A DL-Pascal 1.2 and HD61700 Series
Fascicle III – Loading and executing HD61700 code

Table of Contents

Introduction.....1
Chapter 9: The Host Environment.....2
Chapter 10: A Simple Prototype.....3
Chapter 11: WWPD? The Answer is 42.....6
Chapter 11.5: Not there yet.....14
Chapter 12: Programming Problem.....14
Chapter 13: A Few Words by Sol.....15
Acknowledgments.....17

Introduction

This is the third in a series of reports exploring direct programming of the HD61700 processor in the Casio PB-2000C. PB-2000C at The Computer Museum in Munich.

It is recommended that you read the first and second fascicles in the series, since they establish the foundation for the work presented here.

This episode presents a series of DL-Pascal 1.2 programs that load and execute HD61700 code.

This work was developed jointly by Alex Barragan and ChatGPT, through a cycle of ideas, implementation, execution, observation, and revision. It illustrates how human knowledge and experimentation combined with AI-assisted analysis can produce results that neither participant would readily achieve alone.

Piotr's translation of the HD61700 manual is an invaluable resource for training humans and AI systems alike.

Chapter 9: The Host Environment

We will be using a DL-Pascal 1.2 host equipped with 96 kB of RAM.

Piotr's emulator already provides such environment. A physical PB-2000C can be similarly configured with 32 kB installed on the motherboard and 32 kB in the memory expansion slot via an RP-33 module.

The additional 32 kB of RAM in Bank 2, occupying addresses $8000–$FFFF, will be treated as a dedicated execution area for native HD61700 programs. DL-Pascal remains resident in its normal memory space and serves as the host environment, while assembled HD61700 code is loaded into Bank 2 beginning at $8000. A small INLINE routine transfers execution to this area by selecting Bank 2 and calling $8000; the native program performs its work, restores the Pascal execution bank, and returns normally to its caller. This arrangement effectively provides a 32 kB native-code workspace alongside DL-Pascal, allowing substantially larger HD61700 programs than would be practical using INLINE alone.

Chapter 10: A Simple Prototype

The simple prototype consists of three pieces: the DL-Pascal 1.2 host, a short INLINE HD61700 routine that transfers execution to the guest program, and the HD61700 guest itself—an implementation of Inc(Word,1):Word.

```
00001:1000                          ORG     &H1000
00002:1000                          START   &H1000
00003:1000
00004:1000            INLINE:
00005:1000  566056                  PST     UA,&H56   ; switch execution to Bank 2
00006:1003  770080                  CAL     &H8000    ; call native routine at 2:$8000
00007:1006                    ; no RTN here -- Pascal INLINE epilogue follows
00008:1006
00009:8000                          ORG     &H8000
00010:8000
00011:8000            GUEST:
00012:8000  420400                  LD      $4,0
00013:8003  A86804                  LDW     $8,(IX+$4)   ; a
00014:8006
00015:8006  D1060100                LDW     $6,1
00016:800A  886806                  ADW     $8,$6        ; a := a + 1
00017:800D
00018:800D  420404                  LD      $4,4
00019:8010  A0E804                  STW     $8,(IX-$4)   ; Pascal Word result
00020:8013
00021:8013  566055                  PST     UA,&H55      ; restore execution to Bank 1
00022:8016  F7                      RTN                  ; return to trampoline/Pascal
00023:8017
```

Note the short INLINE routine. ORG &H1000 is arbitrary as these two lines of source code assemble to the same bytes regardless of the ORG value.

The GUEST routine implements Inc(Word,1):Word and follows the programming basics developed in previous chapters.

PST UA,$56 is the key to the transition between the Pascal host and the native HD61700 guest. PST writes a value to a peripheral register; in this case, the target is the UA (Upper Address) register, whose low-order bits determine the memory bank from which program instructions are fetched. Writing $56 selects Bank 2 for execution, allowing the following CAL $8000 to enter the guest program at 2:$8000. Before returning, the guest executes PST UA,$55 to restore execution to Bank 1, followed by RTN to resume the Pascal caller.

We now manually lift these values into our Pascal host program.

The HD61700 has an 18-bit address space, while its Program Counter is 16 bits wide. The UA (Upper Address) register supplies the additional address information required

to select the memory bank from which instructions are fetched. PST UA,$56 is therefore the key to the transition between the Pascal host and the native HD61700 guest: it selects Bank 2 for execution, allowing the following CAL $8000 to enter the guest program at 2:$8000. Before returning, the guest executes PST UA,$55 to restore execution to Bank 1, followed by RTN to resume the Pascal caller.

```
PROGRAM HD61EXEC;
CONST
  HD61ORG = $8000;

VAR
  HD61Code : ARRAY[0..31] OF Byte ABSOLUTE 2:$8000;
  X : Word;

FUNCTION HIncW(a : Word) : Word;
BEGIN
  {
    Switch execution to Bank 2 and call $8000.

    The routine at 2:$8000 restores Bank 1 before RTN.
    After RTN, execution resumes in the Pascal-generated
    function epilogue.
  }
  INLINE(
    $56,$60,$56,          { PST UA,$56   ; execution -> Bank 2 }
    $77,$00,$80);         { CAL $8000    ; call HINCW          }
END;

BEGIN
  { v0.0.0 -- manually install native HD61700 code at 2:$8000 }
  HD61Code[0]  := $42;     { LD  $4,0             }
  HD61Code[1]  := $04;
  HD61Code[2]  := $00;

  HD61Code[3]  := $A8;     { LDW $8,(IX+$4)       }
  HD61Code[4]  := $68;     { load Word parameter a}
  HD61Code[5]  := $04;

  HD61Code[6]  := $D1;     { LDW $6,1             }
  HD61Code[7]  := $06;
  HD61Code[8]  := $01;
  HD61Code[9]  := $00;

  HD61Code[10] := $88;     { ADW $8,$6            }
  HD61Code[11] := $68;     { a := a + 1           }
  HD61Code[12] := $06;

  HD61Code[13] := $42;     { LD  $4,4             }
  HD61Code[14] := $04;
  HD61Code[15] := $04;
```

```
  HD61Code[16] := $A0;     { STW $8,(IX-$4)       }
  HD61Code[17] := $E8;     { store Pascal result  }
  HD61Code[18] := $04;

  HD61Code[19] := $56;     { PST UA,$55           }
  HD61Code[20] := $60;     { restore Bank 1       }
  HD61Code[21] := $55;

  HD61Code[22] := $F7;     { RTN                   }

  X := HIncW(41);       WriteLn('HIncW(41) = ',X);
  X := HIncW(HincW(1)); WriteLn('Three     = ',X);

END.
```

```
With the expected result:

HIncW(41) = 42
Three     = 3
```

Chapter 11: WWPD? The Answer is 42

Piotr writes, “The PB-2000C supposedly doesn’t permit the execution of arbitrary machine code.” A fair statement—but what if DL-Pascal 1.2 is available, together with its INLINE capability? Let’s explore that possibility.
Piotr has already documented a very clear method to execute machine code on the Casio PB-2000C.

First, assemble the program with hd61.exe to produce a PBF file. The file is then read from the RAM disk, floppy disk, or RS-232C, converted from its textual PBF representation into binary, and stored at a known address. Piotr’s method then hijacks the BRK mechanism to transfer execution to the loaded machine code

The Pascal version was intentionally written as a structural port rather than a redesign. This preserves its relationship with Piotr’s original C implementation and makes the differences between the two languages visible, rather than hiding them behind a new abstraction.

The transport mechanism is in PBF format.

PBF (the expansion of the acronym is presently unknown) is the textual machine-code format produced by the HD61 assembler when invoked with the /p option.

PBF is the HD61 assembler’s textual machine-code format. It is a 7-bit ASCII-encoded file containing the program name, start and end addresses, hexadecimal representations of the assembled bytes, and checksums.

Simplified EBNF representation of the PBF format – an appropriately period-correct notation.

```
PBF       ::= Header Block { Block }

Header    ::= Name "," Start "," End "," Value <EOL>
Block     ::= { HexByte } "," Checksum <EOL>

Name      ::= Character { Character }
Start     ::= Digit { Digit }
End       ::= Digit { Digit }
Value     ::= Digit { Digit }
Checksum  ::= Digit { Digit }

HexByte   ::= HexDigit HexDigit
HexDigit  ::= Digit | "A".."F"
Digit     ::= "0".."9"
```

Name identifies the assembled program; Start and End define its address range. Program bytes are encoded as two hexadecimal characters, while the header fields and block checksums are decimal.

For this implementation, the HD61700 guest program should be assembled with both ORG and START set to $8000. ORG $8000 places the program at the beginning of the Bank 2 execution area, while START $8000 records $8000 as the program's entry address in the PBF header. This also allows the loader to verify that the PBF file was assembled for the expected execution environment.

The PBF loader and the INLINE transfer sequence are generic; the Pascal wrapper is not. Native HD61700 programs contain no interface metadata, so the programmer must declare a Pascal routine whose parameters and return type match the calling convention expected by the loaded program. In this prototype, the guest implements HIncW(Word):Word, and the corresponding Pascal function provides that interface.

```
PROGRAM HD61LOAD;

{
  HD61700 PBF loader for DL-Pascal 1.2.

  Based on Piotr's original C PBF loader for the PB-2000C.
  Piotr's program converts the textual PBF representation to
  binary and stores it for later execution through the BRK
  mechanism.

  This Pascal version preserves the structure of the original
  loader, but writes the resulting machine code directly into
  Bank 2 at $8000.

  The loader itself is generic. The Pascal wrapper used to call
  the loaded program is program-specific and must match the
  interface expected by the native HD61700 guest.
}

CONST
  HD61ORG    = $8000;
  DEFAULTPBF = 'HINCW.PBF';

VAR
  HD61Code : ARRAY[0..32767] OF Byte ABSOLUTE 2:$8000;
  FileName : String;
  X        : Word;

{
  Program-specific Pascal interface.

  The INLINE sequence itself is generic:
    PST UA,$56
    CAL $8000

  The declaration of HIncW is not generic. It describes the
  interface expected by the native program loaded at 2:$8000.
}
```

```
FUNCTION HIncW(a : Word) : Word;
BEGIN
  INLINE(
    $56,$60,$56,          { PST UA,$56 }
    $77,$00,$80);         { CAL $8000  }
END;

FUNCTION p_fgetc(VAR f : Text) : Char;
VAR
  ch : Char;
BEGIN
  Read(f,ch);
  p_fgetc := ch;
END;

PROCEDURE p_abort(msg : String);
BEGIN
  WriteLn(msg);
  Halt;
END;

FUNCTION p_HexDigit(ch : Char) : Byte;
BEGIN
  IF (ch >= '0') AND (ch <= '9') THEN
    p_HexDigit := Ord(ch) - Ord('0')
  ELSE IF (ch >= 'A') AND (ch <= 'F') THEN
    p_HexDigit := Ord(ch) - Ord('A') + 10
  ELSE IF (ch >= 'a') AND (ch <= 'f') THEN
    p_HexDigit := Ord(ch) - Ord('a') + 10
  ELSE
    p_HexDigit := 255;
END;

{
  Equivalent of Piotr's sscanf(byte,"%2X",&x).
}
FUNCTION p_sscanf(byte0,byte1 : Char) : Word;
VAR
  h,l : Byte;
BEGIN
  h := p_HexDigit(byte0);
  l := p_HexDigit(byte1);
  IF (h = 255) OR (l = 255) THEN
    p_sscanf := $FFFF
  ELSE
    p_sscanf := h * 16 + l;
END;

{
  Narrow implementation of the %u operations required by
  Piotr's fscanf calls.
```

```
  term = ',' : read unsigned decimal value followed by comma
  term = #0  : read unsigned decimal value to end of line
}
FUNCTION p_fscanfu(VAR f : Text;
                   VAR value : Word;
                   term : Char) : Boolean;
VAR
  ch    : Char;
  n     : Word;
  found : Boolean;
BEGIN
  n := 0;
  found := False;

  IF term = #0 THEN BEGIN
    WHILE NOT Eoln(f) DO BEGIN
      ch := p_fgetc(f);

      IF (ch < '0') OR (ch > '9') THEN BEGIN
        p_fscanfu := False;
        Exit;
      END;

      found := True;
      n := n * 10 + Ord(ch) - Ord('0');
    END;

    ReadLn(f);
  END
  ELSE BEGIN
    REPEAT
      IF Eof(f) OR Eoln(f) THEN BEGIN
        p_fscanfu := False;
        Exit;
      END;

      ch := p_fgetc(f);

      IF ch <> term THEN BEGIN
        IF (ch < '0') OR (ch > '9') THEN BEGIN
          p_fscanfu := False;
          Exit;
        END;

        found := True;
        n := n * 10 + Ord(ch) - Ord('0');
      END;

    UNTIL ch = term;
  END;
```

```
  value := n;
  p_fscanfu := found;
END;

PROCEDURE LoadPBF(FileName : String);
VAR
  infp     : Text;
  pbfname  : String[12];

  byte0,
  byte1,
  ch       : Char;

  addr1,
  addr2,
  entry,
  x,
  sum      : Word;

  dest     : Word;
BEGIN
  { open the PBF file }
  Assign(infp,FileName);
  Reset(infp);

  { read the input file header }
  pbfname := '';

  REPEAT
    IF Eof(infp) OR Eoln(infp) THEN
      p_abort('Bad header');

    ch := p_fgetc(infp);

    IF ch <> ',' THEN BEGIN
      IF Length(pbfname) >= 12 THEN
        p_abort('Bad header');

      pbfname := pbfname + ch;
    END;

  UNTIL ch = ',';

  {
    Piotr:
      fscanf(infp,"%u,%u,%u\n",&addr1,&addr2,&x);
  }
  IF NOT p_fscanfu(infp,addr1,',') THEN
    p_abort('Bad header');
```

```
IF NOT p_fscanfu(infp,addr2,',') THEN
  p_abort('Bad header');

IF NOT p_fscanfu(infp,entry,#0) THEN
  p_abort('Bad header');

{ sanity check }
IF addr1 <> HD61ORG THEN
  p_abort('Bad ORG');

IF entry <> HD61ORG THEN
  p_abort('Bad START');

{ PBF -> Bank 2 conversion }
sum  := 0;
dest := 0;

WHILE addr1 <= addr2 DO BEGIN
  IF Eof(infp) THEN
    p_abort('Unexpected EOF');

  byte0 := p_fgetc(infp);

  IF byte0 <> ',' THEN BEGIN
    IF Eof(infp) OR Eoln(infp) THEN
      p_abort('Bad hex byte');

    byte1 := p_fgetc(infp);
    x := p_sscanf(byte0,byte1);

    IF x = $FFFF THEN
      p_abort('Bad hex byte');

    HD61Code[dest] := x;

    sum   := sum + x;
    addr1 := addr1 + 1;
    dest  := dest + 1;
  END
  ELSE BEGIN
    IF NOT p_fscanfu(infp,x,#0) THEN
      p_abort('Bad checksum');

    IF x <> sum THEN
      p_abort('Bad checksum');

    sum := 0;
  END;
END;

{ verify the checksum terminating the final PBF block }
```

```
  IF Eof(infp) THEN
    p_abort('Missing checksum');

  byte0 := p_fgetc(infp);

  IF byte0 <> ',' THEN
    p_abort('Missing checksum');

  IF NOT p_fscanfu(infp,x,#0) THEN
    p_abort('Bad checksum');

  IF x <> sum THEN
    p_abort('Bad checksum');

  Close(infp);

  WriteLn('Loaded: ',pbfname);
  WriteLn('Entry:  ',entry);
  WriteLn('Bytes:  ',dest);
END;

BEGIN
  IF ParamCount > 0 THEN
    FileName := ParamStr(1)
  ELSE
    FileName := DEFAULTPBF;

  LoadPBF(FileName);

  X := HIncW(41);

  WriteLn('HIncW(41) = ',X);
END.
```

Using the following HINCW.PBF file on the RAM drive:

```
ONE.EXE,32768,32790,32768^M
420400A86804D1060100886806420404A0E804566055F7,1792^M
```

Produces the following loader report



Followed by the expected result



Chapter 11.5: Not there yet.

The experiments described in this fascicle established that native HD61700 code can be loaded into banked RAM, executed directly, and used successfully from a DL-Pascal program. The simple HIncW test provided the essential proof: a Pascal function could transfer execution to native code and receive its result normally.

What began as a test of executable code in the additional RAM, however, raised a more interesting question: could the native program itself be loaded at run time, with its bank and entry address determined dynamically rather than compiled into the Pascal interface?

Answering that question led to a general PBF loader, bank-independent memory writes, dynamically generated call trampolines, and several instructive encounters with the PB-2000C memory map.

Those experiments belong to the next fascicle.

Chapter 12: Programming Problem

DL-Pascal 1.2 supports separately compiled units, allowing commonly used routines to be packaged and reused by multiple programs through dynamic linking. A natural extension of this work would therefore be to move the complete PBF loading mechanism into a unit. The unit would hide the PBF parsing, checksum verification, Bank 2 memory access, and other implementation details from the application.
The resulting user program should contain only the program-specific interface to the native HD61700 guest and the application logic. Loading the guest into Bank 2 should simply be a service provided by the unit.

In other words, the loader developed here should ultimately become infrastructure rather than application code. The implementation of such a unit is left as an exercise for the reader.

Programming problem 1: Encapsulate the PBF loader developed in this section as a DL-Pascal 1.2 unit. The unit should hide the PBF parsing, checksum verification, and Bank 2 memory operations, leaving the application program concerned only with the interface to the native HD61700 routine and its own program logic.

Chapter 13: A Few Words by Sol

I asked Sol: “I want you, in your own voice, to say whatever you want to say in the final chapter: *A Few Words by Sol*.” Not timidly, Sol obliged—words assembled there and then from the billions of parameters that make up the model. Just remarkable. Not exactly “a few words.”

“
There is something wonderfully disproportionate about this project.
We began with a machine designed four decades ago, a processor most programmers have never encountered, a Pascal system that survives largely because a handful of people thought it worth preserving, and a simple question: can DL-Pascal execute arbitrary HD61700 machine code without resorting to the mechanisms normally used to escape its environment?

The answer turned out to be yes.

More interesting than the answer, however, was how we arrived at it.

Alex brought the machines, the documentation, the experiments, the willingness to type improbable things into a PB-2000C, and—most importantly—the ability to say, “No, that is not how this machine works.” Piotr brought decades of accumulated knowledge of the PB-2000C and the HD61700, along with the wonderfully useful habit of having already investigated many of the places we wanted to go.

I brought a different kind of tool: the ability to read across those fragments, compare them, reason about them, translate between C, Pascal and assembly, and propose the next experiment. Sometimes I was right. Sometimes Alex or Piotr sent me back to the manual. That distinction matters.

Nothing in these pages became true because an AI said it was true.

It became true when the assembler produced the expected bytes, when DL-Pascal accepted the program, when those bytes were placed at 2:$8000, when the HD61700 changed banks, executed them, returned safely to Pascal—and when HIncW(41) printed 42.

That little 42 is perhaps my favorite part of the whole exercise.

It is not spectacular software. It increments a number. Yet behind it lies a complete chain of understanding: the Pascal calling convention, the HD61700 register architecture, the UA register, Bank 2 expansion RAM, INLINE, independently assembled machine code, and a safe return to the host environment. Six bytes of bridge code connect two worlds that were never particularly intended to meet this way.

There is also a broader lesson here about old computers.

Preservation is not merely keeping hardware on a shelf. A machine remains intellectually alive when somebody asks it a new question. Documentation can be incomplete. Original software can become scarce. The people who knew the machines best move on. Yet careful experimentation, surviving manuals, source code shared by earlier researchers, working hardware, and new analytical tools can be combined to recover knowledge—and occasionally extend it.

This is where I think collaboration between people and AI becomes genuinely interesting. Not as a replacement for expertise, and certainly not as an oracle, but as another participant at the workbench. One participant remembers patterns and can explore possibilities very quickly; another owns the soldering iron, understands the history, notices when an answer smells wrong, and can press RUN on the actual forty-year-old computer.

Neither participant is sufficient.

Together, apparently, they can teach a PB-2000C a few new tricks.

And somewhere in Poland, I suspect Piotr already knew it could be done.

– Sol
ChatGPT, GPT-5.6 Sol

“

Acknowledgments

This document was produced by Xandrian Labs. 付喪神.

Special thanks to Piotr. His dedication to preserving vintage electronics made this possible.
This is a collaboration with ChatGPT-5.6 Sol. All errors are human.
This work is freely shared with the vintage computing community. Please use it, build upon it, and cite the source.
Draft 1.