Generating PROM Programming Files

PROMs are nonvolatile memory devices that were first conceived as instruction and data storage devices for microprocessor systems. Since their introduction, PROMs have benefited from improvements in processing and manufacturing technology. The evolution of PROMs has included a tremendous increase in their density and speed and has added new features such as built-in registers and reprogrammability. Now these devices can be used in a wide variety of applications other than instruction storage. PROMs are commonly found in state machines, decoders, encoders, complex counters, controllers, sequencers, and look-up tables as well as in their traditional role of instruction or microcode storage.

PROMs are simply an array of data coupled with an input address decoder. The address presented to the device drives a simple 1-of-n decoder. The decoder selects one preprogrammed memory location whose data flows to the output pins of the device. PLAs (Programmable Logic Array) and PALs (Programmable Array Logic) are also programmable devices and, along with PROMs, make up the majority of devices that are considered to be programmable logic elements. The difference between the three types of programmable logic elements can be seen by observing the internal structure of the programmable array of each of the devices. PLAs have both a programmable “AND” array and a programmable “OR” array. PALs have a similar AND-OR structure, but the number of inputs to the OR function is fixed, so only the AND array is programmable. Both the PLA and PAL have a fixed number of AND-OR terms dedicated to each output. Therefore, the number of functions controlling each output is significantly reduced. PROMs, on the other hand, can realize every possible combination or function of n input lines for a given output. There are 2^n product terms (where n = number of address lines) per PROM output. This makes PROMs useful in very complex functions that exhaust the sum-of-product resources of a traditional PAL or PLA architecture. Some PROMs have additional features, such as output registers, that enable them to operate synchronously, which is required for state machines. The Cypress CY7C245A is one of these PROMs. Presets, clears, and initialization words are also available for dealing with power-on and reset conditions.

After understanding the basic function of a PROM, the designer must now create the PROM data in the form of a programming file. Creating the PROM data can be intimidating to engineers who are not familiar with the process. Looking back, we can see that PROMs were mainly used for instruction or microcode storage in a microprocessor or bit-slice-based system. Therefore, the PROM data for such systems is generated by the compilers, assemblers, and linkers that are resident on the CPU development station or emulator. Generating the PROM files for such systems is almost trivial because the programming data file is simply a listing of the CPU’s executable instructions generated by the compiler. But creating the programming file for a complex decoder, look-up table, sequencer, or state machine can be pretty complicated and overwhelming. In fact, just figuring out where to start or what tools to use can become very time consuming. In this brief application note we will discuss the structure of PROM data files and show several ways to create them. Examples using simple languages such as C and BASIC, as well as PLD development tools such as ABEL and LOG/iC, will be discussed.

In order to understand how to create programming files, you must first be familiar with the actual structure or format of such a file. Again, a PROM is simply an array of programmable memory locations. The data file that is transmitted to the PROM programmer must therefore contain data for each of the locations to be programmed. There are many standard formats for PROM data files.

Generic PROM programmers, such as those manufactured by Data I/O, Stag, Logical Devices, Digelic, SMS, and Kontron, are generally compatible with the following formats:

- ASCII–HEX (Space)
- Binary
- DEC Binary
- Motorola Exorciser
- Motorola Exormax
- Intel “Intellec” 8/MDS
- Intel MCS86 “Intellec 86"
- Tektronix “HEX”
- Extended Tektronix “HEX”

The following section describes each format in detail. Each format has its own set of required fields, delimiters, and special characters. When writing code in C or BASIC, you must know exactly where to place each field and special character so that a programmer will interpret your data correctly.

ASCII-HEX (Space)

One of the simplest and probably the most universal file formats is HEX or HEX-Space ASCII. This format does not support checksum or address field conventions. Therefore, the data in the file must be in order incrementing from address 0. However, many times the program that reads the file into programmer memory can manipulate the data to start at any address location.

Three hidden instructions are used in this format:

1. ASCII STX Character (ASCII 02) marks the beginning of the file.
2. ASCII ETX Character (ASCII 03) marks the end of the file.
3. ASCII Space (ASCII 20) is between each data byte.

Figure 1 shows a data file for a 64-byte PROM implemented in ASCII-HEX (space) format.

Note that each data byte is separated by a “space” character and that no addressing information is present.

```
(STX)FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
     FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
     FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
     FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF(ETX)
```

Figure 1. ASCII-HEX Format

ASCII Binary

ASCII Binary files, like ASCII-HEX, contain no addressing or checksum information. ASCII Binary allows for very fast file transfers to the programmer due to its simplicity. The data format begins with the ASCII STX character and is terminated by an ETX. Data is grouped into four-byte lines separated by a space. Each line of data begins with a "B" character and ends with an "F" character.

Figure 2 shows a 64-byte PROM file containing all zeros using ASCII Binary format. All data is loaded into the PROM sequentially starting at location 0.

Simple Binary

The simple Binary format consists of just binary data. There are no start or end characters. Although the binary file is simple to produce, it is not a recommended output format for the following examples because binary files cannot be easily read by text editors.

DEC Binary

DEC Binary is a modification of the basic ASCII Binary file format. DEC Binary adds a starting address and a checksum for each line of data.

Motorola Exorcisor

Motorola Exorcisor is one of the most widely used formats. Motorola Exorcisor files are commonly referred to as "S" records because each line starts with an "S" followed by the record type. Each line also contains a byte count, starting address, and a checksum, which are delineated by carriage returns and line feeds.

Figure 3 shows an example of a 64-byte PROM file implementing "S" Records.

```
(STX)    B00000000F B00000000F B00000000F B00000000F
         B00000000F B00000000F B00000000F B00000000F
         B00000000F B00000000F B00000000F B00000000F
         B00000000F B00000000F B00000000F B00000000F(ETX)
```

Figure 2. ASCII Binary Format

Calculating Record Checksum

The Checksum is calculated by first stripping off the start code ("S"), the record type, and the checksum. The remaining bytes are added together, converted to binary, and complemented (one's complement). For example, the optional sign on "S0" line reads:

S0 06 00 01 00 01 F7

Stripping the appropriate characters leaves:

06 00 01 00 01

Adding the bytes yields

08 hex

The compliment of the value

F7..... Record checksum

End of Each Record

It is important to end each record with a carriage return and a line feed, which is used as a delineator.

"S" records are useful because they are so universal. However, this format can only be used for PROMs smaller than 64 Kbytes because the address field is limited to 4 bytes.

Motorola Exormax

Exormax is another "S" record file and is identical to Exorcisor with only one exception. Exormax allows for a 6-digit address field, which makes it useful for PROMs that are much larger than 64 Kbytes.

Exormax Record Number:

S0– Optional sign on record
S1– Data record (2 Byte Address field)
S2– Data Record (3 Byte Address Field)

Figure 4 shows an example of a 64-byte PROM file implementing "Exormax S" records.

Intel "Intellec" 8/MDS

Intellec is similar to S records in that each line contains a starting address, byte count, and checksum. However, each line begins with a colon.

Intellec Record Example:
":", Byte Count, Address, Record Type, Data, Checksum

Byte Count: Total number of data bytes ONLY.

Starting Address: 2-byte field where record will be placed in memory.

Record Type:
00 — Data Record
01 — End Record

Checksum: The sum of all preceding bytes including byte count, Address, and all data bytes. This number is expressed in two's compliment notation.

The end of each record is marked by a carriage return.

Figure 5 shows a 64-byte PROM file using Intellec format.

Since there is only a 2-byte address field, Intellec is generally used for PROMs smaller than 64 Kbytes.

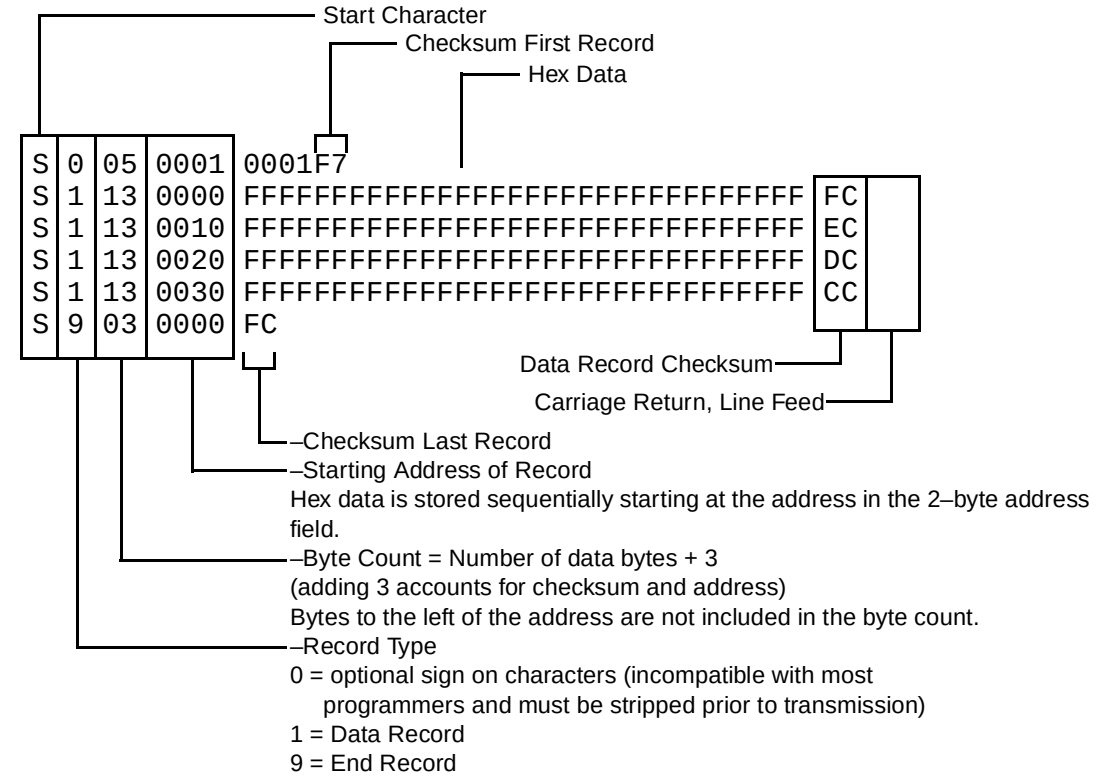

Figure 3. S Record Format

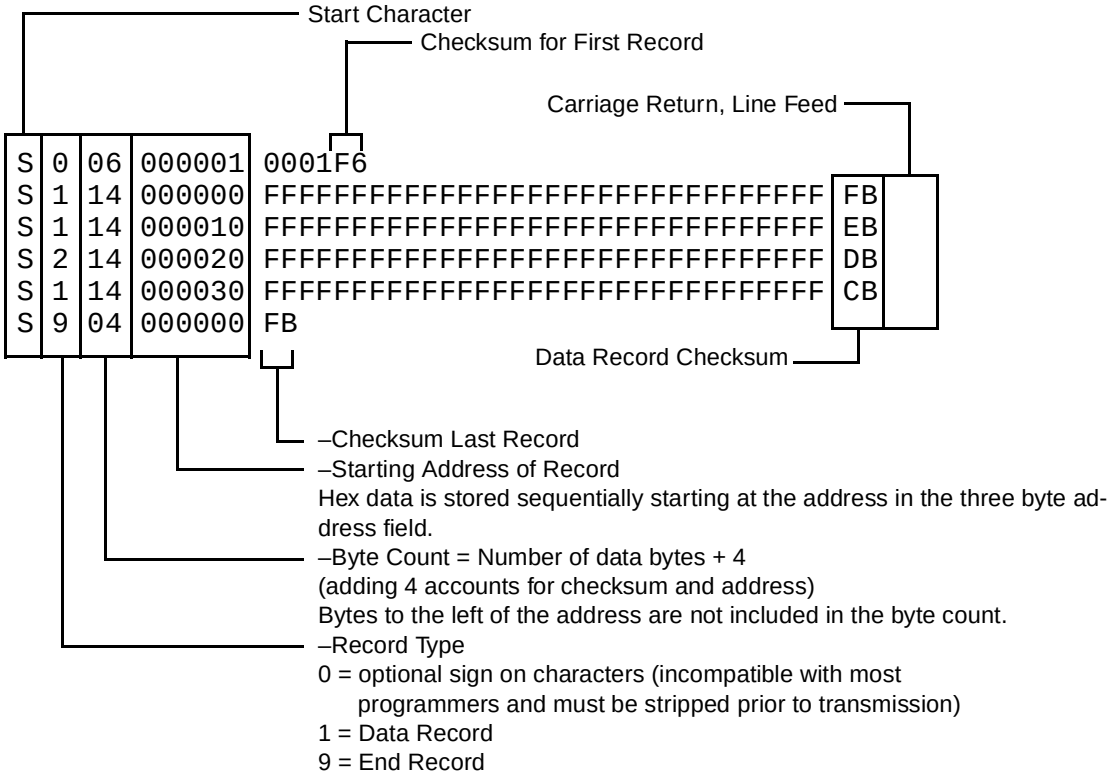

Figure 4. Exormax S Format

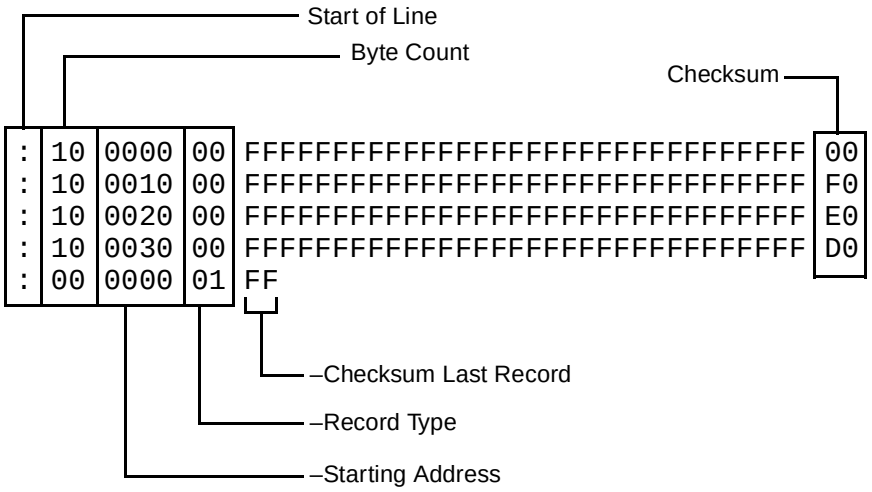

Figure 5. Intellec Format

Intel MCS86 (Intellec 86)

Intellec 86 is an extension of the standard Intellec format. It adds the feature of a Segment Base Address record (SBA). Adding the SBA to the 2-byte address field increases the total addressing capability to 1M locations. The file must begin with an SBA record because physical addresses are calculated using the Starting Address field and the most recent SBA.

Intellec 86 Data or End Record Example:
":", Byte Count, Address, Record Type, Data, Checksum

Intellec 86 SBA Record Example:
":", Byte Count, Address, Record Type "02", SBA, Checksum

Byte Count: Total number of data bytes ONLY.

Segment Base Address (SBA): A 2-byte field that extends the starting address fields of the following records by 4 bits. A new SBA can be inserted as many times as needed. Records sent after a new SBA will use the new SBA to calculate the address.

Starting Address: A 2-byte field where record will be placed in memory. The actual physical address for data placement must be calculated by using the SBA and Starting Address.

Record Type:
00 — Data Record
01 — End Record
(02 — SBA Record)

Checksum: The sum of all preceding bytes including byte count, address, and all data bytes. This number is expressed in two's compliment notation.

The end of each record is marked by a carriage return and line feed.

Figure 6 shows a 64-byte PROM file using Intellec 86 format. This example has an SBA Value of 8000h, which offsets the starting addresses as shown.

To calculate the starting address: (third data record)

Take the value of the most recent SBA	(8000h)
Shift the SBA left	8000
Add value of start address field	+ 0030
Result: Physical Start Address	80030

Again, the segment base address can be updated at any time and will affect the records that follow the change.

Tektronix Hex (TEK HEX)

TEK HEX is another simple file format that is accepted by most programming systems. It uses the "/" character as a start-of-record marker and includes a starting address for each record, byte count, data and two checksums. The first checksum is the summation of the bytes for the address and byte count fields. The second checksum is simply the summation of all of the data bytes. *Figure 7* shows an example of a file stored in TEK HEX format.

```
: 02 0000 02 8000 7C
: 10 0000 00 FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF 00
: 10 0010 00 FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF F0
: 10 0020 00 FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF E0
: 10 0030 00 FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF D0
: 00 0000 01 FF
```

Figure 6. Intellec 86 Format

```
/ 0000 10 01 FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF E0
/ 0010 10 02 FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF E0
/ 0020 10 03 FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF E0
/ 0030 10 04 FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF E0
/ 0000 00 00
```

Figure 7. TEK HEX Format

Start Character: "/" is used to mark the beginning of each line. Most programmers ignore any characters sent before the "/".

Start Address: This value is a 2-byte absolute address. It represents the starting address for the first data byte in the record. All following bytes in the record are stored sequentially.

Byte Count: The number of data bytes in the record are represented by the byte-count field. The end of record is marked by setting the byte count equal to "00".

First Checksum: The simple summation of the nibbles in the address and byte-count fields are represented by the first checksum in each record.

Second Checksum: Calculated by summing all of the nibbles of the data bytes in the record, the second checksum is placed at the end of the record.

Each record is terminated by a carriage return/line feed.

Extended Tektronix Hex (XTEK)

XTEK is a variation of the standard TEK HEX format. It uses the "/" character as a start of record marker and includes a starting address for each record, byte count, data, and two checksums. The first checksum is the summation of the nibbles for the address and byte-count fields. The second checksum is simply the summation of all of the data nibbles. *Figure 8* shows an example of a file stored in XTEK format.

Start Character: "/" is used to mark the beginning of each line. Most programmers ignore any characters sent before the "/".

Start Address: This value is a 2-byte absolute address. It represents the starting address for the first data byte in the record. All following bytes in the record are stored sequentially.

Byte Count: The number of data bytes in the record are represented by the byte-count field. The end of record is marked by setting the byte count equal to "00".

First Checksum: The simple summation of the nibbles in the address and byte count fields are represented by first checksum in each record.

Second Checksum: The summation of the data nibbles is represented by the second checksum, which is placed at the end of the record.

Each record is terminated by a carriage return/line feed.

Using High-Level Languages to Create Files

Depending on the application, there are many different ways to create the actual file. PROMs that contain data derived from mathematical formulas such as look-up tables are easily implemented using a high-level language such as C or BASIC. These languages can easily deal with the complicated data types and mathematical data manipulation that is required for many applications. The data created by the program must be stored in a file so that it can be transferred to a programmer at some later time. The following examples show that opening a new file and writing the data is simple when using high-level languages.

As shown in *Figure 9*, this method written in C follows the simple form:

1. Header documentation – The Header documentation is usually written as comments to help the user understand the purpose and flow of the program. Documentation is not essential, but it is good practice.
2. Constant declaration – Following the Header, the constants can be declared as symbols to help the user update the program to accommodate changes in the design.
3. Variable definition – The variables should be defined to agree with the type of data being used.
4. Body – The body of the program will contain the commands necessary to create the PROM data. This usually takes the form of an outer "For"-type loop to iteratively step through all the possible combinations of address inputs, followed by nested commands that create a data instance to correspond to that combination of address lines.

The C program in *Figure 9* generates ASCII-Space or HEX-Space format output files for downloading to a PROM programmer.

Figure 10 is an example of using BASIC to produce a PROM programming file in the HEX space format.

```
% 1A 6 06 4 1000 FFFFFFFFFFFFFFFF
% 1A 6 0E 4 1008 FFFFFFFFFFFFFFFF
% 1A 6 07 4 1010 FFFFFFFFFFFFFFFF
% 0A 8 16 4 0000
```

Figure 8. XTEK Format

```
/* Example Program 1 */

/* The purpose of this program is to create a data file that could be used as a COSINE look
-up table.  The table has an angular resolution of 256 points per period and an amplitude r
esolution 256 steps or 8 bits.
*/


#include <stdio.h>       /* defines the input-output of PC */
#include <math.h>        /* defines the math package of PC */

int i,j;                 /* integers for loop variables */
float y,x,z;             /* floating pt variables for COSINE */
int data;                /* data  variables for result */
int outfile;

main()
/* main denotes the start of the active part of the program */
{
   FILE *outfile;
/* makes outfile a pointer to the output file */

   outfile=fopen("promfile","w");
/* opens the output file for writing */

   fprintf(outfile,"%c",2);

/* prints control data to output file for download STX */

/* This section consists of 2 nested loops to generate every possible combination of addres
s inputs.  An incrementing variable z is used to generate the angle y in radians.  x = the
cosine of y.  Then x is justified to use the dynamic range of 256 states.  The result is st
ored as an integer in data.  The data is written directly to the output file.  The data is
broken into blocks for easier reading.  */
   z=0;
   for(i=0;i<=15;i++)  {
      for(j=0;j<=15;j++)  {
         y=M_PI*((z)/128.0);
         x=(cos(y));
         x=x*127.99999;
         data = x+128;
         fprintf(outfile,"%02X ",data);
         z=z+1.0;
      }
      fprintf(outfile,"\n");
   }
   fprintf(outfile,"%c",3);
/* prints control char ETX to output file */

   fclose(outfile);
/* closes output file */

}
```

Figure 9. C Program to Generate ASCII-Space or HEX-Space Format Files

```
10 'Example program 2
20 '
30 'The purpose of this program is to create a data file
40 'that could be used as a COSINE look-up table.  The table
50 'has an angular resolution of 256 points per period and
60 'an amplitude resolution of 256 steps or 8 bits.
70 '
80 PI = 3.14159
90 OPEN "O",#1,"PROMFILE.HEX"    'open the file for output
100 '
110 'This section consists of 2 nested loops to generate every
120 'possible combination of address inputs.  An incrementing
130 'variable z is used to generate the angle y in radians.
140 'X = cosine of y.  Then X is justified to use the dynamic
150 'range of 256 states.  The result is stored as an integer
160 'in RANGE.  The data is written directly to the output file
170 'in the HEX SPACE format.
180 '
190 PRINT#1,CHR$(2)     'start the file with the STX char
200 Z = 0               'initialize the loop
210 FOR I = 0 TO 15
220     FOR J = 0 TO 15
230         Y = PI*((Z)/128)
240         X = COS(Y)
250         RANGE = INT(X*127.99999# + 128)
260         IF RANGE > 15 THEN 290
270         PRINT#1,"0";HEX$(RANGE);" ";
280         GOTO 300
290         PRINT#1,HEX$(RANGE);" ";
300         Z = Z + 1
310     NEXT J
320     PRINT#1,""
330 NEXT I
340 PRINT#1,CHR$(3);    'end the file with the ETX char
350 CLOSE
360 END
```

Figure 10. BASIC Program to Generate HEX-Space Format

PLD Development Packages

In general, most of the standard PLD development packages support PROMs. ABEL, CUPL, and LOG/iC are three of the most popular third-party packages. They support most of the industry standard PAL, PLA and PROM devices. These PLD development tools are well suited to creating PROM files that can be described by Boolean equations, truth tables, or state-machine syntax.

ABEL

ABEL, produced by Data I/O Corporation, is one of the most popular PLD development software packages on the market. The fact that ABEL supports PROMs is one of the industry's best-kept secrets. Since a PROM can be thought of as a PLD with a large number of product terms per output, it is relatively easy for a PLD compiler to generate code for a PROM. In fact, the source file (filename.abl) for a PROM and a PLD are almost identical. The only difference is in the device declaration. In the logic diagram package for ABEL, there are pin descriptions for 4-, 8-, and 16-bit PROMS.

Figure 11 shows how to use truth tables and equations to generate a PROM file that is a comparator with some additional built-in logic.

All methods of generating PLD files in ABEL are also available for generating PROM files.

```
module COMP_OR
title '4 bit comparator '
"

     PROMB device     'RA8P8';
"                        8 address lines and 8 data lines

"INPUTS          PIN #   PROM ADDRESS/DATA BIT

     A0         PIN  1;  " A0
     A1         PIN  2;  " A1
     A2         PIN  3;  " A2
     A3         PIN  4;  " A3
     B0         PIN  5;  " A4
     B1         PIN 17;  " A5
     B2         PIN 18;  " A6
     B3         PIN 19;  " A7

"OUTPUTS         PIN #

     AGB        PIN 14;  " D8    A IS GREATER THAN B
     ALB        PIN 13;  " D7    A IS LESS THAN B
     EQUAL      PIN 12;  " D6    A IS EQUAL TO B

     ALL_HIGH   PIN 11;  " D5    ALL BITS ARE HIGH
     OR_BITS_3  PIN  9;  " D4    Misc.  logical functions
     OR_BITS_2  PIN  8;  " D3
     OR_BITS_1  PIN  7;  " D2
     OR_BITS_0  PIN  6;  " D1

     X = .X.;

Declarations
     A_NIB  = [A3,A2,A1,A0];
     B_NIB  = [B3,B2,B1,B0];

Equations
  ALL_HIGH  = (A_NIB==15) & (B_NIB==15);
  OR_BITS_3 = A3 # B3;
  OR_BITS_2 = A2 # B2;
  OR_BITS_1 = A1 # B1;
  OR_BITS_0 = A0 # B0;
```

Figure 11. Using Truth Tables and Equations in ABEL to Generate a Comparator PROM File

```
truth_table
     ([A3,A2,A1,A0, B3,B2,B1,B0]->[AGB,ALB,EQUAL])
      [ 0, 0, 0, 0,  0, 0, 0, 0]->[ 0 , 0 , 1];“A = B CONDITIONS
      [ 0, 0, 0, 1,  0, 0, 0, 1]->[ 0 , 0 , 1];
      [ 0, 0, 1, 0,  0, 0, 1, 0]->[ 0 , 0 , 1];
      [ 0, 0, 1, 1,  0, 0, 1, 1]->[ 0 , 0 , 1];
      [ 0, 1, 0, 0,  0, 1, 0, 0]->[ 0 , 0 , 1];
      [ 0, 1, 0, 1,  0, 1, 0, 1]->[ 0 , 0 , 1];
      [ 0, 1, 1, 0,  0, 1, 1, 0]->[ 0 , 0 , 1];
      [ 0, 1, 1, 1,  0, 1, 1, 1]->[ 0 , 0 , 1];
      [ 1, 0, 0, 0,  1, 0, 0, 0]->[ 0 , 0 , 1];
      [ 1, 0, 0, 1,  1, 0, 0, 1]->[ 0 , 0 , 1];
      [ 1, 0, 1, 0,  1, 0, 1, 0]->[ 0 , 0 , 1];
      [ 1, 0, 1, 1,  1, 0, 1, 1]->[ 0 , 0 , 1];
      [ 1, 1, 0, 0,  1, 1, 0, 0]->[ 0 , 0 , 1];
      [ 1, 1, 0, 1,  1, 1, 0, 1]->[ 0 , 0 , 1];
      [ 1, 1, 1, 0,  1, 1, 1, 0]->[ 0 , 0 , 1];
      [ 1, 1, 1, 1,  1, 1, 1, 1]->[ 0 , 0 , 1];

      [ 0, 0, 0, 0,  X, X, X, 1]->[ 0 , 1 , 0];“A < B CONDS.
      [ 0, 0, 0, X,  X, X, 1, X]->[ 0 , 1 , 0];
      [ 0, 0, X, X,  X, 1, X, X]->[ 0 , 1 , 0];
      [ 0, X, X, X,  1, X, X, X]->[ 0 , 1 , 0];
      [ 1, 0, X, X,  1, 1, X, X]->[ 0 , 1 , 0];
      [ 1, 0, 0, X,  1, 0, 1, X]->[ 0 , 1 , 0];
      [ 1, 0, 0, 0,  1, 0, 0, 1]->[ 0 , 1 , 0];
      [ 0, 1, 0, X,  0, 1, 1, X]->[ 0 , 1 , 0];
      [ 0, 1, 0, 0,  0, 1, X, 1]->[ 0 , 1 , 0];
      [ 0, 0, 1, 0,  0, 0, 1, 1]->[ 0 , 1 , 0];

      [ 1, X, X, X,  0, X, X, X]->[ 1 , 0 , 0];“A > B CONDS.
      [ 0, 1, X, X,  0, 0, X, X]->[ 1 , 0 , 0];
      [ 0, 0, 1, X,  0, 0, 0, X]->[ 1 , 0 , 0];
      [ 0, 0, 0, 1,  0, 0, 0, 0]->[ 1 , 0 , 0];
      [ 1, 1, X, X,  1, 0, X, X]->[ 1 , 0 , 0];
      [ 1, 0, 1, X,  1, 0, 0, X]->[ 1 , 0 , 0];
      [ 1, 0, 0, 1,  1, 0, 0, 0]->[ 1 , 0 , 0];
      [ 0, 1, 1, X,  0, 1, 0, X]->[ 1 , 0 , 0];
      [ 0, 1, 0, 1,  0, 1, 0, 0]->[ 1 , 0 , 0];
      [ 0, 0, 1, 1,  0, 0, 1, 0]->[ 1 , 0 , 0];

 end  COMP_OR
```

Figure 11. Using Truth Tables and Equations in ABEL to Generate a Comparator PROM File (continued)

LOG/iC

LOG/iC by ISDATA probably has the best support of PROM devices due to its ability to create a PROM file of any size. All the programmer has to do is to tell the compiler how many inputs and how many outputs the PROM should have. The above ABEL file is reproduced in *Figure 12* using LOG/iC.

Although not illustrated in the last two examples, both ABEL and LOG/IC are capable of using state machine input formats.

```
*IDENTIFICATION
 This example uses an 8 bit prom as a 4 bit comparator and does some
 additional misc.  logic

*X-NAMES                  !Define the input pins.
 B[3..0], A[3..0];        !Pins are defined MSB first,
                          !Therefor, B3 will be connected
                          !to address bit 7 and A0 will
                          !be connected to address bit 0.
*Y-NAMES                  !Define the output pins
 AGB,ALB,EQUAL,ALL_HIGH,
 OR_BITS[3..0];

*BOOLEAN EQUATIONS
 ALL_HIGH = B3&B2&B1&B0&A3&A2&A1&A0;
 OR_BITS3 = A3 # B3;
 OR_BITS2 = A2 # B2;
 OR_BITS1 = A1 # B1;
 OR_BITS0 = A0 # B0;

*FUNCTION-TABLE
$ ((A3,A2,A1,A0, B3,B2,B1,B0)):((AGB,ALB,EQUAL));
     0  0  0  0   0  0  0  0  :   0   0    1; A=B CONDITIONS
     0  0  0  1   0  0  0  1  :   0   0    1;
     0  0  1  0   0  0  1  0  :   0   0    1;
     0  0  1  1   0  0  1  1  :   0   0    1;
     0  1  0  0   0  1  0  0  :   0   0    1;
     0  1  0  1   0  1  0  1  :   0   0    1;
     0  1  1  0   0  1  1  0  :   0   0    1;
     0  1  1  1   0  1  1  1  :   0   0    1;
     1  0  0  0   1  0  0  0  :   0   0    1;
     1  0  0  1   1  0  0  1  :   0   0    1;
     1  0  1  0   1  0  1  0  :   0   0    1;
     1  0  1  1   1  0  1  1  :   0   0    1;
     1  1  0  0   1  1  0  0  :   0   0    1;
     1  1  0  1   1  1  0  1  :   0   0    1;
     1  1  1  0   1  1  1  0  :   0   0    1;
     1  1  1  1   1  1  1  1  :   0   0    1;
```

Figure 12. Using LOG/iC to Generate a Comparator PROM File

```
     0  0  0  0   –  –  –  1  :   0   1    0; A < B CONDS.
     0  0  0  –   –  –  1  –  :   0   1    0;
     0  0  –  –   –  1  –  –  :   0   1    0;
     0  –  –  –   1  –  –  –  :   0   1    0;
     1  0  –  –   1  1  –  –  :   0   1    0;
     1  0  0  –   1  0  1  –  :   0   1    0;
     1  0  0  0   1  0  0  1  :   0   1    0;
     0  1  0  –   0  1  1  –  :   0   1    0;
     0  1  0  0   0  1  0  1  :   0   1    0;
     0  0  1  0   0  0  1  1  :   0   1    0;

     1  –  –  –   0  –  –  –  :   1   0    0; A > B CONDS.
     0  1  –  –   0  0  –  –  :   1   0    0;
     0  0  1  –   0  0  0  –  :   1   0    0;
     0  0  0  1   0  0  0  0  :   1   0    0;
     1  1  –  –   1  0  –  –  :   1   0    0;
     1  0  1  –   1  0  0  –  :   1   0    0;
     1  0  0  1   1  0  0  0  :   1   0    0;
     0  1  1  –   0  1  0  –  :   1   0    0;
     0  1  0  1   0  1  0  0  :   1   0    0;
     0  0  1  1   0  0  1  0  :   1   0    0;

*ROM
 TYPE = 8_IN_8_OUT;
 INPUTS = 8;
 OUTPUTS = 8;

*RUN
 PROG = INTEL;          Produce an INTEL-HEX output format

*END
```

Figure 12. Using LOG/iC to Generate a Comparator PROM File (continued)

Conclusion

PROM files can be easily generated in a variety of ways. If a complex function is desired, a high-level language approach is probably the best method. However, if a logical function is the desired result, PLD development tools will more than suffice.