tP16

User's Guide

May 3, 2010

Gregg A Tavares & Dan Chang

♻ Printed on recycled paper (contains 50% waste paper including 10% post consumer)

tP16

tP16 is a *tUMEPack* designed for 16-bit console products. The authors of *tUME* do not endorse the methods described in *tP16* as the "one true religion"; in fact, we disagree with many of the design decisions implicit in this *tUMEPack*. However, the product exists, and it works; so study it, and learn from it. Use it if you must, but we recommend that you adapt it to your needs.

Here is a description of the rooms and tilesets *tP16* expects, and what files are generated. *tP16* is designed to work with 8x8 tiles or 16x16 tiles. Here are the tileset types that *tP16* recognizes:

Tileset Type	*User Type*	*User Number*
Image Tiles	0	NA
Contour Tiles	1	NA
Special Tiles	2	NA
Object Tiles	3	NA
4 Color Tiles	4	NA
256 Color Tiles	5	NA
Mode7 Tiles	6	NA

The second column is the tileset number that *tP16* processes, and the first column describes what *tP16* considers that tileset user type to mean. *tP16* ignores the tileset user number.

Image tiles will be converted into 16 color SNES or Genesis characters, 4 color tiles will be converted into 4 color SNES characters, 256 color tiles will be converted into 256 color SNES characters. When *tP16* creates the .CHR font file it puts 4 color characters first, then 16 color characters and finally 256 color characeters.

It is best if all tilesets start with a blank tile in the top left corner in DPaint. This is because any NULL TILEs in your maps are converted to tile 0, colorset 0.

tP16 writes out only one .CHR file; if you create rooms using several different character types, note that all characters used in the tiles in a table room are added to the end of the character font consecutively such that you can use DMA to transfer all of them at once.

Contour tiles may be used in level rooms in the second layer. They generate .CON files. Special tiles may be used in level rooms in the third layer; they generate entries in the .FLR file. Object tiles may be used in level rooms in the fourth layer; they also generate entries in the .FLR file.

Mode 7 tiles are used in Mode 7 rooms.

Here are the room types that *tP16* recognizes:

Room Type	User Type	User Number
Level Room	0	NA
Table Room	1	NA
Picture Room	2	NA
Parallax Room	3	NA
Mode7 Room	4	NA
Flat Conversion Room	5	NA
Layered Conversion Room	6	NA
Animation Room	8	NA

The second column is the room user type that *tP16* processes, and the first column describes what *tP16* considers that room user type to mean. *tP16* ignores the room user number.

Level Rooms

Level rooms generate a .MAP and a .PAL file. Level rooms are treated as 16x16 tiles. The first layer is treated as follows. The program builds a tileset table (a .BLK file) which contains the index numbers of the four 8x8 pixel characters that make up the 16x16 pixel tile along with the flip, colorset and priority information for each character. The program also builds a map (.MAP) file. Each tile on the map is an index into the .BLK file.

The second layer of a level room contains the contour information. Each entry in the .MAP file also indexes a table (a .FLR file) that indexes the .CON file to show which contour tile is used at this map location. The third layer contains special tiles, which is stored in the .FLR file as well. The fourth layer contains object tiles, which is also stored in the .FLR file.

Second Playfields:

A second set of four layers may be added to a level room in which case *tP16* will generate a .MAP file as above for the first 4 layers and an .MP2 file which is in the same format as the .MAP file except for the second set of 4 layers. The layers would be as follows.

layer 1 Image Layer
layer 2 Contour Layer
layer 3 Special Layer
layer 4 Object Layer
layer 5 2nd Image Layer
layer 6 2nd Contour Layer
layer 7 2nd Special Layer
layer 8 2nd Object Layer.

Table Rooms

Table rooms generate a .TBL file. Any characters used by any tiles in the table rooms will have their own entry in the character set, and they won't be checked for duplication when they are added to the character set. The font position of each character is then written out to a .TBL file. The purpose of table rooms is to allow you can find where certain characters are in the font so you may re-define them on the fly to do character set animations.

Another use for table rooms is to force inclusion of contour tiles. This is of interest only if you have flipped contours (-e1) on, as with this option only the contour tiles that are "used" are included in the .CON file. (If flipped contours is not on, all contours get included always.) Including a contour tile in a table room meets the definition of "use". This is useful if you want to have one set contour tiles used for all levels in your game.

Picture Rooms

Picture rooms generate a .MAP file and a .PAL file. Picture rooms use 8x8 characters (the .CHR file), and the map (.MAP) file contains words that can be stored in screen memory directly (it contains the flip, colorset and priority information). One .MAP file is generated for each layer in the room.

Parallax Rooms

Parallax generate a .PAR file and a .PAL file. Parallax rooms use 8x8 characters (the .CHR file), and the map (.PAR) file contains words that can be stored in screen memory directly (it contains the flip, colorset and priority information). Parallax rooms are affected by the command-line -j<WxH> switch. If the (-j) option is specifed than each room is broken up into several screen dumps, each WxH characters in size. One .PAR file is generated for each layer in the room.

Example:

If the original Parallax room was 32x16 16x16 pixel tiles then that would equal 64x32 8x8 characters. Without the (-j) option the .PAR file would be arranged like this:

0	1	...	62	63
64	65	...	126	127
...	...	...	...	...
1920	1921	....	1982	1981
1984	1985	....	2046	2047

With the option (-j32x16) the .PAR file would be arranged as follows:

0	1	...	30	31	512	513	...	542	543
32	33	...	62	63	544	545	...	574	575
...	...	...	...	...	...	...	...	...	...
448	449	...	478	479	960	961	...	990	991
480	481	...	510	511	992	993	...	1022	1023
1024	1025	...	1054	1055	1536	1537	...	1566	1567
1056	1057	...	1086	1087	1568	1569	...	1598	1599
...	...	...	...	...	...	...	...	...	...
1472	1473	...	1502	1503	1984	1985	...	2014	2015
1504	1505	...	1534	1535	2016	2017	...	2046	2047

Mode 7 Rooms (SNES)

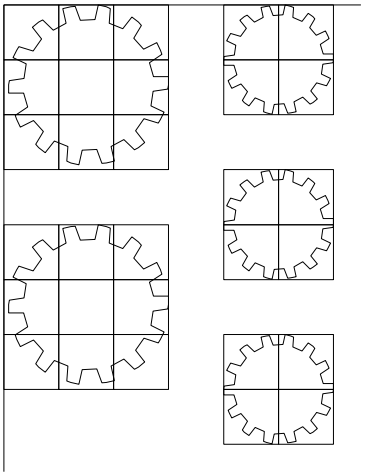
Mode 7 rooms generates a .M7R room file and a .PAL palette file. You should use tiles of type Mode 7 in this room. Since you can not mix Mode 7 with other modes it is best if you do not mix mode 7 tiles with other tiles. The tiles may be 8x8 or 16x16 or 8x8 2x2 composite. Note: All mode 7 limits are enforced. You will be warned if tiles have colorsets or are flipped or if they have priority because all of this is not supported in Mode 7 on the SNES. Mode 7 will only allow 256 characters!

Mode 7 Rooms use 1 byte per tile and are basically retangular screen dump. This means that to use a Mode 7 map you would need to know the width and height of your map. Example: your map is 50 characters wide and 65 characters tall. You would need to copy the first 50 bytes in the mode 7 map file into Video RAM skipping every other byte in Video RAM. Then you would skip 78 WORDS to get to the next mode 7 line in Video RAM and then copy the next 50 bytes and on and on until you have copied all 65 lines.

If you just want a 32K Mode 7 screen dump to can stuff into Video memory and display use the (+7) option below.

Animation Rooms

Animation rooms are used to generate background character animation data. You may place tiles in columns separated by NULL tiles in an animation room. Each NULL tile separates a 'frame' of animation in the column. For example say you had a two spinning gears that you wanted to animate. One gear is 2x2 tiles large and 3 frames and the other is 3x3 and 2 frames. You would place them in an animation room like this:



tP16 will then scan the room and generate animation data for the gears. *tP16* will write out a binary frame data file for the frames and an animation macro file describing how do download the frame data to the character set for animation. Animations that are the same number of frames are merged into one animation. One list of macros is written for each animation unless the (+i) option is given in which case the various animations are interleaved into one macro table. Animations may contain any type of tile (ie. 4 color, 16 color, 256 color, Mode 7) and *tP16* will deal will it. Note: Do not mix Mode 7 tile with other types.

Tiny Mode vs Non-tiny Mode

There are two major modes of operation in tP16, tiny mode and non-tiny mode. The mode affects the output of the .MAP, and the .BLK files. All other files remain the same.

In non-tiny mode, the word in the .MAP table directly indexes the .BLK table. Divide the word in the map by two to index the .FLR table. The .BLK file contains the four characters that should be displayed. All the colorset, flip, and priority information is already encoded into each word in the .BLK file, you can stuff them into screen memory directly.

In tiny mode, the lowest eleven bits of each word (ten bits for SNES) in the .MAP table is multiplied by eight to index the .BLK table, and multiplied by four to index the .FLR table. The upper five bits of each word contains the

colorset, flip, and priority information. These need to be merged with the values in the .BLK file before they can be stored in screen memory.

The advantage of non-tiny mode are simplicity (the display code is simpler), and each tile can be made of four tiles that each have a different colorset. The disadvantage of non-tiny mode is a larger .BLK table size; if two locations in the map have the same image, contour, special, and object layer, but differ only in that one instance of the image tile is flipped, two entries will be generated in the .BLK file. Each entry takes 12 bytes. On one particular project, on just one level there were over 3100 entries, which multiplied by 12 bytes is 37200 bytes, just for the .BLK and .FLR files of one section of the game.

The advantages of tiny mode is smaller .BLK table size; if two locations in the map have the same image, contour, special, and object layer, but differ only in that one instance of the image tile is flipped, only one entry will be generated in the .BLK file, and the flipping is encoded in the .MAP file. The disadvantange is that each tile can use only one colorset four all four characters.

Currently, it takes 12 bytes to represent a tile (four words in the .BLK file, and four bytes in the .FLR file). In non-tiny mode, there is a limit of 8192 tiles. In tiny mode, there is a limit of 2048 tiles in the Genesis, and 1024 tiles in the SNES. However, recall that the definition of a 'tile' differs between tiny and non-tiny mode. If a tile is the same as another tile, except that the image is flipped or has a different colorset or priority in one of them, it is counted as two tiles in non-tiny mode, but as only one tile in tiny mode.

Note that *tP16* does not warn you if you generate too many tiles, you must watch the number yourself.

Decoding a Tiny Mode map

To decode a word in the .MAP file in the tiny mode, you must use the [low 11 bits] * 8 to index the .BLK table. (10 bits on SNES) With the four words you get from the .BLK table, you must OR in the colorset and priority. If the flip bits are set in the .MAP word, you must re-order the four words as appropriate. Finally, you must exclusive-or in the flip bits (as the .BLK words may have flip information in it as well), and store them on screen. In C, it looks something like this:

```
#if SEGA
      #define NDX_BITS        0x07FF
      #define FLIP_BITS       0x1800
      #define PRI_COLOR_BITS  0xE000
      #define XFLIP_BIT       0x0800
      #define YFLIP_BIT       0x1000
#elif SNES
      #define NDX_BITS        0x03FF
      #define FLIP_BITS       0xC000
      #define PRI_COLOR_BITS  0x3C00
      #define XFLIP_BIT       0x4000
      #define YFLIP_BIT       0x8000
#endif

UWORD tile;
UWORD ndx;
UWORD tlchar;
UWORD trchar;
UWORD blchar;
UWORD brchar;
UWORD temp;
```

```
UWORD flipbits;
UWORD pcbits;

tile = map[y * mapwidth + x];       // Get tile from map
ndx  = tile & NDX_BITS;             // get tile ndx (low 11 bits)
tlchar = blocktbl[ndx].tlchar;      // get top left char
trchar = blocktbl[ndx].trchar;      // get top right char
blchar = blocktbl[ndx].blchar;      // get bottom left char
brchar = blocktbl[ndx].brchar;      // get bottom right char

flipbits = tile & FLIP_BITS;        // extract flip bits.
pcbits = tile & PRI_COLOR_BITS;     // keep only color & pri

if (flipbits & XFLIP_BIT) {

      // if tile is xflipped then swap
      // left and right characters.
      temp = tlchar;    tlchar = trchar;  trchar = tlchar;
      temp = blchar;    blchar = brchar;  brchar = blchar;
}
if (flipbits & YFLIP_BIT) {

      // if tile is yflipped then swap
      // top and bottom characters.
      temp = tlchar;    tlchar = blchar;  blchar = temp;
      temp = trchar;    trchar = brchar;  brchar = temp;
}

tlchar = (tlchar ^ flipbits) | pcbits; // put
trchar = (trchar ^ flipbits) | pcbits; // it
blchar = (blchar ^ flipbits) | pcbits; // all
brchar = (brchar ^ flipbits) | pcbits; // together

// now the characters are ready to store
// into screen memory.
```

Object Tiles and Special Tiles

Currently, object tiles and special tiles are encoded into the .FLR file. An alternative approach is to generate a list of objects for each room. Typically, this can be encoded in three to five bytes, the x-coordinate, the y-coordinate, and the object number. *tP16* does not currently support generating a list of objects, however, it is an option we can easily add (call us!).

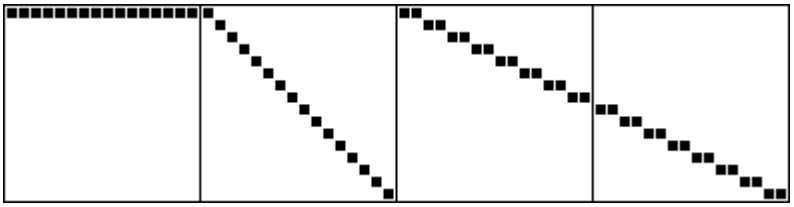
Contour Tiles

Contour tiles are 16x16 pixel tiles. Every tile in the contour tileset is always written to the output .CON file (unless you use the +e switch, in which case it only writes the contour tiles that are actually used). By default, tP16 does NOT process flipped contour tiles; you can configure *tUME* so it does not generated flipped contour tiles. Alternatively, if you decide to use flipped contour tiles, use the +e switch with *tP16*, and it will process flipped contours. It does not check to see if a flipped contour is the same as another non-flipped contour, however (we can easily add this, call us!).

For each pixel column in a tile, scan down the column. If a pixel is found then add the position (+1) of that pixel to the table and continue to the next column. If no pixel is found in the column then add a 0 (zero) to the table. Write out the table to file <*tileset name*>.CON.

Example:

Let say you have CONTOUR.LBM that looks like this:



tP16 will create CONTOUR.CON that looks like this:

```
dc.b  1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1                  ; 0
dc.b  1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16           ; 1
dc.b  1,1,2,2,3,3,4,4,5,5,6,6,7,7,8,8                  ; 2
dc.b  9,9,10,10,11,11,12,12,13,13,14,14,15,15,16,16    ; 3
```

There is another way to deal with contours. Instead of having a separate contour layer, you can create a **conversion room** that associates a contour tile with every image tile. This advantage of doing contours this way is you save memory required to represent your contour. The disadvantage (a minor one) is that any given image will always use the same contour. *tP16* does not currently support contour conversion rooms; however, it is something that can be easily added if you would prefer (call us!).

tP16 Command-Line Switches

Single letter switches with no arguments can be turned on with a '+', and turned off with a '-'. In the following list, the '-' or '+' in front of the switch specifies the default setting of that switch, thus debugging (-d) is off by default, while contour (+c) is on by default.

-a FirstColor. All colorsets in the map (not the palette) are moved up by the amount specified here such that if a tile would normally be displayed in colorset 3, and you use the option -a2, then that tile will be displayed in colorset 5 (3+2). This option is global to all tiles and no error checking is done so that if you do something like have a tile using colorset 5, and you do a -a6, which would make that tile become colorset 11 (5+6), you lose.

-b Binary. Using +b this option will cause ALL files (except table .TBL files) to be written in binary so that they may be easily compressed.

+c Output contour information.

-d Disable debugging messages. +d to enable debugging messages.

-e<ct> Set the flipped contour mode type.

0 = No Flipped contours allowed.

1 = Generate Flipped Contours for Contours that are flipped Instead of converting each contour tile from your DPaint picture directly into a contour table, only those contours that are used are added to the contour table and if a contour is flipped it will be added 'flipped'. Contour #0 is assumed to be blank so make sure it is in your DPaint picture. If you need to know the position of other contours, add them to your TABLE room. Contour tiles in a TABLE room are added to the beginning of the contour table in the order found in your TABLE room, (left to right, top to bottom).

2 = Store flip bit flags in the contour field of the .FLR file to indicate a flipped tile.

bit 0 = X flip (ie. $0001)
bit 1 = Y flip (ie. $0002)

-f<n> Skip the first <n> characters when numbering characters. This option adds a constant to all characters in the block table (.BLK). This means that if a particular tile would normally use characters 10,11,25,27 for its top-left, top-right, bottom-left, bottom-right characters, with -f20 they would become 30,31,45,47. This is all find and dandy; however, if you have mixed tilesets in your map (i.e. 4 color tiles and 16 color tiles) then the -f option makes no sense because 4 color tiles are numbered differently than 16 color tiles and so adding the same constant to both would not work. There are two solutions:

1. Never use mixed tilesets. With this option if you needed to create a 16 color tile level and a 4 color parallax you would create them in separate maps and *tP16* them separately.

2. Use the -r option (see below).

-g Write Room Width/Height and table sizes into files. +g to write size information into the files.

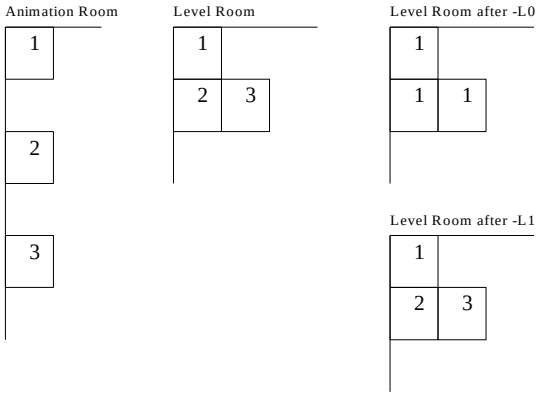
-h SNES 16x16 characters. +h to write SNES 16x16 characters (this is a total hack for a particular project and will not general work as expected)

-j<WxH> Write parallax rooms as several screens, each WxH characters in size.

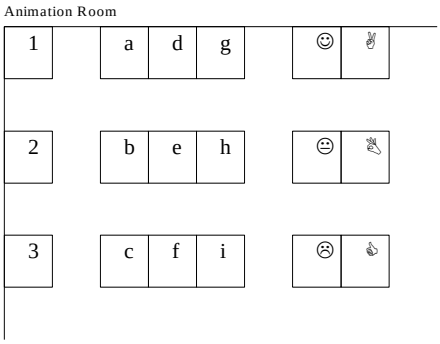
-k Generate only characters used in rooms. Normally all possible characters are generated from the image tilesets whether or not they are actually used in any room. +k will make *tP16* generate only those characters actually used.

-L<animmodes> This arguement takes several sub-parameters.

-TOP : Normally each column of animation converts to the same entry in the block table. In other words if you have a 1 column 4 frame animation in an animation room you may place any of the 4 tiles that make the frames in your LEVEL rooms and they will all be converted to the same animating block. If you specify -L+TOP then only the top tile will be animated. Placing the lower 3 tiles in your LEVEL rooms will generate a standard non-animated block.



+MRG : Normally all animations in one animation room that are the same number of frames in length are merged into one animation. If you specify -L-MRG then, if you separate animations by at least one column of NULL tiles they will generate a separate animation in the .ANI file



Example: Given the animation room above, with -L+MRG there would be one animation in the .ANI file. With -L-MRG there would be 3 animations.

-ZRO : Zero out animation area in character font. When an animation room is found and animation data is created the first frame of animation is also written to the character font since it must be available to initially display the level. With this option specified, the area in the character font where the animation is happening will be zeroed out so that it will compress better and an initialization table is added to the animation macro file specifying what animation data to copy into the character font at runtime to restore the missing characters.

-INT : Interleave animations of different frames. Normally animations of different lengths are written into the macro file as separate lists. -L+INT will write the different animations into one table but it will interleave them. For example if you had a two frame animation of frames ¬ and - and a three frame animation of frames ¶,· and ¸ then with -L+INT the frames would be interleaved like this: ¬, ¶, -, ·, ¬, ¸, -, ¶, ¬, ·, -, ¸.

+CMP: Compress animations. Animations are compressed for example if two columns of characters look and animate exactly the same as another column of characters. -L-CMP turns this compression off.

In tUME, each room may have a comment. Animation use these comments to pass the above parameters so that you may change them from room to room. In otherwords, each room can be

processed with a different set of arguments. You can put any of the -L option in the comment for the room. Do not put the '-L'. Just put the actual options (e.g. '+TOP-MRG+ZRO-INT+CMP') Note: if you use room comments to pass animation arguements it is best to specify all five flags for each room

+m — MergeLayers. Using this option allows you to put contour, special and object tiles on ANY of the layers above layer 1 (ie. layer 2, 3, 4 ....) and *tP16* will still figure everything out.

-o<binlist> — 0 = Palette File, 1 = Contour File, 2 = Font File, 3 = Mode7 Font File, 4 = Table Room File, 5 = Block Table File, 6 = Mode7 Room File, 7 = Level Room File, 8 = Parallax Room File, 9 = Picture Room File, 10 = Floor File, 11 = Animation Macro File. Default = -o0,1,2b,3b,4,5,6b,7b,8b,9b,10, 11. Character files (.CHR) and maps (.MAP & .PAR) are always written out in binary.

+p — Pack Font; find duplicate characters including flipped characters and eliminate from character set.

-r<mem> — Skip first <mem> bytes in font, rounded up to 16 byte boundary. *tP16* figures out first tile index based on <mem> bytes skipped. Thus, if you specify -r2048 then *tP16* would start numbering your characters appropriately, so that if you are using 16 color tiles, the first character would be numbered 64, or if you are using 4 color characters, the first character would be numbered 128. Using this option *tP16* can correctly number all of your characters even if you are using mixed tilesets.

-s — SNES mode. +s to generate SNES instead of Genesis characters and also to make all WORD values be ordered for the 65816 instead of the 68000.

-t — Tiny mode. Using this option, the XY Flip bits, the Priority Bit and the Colorset Bits are stored in the map data instead of the Block table.

+w — Write files. Use (-w) if you just want *tP16* to report errors but not create any files.

+x — Write tiles

+z — Write rooms

-2 — 2Bit colors. +2 causes tP16 to look at colors 0..3, 16..19, 32..35, etc., instead of colors 0..3, 4..7, 8..11, etc. when generating 4 Color Tiles.

-7 — Imbedded font. Using +7 will create a 32K mode7 screen dump that can be copied directly into VRAM for displaying. The .MD7 (font) file will not be generated with this option.

IGNORE — This option allows you to specify which colors are to be ignored in determining the colorset of a particular tile/character. *tP16* used to assume colors 0 and 1 were to be ignored, but now you may specifiy which colors. Example:

```
TP16 MYLEVEL.MAP IGNORE 0 2 5 7
```

This will ignore colors 0, 2, 5 and 7

Making *tP16* Run Faster

tP16 will run 10 times faster if you use EMS memory instead of XMS memory. You can get EMS memory by running DOS 5 EMM386 or QEMM. The reason is EMS is up to 32000 times faster than XMS.

tP16 File Formats

.BLK files (Which Four Characters to Use, Non-tiny Mode)

tP16 will write out a ASCII file that describes which four SNES or Genesis characters make up each tile. Each tile generates a four word entry, which specifies the upper-left, upper-right, lower-left, and lower-right character to use. The flip bits, colorset bits and priority bits are already set in each word. You may stuff these into screen memory directly. Note that the number of tiles is not encoded anywhere in this file.

CHARINDEX4[{number of tiles}] (formatted as follows):

CHARINDEX4 consists of:	
WORD	upper-left_character_index
WORD	upper-right_character_index
WORD	lower_left_character_index
WORD	lower_right_character_index

In Genesis mode, each *x-y*_character_index is formatted as follows:

BITS 00..10	index character table
BIT 11	X flip information
BIT 12	Y flip information
BITS 13..14	colorset information
BIT 15	priority information

In SNES mode, each *x-y*_character_index is formatted as follows:

BITS 00..09	index character table
BITS 10..12	colorset information
BIT 13	priority information
BIT 14	X flip information
BIT 15	Y flip information

.BLK files (Which Four Characters to Use, Tiny Mode)

tP16 will write out a ASCII file that describes which four SNES or Genesis characters make up each tile. Each tile generates a four word entry, which specifies the upper-left, upper-right, lower-left, and lower-right character to use. In each word, only the flip bits are set. You must include the flip, colorset, and priority information from the .MAP before setting screen memory.

Note that each entry in the .BLK file corresponds directly to the same index entry in the .FLR file. Note that the number of tiles is not encoded anywhere in this file.

CHARINDEX4[{number of tiles}] (formatted as follows):

CHARINDEX4 consists of:	
WORD	upper-left_character_index
WORD	upper-right_character_index
WORD	lower_left_character_index
WORD	lower_right_character_index

In Genesis mode, each *x-y*_character_index is formatted as follows:

*x-y*_character_index consists of:	
BITS 00..10	index character table
BIT 11	X flip information
BIT 12	Y flip information
BITS 13..15	set to 0

In SNES mode, each *x-y*_character_index is formatted as follows:

*x-y*_character_index consists of:	
BITS 00..09	index character table
BITS 10..13	set to 0
BIT 14	X flip information
BIT 15	Y flip information

.CHR files (Character data)

tP16 will write out one large BINARY character file for all characters collected from all tilesets of UserTypes 0 (zero), 4 and 5. The name of the file is the same as the name of the map being *tP16*'ed with the extension '.CHR' appended. The format of each character is the native format of the SNES or Genesis; see appropriate documentation about that machine's character format for more details. Note that the number of characters is not encoded anywhere in this file; the first byte of the .CHR file is the first byte of the first character.

When *tP16* creates the .CHR font file it puts 4 color tiles first, then 16 color tiles and finally 256 color tiles. The file would look like this if it contained 4-colors, 16-color, and 256-color tiles:

BYTE[16]	4_color_characters[{all 4 color characters used}]
BYTE[32]	16_color_characters[{all 16 color characters used}]
BYTE[64]	256_color_characters[{all 256 color characters used}]

If the file contains mode 7 characters they need to be copied to every other byte of video memory because that's the way mode 7 works.

.CON files (Contour Tiles)

Please see the description of contour tiles, above. These are usually ASCII files that need to be INCLUDEd by an assembler. Each contour tile generates 16 bytes; each byte is the height of the contour at each of the 16 Y positions. Note that the number of contour tiles is not encoded anywhere in this file.

BYTE[16]	contour_information[{number of contours}]

.FLR files (Contour, Special, and Object Information)

An ASCII file that includes contour, special, and object definitions. For every tile, there is a word and two bytes of information. The word is the index into the contour (.CON) table for this tile. It is already multiplied by 16, so it indexes the .CON table directly. The next byte is set if there is a special tile in the special (third) layer, it will be set to the tile number less one. Similarly, the final byte is set if there is an object tile in the object (fourth) layer, it will be set to the tile number less one.

Note that any time an object tile appears, it will generate a unique entry in the .FLR (and corresponding .BLK) table, even if everything else (image, flip, special, contour, colorset, priority) is exactly the same.

Note that each entry in the .FLR file corresponds directly to the same index entry in the .BLK file. Note that the number of tiles is not encoded anywhere in this file.

CONINDEX_S_O[{number of tiles}] (formatted as follows):

CONINDEX_S_O consists of:	
WORD	index into contour table (.CON)
BYTE	{special tile number}-1
BYTE	{object tile number}-1

.MAP files (Level Room Maps, Non-tiny Mode)

A BINARY .MAP file is generated for each level room. Each is an array <room width> by <room height> of words. Each word indexes the .BLK table directly. Divide each word by two to index the .FLR table.

The file is a two-dimensional array of words:

WORD map_index[{room height}][{room width}]

If the +g option is specified then the file is preceded by the size of the room in blocks.

WORD room_width
WORD room_height

.MAP files (Level Room Maps, Tiny Mode)

A BINARY .MAP file is generated for each level room. Each is an array <room width> by <room height> of words. Each word is an encoded field. The <lowest 11 bits> * 8 indexes the .BLK table. The <lowest 11 bits> * 4 indexes the .FLR table. The upper five bits contain x-flip, y-flip, colorset, and priority information.

Note: on the SNES the <lowest 10 bits> * 8 indexes the .BLK table and the upper 6 bits contain x-flip, y-flip colorset and priority information.

WORD encoded_map_index[{room height}][{room width}]

If the +g option is specified then the file is preceded by the size of the room in blocks.

WORD room_width
WORD room_height

In Genesis mode, each encode_map_index word is formatted as follows:

encoded_map_index consists of:	
BITS 00..10	index into .BLK and .FLR tables
BIT 11	X flip information
BIT 12	Y flip information
BITS 13..15	priority and colorset information

In SNES mode, each encode_map_index word is formatted as follows:

encoded_map_index consists of:	
BITS 00..09	index into .BLK and .FLR tables
BITS 10..13	not defined
BIT 14	X flip information
BIT 15	Y flip information

.MAP files (Picture Room Maps)

A BINARY .MAP file is generated for each picture room. Each is an array <room width> by <room height> of words. Each word indexes the .CHR table directly, and includes encoded flip, colorset, and priority information. This room is basically a **screen dump**.

The file is a two-dimensional array of words:

WORD character_index[{room height}][{room width}]

If the +g option is specified then the file is preceded by the size of the room in characters.

WORD width_in_chars
WORD height_in_chars

In Genesis mode, each character_index is formatted as follows:

BITS 00..10	index character table
BIT 11	X flip information
BIT 12	Y flip information
BITS 13..15	priority and colorset information

In SNES mode, each character_index is formatted as follows:

BITS 00..09	index character table
BITS 10..12	colorset information
BIT 13	priority information
BIT 14	X flip information
BIT 15	Y flip information

.MD7 files (Mode 7 Character data)

Note: This file is no longer written. All mode 7 character data is written to the .CHR file unless the (+7) option is specifed. Do NOT mix mode 7 tiles with other types in the same map.

.M7R files (Mode 7 Room Maps)

A BINARY .MAP file is generated for each picture room. Each is an array <room width> by <room height> of bytes. Each byte indexes the .MD7 table directly. This room is basically a **screen dump**.

The file is a two-dimensional array of bytes:

BYTE character_index[{room height}][{room width}]

If the +g option is specified then the file is preceded by the size of the room in characters (unless the +7 option is specifed).

WORD width_in_chars
WORD height_in_chars

.PAL files (Palettes)

An ASCII .PAL file is generated for each room. In Sega Genesis mode, each entry is a PALETTE value that contains three nibbles, R, G, and B color information. You should define a macro which converts PALETTE into something that works on the Genesis:

PALETTE[{number of palette entries}] (formatted as follows):

PALETTE consists of:	
BITS 00.03	B color information
BITS 04..07	G color information
BITS 08..11	R color information
BITS 12..15	always 0

In SNES mode, each entry is a word value that may be directly loaded into the SNES palette registers:

WORD[{number of palette entries}] (formatted as follows):

WORD consists of:	
BITS 00.04	R color information
BITS 05..09	G color information
BITS 10..14	B color information
BIT 15	always 0

.PAR files (Parallax Room Maps without -j)

.PAR files without using the -j option are exactly the same as .MAP files for Picture Rooms as specified above (except with the +g option, see below).

.PAR files (Parallax Room Maps with -j)

A BINARY .PAR file with several arrays each WxH in size (see -j<WxH> command-line switch), is generated for each parallax room. Each is an array WxH of words. Each word indexes the .CHR table directly, and includes encoded flip, colorset, and priority information. This room is basically a bunch of small **screen dumps**.

The file is a several two-dimensional arrays of words:

WORD character_index[{-jW height}][{-jH width}]

How ever many arrays are required to cover the entire room are written to the file.

If the +g option is specified then the file is preceded by the size of the room in arrays and by the size of one array in characters.

WORD arrays_across_room
WORD arrays_down_room
WORD width_of_one_array
WORD height_of_one_array

In Genesis mode, each character_index is formatted as follows:

BITS 00..10	index character table
BIT 11	X flip information
BIT 12	Y flip information
BITS 13..15	priority and colorset information

In SNES mode, each character_index is formatted as follows:

BITS 00..09	index character table
BITS 10..12	colorset information
BIT 13	priority information
BIT 14	X flip information
BIT 15	Y flip information

.TBL files (Table Room Tagged Characters)

tP16 will write out a ASCII file which contains equates that describe where the four SNES or Genesis characters that make up each tile appear in the character set. Each tile generates a four word entry, which specifies the upper-left, upper-right, lower-left, and lower-right character to use. Instead of actually generating data, currently the program generates an EQUATES file.

CHARINDEX4[{number of tiles}] (formatted as follows):

CHARINDEX4 consists of:	
WORD	upper-left_character_index
WORD	upper-right_character_index
WORD	lower_left_character_index
WORD	lower_right_character_index

.ADT files (Animation Data)

tP16 will write out a binary file which contains all the background character animation data. The data is setup so that one DMA will update an entire frame.

.ANI files (Animation Macros)

tP16 will write out an ASCII file which contains macro lists for animating background characters. The macros look like this:

ani_dma ANIROOMNAME_DATA+$<FOFFSET>,$<COFFSET>,$<SIZE>

where FOFFSET is the offset into the corresponding .ADT file. COFFSET is the offset into the character set where to copy the data in bytes. SIZE is the number of bytes to copy (DMA). ANIROOMNAME will be the name of the animation room.