Inspection immediately prior to any use of get or read.

iso7185prt1715 It is an error if the file is undefined immediately prior to any use of get or read.

iso7185prt1716 It is an error if end-of-file is true immediately prior to any use of get or read.

iso7185prt1717 For read, it is an error if the value possessed by the buffer-variable is not assignmentcompatible with the variable-access.

iso7185prt1718 For write, it is an error if the value possessed by the expression is not assignment-compatible with the buffer-variable.

iso7185prt1719 For new(p,c l ,...,c n,), it is an error if a variant of a variant-part within the new variable becomes active and a different variant of the variant-part is one of the specified variants.

iso7185prt1720 For dispose(p), it is an error if the identifying-value had been created using the form new(p,c l ,...,c n ).

iso7185prt1721 For dispose(p,k l ,...,k, ), it is an error unless the variable had been created using the form new(p,c l ,...,c,,,) and m is equal to n.

iso7185prt1722 For dispose(p,k l ,...,k, ), it is an error if the variants in the variable identified by the pointer value of p are different from those specified by the case-constants k l ,...,k,,,,.

iso7185prt1723 For dispose, it is an error if the parameter of a pointer-type has a nil-value.

iso7185prt1724 For dispose, it is an error if the parameter of a pointer-type is undefined.

iso7185prt1725 It is an error if a variable created using the second form of new is accessed by the identified variable of the variable-access of a factor, of an assignment-statement, or of an actual-parameter.

iso7185prt1726 For pack, it is an error if the parameter of ordinal-type is not assignment-compatible with the index-type of the unpacked array parameter.

iso7185prt1727 For pack, it is an error if any of the components of the unpacked array are both undefined and accessed.

iso7185prt1728 For pack, it is an error if the index-type of the unpacked array is exceeded.

iso7185prt1729 For unpack, it is an error if the parameter of ordinal-type is not assignment-compatible with the index-type of the unpacked array parameter.

iso7185prt1730 For unpack, it is an error if any of the components of the packed array are undefined.

iso7185prt1731 For unpack, it is an error if the index-type of the unpacked array is exceeded.

iso7185prt1732 Sqr(x) computes the square of x. It is an error if such a value does not exist.

iso7185prt1733 For ln(x), it is an error if x is not greater than zero.

iso7185prt1734 For sqrt(x), it is an error if x is negative.

iso7185prt1735 For trunc(x), the value of trunc(x) is such that if x is positive or zero then 0 < x-trunc(x) < 1; otherwise 1 <x-trunc(x) < 0. It is an error if such a value does not exist.

iso7185prt1736 For round(x), if x is positive or zero then round(x) is equivalent to trunc(x+0.5), otherwise round(x) is equivalent to trunc(x- 0.5). It is an error if such a value does not exist.

iso7185prt1737 For chr(x), the function returns a result of char-type that is the value whose ordinal number is equal to the value of the expression x if such a character value exists. It is an error if such a character value does not exist.

iso7185prt1738 For succ(x), the function yields a value whose ordinal number is one greater than that of x, if such a value exists. It is an error if such a value does not exist.

iso7185prt1739 For pred(x), the function yields a value whose ordinal number is one less than that of x, if such a value exists. It is an error if such a value does not exist.

iso7185prt1740 When eof(f) is activated, it is an error if f is undefined.

iso7185prt1741 When eoln(f) is activated, it is an error if f is undefined.

iso7185prt1742 When eoln(f) is activated, it is an error if eof(f) is true.

iso7185prt1743 An expression denotes a value unless a variable denoted by a variable-access contained by the expression is undefined at the time of its use, in which case that use is an error.

iso7185prt1744 A term of the form x/y is an error if y is zero.

iso7185prt1745 A term of the form i div j is an error if j is zero.

iso7185prt1746 A term of the form i mod j is an error if j is zero or negative.

iso7185prt1747 It is an error if an integer operation or function is not performed according to the mathematical rules for integer arithmetic.

iso7185prt1748 It is an error if the result of an activation of a function is undefined upon completion of the algorithm of the activation.

iso7185prt1749 For an assignment-statement, it is an error if the expression is of an ordinal-type whose value is not assignment-compatible with the type possessed by the variable or function-identifier.

iso7185prt1750 For an assignment-statement, it is an error if the expression is of a set-type whose value is not assignment-compatible with the type possessed by the variable.

iso7185prt1751 For a case-statement, it is an error if none of the case-constants is equal to the value of the case-index upon entry to the case-statement.

iso7185prt1752 For a for-statement, it is an error if the value of the initial-value is not assignment-compatible with the type possessed by the control-variable if the statement of the for-statement is executed.

iso7185prt1753 For a for-statement, it is an error if the value of the final-value is not assignment-compatible with the type possessed by the control-variable if the statement of the for-statement is executed.

iso7185prt1754 On reading an integer from a textfile, after skipping preceding spaces and end-of-lines, it is an error if the rest of the sequence does not form a signed-integer.

iso7185prt1755 On reading an integer from a textfile, it is an error if the value of the signed-integer read is not assignment-compatible with the type possessed by variable-access.

iso7185prt1756 On reading a number from a textfile, after skipping preceding spaces and end-of-lines, it is an error if the rest of the sequence does not form a signed-number.

iso7185prt1757 It is an error if the buffer-variable is undefined immediately prior to any use of read.

iso7185prt1758 On writing to a textfile, the values of TotalWidth and FracDigits are greater than or equal to one ; it is an error if either value is less than one.

iso7185prt1759 The execution of any action, operation, or function, defined to operate on a variable, is an error if the variable is a program-parameter and, as a result of the binding of the program-parameter, the execution cannot be completed as defined.

#### Class 3: Advanced error checking

These are advanced error checks that the compiler may or may not check for.

iso7185prt1800 Access to dynamic variable after dispose.

#### Class 4: Field checks

These tests are a collection of issues found while running the compiler.

iso7185prt1900 Access to dynamic variable after dispose.

### Running the PRT and interpreting the results

There is a single script that runs the prt:

$ runprt

The results of all of the tests are gathered and formatted into the file:

iso7185prt.lst

This file has several sections:

1. List of tests with no compile or runtime error.
2. List of differences between compiler output and “gold” standard outputs.
3. List of differences between runtime output and “gold” standard outputs.
4. Collected compiler listings and runtime output of all tests.

Each of these will be examined in turn.

#### List of tests with no compile or runtime error.

The purpose of the rejection test is to cause an error in the test program, either at compile time or runtime, and test its proper handling. By definition, if no error exists, the test has failed. This list gives the tests that need to be examined for why they didn’t detect an error.

#### List of differences between compiler output and “gold” standard outputs.

Shows the number of lines difference between the output test.err or compiler output listing, and test.ecp or “gold standard” compiler output listing, for each test program. A difference of 0 means that nothing has changed in the run.

The compiler output listing flagging an error and the run output flagging an error are mutually exclusive. If the compiler listing shows an error, the program will not be run. If if the compiler output shows an error, it is examined to see if it flagged an appropriate error for the fault, and that it didn’t generate excessive “cascade” or further errors caused by the compiler having difficulty recovering from the error.

Once the test.err file is judged satisfactory, it is copied to the test.ecp file as the “gold” standard. If the test run shows a difference between the current compile and the gold standard file, it does not necessarily mean that it is wrong. It simply means it needs to be reevaluated and perhaps copied as the new gold standard file.

#### List of differences between runtime output and “gold” standard outputs.

If the test file is successfully compiled, it is run and the output collected as test.lst. This is compared to the “gold” run output file test.cmp.

The run output file, if it exists, is checked to see if it indicates an appropriate error for the fault indicated in the test. If the run indicates no error, or an error that is not related to the fault, or simply crashes, that is a failure to properly handle the fault.

Once the test.lst file is judged satisfactory, it is copied to the test.cmp file as the “gold” standard. If the test run shows a difference between the current output and the gold standard file, it does not necessarily mean that it is wrong. It simply means it needs to be reevaluated and perhaps copied as the new gold standard file.

#### Collected compiler listings and runtime output of all tests.

The output of the compiler listing, as well as the output of the run, if that exists, for each test is concatenated and placed at the end of the listing file. This allows the complete output of each test to be examined, without having to look at the individual test.err or test.list file.

### Overall interpretation of PRT results

The rejection test is a “quality” test. There is no absolute right and wrong. The clearest indication of failure is failure to find any error. Past that are issues that indicate the quality of the error handling in the compiler:

1. If the error is indicative of what the failure was.
2. No or few further errors resulted after the failure.
3. The compiler was able to resyncronise with the source.
4. No data was corrupted as a result of the run (dynamic space imbalance, null pointer errors or similar).

You can say that failure to achieve the above marks a poor quality compiler, but not a failing test.

## Sample program tests

The sample program tests give a series of sample programs in Pascal. The idea of the sample programs tests is to prove out operation on common programs, and also to give a newly modified version of P6 a series if tests progressing from the most simple (“hello world”) to more complex tests. If the new version of the compiler has serious problems, it is better to find out with simple tests rather than have it fail on a more complex, and difficult to debug, test.

Hello Gives the standard “hello, world” minimum test.

Roman Prints a series of roman numerals. From Jensen and Wirth’s “User Manual and report”.

Match A number match game, this version from Conway, Gries and Zimmerman “A primer on Pascal”.

Startrek Plays text mode startrek. From the internet.

Basics Runs a subset Basic interpreter. It is tested by running a recoded version of “match” above.

Pascal-S Runs a subset of ISO 7185 Pascal. From Niklaus Wirth’s work at ETH. It is tested by running the “Roman” program above.

## Previous Pascal-P versions test

As part of the regression tests, Pascal-P6 runs the older versions of itself, namely Pascal-P2 and Pascal-P4. These are the only versions of the compiler available. See the section “introduction” on page 10, and also the historical material on Pascal-P on the Standard Pascal website.

The run of previous versions of Pascal-P perhaps constitutes the purest form of regression test. It not only insures that P6 is compatible with previous versions, but that it can actually compile and run all of the previous code. Of course, this is possible in main because these 1970’s versions were adapted to the ISO 7185 standard, but that, fortunately, was a small change.

### Compile and run Pascal-P2

P2 runs on P6 virtually unmodified. It needed only the declarations of the prd and prr files removed. This is required because they are predefined in P6. In a stand-alone use of P2, those files are real external files and have to be declared.

P2 runs as a test a modified version of roman.pas. It has to be modified because of:

1. P2 only supports upper case.
2. P2 does not support the output file as a default parameter to writeln.

These changes are documented in the P2 archive on [http://www.standardpascal.org](http://www.standardpascal.org/).

P2 does not have a full regression test of its own because I didn’t do the work of cutting down the ISO 7185 test to the subset that P2 implements, as I did for P4. This is perhaps a future project.

### Compile and run Pascal-P4

P4 has the same modification of the prd and prr files as P2, but otherwise runs unmodified. P4 runs as its target standardp.pas, which was a version of iso7185pat.pas that was stripped so as to fit the subset of ISO 7185 that P4 runs. Recall that P4 does not run any standard of Pascal at all, it is intentionally a subset.

Standard.pas was modified only in that P4 running under P6 cannot execute a for loop of the form:

for b := false to true do ...

The reason for this is non-trival. The way that P4 executes such a loop is to check that b lies in the range of boolean 0..1, and terminate if it lies outside that range. This actually makes it an illegal program if variable b is treated as boolean, since b will be incremented to 2 and thus be an out of range value.

This is actually a standard issue with Pascal in general, and the ISO 7185 standard took pains to show an equivalent form of the for loop that does not involve creating an out of range boolean. P6 also uses a form internally that fixes this, it has to in order to fully comply with the ISO 1785 standard.

So why does that work with a standalone P4 and not with a P4 running under P6? P4, as well as P6, treat the internal data store as typeless and use type escapes in the form of undiscriminated variants in order to differentiate the different data forms. This means that the result of assigning 2 to b (or incrementing it from 1) are system and compiler dependent. Thus, the direct code generating compilers tolerate this, and P6 simply does not.

## Self compile

One of the more difficult tests for P6 (and the most time consuming) is to have P6 compile and run itself. Actually the entire idea of Pascal-P was to compile and run itself in order to accomplish a bootstrap of the compiler. Pascal-P was never provided in a form able to directly compile itself, it needed a few modifications in order to do that. Also, self compiling a compiler that interprets its final code is different from a machine code generating compiler. Whereas it is conceptually simple to imagine the parser and intermediate code generator compling a version of itself, having the interpreter run another instance of itself on itself is like looking into a series of mirrors. The interpreter will be interpreting itself while that interpreter interprets another program, etc.

This can actually be carried out to any depth of self-interpretation, but because each interpretation level slows down the code by an order of magnitude, such nested self interpretation rapidly becomes impractical to complete. As it is, a single level of self-interpretation takes hours.

The reason to put work into a self compile of P6 is that it is a difficult test that goes a long way to prove out the stability and capacity of the compiler, and also because it proves out a very important function of Pascal-P, that of bootstrapping itself.

### pcom

I was able to get pcom.pas to self compile. This means to compile and run pcom.pas, then execute it in the simulator, pint. Then it is fed its own source, and compiles itself into intermediate code. Then this is compared to the same intermediate code for pcom as output by the regular compiler. Its a good self check, and in fact found a few bugs.

The Windows batch file to control a self compile and check is:

cpcoms.bat

What does it mean to self compile? For pcom, not much. Since it does not execute itself (pint does that), it is simply operating on the interpreter, and happens to be compiling a copy of itself.

#### Changes required

pcom won't directly compile itself the way it is written. The reason is that the "prr" file, the predefined file it uses to represent it's output file is only predefined to P6 itself. The rules for ISO 7185 are that each file that is defined in the header which is not predefined, such as "input" or "output", must be also declared in a var statement. This makes sense, because if it is not a predefined file, the compiler must know what type of file (or even non-file) that is being accessed externally to the program.

Because the requirements are different from a predefined special compiler file to a file that is simply external, the source code must be different for a P6 file vs. another compiler. A regular ISO 7185 Pascal compiler isn't going to have a predefined prr file.

The change is actually quite small, and marked in the source. You simply need to remove, or comment out, the following statements:

{ !!! remove this statement for self compile }

prr: text;                      { output code file }

and

{ !!! remove this statement for self compile }

rewrite(prr); { open output file }

In the batch file above, this modified file is represented as pcomm.pas, or "modified" pcom.pas.

All of the source code changes from pcom.pas to pcomm.pas are automated in cpints.bat.

### pint

Pint is more interesting to self compile, since it is running (being interpreted) on a copy of itself. Unlike the pcom self compile, pint can run a copy of itself running a copy of itself, etc., to any depth. Of course, each time the interpreter runs on itself, it slows down orders of magnitude, so it does not take many levels to make it virtually impossible to run to completion. Ran a copy of pint running on itself, then interpreting a copy of iso7185pat. The result of the iso7185pat is then compared to the "gold" standard file.

As with pcom, pint will not self compile without modification. It has the same issue with predefined header files. Also, pint cannot run on itself unless its storage requirements are reduced. For example, if the "store" array, the byte array that is used to contain the program, constants and variables, is 1 megabyte in length, the copy of pint that is hosted on pint must have a 1 megabyte store minus all of the overhead associated with pint itself.

The windows batch file required to self compile pint is:

cpints.bat

As a result, these are the changes required in pint:

{ !!! Need to use the small size memory to self compile, otherwise, by

  definition, pint cannot fit into its own memory. }

maxstr      = 2000000;  { maximum size of addressing for program/var }

{maxstr      = 200000;}  { maximum size of addressing for program/var }

and

{ !!! remove this next statement for self compile }

prd,prr     : text;(\*prd for read only, prr for write only \*)

and

{ !!! remove this next statement for self compile }

reset(prd);

and

{ !!! remove this next statement for self compile }

rewrite(prr);

All these changes were made in the file pintm.pas.

Pint also has to change the way it takes in input files. It cannot read the intermediate from the input file, because that is reserved for the program to be run. Instead, it reads the intermediate from the "prd" header file. The interpreted program can also use the same prd file. The solution is to "stack up" the intermediate files. The intermediate for pint itself appears first, followed by the file that is to run under that (iso7185pat). It works because the intermediate has a command that signals the end of the intermediate file, "q". The copy of pint that is reading the intermediate code for pint stops, then the interpreted copy of pint starts and reads in the other part of the file. This could, in fact, go to any depth.

All of the source code changes from pint.pas to pintm.pas are automated in cpints.bat.

Self compiled files and sizes

The resulting sizes of the self compiled files are:

pcomm.P6

Storage areas occupied

|  |  |  |
| --- | --- | --- |
| **Contents** | **Range of storage** | **Net size** |
| **Program** | 0-114657 | (114658) |
| **Stack/Heap** | 114658-1987994 | (1873337) |
| **Constants** | 1987995-2000000 | (  12005) |

pintm.P6

Storage areas occupied

|  |  |  |
| --- | --- | --- |
| **Contents** | **Range of storage** | **Net size** |
| **Program** | 0- 56194 | (56195) |
| **Stack/Heap** | 56195-1993985 | (1937791) |
| **Constants** | 1993986-2000000 | (6014) |

1. Many Pascal compilers that feature an ISO 7185 standards compliance flag in fact change the source language to be incompatible with ISO 7185 if the flag is off. Pascal-P6 does not do this. [↑](#footnote-ref-2)
2. The term “dialect” is the correct term for a variant of a language that is not close enough to be mutually understood between different speaking peoples or computer language implementations. [↑](#footnote-ref-3)
3. Although UCSD Pascal is no longer in wide use, that language had much of its constructs copied into other dialects, including Borland, Apple Pascal and many others. [↑](#footnote-ref-4)
4. Typically 8 bit microprocessors and 8 bit microprocessors use the same word length, because the word length goes by address capability, not register length. [↑](#footnote-ref-5)
5. Although it is certainly fair to say that this is not a requirement for a compiler. IP Pascal and I am sure others have an intermediate code that does not rely on machine level characteristics. [↑](#footnote-ref-6)
6. As indeed, original Pascal does not have by design. This may come as a shock to C programmers. [↑](#footnote-ref-7)
7. You may also use LLVM compiler/assembler, which is compatible. [↑](#footnote-ref-8)