P R O G R A M B A R C O D E P R I N T I N G R O U T I N E S

- P B C 1
- P B C 2

SYSTEM CONFIGURATIONS AND SOFTWARE FOR PRINTING PROGRAM BARCODE USING THE HP-41/CV AND THE EPSON MX-80 PRINTER (OR EQUIVALENT), WITH THE SUPPORT OF OTHER SPECIFIED HARDWARE COMPONENTS AND ROM.

by LAURENCE J. LAVINS (7310)
PHILADELPHIA CHAPTER OF PPC
130 RADNOR AVENUE
VILLANOVA, PA. 19085

Tel: (215) 687-1774
293-9208

PROGRAM BARCODE PRINTING ROUTINES

DOCUMENTATION PACKAGE

TABLE OF CONTENTS

PROGRAM BARCODE PRINTING ROUTINES PBC1 AND PBC2

C A U T I O N: USERS SHOULD FIRST READ AND THOROUGHLY UNDERSTAND THE ACCOMPANYING ANALYSIS AND COMMENTARIES BEFORE ATTEMPTING TO RUN THESE PROGRAMS.

1. INTRODUCTION & PURPOSE

PBC1 and PBC2 constitute a set of programs for the HP-41, which were designed for the purpose of producing good quality program type barcode in a reasonably efficient manner. The actual printing is performed by an EPSON MX-80 printer, or equivalent, using its super-high-resolution (SHR) graphics capabilities.

A number of other peripheral devices and ROM's are also required to support these programs. Either of two additional alternative configurations are also delineated, however, in order to make these programs as universal as possible.

2. SYSTEM HARDWARE & ROM REQUIREMENTS

(a) HP-41/CV, or equivalent (2X speedup capability preferable).
(b) PPC ROM Module.
(c) XFM Module, and at least 1 XM Module.
(d) HP Plotter ROM Module (or EPROM equivalent).
(e) HP-IL Module
(f) HP82166A Converter & EPSON MX-80 Printer, or equivalent. (e.g., the HP82905B Printer)
(g) Any one of the following EPROM Sets:

(1) Base Configuration

NFCROM, with XROM Number changed from 17 to 16 to avoid conflict with the Plotter ROM. Also augmented with PPACK and NRCL functions (copied from PPC EPROM-2).

(2) Alternative #1

NFCROM, with XROM Number changed from 17 to 16, but without any augmented functions.

(3) Alternative #2

PPC EPROM-2, in place of the NFCROM.

(h) Sufficient EPROM box and port extender capacity, etc. to support all the above devices and modules.

3. LOADING THE PROGRAMS

It will be easiest and most convenient to copy these programs from some other person's HP-41 calculator or tape file.

Using the wand to read them into the HP-41 from barcode is the next best choice.

If the programs must be keyed in by hand, then by all means, don't hesitate to make use of the LB routine in the PPC ROM to assist in loading the many synthetic alpha strings. Or, if it's available, the ASSEMBLER-3 EPROM Set can also be used to load all these lines even more easily than the PPC ROM.

The decimal byte codes (including the Fx text byte) for all synthetic alpha lines, except for the ANYBLS print routines, are included with the program listings. The ANYBLS code sequences are completely described on a separate enclosure.

4. BASE CONFIGURATION

If the NFCROM is modified and augmented as indicated above, the programs may be executed as provided herewith, without any changes. If, however, the user desires to change the XROM Number to something other than 16, this is OK; but all program lines in PBC2 which use XROM 16 functions will have to be deleted, and then re-entered while the user's ROM is connected.

After changing the XROM Number, PPACK and NRCL functions may be copied directly from the PPC EPROM-2 into the NFCROM, if the proper equipment is on hand. Owners of ProtoCODER or MLDL devices should have no difficulty in changing the XROM Number and adding these functions into any unused portion of the NFCROM. Just don't forget to make the appropriate directory entries and to change the total number of functions at ROM address x001.

5. ALTERNATIVE #1

If the user is able to change the XROM Number of the NFCROM, but can't provide the two augmented functions, then some simple changes and additions can be made in both PBC1 and PBC2 to enable them to work under these circumstances.

(a) Changes to PBC1 program:

Insert two new program steps after Line 65, as follows:

```
          65 ACA
New Line  66 SF25
New Line  67 1
          ......
          70 END
```

(b) Changes to PBC2 program:

Line 193 Delete 21 and insert 474 in its place.
Line 194 Delete NRCL and insert RCLA in its place.
Line 199 Delete 22 and insert 475 in its place.
Line 200 Delete NRCL and insert RCLA in its place.

Line 134 Delete the existing synthetic alpha string. Insert the following synthetic alpha string in its place: (247) 0 168 1 6 230 143 227.

Line 132 Delete PPACK and insert BCP in its place.
New Line 133 Insert FC?25 after BCP.

After these changes have been entered to accomodate a non-augmented NFCROM, the programs will run with no noticeable difference from the original version. Don't forget, though, that the XROM Number of the NFCROM must first be altered, or it won't work. Since the size of PBC2 is increased by 4 bytes, the maximum size of an object program that can be accomodated in memory will be decreased by the same 4 bytes.

6. ALTERNATIVE #2

For those users who don't have an NFCROM or the capability to modify the XROM Number, these programs will work equally well with the PPC EPROM-2, which has been rather widely distributed throughout the world. As was the case with Alternative #1, there are a few changes which must first be made, as follows:

(a) Changes to PBC1 program:

No changes to the base program, as provided herewith, are required for Alternative #2.

(b) Changes to PBC2 program:

Line 165	Delete <> and insert NRCL in its place
New Line 166	Insert STO IND Y after NRCL
Line 137	Delete STA and insert CHS in its place.
New Line 138	Insert NSTO after CHS.
Line 134	Delete the existing synthetic alpha string. Insert the following synthetic alpha string in its place: (247) 168 1 6 228 143 227 167.

After these changes have been entered to accomodate a PPC EPROM-2 in place of a modified/augmented NFCROM, the programs will run without any noticeable difference from the original version. These changes increase the size of PBC2 by 3 bytes, and therefore, the maximum size of an object program that can be accomodated in memory will be decreased by the same 3 bytes.

7. MISCELANEOUS CHANGES (OPTIONAL)

(a) If a PPC ROM isn't available, or if it must be removed due to lack of ports, etc., ADV functions can be substituted wherever there is an XROM PO. Substitute as many ADV's as desired in any of the lines in either PBC1 or PBC2. Since all the XROM PO's in PBC2 are further down in the program than Line 222, there will be no effect upon the pre-compilation of Line 222.

(b) If a switched 2X speedup capability isn't available on the HP-41, the STOP function at Line 133 of PBC2 can be deleted. If it is deleted, it's absolutely necessary to add a CLX in its place, in order to maintain a constant byte count above Line 222. (More about this below.)

8. RUNNING THE PROGRAMS

(a) Preparatory Steps

Store PBC1, PBC2 and all object programs (i.e., those programs for which barcode representations will be printed) in XM. A total of 178 XM registers are required for just PBC1 and PBC2, in their base configuration versions.

It is important for all programs being stored in XM to be first packed with GTO.. prior to being stored in XM, to insure that all non-essential nulls are deleted, that normal program END's are present, and that valid byte counts will result.

It is also necessary to clear all programs from main memory.

Now execute GETP to bring PBC1 into main memory from XM. DO NOT PACK!!

PROGRAM BARCODE PRINTING ROUTINES P B C 1 AND P B C 2

(a) Preparatory Steps (Continued)

Make sure that all elements of the system are properly connected and plugged in, turned on, switches properly set, etc.

Now, execute PBC1.

(b) Printer Anomaly?

For reasons which are not clearly understood, the EPSON Printer does not always (but it does sometimes) respond to an initial set of printer commands, after being turned on. It may, therefore, be necessary to repeat an initial sequence.

When PBC1 is executed, a series of instructions to the user should be immediately printed, together with the prompting message "SEE PRINTOUT" in the HP-41 display. The program then stops until R/S is pressed. The program may frequently reach this point, as evidenced by the display, without any sign of life by the printer. If this happens, don't panic. Just do the following 3 simple steps:

(1) Press the RTN key on the HP-41.
(2) Turn the HP-41 OFF. Then turn it ON again.
(3) Press R/S key to re-initiate PBC1.

This sequence of operations causes the printer to function normally from here on. Very, very strange!

(c) Initial Paper Adjustment

The prompt mentionned just above stops the program to allow the user to adjust the fanfold paper to the very top of the next page form (the TOF). Don't forget to toggle the printer power switch OFF and then ON again, after adjusting the paper.

(d) Label Entry

The program will stop again to prompt the user to enter the name of the object program. The HP-41 will already be in the Alpha mode. Just key in the label, and then press the R/S key to restart the program.

(e) Switching to 2X Speed

If a ProtoSYSTEM is connected, and a triple XFM/XM/XM module is also being used, the author experienced problems with the execution of XM functions when the HP-41 was set to 2X speed, during the hot and humid mid-summer season. That's the reason for the STOP at Line 133 of PBC2. Troubles (like a crash) were avoided by running at normal speed up to this point, and then switching into 2X after all XM operations were completed. If this STOP step isn't needed, it may be easily deleted, as per the instructions in Para. 7(b) above.

PROGRAM BARCODE PRINTING ROUTINES PBC1 AND PBC2

9. ALLOCATION OF MAIN MEMORY

(a) Data Storage Registers & SIZE

SIZE is set to 59.

REGISTER No.	REGISTER USAGE
R-00	Instruction codes for XEQ 71 and RTN.
R-01 to R-16	XEQ's for up to 32 print routines.
R-17	Instruction codes for LBL 99 and XEQ 47.
R-18	Label of the object program.
R-19	Uncompiled code for XEQ 71 and RTN.
R-20	Sequential barcode row number.
R-21	Temporary storage for N-register contents.
R-22	Temporary storage for M-register contents.
R-23 to R-25	Addressing & loop controls.
R-26	Rows per page counter.
R-27 to R-58	Storage for 32 hex digit character codes.

(b) Program Memory Usage

Total Memory = (319 + 4/7) registers x 7		=	2,237 bytes
Less: 59 data storage registers	= 413		
26 I/O buffer registers	= 182		
PBC2 program size	= 969		
	1,564		-1,564 bytes
Net memory available for object program		=	673 bytes

This assumes that there are no registers tied up with key assignments.

If the object program exceeds the maximum available program memory, it is recommended that it be split up into two or more smaller programs before attempting to print the barcode.

10. PRINTING PBC1 AND PBC2

(a) Barcode Printing for PBC1

No special handling is required for printing the barcode representation of PBC1.

(b) Barcode Printing for PBC2

At Line 131, the program will stop and print an error message "NO ROOM." When this happens, press SST to advance the program to the next step. Then press R/S to continue on. Barcode for PBC2 will then be printed.

11. SPEED OF EXECUTION

On the author's system, the following average times were observed for execution of a complete (double-printed) 16-byte row of barcode, including all computation time as well as the printing time for each row.

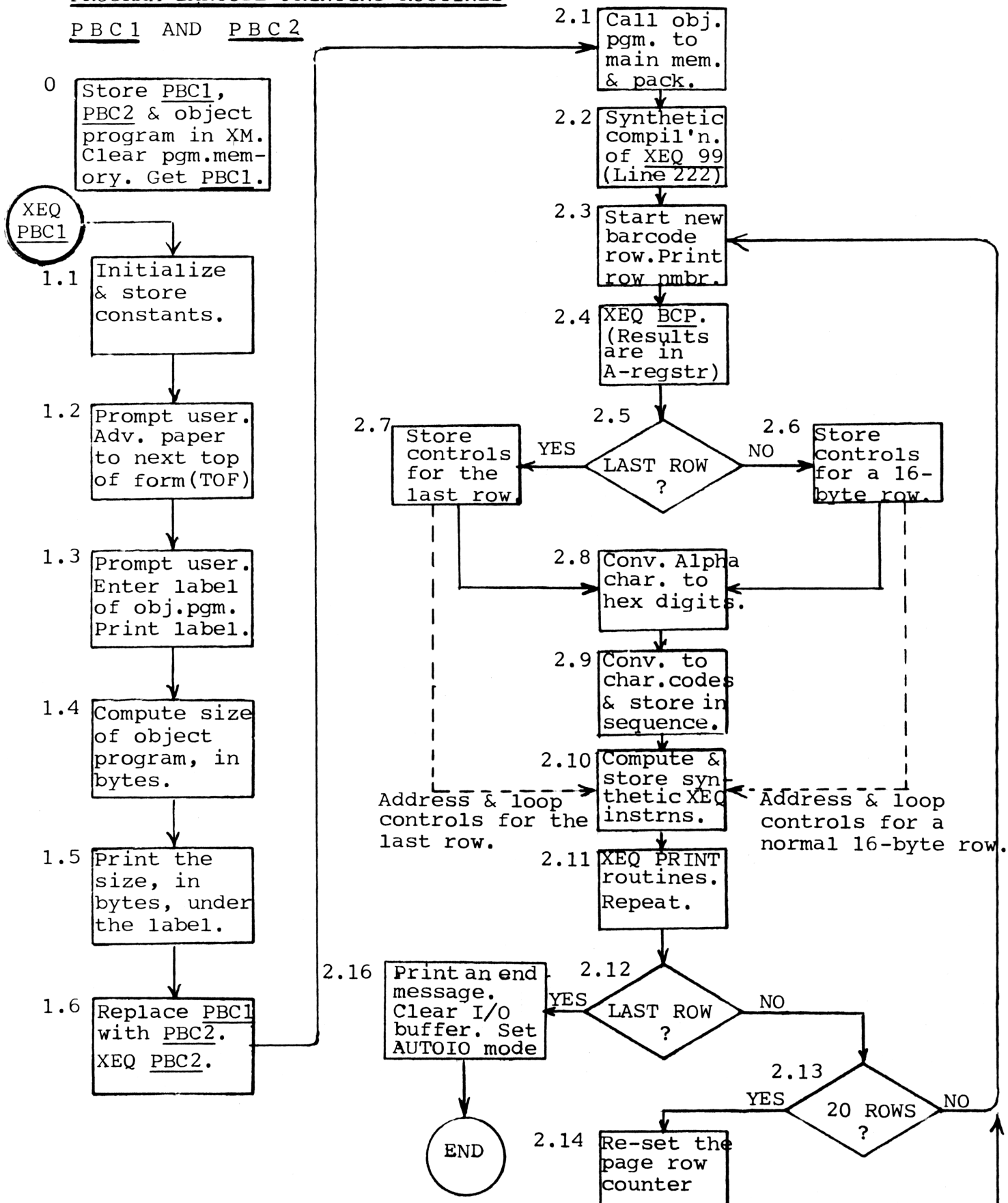
(a) HP-41 at std. speed -------- 104.9 sec. per row (av'ge.)
(b) HP-41 at 2X speed ---------- 61.5 sec. per row (av'ge.)

G E N E R A L I Z E D F L O W C H A R T D I A G R A M

Rev: 8-29-83

PROGRAM BARCODE PRINTING ROUTINES

P B C 1 AND P B C 2



by LAURENCE J. LAVINS (7310)
130 RADNOR AVENUE
VILLANOVA, PA. 19085
Tel: (215) 687-1774
293-9208

PROGRAM BARCODE PRINTING ROUTINES

PBC1 - PART 1 OF 2 PARTS

by Larry Lavins (7310)
130 Radnor Avenue
Villanova, Pa. 19085
Tel: (215) 687-1774
293-9208

CAUTION: It is most important for users to read and understand all the attached explanatory materials before attempting to use these programs.

```
AUG. 30,1983
        CAT 1
LBL'PBC1
END
266 BYTES
```

```
 01◆LBL "PBC
1"

 02 AUTOIO
 03 SF 25
 04 PINIT

 05 MANIO
 06 FIX 0
 07 CF 29
 08 SF 17
 09 59
 10 PSIZE
 11 20
 12 STO 26
 13 1
 14 STO 20

 15 "c
◆/"
 16 RCL [
 17 STO 17

 18 "
◆G"
 19 RCL [
 20 STO 00
 21 STO 19
 22 "ADVANCE
 PAPER "
 23 ACA
 24 "TO TOP
OF NEXT "
 25 ACA
 26 "PAGE"
 27 ACA
 28 ADV
 29 "THEN TO
GGLE "
 30 ACA
 31 "PRINTER
 OFF AND"
 32 ACA
 33 " ON AGA
IN"
 34 ACA
 35 7
 36 ACCHR
 37 ADV
 38 "PRESS R
/S KEY "
 39 ACA
 40 "TO CONT
INUE"
 41 ACA
 42 7
 43 ACCHR
 44 XROM "PO
"
 45 "SEE PRI
NTOUT"
 46 PROMPT
 47 "┌ÆEÆ1"
 48 ACA
 49 "LABEL?"
 50 AON
 51 PROMPT

 52 AOFF
 53 ASTO 18
 54 PRA
 55 ADV
 56 ADV

 57 RCLPTA

 58 "PROGRAM
 SIZE = "
 59 ARCL X
 60 "├ BYTES
"
 61 PRA
 62 ADV
 63 ADV
 64 "ÆF"
 65 ACA

 66 "PBC2"
 67 GETP

 68 END
```

SYNTHETIC PROGRAM LINES

LINE	DEC. CHARACTER CODES	DESCRIPTION
15	(245) 207 99 224 0 47	Codes for LBL 99 and XEQ 47.
18	(244) 224 0 71 133	Codes for XEQ 71 and RTN.
47	(245) 14 27 69 27 49	Printer control codes.
64	(242) 27 70	Printer control codes.

SYSTEM REQUIREMENTS

HP-41CV XFM + XM HP-IL PPC ROM PLOTTER ROM

EPSON MX-80 PRINTER & HP82166A CONVERTER (or equivalent)

MODIFIED NFCROM EPROM SET (or specified alternatives)

PROGRAM BARCODE PRINTING ROUTINE

PBC 2 - PART 2 OF 2 PARTS

CAUTION: Users are urged to read and understand all the attached explanatory materials before attempting to use these programs.

by Larry Lavins (7310)
130 Radnor Avenue
Villanova, Pa. 19085
Tel: (215) 687-1774
293-9208

AUG. 29,1983

```
LBL"PBC2
END
969 BYTES
```

```
 01◆LBL "PBC
2"
 02 GTO 98



 03◆LBL 48
 04 "ÆLä◆"
 05 ACA
 06 "◆◆◆◆◆◆◆
"
 07 "├◆◆◆◆◆"
 08 ACA
 09 RTN


 10◆LBL 49
 11 "ÆLä◆"
 12 ACA
 13 "◆◆◆◆◆◆◆
"
 14 "├◆◆◆◆◆"
 15 ACA
 16 RTN


 17◆LBL 50
 18 "ÆLä◆"
 19 ACA
 20 "◆◆◆◆◆◆"
 21 "├◆◆◆◆◆◆
"
 22 ACA
 23 RTN


 24◆LBL 51
 25 "ÆLö◆"
 26 ACA
 27 "◆◆◆◆◆◆"
 28 "├◆◆◆◆◆◆
"
 29 ACA
 30 RTN
```

```
 31◆LBL 52
 32 "ÆLä◆"
 33 ACA
 34 "◆◆◆◆◆◆"
 35 "├◆◆◆◆◆◆
"
 36 ACA
 37 RTN


 38◆LBL 53
 39 "ÆLö◆"
 40 ACA
 41 "◆◆◆◆◆◆"
 42 "├◆◆◆◆◆◆
"
 43 ACA
 44 RTN


 45◆LBL 54
 46 "ÆLö◆"
 47 ACA
 48 "◆◆◆◆◆◆"
 49 "├◆◆◆◆◆◆
"
 50 ACA
 51 RTN


 52◆LBL 55
 53 "ÆLü◆"
 54 ACA
 55 "◆◆◆◆◆◆"
 56 ACA
 57 "◆◆◆◆◆◆"
 58 ACA
 59 RTN


 60◆LBL 56
 61 "ÆLä◆"
 62 ACA
 63 "◆◆◆◆◆◆"
 64 "├◆◆◆◆◆◆
"
 65 ACA
 66 RTN
```

```
 67◆LBL 57
 68 "ÆLö◆"
 69 ACA
 70 "◆◆◆◆◆◆"
 71 "├◆◆◆◆◆◆
"
 72 ACA
 73 RTN


 74◆LBL 65
 75 "ÆLö◆"
 76 ACA
 77 "◆◆◆◆◆◆"
 78 "├◆◆◆◆◆◆
"
 79 ACA
 80 RTN


 81◆LBL 66
 82 "ÆLü◆"
 83 ACA
 84 "◆◆◆◆◆◆"
 85 ACA
 86 "◆◆◆◆◆◆"
 87 ACA
 88 RTN


 89◆LBL 67
 90 "ÆLö◆"
 91 ACA
 92 "◆◆◆◆◆"
 93 "├◆◆◆◆◆◆
◆"
 94 ACA
 95 RTN


 96◆LBL 68
 97 "ÆLü◆"
 98 ACA
 99 "◆◆◆◆◆◆"
100 ACA
101 "◆◆◆◆◆◆"
102 ACA
103 RTN
```

```
104◆LBL 69
105 "ÆLü◆"
106 ACA
107 "◆◆◆◆◆◆
108 ACA
109 "◆◆◆◆◆◆
110 ACA
111 RTN


112◆LBL 70
113 "ÆLœ◆"
114 ACA
115 "◆◆◆◆◆◆
116 ACA
117 "◆◆◆◆◆◆
118 ACA
119 RTN


120◆LBL 47
121 "ÆLλ◆◆◆◆
◆◆◆◆"
122 ACA
123 RTN


124◆LBL 71
125 "ÆLσ◆◆◆◆
"
126 ACA
127 RTN
```

(Continued on the next page

```
128◆LBL 98
129 CLA
130 ARCL 18
131 GETSUB
132 PPACK
133 STOP
134 "
J              ×
"
135 RCL [
136 327
137 STA

138◆LBL 15
139 "ROW "
140 RCL 20
141 ARCL X
142 PRA
143 BCP
144 STO 20
145 X=0?
146 GTO 02

147 27
148 STO 24
149 16
150 STO 25
151 GTO 03

152◆LBL 07
153 ATOX
154 STO IND
23
155 ISG 23
156 GTO 07
157 RTN
158◆LBL 02
159 RCL T
160 ENTER↑
161 CHS
162 16
163 +
164 19
165 <>
166 RDN
167  E-3
168 *
169 16
170 +
171 STO 25
172 RDN
173 ST+ X
174 CHS
175 59
176 +
177 STO 24

178◆LBL 03
179 RCL [
180 STO 22
181 RCL \
182 STO 21
183 RCL ]
184 DECODE
185 ASHF
186 ATOX
187 ATOX
188 ATOX
189 ATOX
190 27.03
191 STO 23
192 XEQ 07
193 21
194 NRCL
195 DECODE
196 31.044
197 STO 23
198 XEQ 07
199 22
200 NRCL
201 DECODE
202 45.058
203 STO 23
204 XEQ 07
205◆LBL 04
206 "
◆"
207 RCL IND
24
208 XTOA
209 ISG 24
210 CLX
211 "⊢
◆"
212 RCL IND
24
213 XTOA
214 ISG 24
215 CLX
216 RCL [
217 STO IND
25

218 DSE 25

219 GTO 04

220 SF 01

221◆LBL 05

222 XEQ 99
223 PRBUF

224 FS?C 01

225 GTO 05

226 ADV
227 ADV

228 RCL 20
229 X=0?
230 GTO 06

231 DSE 26

232 GTO 15

233 20
234 STO 26

235 "µ↑ÆE"
236 ARCL 18
237 "⊢á  ("
238 ACA
239 "CONTINU
ED)"
240 ACA
241 PRBUF
242 "ÆF"
243 ACA
244 XROM "PO
"
245 GTO 15

246◆LBL 06

247 "END OF
PROGRAM"
248 PRA
249 XROM "PO
"
250 XROM "PO
"
251 PCLBUF
252 AUTOIO

253 END
```

For details regarding coding of Lines 3-127, refer to the ANYBLS description & commentary.

S Y N T H E T I C P R O G R A M L I N E S

LINE	DECIMAL CHARACTER CODES	DESCRIPTION
134	(247) 236 142 227 167 74 170 1	Compilation data for Line 222.
206	(242) 224 0	Code for XEQ prefix.
211	(243) 127 224 0	Code for APPEND XEQ prefix.
235	(244) 12 14 27 69	Printer control codes.
237	(245) 127 20 32 32 40	Printer control codes.
242	(242) 27 70	Printer control codes.

A N Y B L S

BARCODE PRINTING ROUTINES FOR EPSON MX-80 PRINTER

EACH NUMBERED LABEL PRINTS A HEX DIGIT WHOSE DECIMAL CHARACTER CODE IS EQUAL TO THE LABEL NUMBER. ALSO, LABELS 47 & 71 PRINT LEFT AND RIGHT END BARS, RESPECTIVELY.

by LAURENCE J. LAVINS (7310)
130 RADNOR AVENUE
VILLANOVA, PA. 19085
Tel: (215) 687-1774
293-9208

```
LBL "ANYBLS
END
681 BYTES
```

```
01♦LBL "ANYBLS"

02♦LBL 48
03 "ELa♦"
04 ACA
05 "♦♦♦♦♦♦♦"
06 "┣♦♦♦♦♦"
07 ACA
08 RTN

09♦LBL 49
10 "ELä♦"
11 ACA
12 "♦♦♦♦♦♦♦"
13 "┣♦♦♦♦♦"
14 ACA
15 RTN

16♦LBL 50
17 "ELâ♦"
18 ACA
19 "♦♦♦♦♦♦"
20 "┣♦♦♦♦♦♦"
21 ACA
22 RTN

23♦LBL 51
24 "ELõ♦"
25 ACA
26 "♦♦♦♦♦♦"
27 "┣♦♦♦♦♦♦"
28 ACA
29 RTN

30♦LBL 52
31 "ELã♦"
32 ACA
33 "♦♦♦♦♦♦"
34 "┣♦♦♦♦♦♦"
35 ACA
36 RTN

37♦LBL 53
38 "ELô♦"
39 ACA
40 "♦♦♦♦♦♦"
41 "┣♦♦♦♦♦♦"
42 ACA
43 RTN

44♦LBL 54
45 "ELö♦"
46 ACA
47 "♦♦♦♦♦♦"
48 "┣♦♦♦♦♦♦"
49 ACA
50 RTN

51♦LBL 55
52 "ELü♦"
53 ACA
54 "♦♦♦♦♦♦"
55 ACA
56 "♦♦♦♦♦♦"
57 ACA
58 RTN

59♦LBL 56
60 "ELà♦"
61 ACA
62 "♦♦♦♦♦♦"
63 "┣♦♦♦♦♦♦"
64 ACA
65 RTN

66♦LBL 57
67 "ELò♦"
68 ACA
69 "♦♦♦♦♦♦"
70 "┣♦♦♦♦♦♦"
71 ACA
72 RTN

73♦LBL 65
74 "ELõ♦"
75 ACA
76 "♦♦♦♦♦♦"
77 "┣♦♦♦♦♦♦"
78 ACA
79 RTN

80♦LBL 66
81 "ELü♦"
82 ACA
83 "♦♦♦♦♦♦"
84 ACA
85 "♦♦♦♦♦♦"
86 ACA
87 RTN

88♦LBL 67
89 "ELö♦"
90 ACA
91 "♦♦♦♦♦"
92 "┣♦♦♦♦♦♦♦"
93 ACA
94 RTN

95♦LBL 68
96 "ELü♦"
97 ACA
98 "♦♦♦♦♦♦"
99 ACA
100 "♦♦♦♦♦♦"
101 ACA
102 RTN

103♦LBL 69
104 "ELü♦"
105 ACA
106 "♦♦♦♦♦♦"
107 ACA
108 "♦♦♦♦♦♦"
109 ACA
110 RTN

111♦LBL 70
112 "ELe♦"
113 ACA
114 "♦♦♦♦♦♦"
115 ACA
116 "♦♦♦♦♦♦"
117 ACA
118 RTN

119♦LBL 47
120 "ELλ♦♦♦♦♦♦♦♦"
121 ACA
122 RTN

123♦LBL 71
124 "ELσ♦♦♦♦"
125 ACA
126 END
```

L. J. LAVINS (7310)
130 Radnor Avenue
Villanova, Pa. 19085
Tel: (215) 687-1774

I N P U T S E Q U E N C E S F O R " A N Y B L S "

(Use XROM "LB" routine for loading the specified numeric strings)

NYBBLE	LABEL	DECIMAL INPUT SEQUENCES FOR USE WITH XROM "LB"
0000	48	ACA 244 27 76 20 0 253 255 255 0 0 0 255 255 0 0 0 255 255 0 ACA 248 127 0 0 255 255 0 0 0
0001	49	ACA 244 27 76 22 0 253 255 255 0 0 0 255 255 0 0 0 255 255 0 ACA 250 127 0 0 255 255 255 255 0 0 0
0010	50	ACA 244 27 76 22 0 253 255 255 0 0 0 255 255 0 0 0 255 255 255 ACA 250 127 255 0 0 0 255 255 0 0 0
0011	51	ACA 244 27 76 24 0 253 255 255 0 0 0 255 255 0 0 0 255 255 255 ACA 252 127 255 0 0 0 255 255 255 255 0 0 0
0100	52	ACA 244 27 76 22 0 253 255 255 0 0 0 255 255 255 255 0 0 0 255 ACA 250 127 255 0 0 0 255 255 0 0 0
0101	53	ACA 244 27 76 24 0 253 255 255 0 0 0 255 255 255 255 0 0 0 255 ACA 252 127 255 0 0 0 255 255 255 255 0 0 0
0110	54	ACA 244 27 76 24 0 253 255 255 0 0 0 255 255 255 255 0 0 0 255 ACA 252 127 255 255 255 0 0 0 255 255 0 0 0
0111	55	ACA 244 27 76 26 0 ACA 253 255 255 0 0 0 255 255 255 255 0 0 0 255 ACA 253 255 255 255 0 0 0 255 255 255 255 0 0 0 ACA
1000	56	ACA 244 27 76 22 0 253 255 255 255 255 0 0 0 255 255 0 0 0 255 ACA 250 127 255 0 0 0 255 255 0 0 0
1001	57	ACA 244 27 76 24 0 253 255 255 255 255 0 0 0 255 255 0 0 0 255 ACA 252 127 255 0 0 0 255 255 255 255 0 0 0
1010	65	ACA 244 27 76 24 0 253 255 255 255 255 0 0 0 255 255 0 0 0 255 ACA 252 127 255 255 255 0 0 0 255 255 0 0 0
1011	66	ACA 244 27 76 26 0 ACA 253 255 255 255 255 0 0 0 255 255 0 0 0 255 ACA 253 255 255 255 0 0 0 255 255 255 255 0 0 0 ACA
1100	67	ACA 244 27 76 24 0 253 255 255 255 255 0 0 0 255 255 255 255 0 0 ACA 252 127 0 255 255 0 0 0 255 255 0 0 0
1101	68	ACA 244 27 76 26 0 ACA 255 255 255 255 255 0 0 0 255 255 255 255 0 0 0 255 ACA 251 255 0 0 0 255 255 255 255 0 0 0 ACA

Continued - 2nd page

L. J. LAVINS (7310)
130 Radnor Avenue
Villanova, Pa. 19085
Tel: (215) 687-1774

"ANYBLS" INPUT SEQUENCES (CONTINUED)

NYBBLE	LABEL	DECIMAL INPUT SEQUENCES FOR USE WITH XROM "LB"
1110	69	ACA 244 27 76 26 0 ACA 255 255 255 255 255 0 0 0 255 255 255 255 0 0 0 255 ACA 251 255 255 255 0 0 0 255 255 0 0 0
1111	70	ACA 244 27 76 28 0 ACA 254 255 255 255 255 0 0 0 255 255 255 255 0 0 0 ACA 254 255 255 255 255 0 0 0 255 255 255 255 0 0 0
ROW LEADER		
00	47	ACA 255 27 76 11 0 0 255 255 0 0 0 255 255 0 0 0
ROW TRAILER		
10	71	ACA 253 27 76 9 0 255 255 255 255 0 0 0 255 255

TECHNICAL NOTES

1. These input sequences were designed specifically for use with the Epson MX-80 printer to enable the generation of barcode symbols for the HP-41 wand. The start of each sequence is a control code which will turn on the Epson's super high resolution graphics mode. As a matter of superficial interest, it was found, quite by accident, that this method of programmatically generating super high resolution graphics, using synthetic alpha strings, will work only if there is a break point and separate entry immediately after the 27, 76, N, 0 control code group; followed by a secondary entry (or entries) of the columnar data. Or alternatively, put it all in 1 string.
2. The labeling scheme for the 16 nybbles is strongly recommended in order to make it easier to address these subroutines. For example, the labels correspond exactly to the numbers which are returned to X by the function ATOX, for each nybble symbol from 0 to F. Then all we need to do is XEQ IND X, as one example.
3. An extra "0" was inserted at the start of the columnar data for the row leader sequence in order to have 11 characters rather than 10. The Epson printer will not respond to a 10.
4. The total length of the sequences for nybbles 0111, 1011, 1101, 1110 & 1111 will not permit an alpha append function. Therefore each sequence must be separately entered into the buffer, with an ACA function.
5. Labeling suggested for the leader and trailer is purely arbitrary. Any labeling will do as well, except 00 or 02, which are more obviously convenient, but will probably be used elsewhere in most programs, and will be inefficient in speed if the jump distance exceeds 16 registers.

PROGRAM BARCODE PRINTING ROUTINES

TECHNICAL ANALYSIS AND COMMENTARIES

In mid-1982, the author purchased an EPSON MX-80 III F/T printer, and connected it into the HP-IL loop via an HP-82166A converter which had been previously purchased for the purpose.

A great deal of experimentation was performed to learn the nature of the interface between this printer and the HP-41CV, and exactly what the capabilities, limitations and restrictions might be.

As one outgrowth of this experimentation, the idea of generating barcode patterns seemed to appear feasible, using the super-high-resolution graphics capabilities of the MX-80. There followed an on-again/off-again effort for almost a year, culminating in the successful development of a set of routines to print program barcode in a reasonably efficient manner.

These program barcode printing routines, PBC1 and PBC2, were, in fact, designed to take advantage of the super-high-resolution (SHR) graphics capabilities of the EPSON MX-80 printer to produce bar-code patterns of consistently acceptable quality.

Bar and space widths were selected such that a full 16-byte row could be printed within the constraints of a standard 8½ inch paper width. Extensive tests were then performed with a wand to insure that these barcode patterns could be reliably and accurately scanned, via free-hand scanning.

It became immediately evident that the height of the bar symbols generated by a single printing, approximately 8/72 inch, was not sufficient to allow for reliable scanning over a 16-byte row without the use of a guide ruler. And even with a ruler, some difficulty was still experienced. Consequently, it was decided to run further tests with double and triple printed rows, where-in the printer line spacing was set to print the top "dots" of each succeeding row 1/72 inch below the bottom "dots" of the previous row.

Based upon further tests, the double rows seemed to be tall enough to permit reliable free-hand scanning, provided that the operator would exercise reasonable attention. Triple printed rows were obviously better and easier to scan, but not worth the additional time penalty in the author's opinion.

Therefore, from this point onward, all planning was done on the basis of a double printed row for each line of barcode. This will permit a maximum of 20 rows to be printed on each 11 inch page, without any excessive overcrowding.

Although it was conclusively demonstrated that the EPSON MX-80 is capable of generating good quality barcode, additional factors still had to be considered in regard to the overall feasibility of using the HP-41CV to drive the printer, and the design of programs to accomplish the complex task of generating, processing and printing program barcode. The principal considerations were as follows:

- o MEMORY USAGE
  How much memory will be required? Will there be enough storage capacity left over to accomodate a reasonably sized object program (the program to be printed) ?

- o SPEED OF EXECUTION
  What is the fastest rate at which a predetermined full 16-byte double row of barcode can be printed? How close to this speed can we come in an actual operational program?

- o MEMORY vs TIME CONSIDERATIONS
  What are the tradeoffs, if any, between memory usage and the speed of execution? What constitutes an optimum balance?

- o SYSTEM CONFIGURATIONS
  How many ROM modules, special EPROM sets, etc. should be included?

- o USER CONSIDERATIONS
  What user considerations should be included to allow for ease of operation, minimum cost, most universal acceptance, etc.

The very first major task was to develop the specific methodology and software for driving the printer - i.e. the print routines. Here, memory and speed were the two principal concerns.

Intuitively, it seemed that the highest printing speed would result if the printing were to be done a byte at a time. Since there are 256 different permutations of an 8-bit byte, then the storage capacity would have to be sufficient for each of 256 individual print routines, averaging about 115 bytes each, for a total of almost 30,000 bytes, just for driving the printer in the SHR graphics mode. This, obviously, is completely out of the question, and will not be given any further consideration.

Some analysis and experimentation were then done to print the barcodes by the Hex digit (nybble), a half-nybble of two bits, and a single bit. Single bit printing was eliminated for reasons of inefficiency and slow speed. Now the choice was narrowed down to only two alternatives.

Printing on a 4-bit Hex digit basis would require 16 routines for all of the Hex digits, plus a left end and a right end 2-bar pattern. Straightforward programming methods would result in a total of a little over 1,100 bytes for these 18 print routines. This memory requirement is high, but it is somewhat offset by relatively fast printing speed, as well as the ease with which the HP-41 can process hex digits, using clever programming methods.

On the other hand, it could be seen that printing by a half-nybble (2 bits), would only require four different print routines. Not even any extras for the left and right end patterns. Memory requirements would come to only about 150 bytes. But this advantage had to be weighed against an approximately 50% slower printing speed and other inefficiencies brought about by the necessity to process 2-bit entities.

A decision was finally made (and I'm still not completely sure that it was the best one) to go with the 4-bit Hex digit approach in order to maximize speed and programming efficiency - but with a visceral feeling that some means could be found to achieve a substantial reduction in the 1,100+ bytes of memory required for the storage of these print routines.

Some discussion is now in order regarding the specific details of how the EPSON MX-80 printer generates barcode patterns. As mentionned elsewhere, the SHR graphics mode must be used since it provides a horizontal density of 120 dots per inch. Therefore, we can generate a vertical column of dots every 1/120 inch along the horizontal dimension. For barcode purposes, this column may be either a blank space or a full 8 (vertical) dots. It was found, by trial and error, that 2 contiguous columns of 8 dots for a "zero" bar, 4 contiguous columns of 8 dots for a "one" bar, and 3 contiguous blank columns for the spacing between bars, produced barcode patterns which could be reliably scanned, and which would fit well within the width of a standard 8½" paper form, even for an all F's row of 16 bytes plus end bars. Since each vertical column of 8 dots has a height of about 8/72 inch, a second duplicate pattern had to be printed 1/72 inch immediately below the first. The resulting height of 16/72 inch was quite satisfactory for free-hand scanning with a standard optical wand.

It might here be worthwhile to note that the EPSON MX-80 also provides a normal high-resolution (HR) graphics mode which has a horizontal density of only 60 dots per inch, compared to 120 dots per inch for the SHR graphics mode. Testing showed that the 60 dots per inch could not generate readable barcode. Unfortunately, the earlier EPSON MX-70 model has only the HR capability, and thus can not be used for barcode printing.

The basic programming methodology for SHR graphics is to enter the control sequence of 27, 76, N_1, N_2 into the printer buffer, followed by the binary codes for each vertical dot column, where:

> 27 & 76 are the escape code & control code for SHR mode,
> N_1 is a number of dot columns from 0 to 255,
> N_2 is a number of multiples of 256 dot columns, from 0 to 7.
>
> Each dot column is encoded from 0 to 255, the binary pattern of which determines which of the 8 vertically stacked printer head pins are fired up. (e.g., 0 is a blank; 255 fires all 8.)

Once the complete control sequence of 4 numbers has been entered into the printer buffer, all subsequent inputs, up to the total number of columns specified by N_1 and N_2, are interpreted only as dot column codes. Therefore, until this total number of entries has been made, the HP-41 (or any other driving computer) can not execute any instructions whatsoever. Such instruction inputs would be interpreted as coded dot columns, and printed accordingly. The only

exceptions to this rule are for those instructions which transfer data from the X-register or Alpha-register into the printer buffer.

A sample set of program steps for the Hex digit "A", whose binary code is 1010, takes the following form:

```
   MANIO          01 LBL 65      10 OUTA        22 OUTA
   FIX 0          02 27          11 OUTA        23 OUTA
   CF 29          03 ACCHR       12 OUTA        24 OUTA
   SF 17          04 76          13 OUTA        25 OUTA
   CLA            05 ACCHR       14 ACCHR       26 ACCHR
   255            06 24          15 ACCHR       27 ACCHR
   XTOA           07 ACCHR       16 ACCHR       28 ACCHR
 ---------------  08 0           17 OUTA        29 OUTA
                  09 ACCHR       18 OUTA        30 OUTA
                                 19 ACCHR       31 ACCHR
                                 20 ACCHR       32 ACCHR
                                 21 ACCHR       33 ACCHR
                                                34 RTN
```

Note that a total of 24 columns were specified by N_1 and N_2 in the control sequence, and by initializing the Alpha register at a prior time, we are able to use the ACCHR function to load a blank column, for the between-bars spaces, from a zeroed X-register; and OUTA is used to load a full 8-dot column from the Alpha-register. Execution of this LBL 65 routine will produce the printout of the appropriate 1010 barcode pattern. For reasons which will become apparent later on, please note that the label numbers for the Hex digit print routines were made identical to the HP-41 character codes for those digits. (The reader is referred to the HP-41 byte table)

The total size of just this one label routine is 66 bytes. In the aggregate, similar routines for all 16 Hex digits, plus a left and a right end pattern, come to just over 1,100 bytes.

Speed tests were performed to see how fast a full row of 32 hex digits (16 bytes), plus left and right end patterns, could be printed using these routines. The results were disappointingly slow, even with an HP-41CV running at 2X speed. Moral of the story: Races will never be won using SHR graphics!

At this point in the development, progress came to a complete stop, while the author contemplated his navel, licked his spiritual wounds, and tried to think of a more efficient approach.

After several weeks of incubation, the answer came one day - all of a sudden. So simple — so elementary — why don't we ever think of these things right away?? The solution: Convert the program sequences to synthetic alpha strings. By so doing, we might be able to realize large reductions in storage requirements as well as a very substantial increase in speed. There were some serious doubts, however, as to whether this technique would really be compatible with the touchy SHR graphics mode, or whether all kinds of unpredictable printer responses might ensue.

Now, with new enthusiasm, work was resumed, and again, after much trial and error, a set of routines was perfected, using synthetic alpha strings. The total size was reduced from 1,110 bytes down to 669 bytes, a reduction of 40%. And even more exciting, a full 16-byte, single row of barcode could be printed in just under 15 seconds, using direct execution of these routines, again with an HP-41 modified to run at 2X speed.

A covering Global Label of "ANYBLS" was given to these routines, and the complete program set was stored on tape. A printout of the "ANYBLS" program, the detailed coding of the synthetic alpha strings, and full explanatory notes are attached to this commentary discussion as a separate enclosure. Readers are encouraged to read this attachment, and to become familiar with the methodology before continuing with this dissertation, since "ANYBLS" is incorporated in-toto, less the global label, into PBC2.

Alas, the 2nd corollary to Reagan's "No Free Lunch Principle" states very clearly that there's always a hidden price to be paid for such miracles! And this one is no exception to the rule.

The astute reader may have observed that in the original program sequence, the ACCHR function was used to transmit an all-zeros column from the X-register to the printer buffer; and the OUTA function was used to transmit an all-ones column from Alpha-register to buffer. In the synthetic alpha string version, however, the ACA function must be used to load the buffer from Alpha, because we have combined zero sequences (nulls) and ones sequences in the same strings, and OUTA will NOT pass any null characters whatsoever, even those in the middle of a string. (Try it yourself, and you'll soon see.) But, on the other hand, OUTA will pass a full 8-bit character to the buffer, and ACA will pass only 7 bits. (Minor details which were never revealed by "Mother HP") So the bottom line for us is that, in return for the savings in storage and the increase in speed, we have to live with a vertical column height of 7 dots instead of 8 dots - a reduction of 1/72 inch. With the double printing, however, which was necessary even with the 8 dots, further tests demonstrated that free-hand scanning can still be reliably performed on a double row which has a net 1/36 inch less height than originally planned.

Now that the problem of the print routines had been solved by reducing the memory requirements to a reasonable size and being able to demonstrate a very respectable print speed, an operational program had to be developed to generate and process program barcode, using the "ANYBLS" routines to execute the printout functions.

All five principal concerns outlined earlier in this discussion now had to be considered in the establishment of a program design plan. Again, there's no certainty that the selections made were really optimum, but the necessary selections were made, and it was decided to proceed along the following lines:

1. An EPSON MX-80 (or other EPSON printer with SHR graphics capabilities) connected to the HP-IL via an HP-82166A converter is the basis of the entire plan. However, it is assumed that an HP-82905B printer, which is equivalent to an EPSON MX-80 with a built-in converter, will also work equally well.

2. The HP Plotter ROM module will be used, if possible, to generate the program barcode bit pattern in the Alpha register, due to its speed and simplicity. Additionally, its capability to generate bit patterns for all other types of barcode could be used with future modifications of this program to print such barcode. By using this ROM, we also eliminate the need for any program storage to generate the bit patterns, except that the Plotter ROM does require the establishment of a 26 register I/O buffer in RAM when generating program type barcode.

   Other methods of generating the bit pattern, such as that used in the "BAP" program by W. Maschke (7356), as published in the PPCCJ V9N4 pp. 44-45, were rejected due to much slower speed of execution, requiring substantial amounts of memory, and lacking the flexibility for producing other types of barcode. One point here, however, that might appeal to some users is the feature of Maschke's program which allows the object program to be stored and operated upon in XM. Thus, there is no main memory requirement at all for the object program. The disadvantage is that the object program must be the very first file in the XFM module - which means, of course, that all of XM must be cleared out of everything else which the user might have stored in XM, in order to position the object program at the very beginning of the XM.

   The use of the Plotter ROM does place a financial burden on the user for the module (or EPROM copy) and enough port capacity (or EPROM box capacity) to incorporate it into the system. As a possible alternative, Maschke's method can probably be adapted to this program by replacing the "BCP" function of the Plotter ROM used here with Maschke's front end method. It doesn't seem to be an impossible task.

3. As an additional method of increasing speed of execution, it was also decided to use several functions of a modified and augmented NFCROM EPROM set; with an alternative method which can be easily implemented for a modified but not augmented NFCROM; or as another alternative, the use of a PPC EPROM-2** EPROM set in lieu of the NFCROM*.

   If the NFCROM EPROM set is to be used at all, then at the very least, its XROM number 17 must be changed to avoid a conflict with the lower 4K page of the Plotter ROM which is also XROM 17, and which must remain fixed as is.

---

* The NFCROM EPROM contains a useful set of assembly language functions for use with the ProtoSYSTEM, as well as additional general functions for the HP-41 which are not otherwise provided by HP. NFCROM can be obtained from Nelson F. Crowle, PROTOTECH Inc., P.O.Box 12104, Boulder, Colorado 80303.

** The PPC EPROM-2 contains two sets of generally useful assembly language functions for the HP-41, originally known as "JIMROM" (for Jim DeArras) and "MELBROM" (for Melbourne Chapter of PPC). It has been widely distributed. PPC members should have little or no difficulty in locating a set from which duplicates can be burned.

The author changed the XROM Number of his NFCROM to 16, but any other generally non-conflicting number would do just as well. Additionally, several new assembly language functions were added to the author's NFCROM. Two of these augmented functions, NRCL and PPACK, are used in PBC2. These two particular functions originated in PPC EPROM-2, and were copied from there into the modified NFCROM.

Since there were also bugs in two of the other functions of the original NFCROM-1B, it was the author's personal preference to burn a new chip set after making all the modifications and additions, and correcting the two bugs.

Owners of an MLDL or a ProtoCODER should have no problem in adapting the NFCROM for use with these programs. Other users, however, who don't have these capabilities, will have to use the PPC EPROM-2. Detailed instructions are provided elsewhere for modifying these programs to work with each of the two alternative EPROM set options described above.

4. The PPC ROM is used in a very minor way, as a convenience for paper advances, with the PO function. If a PPC ROM is not available, the user will have to replace the PO functions with ADV's in accordance with the specific change instructions provided elsewhere.

   The LB function of the PPC ROM is extremely useful, however, for loading the many synthetic Alpha strings into program memory. If a PPC ROM module isn't readily available for this purpose, the user should plan to spend approximately $2^{18}-1$ hours loading all the ANYBLS synthetic alpha strings. Alternatively, for those users who may have one, the ASSEMBLER-3 EPROM set, developed by our colleagues "down under" in the Melbourne Chapter, provides even easier and faster means.

5. XFM/XM will be utilized for several extended functions, as well as for storage of PBC1, PBC2 and the object program. PBC1 and PBC2 require a combined total of 178 XM registers. Each object program will be limited to approximately 95 registers in size. These programs may be stored anywhere within the XM.

6. The total program will be split into two parts: PBC1 and PBC2. PBC1 will be called into main memory and executed by the user, whereby it will carry out all initializing functions, etc. Its very last step will be to replace itself with PBC2. Use of this technique allows more storage capacity for an object program than would otherwise be available.

Once these system parameters and configurations had been specified, a generalized flowchart diagram of the entire plan was prepared to serve as a roadmap for the preparation of more detailed sequences and the subsequent programs. A detailed flowchart diagram is provided as a separate enclosure, and references will be made to the various numbered boxes, as well as to specific program lines, in the later discussion of the program sequences.

Before going into these flowchart sequences, the methodology used for executing the synthetic printing routines must be explained in detail. Unless the specific technique is very clearly understood, the reader may not be able to understand other related elements of these programs.

As the reader will recall, the labeling of the print routines was established in such a way as to take advantage of equivalency between the character code for each hex digit and the label number. Thus, when processing a string of hex digits in the Alpha register, if the hex digit "A" happened to be the next digit in sequence, the function ATOX will remove the "A" from the left side of Alpha and enter its character code of 65 into the X-register. This numeric character code can then be sequentially stored in some register, say R-nn. A little later on, when we do the actual printing, all we need to do is an XEQ IND nn, which executes LBL 65, which will print the appropriate barcode pattern. By storing the character codes for all 32 digits of a row of barcode into 32 sequential registers, it's a very easy matter to use this method of indirect addressing together with the ISG function to increment the value of nn, and print the entire row.

The very first operational program that was written actually used this indirect addressing technique to execute the print routines. Although this first program did indeed prove that the HP-41CV & EPSON MX-80 combination could do the job, the speed performance was most disappointing. Whereas it had already been demonstrated that direct execution of the print routines could print a full double-printed 16-byte row of barcode in less than 30 seconds, the actual operational program was taking almost 2 minutes per row (again, with an HP-41CV modified for 2X speed).

This would never do. But what are the alternatives? Since it's not known à-priori what labels are to be executed until the barcode bit patterns are generated and processed by the program itself, the indirect addressing technique seems to be the only means which HP has provided for handling such situations. It's easy to use, but the execution time leaves much (like everything) to be desired. There's no compilation, and each & every execution can take up to or over a full second of time! The proprietor of my favorite Chinese restaurant can operate his abacus faster than that!

Very clearly (or maybe even not so clearly) it was now time, again, to stop and contemplate my navel, and maybe even fantasize about the possibilites of owning a real computer someday soon. (Shades of deja-vu! Weren't we here once before?)

In the midst of one of these transcendental reveries, a faint voice was heard from somewhere (maybe out in PPC Land?) on the other side of that curtain of the mind that separates reality from dreams. Very, very slowly, there began to emerge a coherent structure of an idea, until at last a possible solution became apparent.

Since HP doesn't provide the capability to create instruction steps within a running program, which can then be executed by that program, let's fool the operating system by creating these instructions synthetically, storing them as NNN data on the other side of the curtain, then raising the curtain, and executing these instructions. We did, in fact, manage to fool the system once already. Could we push our luck and do it again?

To summarize, a lot of experimentation was done, and after many iterations and failures, a rather simple technique was devised to create instructions synthetically and store their NNN bit patterns into data registers above the curtain. Furthermore, if the proper magic words are uttered, these sequences can be executed without the necessity of even bothering to raise the curtain at all! Execution of steps that don't exist? Now that's real magic! Unfortunately, there's still no exception made, even in this instance, to Reagan's "No Free Lunch Principle." As will be explained below, it costs us 18 registers of memory for these "non-existent" program steps.

Notwithstanding the price, we can now realize the full speed advantages of direct execution. As it turned out, the first pass along each double-printed row takes about twice as long as the repeat printing because the XEQ's jump distances are not compiled the first time. After the first printing of each row, however, all jump distances have been compiled, and the repeat printing is reduced from about 30 seconds down to 15 seconds, right at the benchmark limit which we had earlier established.

The "magic" that gets us up there to begin with is the synthetic compilation of the XEQ 99 step in Line 222 of PBC2, which is done by the operations in Lines 134-137.

By synthetically compiling this XEQ function, we can cause a jump to be made into any part of memory, and the O/S doesn't seem to care which side of the curtain it's on. Once there, it will execute any instructions in the same manner as if they were below the curtain. It follows, of course, that any changes made to the program which will change this jump distance will require us to also make a change in the synthetic compilation steps of Lines 134 and 136.

In addition, for all rows of barcode, the entry point is LBL 99, followed by XEQ 47 for the left end bars. Similarly, the last two steps of all rows are an XEQ 71 for the right end bars, followed by a RTN. Consequently, we can create these steps synthetically in the PBC1 program, and store them in R-17 and R-00, at two program steps per register, respectively, wherein they will reside as constants throughout the duration of the programs. Since the final row of barcode is usually less than a full 16 bytes long, we will also save the uncompiled last two steps, and store them into the proper register later on when the final row is processed. In between R-17 and R-00, the 16 intervening registers are used to store the synthetic XEQ's for all 32 hex digits of each normal 16-byte row, at two steps per register.

One other special technique was also used to help speed up the print operations. This was to place the entire sequence of synthetic print routines (669 bytes) at the very top of PBC2, immediately after the global label and a GTO step. By locating these routines in this area, from Lines 3 to 127, the hidden XEQ steps, which reside immediately above the curtain, do not have to spend as much time searching for their destination labels when they are executed in uncompiled form for the first printing of each row.

THE PROGRAM SEQUENCES

We have now covered most of the pertinent system considerations, as well as the hardware configurations and details of the print routines. Any reader who has managed to absorb everything thus far, and still remain interested, is hereby made a Provisional Member of "The Order of the Nybble" and is entitled to all the rights and privileges thereto.

Hopefully, all of this detailed background information will enable the new provisional members to keep up a fast pace as we proceed through the entire sequence of program operations. The flowchart diagram and the program listings will both be used as references.

The following paragraphs will deal primarily with the more interesting or important elements of the programs. Little or nothing will be said about those portions of the programs which have been discussed previously, or which should be readily understood by most readers of this paper (a generally sophisticated group of users).

BLOCK 0

If they have not already been previously stored there, enter PBC1, PBC2 and the object program into XM. It is most important that main memory is packed with GTO.. before storing these programs into XM, to insure that the byte counts will be correct.

Clear out program memory of all other programs. Bring PBC1 in from XM storage. Do NOT pack with GTO.. now because this will add a normal program END in addition to the permanent .END., and will thus prevent PBC1 from replacing itself with PBC2 later in the program sequence (BLOCK 1.6).

BLOCK 1.1 (Lines 02-21)

Execution of PINIT establishes a 26-register I/O buffer at the bottom of program memory. This is required by the Plotter ROM for program type barcode generation. Flag 25 (the error ignore flag) must first be set if there is no plotter in the loop. Furthermore, the HP-41 must be in the AUTOIO mode (Flag 32 clear) or PINIT will not execute. We're really tricking the Plotter ROM into seeing an imaginary HP82162A thermal printer. Otherwise, if the MANIO mode is set at this time, which would be the normal procedure with the EPSON printer in the loop, PINIT will not execute, and we will just get a "NONEXISTENT" error message in the display. It took several weeks to figure this one out by careful study of an HP ERS document (also, by a lucky guess). This is the reason for the phrase "if possible" in the first line of Paragraph 2, on Page 6 of this paper. At the time of making the decision to use the Plotter ROM, the PINIT execution problem had not yet been solved, and there was no real confidence that it could be solved.

Once PINIT has been executed, we can then go right into MANIO mode (Flag 32 set), which is necessary for any peripheral device connected to the HP-IL converter. Flag 17 is also set in order to inhibit CR/LF after each hex digit barcode pattern is printed. At the end of an entire row, PRBUF will be executed to override this inhibition. SIZE is set to 59; the display mode is set to avoid

decimal points and fractions; the page-row counter is set to 20 for keeping count of the number of rows per page.(We're going to print 20 rows of barcode on each 8½"x 11" form); and the sequential row counter is set to 1.

As explained earlier, synthetically coded program lines for the instruction steps LBL 99 and XEQ 47 are generated as Alpha strings and stored in NNN form in R-17. Similarly, the codes for XEQ 71 and RTN are generated and stored in R-00, with an uncompiled copy being saved in R-19 for later use when the final row of barcode will be processed.

BLOCK 1.2 (Lines 22-46)

The program is halted, and instructions for the user are printed out, requesting the user to take the following steps:

(1) Manually advance the paper to the top of the next page form (i.e., right under the next horizontal perforated line).

(2) Toggle the printer power switch off and on.

(3) Press the R/S key to continue the program.

This sequence of operations is required to enable printing to start at the top of a fresh 11 inch page, and to enable the barcode to be correctly printed at 20 rows per page.

BLOCK 1.3 (Lines 47-56)

The user is prompted for the name (label) of the object program. After keying the label into Alpha, pressing the R/S key will start the program running again. The name of the object program is stored for future use (for page headers if additional pages are needed), and is also printed at the top of the page in double-width emphasized format. And line spacing is also set to 7/72".

BLOCK 1.4 (Line 57)

The RCLPTA function is executed to return the size, in bytes, of the object program (whose label is still in Alpha) to the X-register.

BLOCK 1.5 (Lines 58-65)

The size, in bytes, of the object program is printed under the name at the top of the page, in normal size emphasized format. The printer is then restored to normal printing mode.

BLOCK 1.6 (Lines 66-67)

PBC2 is called into main memory from XM. Since we took care (in BLOCK 0) not to provide PBC1 with a normal program END, GETP will cause PBC1 to be replaced by PBC2, and program execution of PBC2 will automatically commence.

We have now come to the "END" of PBC1, and perhaps a short tutorial might be of interest at this point in regard to some of the more obscure characteristics of END's and .END.'s.

When any program is brought into main memory from XM, the normal program END (ordinarily, the last numbered step of a program) is lost. Since this program has now become the very last program in main memory, the normal penultimate step of this program is there-

fore followed by the permanent .END.. And since this permanent .END. always occupies only the three righthandmost bytes of a register, then there may be anywhere from 0 to 6 null bytes in between the normal penultimate step of this last program and the permanent .END.. The exact number of null bytes depends upon the position (in its register) of the last byte of the penultimate program step.

If a GTO.. is now executed, the last program will be packed, and the normal program END will also be restored as the last step of the program. (Note that GTO.. is non-programmable.) Now there may be nulls in between the END and the permanent .END., depending upon the position, in its register, of the END.

Alternatively, if a PACK or PPACK function is executed instead of a GTO.., any non-essential nulls within the program will be eliminated, but a normal program END will NOT be restored to this last program, and anywhere from 0 to 6 nulls may still exist between the penultimate function and the permanent .END..

If, now, a GETSUB instruction is executed to bring in some other program from XM to main memory, the previous last program will become the next-to-last. Furthermore, the previous permanent .END. will be automatically changed into a normal program END, but will continue to occupy the same three bytes as it did before GETSUB was executed. Since there is no packing under such circumstances, then any nulls which may have existed in front of the previous permanent .END. will still exist. Now, however, a PACK or PPACK can be used to remove these particular nulls.

It can be seen, therefore, that while a program is running, the only way in which a normal program END can be restored to the last program in main memory, which doesn't have such an END, is for the running program to execute a GETSUB instruction which will bring in another program from XM.

In order to conserve as much memory as possible, it was decided not to include any steps which would restore a normal program END to the object program after it is called in from XM (BLOCK 2.1). A PPACK step has been included, however, to remove any nulls from in front of the END of PBC2, and more importantly, to meet the conditions for BCP (BLOCK 2.4) which will not initiate an execution upon an unpacked object program.

As a consequence of these null bytes which may exist just in front of the permanent .END., the program size data,which was printed at the top of the page will not necessarily be the same as the number of bytes of barcode. After all the barcode is scanned into the HP-41 with the wand, however, a GTO.. will eliminate any such nulls, and also restore the normal program END. Since the barcode .END. is a variation of a permanent .END., it will locate itself in the righthandmost three bytes of the appropriate register in any event, and nulls would probably result until cleared by a GTO..

For the benefit of those very few truly exceptional readers whose obsessive curiosity will not allow them any peace of mind until they can see the coding structure of these END's and .END.'s, the following tabulation is provided. Six different variations in the coding of the third byte are delineated, as well as the circumstances under which each such variation may be found:

TABULATION: The ends of END's and .END.'s*

	HEX CODE	
(1)	Ca bc ØD	Normal program END. The program is NOT packed.
(2)	Ca bc Ø9	Normal program END. The program IS packed.
(3)	Ca bc 2D	Permanent .END.. The last program in memory does NOT have a normal END, and has NOT been packed.
(4)	Ca bc 29	Permanent .END.. The last program in memory does NOT have a normal END, but it IS packed.
(5)	Ca bc 2Ø	Permanent .END.. The last program in memory HAS a normal END. Individual programs in memory may be either packed or unpacked.
(6)	Ca bc 2F	Special permanent .END. generated by the BCP function of the Plotter ROM. This may be observed in the barcode pattern which is printed.

BLOCK 2.1 (Lines 128-133)

For reasons which were explained previously, the 125 steps of the print routines (ANYBLS) were located at the very beginning of PBC2. We will, therefore, jump around them to Label 98 (at Line 128) to begin the new sequence of operations in this program.

GETSUB first calls in the object program from XM; PPACK then clears out any nulls which may exist in front of the END of PBC2, and also prepares the object program for subsequent execution by BCP (in BLOCK 2.4). If these steps are not completely understood, the reader should go back and review the previous discussion of END's and .END.'s, under BLOCK 1.6.

The sole purpose of the STOP function at Line 133 is to provide the author with an opportunity to switch the HP-41 into 2X speed at this point. Just press the R/S key to continue processing. The normal 1X speed was used up to this point because it seemed that certain conditions of mid-summer temperature and humidity might be the cause of problems with the author's triple XFM/XM/XM module when the ProtoCODER was also connected, and an attempt was made to execute any XM function when switched to the 2X speed setting.

(This was just one of many frustrating hardware problems that have plagued the author over the last few months. An intermittently bad HP-IL module was another example. It caused the HP-41 to crash at frequent and unpredictable times. The module finally "died", and only then could the cause of the previous crashes be diagnosed. Connectors are yet another source of transient poltergeists. In summary, it seems that the HP-41 may not have been designed well enough to withstand all the mechanical and electrical stresses, over prolonged periods of time, which we impose upon it. To the extent that any of these super-system configurations running at 2X speed, etc. do seem to work most of the time, maybe we're just plain lucky. Ultimately, Murphy's Laws always seem to catch up with us!)

Users who don't have a switchable speed capability should delete this STOP. Or anyone else who may not need it may do likewise. If it is deleted, however, a 1-byte NOP must be inserted in its place.

*NOTE: For detailed information regarding the coding of hex digits a, b and c, the reader is referred to pp. 15-17 of "Synthetic Programming on the HP-41C" by W.C.Wickes, Larken Publications, 1980.

BLOCK 2.2 (Lines 134-137)

This is one of the most interesting parts of the program, where a synthetic compilation is executed upon the XEQ 99 instruction of Line 222, as generally described previously on Page 9.

In PBC2, as provided herewith, all three bytes of this instruction reside in absolute address 327_d. Label 99, the destination point, was synthetically created in PBC1 (BLOCK 1.1) and stored in data register R-17, which is absolute address 470_d. The jump distance is equal to the total number of registers and bytes, starting from the first byte of the XEQ function to the byte immediately preceding the designated label. Prior to any compilation, the hex codes for these two registers are as follows:

(1) Abs. Address 470_d (Data Reg. 17)----- 00 00 CF 63 E0 00 2F

(2) Abs. Address 327_d (Uncompiled)------- E0 00 63 A7 4A AA 01
(XEQ 99)

It can be seen that the jump distance amounts to 142 registers plus 6 bytes, and compilation of the XEQ 99 instruction will result in the following coded sequence:*

Abs. Address 327_d (Compiled) -------- EC 8E E3 A7 4A AA 01
(XEQ 99)

The Alpha string in Line 134 contains the decimal byte equivalents to these hex codes. The functions in the next 3 steps store this sequence as an NNN into program memory, where it overwrites the original uncompiled contents of absolute address 327_d.

It can be very clearly seen that any program changes which would change this jump distance must be analyzed, and new compilation data would have to be entered into Line 134, and possibly Line 136, in place of the present data. Moreover, in order to avoid the problem of entering compilation data into two registers, we should also insure that all three bytes of the XEQ 99 instruction will reside in one single register. In the instant case, the three XEQ 99 bytes are located in the lefthandmost three byte positions of register 327_d. If the program is changed in such a way that will delete any bytes from areas above Line 222, then these three bytes will be shifted to the left and upwards. Thus a decrease of only 1 or 2 bytes will result in a split between 327_d and 328_d; but a decrease of 3 to 7 bytes will cause all three bytes of this instruction to move into 328_d. Alternatively, an increase of 1 to 4 bytes will shift the 3-byte instruction to the right, but totally within the same register; but an increase of 5 or 6 bytes will cause a split between 327_d and 326_d. Where splits will otherwise occur, it is preferable to add NOP filler steps at some convenient location in order to retain the entire 3-byte instruction within a single register.

BLOCK 2.3 (Lines 138-142)

Label 15 marks the point where processing will begin for each new row of barcode. A label number greater than 14 was used because GTO's will be executed to return to this point from other locations which are greater than 16 registers distant. The 3-byte GTO, which

---

*NOTE: The reader is referred, again, to pp. 15-17 of Wickes's book for an excellent tutorial discussion of jump distance coding.

is automatically called up by the O/S whenever the label number is greater than 14 results in the fastest possible execution time for these longer jump distances.

RCL 20 brings the new row number into X. ARCL X then appends it to Alpha, wherefrom the sequential row number is printed in normal print mode, directly above each barcode row. The reason for using both a RCL and ARCL when just a simple ARCL 20 would suffice, is because BCP, which follows in Line 143, requires the row number to be in the X-register as a precondition.

BLOCK 2.4 (Lines 143-144)

BCP, a Plotter ROM function, generates the bit pattern for an entire row of program barcode in the Alpha register. A full 16-byte row includes 7 bytes in the M-register, 7 bytes in the N-register and 2 bytes in the O-register.

Based upon information which wasn't seen in the Plotter ROM manual, but was found in the ERS document, a couple of shortcuts were taken to save time and bytes.

(1) According to the manual, it's necessary to enter the name of the object program into the Y-register. The ERS, however, indicates that BCP will be executed upon the last program in memory in the absence of any Alpha data in the Y-register.

(2) Again, according to the manual, we must enter a number, in the form rrr.bb, into the X-register, where rrr is the barcode row number to be produced and bb is the number of bytes per row. The ERS indicates that if bb is absent, a full 16-byte row will be produced.

Since it was decided to print with full 16-byte rows, only an integer (rrr) is loaded into X from R-20, which was originally initialized with a 1 in PBC1. Each time that BCP is executed, the row number is automatically incremented by 1. This incremented row number is then saved in R-20 for subsequent recall when we return later to begin processing of the next row.

After the pattern for the last row of barcode has been produced, a zero is left in X, which makes it very simple to test for a last row situation (BLOCK 2.5).

BCP also leaves a number in the Z-register, in the form fff.lll, which represents the first and last program line numbers in this row of barcode. Some people do find it useful for such program line numbers to be printed alongside the sequential row number. However, provisions have not been made to print program line numbers here because it wasn't judged to be of sufficient universal importance to be worth the time and memory which would be required. Any user who feels the need to print this data should be able to add the necessary program steps without any great difficulty. They must bear in mind that any such changes will also change the jump distance for XEQ 99, presently at Line 222. And this, in turn, will require changes to be made in Lines 134 and 136 (BLOCK 2.2).

There is yet one more useful output of BCP which we will put to good advantage here. This feature places the number of bytes actually produced into the T-register. Obviously, this number will be a 16 for all (normal) rows except the last, which will usually be some number less than 16. We will make use of this feature in the next BLOCK.

BLOCK 2.5 (Lines 145-146)

The contents of the X-register are tested for zero to determine whether the barcode pattern for the last (final) row of the object program has just been generated. If so, then it's necessary to go to Label 02 to compute special addressing and loop controls for the processing of this last row (BLOCK 2.7). Otherwise, addressing and loop controls will be established for a normal 16-byte row (BLOCK 2.6)

Before proceding any further with the program sequence, and at the risk of being considered somewhat redundant, this may be a good place to first provide a detailed explanation of the methodology which will be used to convert the bit pattern in Alpha, which was generated by BCP in BLOCK 2.4, into a sequence of instruction codes. These codes, in turn, will be stored as NNN's in data registers, and subsequently executed as program steps to print the required barcodes. Once this process becomes clear, readers should have no difficulty in understanding the program steps in BLOCKS 2.6 to 2.10.

The methodology can be summarized by 5 general steps:

(a) Store synthetically generated codes for LBL 99 and XEQ 47 into data register R-17. (This was previously done in BLOCK 1.1.)

(b) Store synthetically generated codes for XEQ 71 and RTN into data registers R-00 and R-19. (This was also previously done in BLOCK 1.1.)

(c) Convert 16 bytes of the Alpha register (i.e., all of M, all of N and 2 bytes of O) to the equivalent 32 hex digits.

(d) Convert the 32 hex digits to numeric character codes, and store into 32 consecutive data registers, R-27 to R-58.

(e) Use the stored character codes in R-27 to R-58 to synthetically generate codes for up to 32 instruction steps which will then execute the appropriate print routines. Store these codes as NNN data in R-16 to R-01, at two instructions per register.

For all normal 16-byte rows, execution of the print routines will subsequently start at Label 99 (located in R-17) and continue on through all the stored instruction steps, until the RTN (in R-00) causes a return to Line 223 of the main program.

Figure 1 illustrates the storage register usage for the character codes and the synthetically generated program steps which pertains when processing a normal 16-byte row.

For reasons of convenience and program simplicity, the procedures delineated in steps (c) and (d), above, were made the same for a last (final) row as for all normal 16-byte rows. Since this last row may vary from 4 bytes up to 16 bytes in size, then there may consequently be anywhere from 0 to 24 "zero" hex digits at the beginning of the 32-digit sequence before the first valid digit pops up.

F I G U R E 1

STORAGE REGISTER CONTENTS WHEN PROCESSING A NORMAL 16-BYTE BARCODE ROW

REGISTER NUMBER	STORED INFORMATION
R-58	Char.Code for Digit #32
R-57	Char.Code for Digit #31
...	...
...	...
R-28	Char.Code for Digit #2
R-27	Char.Code for Digit #1

FIGURE 1(A) - CHARACTER CODES

REGISTER NUMBER	INSTRUCTION CODES	
R-17	LBL 99	XEQ 47
R-16	XEQ #1 of 32	XEQ #2 of 32
...	...	
...		
R-01	XEQ #31 of 32	XEQ#32 of 32
R-00	XEQ 71	RTN

FIGURE 1(B) - INSTRUCTION CODES

F I G U R E 2

STORAGE REGISTER CONTENTS WHEN PROCESSING A FINAL BARCODE ROW OF 10 BYTES

REGISTER NUMBER	STORED INFORMATION
R-58	Char.Code for Digit #20
R-57	Char.Code for DIGIT #19
...	...
...	...
R-40	Char.Code for Digit #2
R-39	Char.Code for Digit #1
R-38	0
R-37	0
...	0
R-28	0
R-27	0

FIGURE 2(A) - CHARACTER CODES

REGISTER NUMBER	INSTRUCTION CODES	
R-17	LBL 99	XEQ 47
R-16	XEQ #1 of 20	XEQ #2 of 20
...	...	
...	...	
R-07	XEQ #19 of 20	XEQ #20 of 20
R-06	XEQ 71	RTN
R-05		
R-04		
R-03		
R-02		
R-01		
R-00		

FIGURE 2(B) - INSTRUCTION CODES

Therefore, when executing step (e) for the last row of barcode, we must bypass all these "zero" digits, and start with the first valid digit. Furthermore, we must now also store the uncompiled code for XEQ 71 and RTN, a copy of which was previously stored in R-19 (Remember that?), into the appropriate location.

Figure 2 illustrates the storage register usage for the character codes and instruction steps when processing a last row which, for sake of a typical example only, is 10 bytes long.

BLOCK 2.6 (Lines 147-151)

If the result of the zero test in Line 145 was negative, then two initial parameters will be established here, for later use in BLOCK 2.10, for processing a normal 16-byte row.

(1) The number 27 is stored in R-24. It represents the address of the first of 32 consecutive data registers used to store the hex digit character codes of a normal 16-byte row. This stored number will be used in BLOCK 2.10 for indirect addressing purposes.

(2) In addition, a number 16 is also stored in R-25. The integer portion represents the first location for storage of the synthetically created instruction steps. The decimal portion (.000, in this case) will be used to control the number of loop iterations for a normal 16-byte row.

A jump is then made down to BLOCK 2.8, to continue the normal processing sequence.

BLOCK 2.7 (Lines 158-177)

A positive zero test in Line 145 indicates that the last row of the object program has now been reached, and this row may be anywhere from 4 to 16 bytes in length. Since the exact length, in bytes, was entered into the T-register by BCP, this number is now recalled to X, where it will be used to compute three necessary parameters for processing the last row of barcode.

(1) The length of the last row is subtracted from 16 to determine the correct address into which the codes for XEQ 71 and RTN must be stored. In the typical example, illustrated in Figure 2, where the length of this last row is equal to 10 bytes, this computation results in a 6. The <> function of the NFCROM is then executed in Line 165 to interchange the contents of this address (R-06, in our example) with those of R-19, which contains an uncompiled set of codes for these two instructions. The <> function does not normalize any NNN's, but readers are warned that the REGSWAP function of XFM will not work with any NNN's.

(2) The next few lines divide this previous result by 1,000, add it to 16 and store the results into R-25. Again, referring to the example of Figure 2, this computation produces the number 16.006. Note that the integer portion of this number is the same as that used for normal 16-byte rows to specify the first address for storage of the synthetically created instruction steps. The decimal portion, however, is different from the

normal rows. When the integer portion is later decremented by a DSE function, the fractional number, .006, will terminate looping after instruction codes have been stored in R-07, thus loading these registers for the printing of exactly 20 hex digits (10 bytes).

(3) The third parameter is the quantity (59 - twice the length). This computation generates the address of the character code of the first valid hex digit. For normal 16-byte rows, this was R-27 (see Figure 1A). In our typical example where the last row's length is 10 bytes, this comes out to be R-39. It is left as an exercise for the reader to confirm the validity of this result, with the assistance of all 10 fingers, the Figure 2A, pencil & paper, and an HP-41. The result is then stored in R-24, and we're ready to move into BLOCK 2.8.

BLOCKS 2.8 and 2.9 (Lines 178-204 & 152-157)

Lots of program lines here, but nothing exotic. The NNN contents of M and N registers are temporarily stored for safekeeping while the Alpha register is being used for decoding, etc. Then we decode the 7 bytes of the 0-register and dump the leftmost 10 hex digits, saving only the last 4. These are converted in the Label 07 subroutine by the ATOX function and stored in memory. Then all 7 bytes of N and M registers, respectively, are similarly decoded, converted and stored. Figure 1A illustrates the ultimate result of these steps
- the storage of 32 character codes into R-27 to R-58.

BLOCK 2.10 (Lines 205-219)

This is another of the more interesting sections of the program: the place where we will synthetically create program code which will later be executed. Line 206 is a synthetic alpha string for the 2-byte prefix portion of a 3-byte XEQ instruction, which is E0 00 in hex code symbols. Now we recall the first of the stored hex digit character codes, convert it to an Alpha character and append it to the 2-byte prefix already in Alpha.

They said "It couldn't be done." But our program has now synthetically created code for new instructions which will later be executed by the program!

Returning now to this first instruction, if the first hex digit postfix was an "A", then the hex code for the complete 3-byte instruction is E0 00 41 (in uncompiled form). Translated to program line format, this can be read as XEQ 65. The reader is now reminded of the way in which the numbering was originally established for the print routine labels (See p.4, p.8 and the ANYBLS paper) so that the label numbers would be identical to the character codes. In this example, XEQ 65 will cause the barcode pattern for hex digit "A" to be printed, without any further conversion or translation. And similarly, for all other hex digits.

The remaining program steps in this BLOCK are reasonably straightforward. The contents of R-24 are incremented after XTOA. CLX is merely a 1-byte NOP step. Then we append the code for a second instruction, increment R-24 again, and store the combined code for the two instruction steps as per the address contained in R-25, which is also used for loop control.

Instruction codes are thus entered into data registers, from R-16 down to R-01 for normal 16-byte rows, or down to some other address for shorter last rows.

Some of the more astute readers may have observed that there are 2 null bytes in R-17, 1 null in each of R-16 to R-01, and 3 nulls in R-00. These nulls have not caused any noticeably adverse effects. The time and memory required to eliminate them would be most disproportionate to the marginal increase in speed, if any, which might result from their removal.

BLOCK 2.11 (Lines 220-227)

At long last, this is IT! The XEQ 99 (Line 222), which we had precompiled by synthetic means in BLOCK 2.2, now moves the pointer back behind the curtain to Label 99, and the EPSON printer will be off and running! Note that Flag 1 was set to provide a simple means for printing the row a second time, 7/72" below the first printing, as described previously. And a PRBUF at Line 223 takes care of the CR/LF requirement at the end of each of these printings.

Observe carefully. You will be able to discern the printing of each individual hex digit and end pattern as the print head moves across the paper, making its characteristic sounds.

Again, notice that after the first CR/LF, the repeat printing of the row will be done at approximately twice the speed of the first printing because all the XEQ steps became compiled after their first execution.

BLOCK 2.12 (Lines 228-230)

Upon completion of the printout of each row of barcode, another zero test is done (on the contents of X) to test for the last row. Go to Label 06 for a positive test result. Otherwise, continue on.

BLOCK 2.13 (Lines 231-232)

If the zero test for a last row was negative, the lines per row counter in R-26 is decremented by 1. When it reaches zero, we've printed 20 rows of barcode on the current page. If not, then the program sequence will return to Label 15 to start processing of the next row (BLOCK 2.3).

BLOCK 2.14 (Lines 233-234)

After R-26 is decremented to zero, the counter is re-set to 20, for starting a fresh row count on the next page.

BLOCK 2.15 (Lines 235-245)

Th paper is advanced to the top of the next page form (TOF), and header information is printed at the top of this new page: The label (name) in double-width/emphasized format, followed by a "(CONTINUED)" in standard-width/emphasized format. Now we can return to Label 15 to start processing of the next row in the usual manner (BLOCK 2.3).

BLOCK 2.16 (Lines 246-253)

If the zero test of BLOCK 2.12 was positive, this means that the last row of barcode has now been printed, and we're all through.

An "END OF PROGRAM" message is now printed just below the last row of barcode, the special I/O buffer is cleared from memory, and the AUTOIO mode is restored.

THE .END.

You, dear readers, have also now reached the .END.. CONGRATULATIONS!

C I T A T I O N

IN RECOGNITION OF THEIR NOTABLE ACHIEVEMENT IN REACHING THE .END. OF THIS LENGTHY DISSERTATION, FULL AND IRREVOCABLE LIFETIME MEMBERSHIP STATUS IN THE ORDER OF THE NYBBLE IS HEREBY AND HEREWITH BESTOWED UPON ALL FORMER PROVISIONAL MEMBERS WHOSE FORTITUDE HAS ENABLED THEM TO REACH THIS POINT. GO FORTH NOW AMONGST THE MULTITUDES, AND MAY YOUR BARCODES PAPER THE EARTH FOREVER!

PBC1

PROGRAM SIZE = 266 BYTES

ROW 1

ROW 2

ROW 3

ROW 4

ROW 5

ROW 6

ROW 7

ROW 8

ROW 9

ROW 10

ROW 11

ROW 12

ROW 13

ROW 14

ROW 15

ROW 16

ROW 17

ROW 18

ROW 19

ROW 20

ROW 21

END OF PROGRAM

PBC2

Rev: Aug. 29, 1983

PROGRAM SIZE = 969 BYTES

ROW 1

ROW 2

ROW 3

ROW 4

ROW 5

ROW 6

ROW 7

ROW 8

ROW 9

ROW 10

ROW 11

ROW 12

ROW 13

ROW 14

ROW 15

ROW 16

ROW 17

ROW 18

ROW 19

ROW 20

PBC2 (CONTINUED)

ROW 21

ROW 22

ROW 23

ROW 24

ROW 25

ROW 26

ROW 27

ROW 28

ROW 29

ROW 30

ROW 31

ROW 32

ROW 33

ROW 34

ROW 35

ROW 36

ROW 37

ROW 38

ROW 39

ROW 40

PBC2 (CONTINUED)

ROW 41

ROW 42

ROW 43

ROW 44

ROW 45

ROW 46

ROW 47

ROW 48

ROW 49

ROW 50

ROW 51

ROW 52

ROW 53

ROW 54

ROW 55

ROW 56

ROW 57

ROW 58

ROW 59

ROW 60

ROW 61

ROW 62

ROW 63

ROW 64

ROW 65

ROW 66

ROW 67

ROW 68

ROW 69

ROW 70

ROW 71

ROW 72

ROW 73

ROW 74

ROW 75

END OF PROGRAM