SPACE 64 COLOR MASKS APPLICATION NOTE

S. de Graaf

Circuits and Systems Group
Faculty of Electrical Engineering
Delft University of Technology
The Netherlands

Report ET-CAS 01-02
May 4, 2001

Last revision: January 4, 2011.

1. INTRODUCTION

This report describes something about the internal structure of the *space* program. What is changed and must be known about the 64 masks implementation.

The files "color.h" and "color.c" are added to directory "space/auxil". And the object modules are added to library "auxil.a".

A color is a bitmap. Each ’1’ bit in the color bitmap represents the presence of a mask. Each mask has an unique color, whereby only one bit is set. The program *tecc* has given each mask a color in the used technology file. Thus mask combinations make new colors, which are assigned by *space* to edges and tiles, and are used as searching key to find circuit elements.

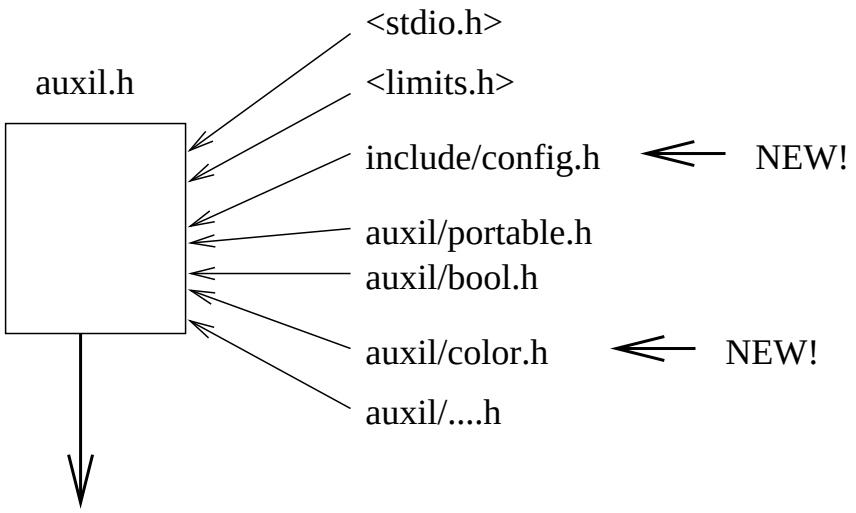
To use 64 masks, the color is implemented as an array of integers. Because an integer represents 32 bits in C, we only need two array elements (NCOL = 2). It is also possible to use other basic types (char, short, long) to get the same result. Note that NCOL must be set to 8 for ’char’ and to 4 for ’short’.

But, this implementation use type ’int’ as the base color type ’mask_base_t’. All color variables in the program sources are of type ’mask_t’.

Now, these variables can not simple be initialized to zero by assigning ’0’ to them. Because these variables represent arrays. Also, can not simple ’0’ be the argument of a function call. For these cases a special color variable **cNull** is defined, and must be used. Otherwise, it is possible to assign a color variable to another color variable. Thus you can copy a color variable to another. And a color variable can possibly be the return value of a function (if needed).

For the new (64 bits) color datatype ’mask_t’ there are made a number of macro’s and functions to use them. They will be explained in this document.

2. SPACE PROGRAM STRUCTURE



The color implementation is added as a part of the library "auxil.a". This was the most simple solution, because allmost all SPACE programs are using this library. The color datatype ’mask_t’ was already in use and was defined in "include/tile.h". Now this datatype is moved to "auxil/color.h". Thus, all program sources which are including "tile.h" must now also include "auxil/color.h" and "include/config.h". The last include is needed for the value of NCOL. Sources, which already include "auxil/auxil.h", do not need to be changed (see the above figure).
It was needed, also to add "include/config.h" into "auxil/auxil.h". Thus, the settings of ‘‘config’’ may take over previous local settings.

See for (a part of) the listings of these files the appendices.

3. COLOR MACRO'S

3.1 Color Special Macro's

```
COLOR_INT (x)         mask_t x
COLORINIT (x)         mask_t x          [ x = 0 ]
COLORCOPY (x, y)      mask_t x, y       [ x = y ]
```

The above macro's are special, because they are seldom used. The COLORINIT and COLORCOPY macro's are not really needed, because 'COLORINIT(x)' is equal to 'x = cNull' or 'COLORCOPY(x,cNull)', and 'COLORCOPY(x,y)' is equal to 'x = y'. Note that 'cNull' is a global color variable, which must have a zero bitmap. This 'cNull' variable must somewere be defined. In *space* this is done in the file "extract/gettech.c".
The COLOR_INT macro is only used in "extract/recog.c" to get the low order 32 bits. These bits are put into an integer, which is used as a key index into the key-table. Note that this key-table has never more than 2^32 entries. The current implementation of macro COLOR_INT, see appendix C, is only correct for 'mask_base_t' type 'int32' (int32 = int).

3.2 Color Init Macro's

```
COLORINITALLBITS (x)  mask_t x          [ x = ~0 ]
COLORINITBITS  (x,y)  mask_t x; int y   [ x = cbits(y) ]
COLORINITINDEX (x,y)  mask_t x; int y   [ x = color(y) ]
```

Init means in this case, set initial value of color 'x'. The macro COLORINITALLBITS sets all bits of 'x' to '1'.
The macro COLORINITBITS sets only 'y' low order bits of 'x'. The value of y must be greater than 0. For y equal to 0 the macro is equal to COLORINIT (see above).
The macro COLORINITINDEX sets only one bit in 'x'. The value of y must be equal or greater than 0 (y >= 0). A value greater than 63 sets no bit is 'x' (makes 'x' zero). For y equal to 0 the low order bit is set.

3.3 Color Change Macro's

```
COLOR_ADD (x,y)       mask_t x, y       [ x |= y ]
COLOR_XOR (x,y)       mask_t x, y       [ x ^= y ]
COLOR_AND (x,y)       mask_t x, y       [ x &= y ]
COLOR_ADDINDEX(x,y)   mask_t x; int y   [ x |= color(y) ]
COLOR_SLEFT  (x)      mask_t x          [ x <<= 1 ]
COLOR_SLEFT1 (x)      mask_t x          [ x <<= 1 , ++x ]
```

The macro COLOR_ADD does an inclusive-or of 'y' with 'x'. This means that the '1' bits in bitmap color 'y' are added to color 'x'.
The macro COLOR_ADDINDEX adds the y-index bit to 'x' (y >= 0). The type of 'y' is in this case an integer.
The macro COLOR_XOR does an exclusive-or of 'y' with 'x'. This means that the '1' bits in color 'y' invert the bits in color 'x'.
The macro COLOR_AND does an and-operation of 'y' with 'x'. This means that the '0' bits in color 'y' are added to color 'x', or that only the '1' bit positions in 'y' don't

change (clear) bits in ’x’.
The last two macro’s above shift the bits in ’x’ one position to the left.
By COLOR_SLEFT the low order bit becomes ’0’ and by COLOR_SLEFT1 the low order bit is set to ’1’.

3.4 Color Test Macro’s

```
COLOR_GT_COLOR(x,y)   mask_t x, y       [ x >  y ]
COLOR_EQ_COLOR(x,y)   mask_t x, y       [ x == y ]
COLOR_PRESENT (x,y)   mask_t x, y       [ (x & y) == y ]
COLOR_ABSENT  (x,y)   mask_t x, y       [ (x & y) == 0 ]
IS_COLOR (x)          mask_t x          [ x != 0 ]
```

Examples:

```
if (IS_COLOR (x)) ...
if (!COLOR_ABSENT (x, y)) ...
```

The COLOR_GT_COLOR macro returns TRUE when color ’x’ has a ’1’ bit on a higher position than color ’y’. The COLOR_EQ_COLOR macro returns TRUE when color ’x’ and ’y’ are completely equal.
The COLOR_PRESENT macro returns TRUE when color ’y’ ’1’ bits are completely present in color ’x’. The color ’y’ ’0’ bit positions do not need to be ’0’ in color ’x’. Note that !COLOR_PRESENT means ’not completely present’.
The COLOR_ABSENT macro returns TRUE when the color ’y’ ’1’ bits are completely absent in color ’x’. Note that !COLOR_ABSENT means ’not completely absent’ (= ’some are present’) and that is not always equal to COLOR_PRESENT.
Thus means !COLOR_PRESENT ’not completely present’ (= ’some are absent’) and that is not always equal to ’all are absent’. Note that !COLOR_ABSENT is used for COLOR_PRESENT, when color ’y’ contains only one ’1’ bit. When color ’y’ represents only one mask.
At last, the macro IS_COLOR is TRUE when one or more bits are ’1’ in ’x’. In that case we can say, that ’x’ represents a color (contains one or more masks).

4. COLOR FUNCTIONS

4.1 Color Internal Functions

```
int isCOLOR_GT_COLOR (mask_t x, mask_t y)
int isCOLOR_EQ_COLOR (mask_t x, mask_t y)
int isCOLOR_PRESENT  (mask_t x, mask_t y)
int isCOLOR_ABSENT   (mask_t x, mask_t y)
int isCOLOR (mask_t x)
```

The above 5 functions are only used by setting NCOL greater than 2. By NCOL less or equal to 2 only the inline macro code is used. See also file "auxil/color.h" at the end of appendix C.

4.2 Color Print Functions

```
char *colorBitStr (mask_t a)
char *colorOctStr (mask_t a)
char *colorHexStr (mask_t a)
char *colorIntStr (mask_t a)
```

The above 4 functions convert the color ’a’ into a string. The result is put into a static string buffer, which address is returned as the function return value. Note that a buffer can hold only one function result.

4.3 Color Index Functions

```
int colorindex (mask_t a)
```

The above function returns the index of the highest bit that is set in color ’a’. The value -1 is returned, when no bits are set in color ’a’. In the current implementation (64 bits), the index return value can be 0 to 63.

4.4 Color Init Functions

```
void initcolorhex  (mask_t *a, char *hex)
void initcolorint  (mask_t *a, char *ins)
void initcolorbits (mask_t *a, char *bit)
```

Example:

```
initcolorhex (&a, "ffff0000abcd1111");
```

The above 3 functions can init the color ’a’ bitmap by using a string. Do not use incorrect characters in the string and do not try to put to many bits into a color bitmap (see code in appendix D). Not set bits remain zero. Note that the *tecc* program uses function colorIntStr to write very long integers, and that these very long integers can be converted back to a color bitmap with function initcolorint.

5. REFERENCES

For SPACE see:

The usage of Space is described in the user manuals:

- Space Tutorial, October 2000
- Space User’s Manual, September 2000
- Space3D Cap. Extraction User’s Manual, October 2000
- Space Substrate Res. User’s Manual, May 2000