

# Progress in Pascal compiler for HP 48/38 calculators

YUAN, Feng  
Software Engineer  
(Hewlett Packard Singapore)  
email: yuanfeng@hpsgrt1.sgp.hp.com

Paper presented at 1995 HP Handheld Users Conference  
Aug 5&6, 1995, Mall of America, Bloomington, Minnesota

## Abstract

This paper continues discussion of how Pascal programs can be compiled to system RPL code, and how applications for the HP48S/48G and the latest HP38G calculator can be developed in Pascal.

### 1. Introduction

The popular HP 48S/48G series of calculator is built using mainly a special programming language system RPL, so is the latest HP graphical calculator HP38G. Although system RPL is a very powerful language, especially in dynamic memory management, versatile control structure, symbolic processing, error handling, etc., programming in system RPL is very hard.

Recently, there have been some interesting development in compiling C++ into Saturn Assembly language, and compiling Pascal into system RPL. This paper continues discussion reported in a paper submitted to Prompt 94 Conference on compiling Pascal into system RPL and shows how Pascal can be used to write applications for HP38G ( also called aplets).

### 2. Compilation of Pascal constructs

System RPL is an intermediate code that works on a software emulated stack machine. Another popular stack based intermediate language is P\_code, which was first used in the public domain P4 compiler written by Urs Ammann and others from ETH Zurich and documented in S.Pemberton's book **Pascal Implementation**. The work reported here is based on the P4 compiler in the front end with substantial redesign in the code generation part.

Compiling of major Pascal language constructs into system RPL can be illustrated below use simple examples.

#### Integer operations

Although it sounds very simple, compiling of Pascal **integer** into RPL is not straightforward. This is because RPL system only supports 20 bits unsigned integer operations, and conversion routines between real and unsigned integers. The following routines have been created to support signed **integer** operations:

#ODD	( d -> T/F )	<b>integer</b> odd test
#NEG	( d -> d )	<b>integer</b> negation
#ABS	( d -> d )	<b>integer</b> absolute value
d>	( d d -> T/F )	<b>integer</b> > comparison
d<	( d d -> T/F )	<b>integer</b> < comparison
d>=	( d d -> d>= )	<b>integer</b> >= comparison
d<=	( d d -> d<= )	<b>integer</b> <= comparison
d*	( d d -> d )	<b>integer</b> multiplication
dDIV	( d d -> d )	<b>integer</b> division
dMOD	( d d -> d )	<b>integer</b> remainder
d>%	( d -> % )	<b>integer</b> to <b>real</b> conversion
%>d	( % -> d )	<b>real</b> to <b>integer</b> conversion

### Local variable and function definition

Local variables are allocated on the data stack together with parameters, so is function return value.

For example, the following Pascal function declaration:

```
function p(a,b,c: real): real;
begin
  p:=sqrt(b*b-4.0*a*c);
end;
```

can be compiled into:

```
NAMELESS proc_3 ( a b c -> sqrt(b*b-4.0*a*c) )
::
  ZERO                                ( a b c ZERO )
  3PICK 4PICK %*                      ( a b c ZERO b*b )
  %4 6PICK %* 4PICK %* %- %sqrt      ( a b c zero p )
  SWAPDROP                           ( a b c p )
  4UNROLL3DROP                       ( p )
;
```

### Global variables

For HP48 calculators, global variables can be stored into temporary environment that is created on entry into program and destroyed on quit from program. Global variables can be accessed using GETLAM and PUTLAM. For HP38 calculators, there is a special region of updateable pointers TopicVar1 through TopicVar91 that can be used by individual applications ( aplets in HP38G's term ). This provides a much faster way to access global variables through special routines like TopicVar1@, TopicVar1!, etc.

```
program test(input,output);
var a,b,p: real;
begin
  p:=a+b;
end.

NAMELESS proc_3 ( -> )
::
  %0AllTopicVars                     ( CLEAR ALL TOPICVARS )
  TopicVar3@ TopicVar2@ %+          ( a+b )
  TopicVar1!
  %0AllTopicVars
;
```

### Array, Record and String

**Array** as in Pascal is a composite data type, with each element being a valid data object. In current implementation, Pascal **array** is mapped to a chunk of stack locations or topicvars. The basic program of compiling **array** access is to generate code to compute the address of `a[i]`, which may be defined as:

```
a: array [min..max] of type;
```

For local variable access, we need to generate code for computing offset of `a[i]` as:

```
offset(a[i])=offset(a)-(i-min)*sizeof(type)
              =(offset(a)+min)*sizeof(type)-i*sizeof(type);
```

Compilation of **record** is quite straightforward, because only fixed offsets are involved. The following example contains both **array** and **record**:

```
var p: array [1..5] of record x,y: integer end;

procedure movex(i,dx: integer);
begin
  p[i].x:=p[i].x+dx
end;

NAMELESS _proc_3                                ( i dx -> )
::
  OVER TWO d* ZERO #+ TopicVarN@                ( i dx p[i] )
  OVER #+                                         ( i dx p[i]+dx )
  3PICK TWO d* -1 #+ #1+ TopicVarN!              ( i dx )
  2DROP                                           ( i dx )
;
```

It would be very inefficient if **string** type is also handled in the same way as **array**, because it would require 81 updateable pointers or LAMs to implement a variable length **string** with maximum length of 80 characters. A much efficient way to implement **string** is to map it to RPL character string object as illustrated below:

```
program stringtest;
  var s:string;
begin
  s:=' Pascal'+ '-' + 'RPL';
  s[1]:=chr(length(s));
end.

NAMELESS _proc_24 ( -> )
::
  $ " Pascal"                                     ( string constant )
  FORTYFIVE                                       ( ASCII code for '-' )
  >T$                                             ( append character to string )
  $ "RPL" &$                                     ( string concatenation )
  TopicVar1!                                     ( assign to s )
  TopicVar1@ ONE                                 ( s 1 )
  TopicVar1@ LEN$ #>CHR                          ( s 1 chr(length(s)) )
  CHR># $PutByte
;
```

where `$PutByte` is a newly added Saturn assembly routine to overwrite a character within a string, which can also be interpreted as changing one byte within a byte array:

```
$PutByte ( $ index #ch -> )
```

## Control Structures

System RPL has a very rich set of control structure words like , SEMI, COLA, RDROP, SKIP, IT, ITE, DO\_LOOP, BEGIN\_WHILE\_REPEAT, BEGIN\_AGAIN, BEGIN\_UNTIL, case and even GOTO. We base our implementation here on relative jumps, as an optimization for space system RPL control constructs can also be generated when possible. Three new routines ( JMP, FJMP, CaseJump) are defined to implement Pascal control structures: IF\_THEN, IF\_THEN\_ELSE, WHILE\_DO, REPEAT\_UNTIL and CASE.

JMP expects a binary integer object as next object in the run stream, adding this offset to the next RPL instruction pointer will get the target instruction pointer.

FJMP expects a T/F flag on the stack and a binary integer object as next object in the run stream. If the flag is FALSE, JMP is performed, otherwise the binary integer object is skipped and execution continues.

CaseJump expects a binary integer on the stack and a character string in the run stream. The integer is an index into a series of relative jumping addresses encoded in the character string. Default jump address is stored first in the character string.

The following example demonstrates compilation for major Pascal control structures:

```

program control;
  var i,n: integer;
  begin
    i:=1961;
    for n:=1 to 20 do
      if odd(i) then i:=i*3+1 else i:=i div 2;

    case i of
      1: while i<10 do i:=i+1;
      2: repeat i:=i-1 until I=0
    end
  end.

NAMELESS _proc_3 ( -> )
  ::
    ZERO
    1961 TopicVar2!
    ONE TopicVar1!
    TWENTY SWAPDROP
  LOCALLABEL lab_4
    DUP TopicVar1@ d>=
    FJMP DOBINT
  ASSEMBLE
    REL(5) lab_5
  RPL
    TopicVar2@ #ODD ITE
    :: TopicVar2@ THREE d* #1+ ;
    :: TopicVar2@ TWO dDIV ;
    TopicVar2!
    TopicVar1@ #1+ TopicVar1!
    JMP DOBINT
  ASSEMBLE
    REL(5) lab_4
  RPL

  LOCALLABEL lab_5
    TopicVar2@ #1- CaseJump
  ASSEMBLE
    CON(5) =DOHSTR
    CON(5) 20

```

( \* for \*)  
( \* if\_then\_else \*)  
( \* case \*)  
( \* while \*)  
( \* repeat \*)  
( alloc for\_end index )  
( i:=1961 )  
( n:=1 )  
( for\_end:=20 )  
( for\_end>=i )  
( if not goto end\_loop )  
( ITE odd(i) )  
( I\*3+1 )  
( I div 2 )  
( I:= )  
( n:=n+1 )  
( loop\_back )  
( case i:zero\_based )  
( three cases )

```

REL(5) lab_9                ( default )
REL(5) lab_10               ( i'=0 )
REL(5) lab_13               ( i'=1 )

RPL
LOCALLABEL lab_10
BEGIN                        ( system RPL control )
  TopicVar2@ TEN d<
  WHILE                      ( while I<10 )
    TopicVar2@ #1+ TopicVar2! ( i:=i+1 )
  REPEAT                     ( loop_back )
    JMP DOBINT               ( goto end_case )
ASSEMBLE
  REL(5) lab_9

RPL
LOCALLABEL lab_13
BEGIN                        ( system RPL control )
  TopicVar2@ #1- TopicVar2!   ( i:=i-1 )
  TopicVar2@ ZERO #=
  UNTIL                      ( until I=0 )
    JMP DOBINT               ( goto end_case )
ASSEMBLE
  REL(5) lab_9

RPL
LOCALLABEL lab_9
DROP                         ( drop for_end index )
;

```

### 3. Optimizations

Although the code examples shown above still look quite naive to experienced system RPL programmers or any one with some basic knowledge of modern compilers features, some level of optimization has already been implemented.

The original P4 compiler generates P\_code by writing text representation of P\_codes directly into output file, without any optimization. Generating text output also makes optimization very hard. To be able to extend the limited P\_code instruction set to the rich system RPL instruction set and optimize the code generation, the code generation phase of the P4 compiler has been rewritten to generate binary code. A separate code optimization and ( text form) system RPL generation phase is added.

The following words have been added to extend P\_code, all of them have corresponding system RPL words:

it	IT
semi	;
ite	ITE
colon	::
begin	BEGIN
while	WHILE
repeat	REPEAT
until	UNTIL
nop	NOP
tick	
list	DOLIST
endlist	SEMI
takeoverTakeOver	
getch	\$Putbyte
putch	\$SUB1#

The following library function words have been added to extend Pascal built\_in functions and implement **string** as a simple object:

push	NOP ( ob -> ob )
------	------------------

pop	DROP
length	LEN\$
asin	%ASIN
log	%LOG
tan	%TAN
acos	%ACOS
sinh	%SINH
cosh	%COSH
tanh	%TANH
asinh	%ASINH
acosh	%ACOSH
atanh	%ATANH
chrstr	>H\$
strchr	>T\$
strstr	&\$

Most of the currently implemented peephole optimizations can be described using the following pattern and replacement rules:

<const> <const> #+	=> <const>+<const>
<const> <const> #*	=> <const>*<const>
<const> <const> #-	=> <const>-<const>
<object> flo	=> flt <object>
<const> chs	=> -<const>
<const> #+	=> inc(<const>)
<const> #-	=> dec(<const>)
<const> <object> #+	=> <object> inc(<const>)
0 <object> #+	=> <object>
<const> <object> <const> > + +	=> <object> inc <const>+<const>
<const> <object> <const> > - +	=> <object> inc <const>-<const>
<const> ind	=> ldo(<const>)
<const> <object> str	=> <object> sro(<const>)
if . then . ob else . ob endif	=> if . then . else . endif ob

Optimizations as currently implemented are far from enough. System RPL programs generated by current implementation can be reduced by as much as 40%, as shown by the following manually rewritten version of the control structure example shown above:

NAMELESS _proc_3 ( -> )	
::	
1961 TopicVar2!	( i:=1961 )
TWENTYONE ONE_DO (DO)	( for n:=1 to 20 do )
TopicVar2@ DUP #ODD ITE	( ITE odd(i) )
:: #3* #1+ ;	( i*3+1 )
#2/	( i div 2 )
TopicVar2!	( i:= )
LOOP	
TopicVar2@ #1=case ::	
BEGIN	
TopicVar2@ TEN d<	
WHILE	( while I<10 )
TopicVar2@ #1+ TopicVar2!	( i:=i+1 )
REPEAT	( loop_back )
;	
TopicVar2@ #2= NOT?SEMI	
BEGIN	( system RPL control )
TopicVar2@ #1- TopicVar2!	( i:=i-1 )
TopicVar2@	
#0=UNTIL	( until i=0 )
;	

This also shows that deep understanding of system RPL is still very useful even when languages as Pascal can be compiled into system RPL automatically. We may even say knowledge of system RPL is more useful than ever before because Pascal compiler opens a bigger entrance into the wonderful world of system RPL. This is achieved through Pascal compiler's variable allocation, static type checking, parameter checking, address calculation, infix expression to postfix translation, etc., which are handled manually in traditional system RPL program development. It will be shown later in this paper how Pascal programs can be debugged first on the DOS platform before they are compiled into system RPL.

#### 4. Generating HP48/38 object code

To make the program runnable on HP48/38 calculators, all the pieces described above should be glued together into system RPL external library object. External library should contain library name, library id, configuration code, hash table, link table, and 4\_nibble CRC field besides the actual code. A file header is also needed for downloading.

There are several steps involved in generating HP48/38 object code from the original Pascal source code:

- PAS2RPL: syntax check, binary internal code generation
- GENRPL: optimization, system RPL source generation
- RPLCOMP: Saturn assembly source generation
- SASM: Saturn object code generation
- SLOAD: Generating HP48/38 downloadable object code

Let's take a simple program:

```
program main(input,output);
var n: integer;
function add1(i: integer): integer;
begin
  add1:=i+1;
end;
begin
  n:=add1(10);
end.
```

System RPL source generated by GENRPL will look like:

```
xROMID 2FC                                ( external ROMID )

ASSEMBLE
NIBASC /HHP48-A/                          ( binaray header )
=ROMID2FC EQU #2FC
CON(5) =DOLIB                             ( library prologue )
=LIB2FC REL(5) =SYSEND2FC
CON(2) 10                                ( library name )
NIBASC \PascalDemo\
CON(2) 10
=SYS2FC CON(3) #2FC
REL(5) =Hash2FC
CON(5) 0
REL(5) =LinK2FC
REL(5) =CnfG2FC

=CnfG2FC                                ( configuration code )
RPL
:: #2FC XEQSETLIB ;
```

```

NULLNAME proc_3                                ( procedure add1 )
:: #1+ ;

ASSEMBLE
CON(1)      8
RPL
xNAME Demo                                ( main program )
:: ZERO ' NULLLAM ONE DOBIND              ( allocate globals )
TEN proc_3 1PUTLAM                        ( main action )
ABND                                             ( deallocate globals )
;

ASSEMBLE                                ( hash table )
=Hash2FC CON(5) =DOHSTR
REL(5) TaBlEnD
CON(5) 0
CON(5) 0
CON(5) 0
REL(5) oBleN4
...
REL(5) naMEend
oBleN4 CON(2) 4
NIBASC \Demo\
CON(3) 0
naMEend
=~xDemo EQU #2FC+4096*0 ( romptr def )
CON(5) (*)-(oBleN4)-0
=~proc_3 EQU #2FC+4096*1
TaBlEnD

=LinK2FC CON(5) =DOHSTR
REL(5) enDLink
REL(5) =xDemo
REL(5) =proc_3
enDLink CON(4) 0 ( crc )
=SYSEND2FC END
RPL

```

When the final object code is downloaded into HP48 and installed into Port0, you can run the program by typing Demo from keyboard or choose it from the port menu.

The latest HP38G graphic calculator is based on a concept called aplet, which is a small application packaged as an electronic lesson. Aplet has a data storage part containing information like plot setups, numeric setup, symbolic expressions, text note and graphical sketches. It has a code part that supports the running of 5 main views and 3 setup views of aplets, where people can do setup and explore. The aplet data part can have multiple instances to solve different problems of the same class. To be more specific, aplets can have:

- symbolic view, for entering functions and manipulating them
- plotting view, for generating graphs of functions
- numeric view, for generating tables of functions
- note view, for reading and editing text description of problem
- sketch view, for watching and editing sketch description of problem
- symbolic setup view
- plot setup view
- numeric setup view

Down to the implementation level, aplet data are stored into a system RPL directory object with fixed number of nameless slots. Aplet code is packaged into a system RPL library object that is referenced to by aplet directory. Aplet code should be able to supply specific view handlers when certain predefined view keys are pressed.



To benefit from the new HP38 aplet structure, the Pas2Rpl compiler can generate a simple degenerated form of aplets that has the code part embedded into the directory part, making it a single object. This simple form of aplets still supports common aplet variables like Xmin, Xmax, Notetext, Pagenum, Page. It supports note and sketch view, but all the other views are mapped into a single view by default. Experienced system RPL programmers will be able provide different handling to different views by working on the system RPL source code generated.

As the final example of this paper, let's try a big' yet still toy program. The vector program shown below stores coordinates of 5 points of a pyramid, displays it on screen, and lets the user to rotate it interactively around x, y, z axes in steps of 15 degrees.

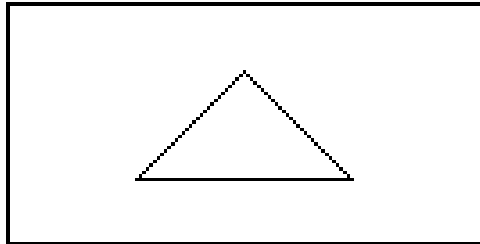


Fig 1: initial screen

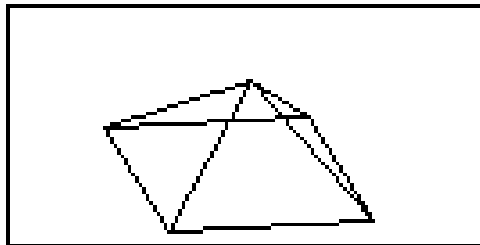


Fig 2: sample screen after rotation

```
program vector(input,output);

uses lcd;

const sina=0.258819;
      cosa=0.965926;

var   obj : array [1..5,1..3] of real;
      x   : array [1..5] of integer;
      y   : array [1..5] of integer;

procedure init; far;
begin
  open_lcd;

  obj[1,1]:= 5.0; obj[1,2]:=-5.0; obj[1,3]:= 2.5;
  obj[2,1]:= 5.0; obj[2,2]:= 5.0; obj[2,3]:= 2.5;
  obj[3,1]:=-5.0; obj[3,2]:= 5.0; obj[3,3]:= 2.5;
  obj[4,1]:=-5.0; obj[4,2]:=-5.0; obj[4,3]:= 2.5;
  obj[5,1]:= 0.0; obj[5,2]:= 0.0; obj[5,3]:=-2.5;
```

```

end;

procedure connect(i,j: integer);
begin
  lineon(x[i],y[i],x[j],y[j])
end;

procedure display; far;
var i: integer;
begin
  for i:=1 to 5 do
    begin
      x[i]:=trunc(obj[i,2] * 6) + 64;
      y[i]:=trunc(obj[i,3] * 6) + 32;
    end;
  clear(0,0,131,64);
  connect(1,2); connect(2,3); connect(3,4); connect(4,1);
  connect(5,1); connect(5,2); connect(5,3); connect(5,4)
end;

procedure rotate(s: real; x,y: integer);
var i: integer; xx,yy: real;
begin
  for i:=1 to 5 do
    begin
      xx:=obj[i,x]; yy:=obj[i,y];
      obj[i,x]:=xx*cosa - yy*sina*s;
      obj[i,y]:=xx*sina*s + yy*cosa
    end;
  end;

procedure handle; far;
begin
  case key of
    key_1 : rotate(-1.0,2,3);
    key_3 : rotate( 1.0,2,3);
    key_2 : rotate(-1.0,3,1);
    key_8 : rotate( 1.0,3,1);
    key_6 : rotate(-1.0,2,1);
    key_4 : rotate( 1.0,2,1);
  end;
  keymap
end;

begin
  setviewui(init,close_lcd,display,handle)
end.

```

View handler required by HP38 is similar to the parameterized outer loop user interface specification required by HP48 parameterized outer loop. A simplified version is supported here by Pas2Rpl compiler through the routine **setviewui**. **Setviewui** expects four parameter\_less procedures as view entry routine, view exit routine, view display handler and view keyboard handler.

In the vector example, view entry routine **init** clear screen and all topic variables, initializes global variables used by the program; view exit routine **close\_lcd** clears topic variables again and signals screen should be updated; display handler **display** redraw the pyramid; key handler **handle** takes over 6 keys for rotating, handle pre\_defined task switch keys and menu keys (if they are defined) , shows visual warning for other keys pressed. The compiler has special code generation algorithm for key handler and the **setviewui** procedure call. The final object code is 1416 bytes in size.

The procedure handler will be compiled into:

```
NAMELESS _proc_32 ( #code #plane -> keyob TRUE | FALSE )
::
  { { FORTYONE      :: TakeOver %-1 TWO   THREE proc_29 ; ( kc1 )
    FORTYTHREE     :: TakeOver %1  TWO   THREE proc_29 ; ( kc3 )
    FORTYTWO       :: TakeOver %-1 THREE ONE   proc_29 ; ( kc2 )
    THIRTYTWO      :: TakeOver %1  THREE ONE   proc_29 ; ( kc8 )
    THIRTYEIGHT    :: TakeOver %-1 TWO   ONE   proc_29 ; ( kc6 )
    THIRTSIX       :: TakeOver %1  TWO   ONE   proc_29 ; ( kc4 )
    }
    { } { } { } { } { } { }      ( ls, rs, ans, als, ars planes )
  }
  ONE KeyFace                    ( HP38 built_in key handling routine )
;
```

The call to **setviewui** will be compiled into:

```
' proc_24
' LeaveGraphView
' proc_26
' proc_32
DummyMenuErr
```

where DummyMenuErr supplies a dummy menu and error handler.

While Pascal compiler can do a nice job of detecting syntax errors, programs generated correctly by Pas2Rpl compiler can easily fail badly when run on HP calculators because of semantic errors. The edit, compile and test loop can be quite long if user has to download object code into HP calculator for testing every time a modification is made.

To make things much easier for debugging and testing, the Pas2Rpl compiler has been designed in such a way that accepted programs are compilable and runnable using Borland International's famous Pascal compiler on the DOS platform. You can even do symbolic debugging using Borland Pascal compiler. This is achieved by conforming to syntax acceptable by Borland Pascal ( e.g.: **far** is needed for procedures passed to **setviewui** ) and providing a lcd module ( unit ) to simulate the 131x64 lcd display, text output commands, line drawing commands, key handling and setviewui.

For example, setviewui ( together with view outer loop) can be implemented using:

```
type proc    = procedure;
var appexit: boolean;
    key     : integer;

procedure setviewui(init,term,disp,handle: proc);
begin
  init;
  appexit:=false;
  while not appexit do
  begin
    disp;
    key:=getkey;
    if key=27 then
      appexit:=true
    else
      handle;
    end;
  term;
end;
```

## 5. Future developments

The preceding sections describe how Pascal programs can be compiled into object code for HP48/38 calculators. Comparing with last year's work, the following has been done:

- Basic support routines built into HP calculator.
- Signed integer arithmetics support
- **String** as a simple type
- Peephole optimizations on a two pass compiler structure
- Generate code for HP48 style external ROMPART structure
- Generate code for HP38 style simple aplet structure
- Turbo Pascal compatibly and the lcd module for simulation and debugging on PC

There are still lots of work to be done to support more Pascal features ( e.g.: variable parameters, **with** statement, **set** type ), more system features ( e.g.: vector, matrix as a simple object, choose box, input form ) and more powerful simulation of HP48/38 on the PC side ( e.g.: small/big font, bitmap display, choose box, input form, multiple views of HP38 ).

## References

1. S.Pemberton & M.Daniels, **Pascal Implementation**, Ellis Horward, Chichester, UK
2. J.Donnelly, **An Introduction to HP 48 System RPL and Assembly Language Programming**, 1995, Armstrong Publishing Company, OR, USA
3. F.Yuan, **Towards a Pascal compiler for HP RPN calculator**, Prompt Anniversary Conference, 1994, pp.7-20, Netherlands