C3 Codes - A Simple, but Effective Method to Increase Barcode Data Capacity.

In this tutorial/proof-of-concept demonstration, I'll be showing you how to triple the data capacity of 2D barcodes very easily, using a few simple Linux commands.

First of all, you need to install qrencode and zbar-tools to encode and decode standard QR Codes. While you're at it, also make sure you have imagemagick installed. If your distro is Debian based (Mint, Ubuntu, Raspbian, etc.), this can all be done by typing:

```
sudo apt-get update
```

```
sudo apt-get install qrencode zbar-tools imagemagick
```

Now we're ready to start. In order to make our first standard QR Code, try the following command:

```
qrencode "Whatever you want the message to be" -o output.png
```

Yes it's that easy! You can decode the QR Code with this:

```
zbarimg output.png
```

Now that we know how to encode and decode standard QR Codes, we're going to make three of them, each containing the same amount of characters. This means we should get QR Codes of the same size/version, which is crucial in order for this method to work. More on this later.

These are the three commands we will be using:

```
qrencode "[RED RED RED RED] This is the Red Channel! Here is a list of some common Red things:
Tomatoes, Rubies, Blood, Fire, Roses, Mars, Strawberries. [RED RED RED RED]" -o channel-0.png
```

```
qrencode "[GREEN GREEN GREEN] This is the Green Channel! Here is a list of some common Green things:
Forests, Limes, Frogs, Broccoli, Emeralds, Grass. [GREEN GREEN GREEN]" -o channel-1.png
```

```
qrencode "[BLUE BLUE BLUE] This is the Blue Channel! Here is a list of some common Blue things:
Sapphires, Water, The Sky, Blueberries, The Ocean, Jeans. [BLUE BLUE BLUE]" -o channel-2.png
```

If you want, use zbarimg to check if they all decode and then move on to the next step. If you are encoding your own message, ensure each of the three QR Code images are exactly the same width and height. I repeat, it is VERY important that each of the three QR Code images are exactly the same width and height. This composite colour method will not work if they are not! In my case they were each 171 by 171 pixels, but you may get a different result.

Note: qrencode will pad out any unused data capacity and will also automatically choose the smallest QR version size that can contain your message. Just keep in mind there is a little bit of leeway here. For example, a message that is 70 characters long will likely produce a QR Code the same image height and width as a QR Code that contains a message 62 characters long. You don't have to have all three messages exactly the same length. We are only doing it in this example for simplicity's sake. For more information: `man qrencode`

If you've used QR Codes in linux before, everything up to this point is nothing new to you, but now is when (excuse the pun) the magic happens! We'll be using imagemagick to create a Composite Colour Code, merging all three QR Codes into the footprint of one. Type out the following:

```
convert channel-0.png channel-1.png channel-2.png -combine composite.png
```

Now we have a colourful image that contains three monochrome QR codes! Display it on a screen and then take a photo of it. (You could just skip this step and continue using the same image, but that wouldn't really be a fair representation of how it would normally be done.)

If you took the photo with a high resolution camera you may want to shrink the image. A resolution of around 800 by 600 or smaller seems to decode well when using zbarimg. This gets the job done for me in most cases:

```
convert image.jpg -resize 600 image.png
```

To decompose into channels, we need to do this:

```
convert image.png -separate output.png
```

This will create three standard QR Codes named: output-0.png, output-1.png and output-2.png

You can decode them as normal with zbarimg:

```
zbarimg output-0.png output-1.png output-2.png
```

What if I have one big message, rather than three equal sized messages?

Put your full message in a text file called "message.txt":

```
nano message.txt
```

Type whatever you want and save.

Then split the message into three roughly equal sized parts:

```
split -n 3 message.txt
```

This will create three files named xaa, xab and xac.

Rather than type out the messages in the qrencode command, you'll find this easier to do:

```
cat xaa | qrencode -o 0.png
```

```
cat xab | qrencode -o 1.png
```

```
cat xac | qrencode -o 2.png
```

Then make the composite as usual:

```
convert 0.png 1.png 2.png -combine composite.png
```

For decomposing it's also the same process as before:

```
convert composite.png -separate channel.png
```

When you want to recombine the message on the other end do either this: (to display in terminal)

```
zbarimg --raw channel-0.png channel-1.png channel-2.png
```

Or this: (to recreate a single text file)

```
zbarimg --raw channel-0.png | head -c-1 > channel-0.txt
```

```
zbarimg --raw channel-1.png | head -c-1 > channel-1.txt
```

```
zbarimg --raw channel-2.png | head -c-1 > channel-2.txt
```

```
cat channel-0.txt channel-1.txt channel-2.txt > rejoined.txt
```

We need to add the head command here because zbarimg adds a newline to the output, which would alter the formatting of the final file.

Note: These composite colour codes don't decode well once printed. You should probably only use them in applications where they will be displayed on a screen of some kind.

How I figured out the composite colour method:

You may have heard of HCC2D codes, which are according to Wikipedia, currently in prototyping stage. I hadn't done much research on it, but after seeing a picture of one while researching standard QR codes and seeing specifically which colours are used, I guessed it most likely worked by overlaying three QR Codes, each with a different colour channel. It appears I guessed wrong and that their method works by encoding / decoding on a per square basis and uses complex colour sensing techniques.

I decided to see if I could reproduce some working examples myself and spent a while messing about in GIMP with transparency, layers, screen mode. I even thought I'd need to learn Gimp's script-fu to automate the process. After a bit more research I realised it could all be done quite easily with some very short and simple imagemagick commands.

I call this the "composite colour method" to distinguish it from whatever far more advanced techniques and protocols end up being incorporated into the HCC2D standard. It should not be assumed Composite Colour Codes will ever be compatible with any real standards. This is literally just a quick hack that currently has minimal testing. It works, but can be a bit buggy.

How Composite Colour Codes work:

The way this method works is that the colour (red, green or blue) has to be bright in order to simulate white once it is decoded. Within each channel the other colours should just appear black.

Let's look at this simplified two channel example:

Look at the bottom right corner of the yellow composite. There are two green pixels here. According to the red channel, these are black pixels, because when viewed in the red channel, green shows as black. The same goes for the red pixels near the middle on the right. Here the pixels are red because they are deactivated (black) in the green channel, but activated (red) in the red channel. If a pixel is active in both channels it will turn yellow in this composite example. If a pixel is inactive in both channels, it will be black in the composite.

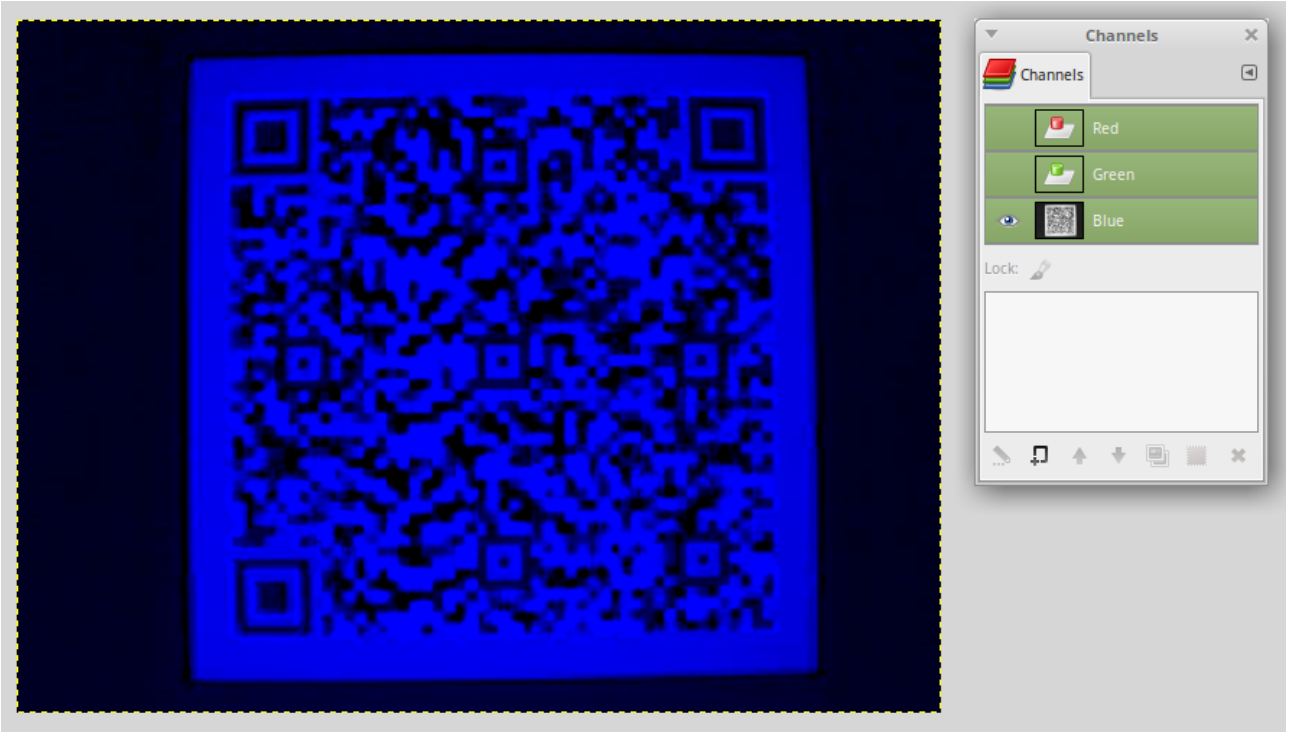
The same concepts apply when using all three (red, green and blue) channels. Black should be thought of as zero colours and white should be thought of as all colours. These are the possible colour combinations:

Black	+ **Black**		= **Black**	(0 + 0 = 0)
Red	+ **Black**		= **Red**	(1 + 0 = 1)
Green	+ **Black**		= **Green**	
Blue	+ **Black**		= **Blue**	
Red	+ **Green**		= **Yellow**	
Green	+ **Blue**		= **Cyan**	
Blue	+ **Red**		= **Magenta**	
Red	+ **Green**	+ **Blue**	= **White**	

Look at how this composite appears in each channel when using the "Channels Dialogue Window" in GIMP:


Channels
Channels
Red
Green
Blue
Lock:


Channels
Channels
Red
Green
Blue
Lock:



(I used cheese webcam application to capture the images on a low resolution webcam. As you can see, the image quality isn't brilliant, but each channel still decodes well if you separate them out from the composite. Increasing the saturation setting in cheese to maximum can sometimes help.)

Study closely how each colour in the composite appears within each channel and you'll start to understand how this system works.

Naming:

I call the images created with this method "Composite Colour Codes" or C3 for short. C3 also applies because it generally works with three channels. Even though "code" is repeated twice, it sounds better to say "a C3 Code" when talking about one of the images in the singular, so that is what I'll be doing in this tutorial.

Current issues with this method:

Please note that these C3 Codes don't decode well once printed on paper. At the moment this method is only recommended for applications where the composite will be displayed on a screen of some sort. If you want a printed 2D barcode, you will probably be better off sticking with standard monochrome QR Codes.

In my testing with a low budget Canon ink-jet printer and composites of QR Codes with low level error correction, the red channel nearly always decodes (with flash/bright light or without). The blue channel is generally decodable as long as you use a flash or very bright white light to take the photo. The green channel sometimes decodes, but much more rarely. Green seems to be the hardest to decode. I think this is due to green being a very dark colour once printed in comparison to how it is displayed on a screen. This may not be the case with other types of printers/ink, so please test it out for yourself. One thing I have noticed is that it may be easier to decode the green channel without using a flash, but that may have just been due to ambient light when I took the photo.

You could always make your own C3 Codes that contain just two channels (red and blue for example) and you'd still get double the normal data capacity of a QR Code. If you want to do this, make sure you create a black square that is the same pixel height and width as the other QR Code images and then combine all three with imagemagick as usual.

Use this to create a black square image: (Change the size parameter to whatever you need!)

```
convert -size 50x50 canvas:black output.png
```

You'll get unexpected colour outputs if you combine only two channels without the black there.

Put the black square image in place of where the missing channel would be. For example, if you want a two channel composite that is missing the green channel, do this:

```
convert red_channel.png black_square.png blue_channel.png -combine red_blue_composite.png
```

Troubleshooting and Tips: (When a photo is taken of a C3 Code displayed on a screen)

If you're having trouble getting anything to decode it sometimes helps to shrink the resolution of the composite image before decomposing. This is especially true if you took the picture with a high megapixel camera. A resolution of around 800 by 600 or less seems to decode well when using zbarimg. This gets the job done for me in most cases: (regardless of orientation)

```
convert image.jpg -resize 600 image.png
```

One way I was able to salvage an unreadable channel after it had been decomposed, was by changing the threshold value or the brightness and contrast in GIMP colour settings. This is also possible with imagemagick. The issue with trying to enhance the picture this way is that it's mostly a manual adjustment process until it looks about right.

You'll find that generally the green channel will decode directly from the composite image, without needing to separate it out from the other channels. This does not apply if it was printed on paper.

Some cameras (specifically a webcam I've tested) seem to capture blue as purple. As a result, you'll find that the red channel is not decodable. I believe this is due to not having an infra-red filter, although I could be completely wrong about that.

Smaller capacity C3 Codes that have less squares, decode easier than larger ones that have more squares. This is probably due to chromatic distortion being an issue the more squares there are. Once you start to get to high square counts, some channels may not decode. This also has a lot to do with camera quality and image resolution. I normally get better results using a high resolution camera and then shrinking the image than if I use a low resolution camera.

Note: You may find that when decoding large QR Codes, zbarimg sometimes erroneously detects a different type of barcode such as EAN-8. If this happens zbarimg will append unwanted data to the message. Use the "-S" option of zbarimg to avoid this. For example:

```
zbarimg -Sdisable -Sqrcode.enable image.png
```

This will disable scanning of all barcodes and then re-enable QR Code only scanning.

For more information: `man zbarimg`

Possible Improvements:

One thing that would be nice to implement would be an extra level of error correction. It is possible to use one channel solely for the parity of the other two channels. This is the way RAID 4 works, but with hard drives instead of QR Codes. Imagine the red channel and the green channel contain exactly half of the total data each, the blue channel could encode the XOR result of them. This way if one of the channels can not be decoded it would always be possible to recover the information. This would of course mean that total data capacity would be reduced by a third, but it may be worth it in some circumstances, such as when the C3 Code is going to be printed out.

Note: You should probably try increasing the standard QR code error correction levels before using this method. This can be done very easily with qrencode.
For more information: `man qrencode`

Try the following if you want to create a C3 Code with a parity channel:

Put your full message in a text file called "message.txt":

```
nano message.txt
```

Type whatever you want and save it.

Next we need to make sure it has an even amount of bytes:

```
wc -c message.txt
```

If the number is even, the message is ready for splitting.
If not, go back into the text file and add a space or something as padding.

Then do the following:

```
split -n 2 message.txt
```

This will split the message into two, creating the files "xaa" and "xab". These files should be EXACTLY the same size. This is very important if we are to create the parity correctly.

Next we'll make sure our system can compile and also download a utility to do XOR operations. On Debian based systems, do this:

```
sudo apt-get install build-essential
```

```
wget http://raw.githubusercontent.com/scangeo/xor-files/master/source/xor-files.c
```

Compile the xor-files.c source code to get the executable program:

```
gcc xor-files.c -o xor-files
```

Now we are going to calculate the XOR result of the two halves of our message with xor-files:

```
./xor-files -r xor xaa xab
```

This will create the file "xor", which will be the XOR parity calculation of xaa and xab.
For more information: `./xor-files -h`

Then qrencode each file:

```
cat xaa | qrencode -8 -o 0.png
```

```
cat xab | qrencode -8 -o 1.png
```

```
cat xor | qrencode -8 -o 2.png
```

Make a composite of all three channels using the normal method:

```
convert 0.png 1.png 2.png -combine composite.png
```

To decompose into channels after displaying and taking photo: (Remember to resize your image if needed!)

```
convert input.png -separate output.png
```

Decode whatever you can. zbarimg each channel image and pipe it to head to remove the trailing newline:

```
zbarimg -Sdisable -Sqrcode.enable --raw output-0.png | head -c-1 > output-0
```

```
zbarimg -Sdisable -Sqrcode.enable --raw output-1.png | head -c-1 > output-1
```

```
zbarimg -Sdisable -Sqrcode.enable --raw output-2.png | head -c-1 > output-2
```

Note: If you can't decode a channel at this stage, you'll have to repeat the XOR calculation you did earlier on whichever two decoded messages you have. This will recover the data of the missing channel:

```
./xor-files -r output_that_is_missing output_that_you_have other_output_that_you_have
```

Finally rejoin the two files that contain the message (not the parity) with cat:

```
cat output-0 output-1 > rejoined.txt
```

Note: The issue I had when using a parity channel (which is non-text binary data) was in the final decoding stage. I couldn't get zbarimg to decode pure bytes, so you may need to use a different decoder instead (see below). Encoding is a simple process, as qrencode handles this functionality fine.

Note: Use the "-8" switch when using qrencode to encode in 8 bit mode.
For more information: `man qrencode`

Can C3 Codes store data other than text?

In theory yes. C3 Codes should be able to encode any type of file (images, programs, zip files, maybe even very short audio!) just like QR Codes can. It's just that at the moment, zbar doesn't seem to be able to handle pure data (I couldn't get it to work in Debian, but you may have better luck than me if you are using a different distro or OS).

Apparently ZXing (which is available as a library for various programming languages) has no problem decoding non-text data. I couldn't find a command line utility that uses it though, so you'll have to mess about with online applications or create your own.

Should I use this method with base64 or similar to encode data to text?

Probably not. Try to use a different decoder instead.

What we are doing with this method, is cramming three QR Codes into the space of one. That triples the amount of data you can store in a single QR footprint. We are trying to maximise data capacity.

Using base64 means wasting 2 bits per byte, which adds up to a lot of wasted of data. Nowadays, that normally isn't such a big deal with fast networks and terabytes of data storage everywhere, but the capacity of a QR Code by comparison is very small.

What is really needed is a way to use the full 8 bits of data per byte and be efficient.

Note: If you want to try to get zbar working in 8 bit mode, you may want to look into ISO-8859-1 (in theory the byte encoding QR Codes support) and UTF-8 (The byte encoding most QR software seems to support). Also check out iconv.

Could we take this to another level and make C3 Codes store even more information?

Well, kind of. We can create an animation of multiple C3 Codes, which I call "Collated Composite Colour Codes".

In this example I want to put my message into 4 separate composites, each containing 3 channels, so I need to split my message into 12 parts:

```
split -n 12 message.txt
```

Create a.png, b.png, c.png and d.png using three message portions for each composite:

```
cat xaa | qrencode -o 0.png
```

```
cat xab | qrencode -o 1.png
```

```
cat xac | qrencode -o 2.png
```

```
convert 0.png 1.png 2.png -combine a.png
```

(Repeat the process to create b.png, c.png and d.png)

Note: qrencode will occasionally create QR codes that have different image sizes, even if the message portions have the same amount of bytes. This is due to the way QR Codes store text and how qrencode automatically selects the smallest possible QR version. If this happens use the "-v" option to manually specify the symbol version needed. For more information: `man qrencode`

Then to create the collated composite run this:

```
convert -delay 100 a.png \( b.png -rotate 90 \) \( c.png -rotate 180 \) \( d.png -rotate 270 \) -loop 0 collated_composite.gif
```

(I use the rotate option here, because as far as I know it doesn't affect decoding and it makes it easier to see which channel came from which composite later on.)

To decompose each channel of each composite:

```
convert collated_composite.gif -separate output.png
```

Decode each output with this: (repeat for as many outputs as you have)

```
zbarimg -Sdisable -Sqrcode.enable --raw output-0.png | head -c-1 > output-0.txt
```

Then rejoin all the outputs with cat:

```
cat output-*.txt > rejoined.txt
```

Note: When creating the composite, it might be a good idea to start with something like a plain black or white square in order to indicate where the sequence begins and ends.

Note: An alternative way to decode a collated composite is to output each channel as a single animated gif:

```
convert collated_composite.gif -separate output.gif
```

zbarimg can decode the animated gif directly with this command:

```
zbarimg -Sdisable -Sqrcode.enable --raw output.gif > output.txt
```

(The issue with this alternative method, is that there isn't an easy way to remove the trailing newlines zbarimg appends to each portion of the message.)

Can C3 Codes be a valid way to stream data?

I suppose so, but in comparison to any modern type of data transfer it would be incredibly slow.

If you only take into account how long a QR decoder program will take (not including channel decomposition or encoding/composition on the other end), then at best you'd probably be looking at around 2 kilobytes per second (See Note 1 below). This is a guess based on a few tests I did which involved various QR versions on the lowest error correction setting, so these are all ideal/maximum byte per QR Code settings. Obviously at speeds like 16Kb/s not many people would consider it worth looking into.

One situation where this sort of data transfer would be much faster is when a stream has already been encoded, is ready to send and can be decoded very slowly later on. For example, you walk up to a display, switch on your high speed camera and start the data stream on the display. When the stream finishes the transfer is complete, but the message has not yet been decoded and won't be for quite some time. You are free to walk off with your camera though, as a connection is no longer required. If it were possible to capture around 50 images per second, then 2 megabits per second transfer speeds might be achieved (See Note 2 below). Again I have no idea if that would actually be possible due to the way screens (and cameras?) use multiplexing to refresh line by line, not an entire image at a time. An LED matrix might do the job better, but it would have to be wired to refresh one full image at a time.

I apologise that this final section is pretty much all speculation. I only mention it here in the hopes that others might test this out and see what can be done.

Note 1: (Decoding Speeds)
Tests done using standard QR Codes with zbarimg on a Dell laptop with 3GB of RAM and a Core i3 CPU running at 2.27GHz. Better hardware is available, so please run your own tests!

Message	Decoded		Speed
0.5KiB	0.33 sec	=	1.50KiB/s
1.0KiB	0.50 sec	=	2.00KiB/s
1.5KiB	0.66 sec	=	2.25KiB/s
2.0KiB	1.3 sec	=	1.50KiB/s
2.5KiB	2.8 sec	=	0.90KiB/s
All 5	5.3 sec	=	1.40KiB/s

Note 2: (Transfer Speeds)
If each image contained 6KiB of data (3 channels containing 2KiB each) at 50 images per second = 300KiB per second = 2400Kib per second. Each channel could theoretically contain 2953 bytes maximum which is close to 3KiB, so faster speeds might be achievable. Obviously a faster camera would also increase speeds. The GoPro HERO7 can apparently record 1080p video at up to 240 fps. At 240 images per second and 6KiB per image, you would get an incredible theoretical maximum speed of 11.25Mib/s. Just keep in mind it'll take forever to decode!

Can this method be applied to other barcode types?

Yes. It is possible this method could be applied to pretty much any monochrome barcode (1D or 2D) in order to triple the amount of data stored within the original barcode footprint. Whether or not it would be worthwhile needs to be tested.

Written by Alex Inglis, April 2019 | aij@users.ourproject.org | http://c3codes.ourproject.org

---

Legal Stuff:

C3 Codes are NOT a new version of QR codes or even a type of QR Code. This method and the resultant graphical images which I term "C3 Codes" are simply a way of encapsulating, packaging or encoding multiple QR Codes. This method is merely a new addition to the myriad of formats and ways it is currently possible to store or encode standard QR Codes.

QR Codes can be, and are on a regular basis, encoded and stored in thousands of possible formats and format combinations.

For example:
Numerous possible image formats.
Numerous possible file formats (including compressed form).
Numerous possible video formats.
Numerous possible network packet and stream formats.
Numerous possible encrypted formats.
Numerous possible file system formats.
Even multiple QR codes stored within a single image, which is what is done with this method.

To make it specifically clear: The Composite Colour Code Method described in this tutorial is simply a method of encapsulation of standard QR Codes.

This method uses certain software, but is not a piece of software in itself. It is just a series of concepts.