Alex Murray
Marcel Kaltenrieder

6. Mai 2013

Inhaltsangabe

1. Abstract

In dieser Arbeit wird die Entwicklung, Herstellung, und Programmierung einer LED Matrix beschrieben.

Das Ziel dieses Projektes war es, eine Matrix aus RGB LEDs der Grösse 16x16 anzusteuern. Dabei wollten wir nicht einfach nur ein paar Farben zeigen können, sondern einen voll Funktionstüchtigen, modularen Display mit vielen Funktionen realisieren.

Entstanden ist eine kleine Konsole mit Platz für 4 Spieler, und ein 16x16 LED Display mit einer Farb-Tiefe von 12 Bit.

Die entstandene Software dazu ist das Spiel „Snake“, ein Farb-Demo, und Conway's Game of Life.

2. Einführung

Wir wollten, dass das Projekt modular bleibt, und über eine universelle Schnittstelle angesteuert werden kann. In der Theorie könnte man also mehrere LED-Matrizen herstellen und parallel schalten, um einen grösseren Display zu realisieren.

Um das ganze auch vorführen zu können, wollten wir das klassische Computerspiel *Snake* – aber für 4 Spieler anstatt nur für Einen – zeigen. Dazu wollten wir eine zweite Schaltung realisieren, welches über diese Schnittstelle mit dem Display kommunizieren kann, denn alleine ist das Display nur ein Grafiktreiber, und würde nichts machen.

Damit wir auch etwas über Grafiktreiber, Farben-Berechnungen, Signalverarbeitung und Programmieren lernen, wurde auf bestehende Lösungen verzichtet und alles von Grund auf entwickelt und realisiert.

3. Hardware

3.1. Überblick

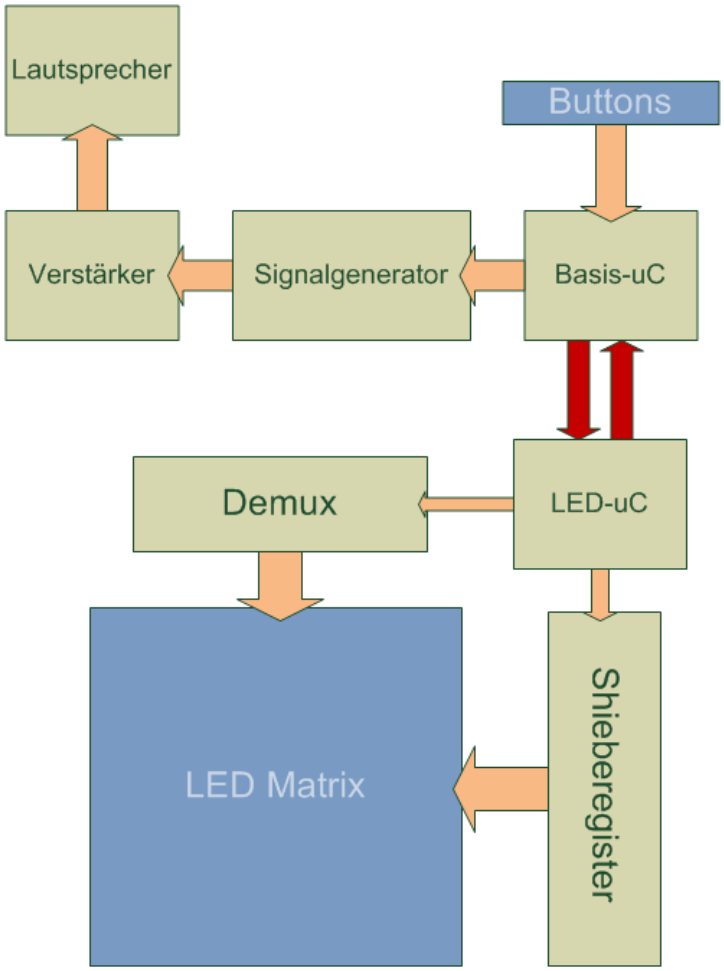
Es sind zwei separate Schaltungen realisiert worden, die miteinander über UART kommunizieren. Einer ist dazu zuständig, die LEDs kontinuierlich anzusteuern, der andere ist dazu zuständig, Computerspiele zu verarbeiten.

Für die Matrix wurde eine schon vorgefertigte Array von LEDs benutzt. Weil sie nicht dazu fähig ist, Zustände zu speichern, muss sie immer wieder angesteuert werden. Dies wird von einem Mikrocontroller übernommen, was auch für alles andere wie Zeichnungsbefehle zuständig ist.

Die Schaltung mit dem Computerspiel *Snake* beinhaltet auch ein Mikrocontroller, welches dem Display Befehle erteilt. Insgesamt können vier Spieler mitspielen, und jeder Spieler erhält fünf Eingabetaster für die Steuerung.

Zusätzlich wird noch ein Signalgenerator eingebaut, um Töne bzw. sogar Musik abspielen zu können. Dies ist aber nur aus Spass und bleibt ein Experiment.

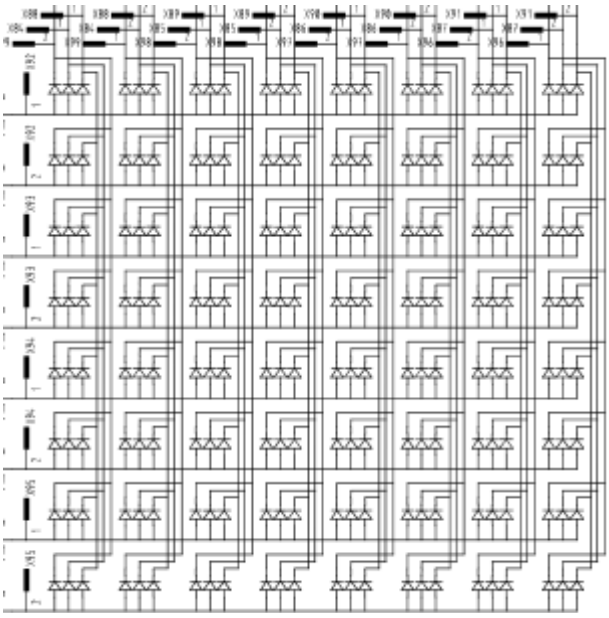
Im folgendem Diagramm ist ein Blockschaltbild des ganzen Projektes aufgezeichnet.



3.2. Display

3.2.1. Aufbau der Matrix

Die Matrix besteht aus vier 8x8 schon integrierte Schaltung von RGB LEDs, wie im Folgendem Diagramm zu erkennen ist. Zusammen ergeben sie die 16x16 Matrix.



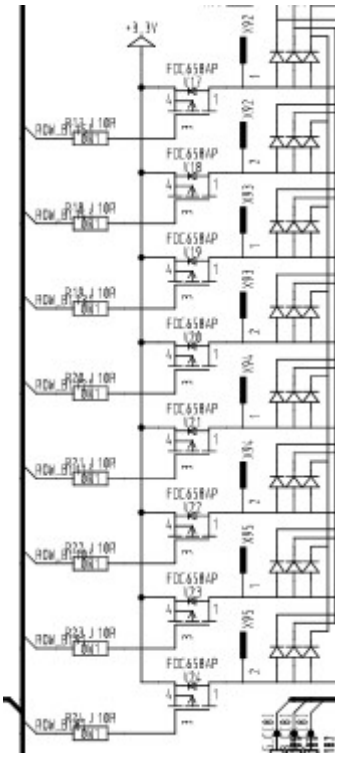
Die LEDs können physikalisch nicht gleichzeitig angesteuert werden. Auch wenn die Möglichkeit bestünde, sie alle parallel anzusteuern, gäbe es nur wenige Mikrocontroller mit 768 IO-Pins, also ist das Anwenden von Multiplexer unausweichlich.

3.2.2. Ansteuerung der Matrix

Es wird von Reihensteuerung und Datensteuerung unterschieden. Mit Reihensteuerung ist die Auswahl einer der 16 Reihen zu verstehen. Mit Datensteuerung ist die Ausgabe der Einzelnen Farben für die ausgewählte Reihe zu verstehen.

Nur wenn eine Anodenseite der Matrix auf *high* gezogen wird, und eine Kathodenseite der Matrix auf *low* gezogen wird, leuchtet die Entsprechende LED. Die LEDs haben jeweils gemeinsame Anoden, also muss die Datensteuerung auf der Kathodenseite erfolgen und entgegensetzt die Reihensteuerung auf der Anodenseite erfolgen.

Ein 4 zu 16 Demultiplexer wurde eingesetzt, um p-MOSFETs auf der Anodenseite zu schalten. So wird jeweils immer nur eine Reihe aufs mal ausgewählt.



Pro Reihe müssen 48 Bit an das Display geschrieben werden. Auf der Kathodenseite hängen 48 NPN-Transistoren, welche die entsprechende LEDs auf *low* ziehen, damit sie leuchten können.

Die Daten können schlecht parallel vom Mikrocontroller ausgegeben werden, weil so viele Leitungen vorhanden sind. Deshalb werden sechs 8-Bit Schieberegister eingesetzt. Die Daten können jetzt zwar von nur einem Port ausgegeben werden (6 bit), müssen aber acht mal schneller ausgegeben werden um die gleiche Geschwindigkeit zu erreichen.

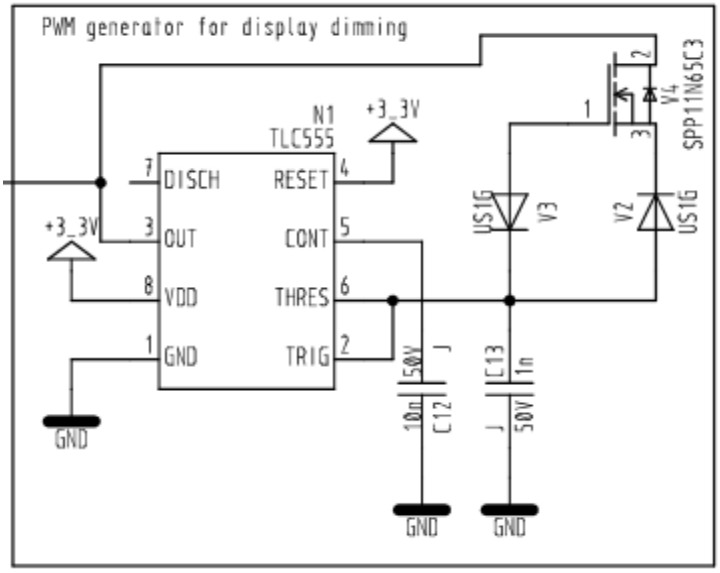
Nochmal als Zusammenfassung: Es wird zuerst eine von den 16 Reihen ausgewählt. Danach werden jeweils für die 6 Schieberegister 8 Bit seriell ausgeschrieben (insgesamt 48 bit). Wenn das soweit ist, werden die 48 Bit, welche nun in den Schieberegistern sind, direkt an die LEDs geschrieben. Der Prozess wird dann für die nächste Reihe wiederholt, und so weiter.

Die Vorwiderstände für die LEDs wurden wie folgt berechnet.

$$R_V = \frac{V_{CC} - U_V - U_{DS} - U_{CE}}{I_F * n}$$

Da jede LED im schlimmsten Fall $\frac{1}{16}$ der Zeit eingeschaltet ist, kann die LED auch mit Faktor 16 übersteuert werden, und erhalten so einen relativ niederohmigen Widerstand von 75Ω.

Der PWM-Generator (siehe folgendes Bild) wurde nicht bestückt.



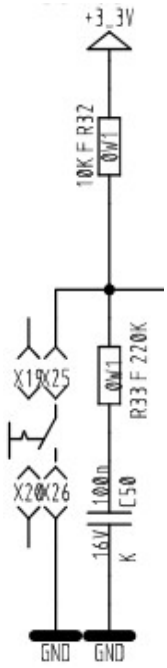
3.3. Base

3.3.1. Aufbau der Base

Das Ziel hier war es ein Schema zu entwickeln, das Eingaben von einem Benutzer nehmen kann, um das Display ansteuern zu können.

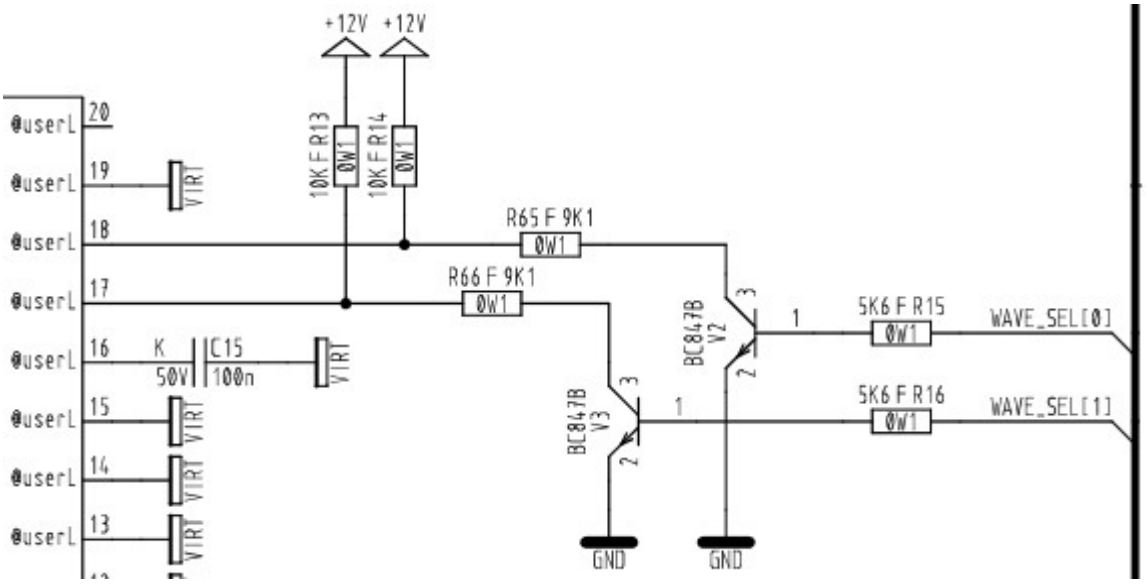
Die Base besitzt 20 Taster und ein Signalgenerator, was später vielleicht für Musik oder Töne eingesetzt werden kann. Das bleibt aber experimentell und wird wahrscheinlich aus Zeitgründen nicht umgesetzt.

Die Taster sind durch einem RC-Netzwerk hardware-entprellt, wie hier zu sehen ist:

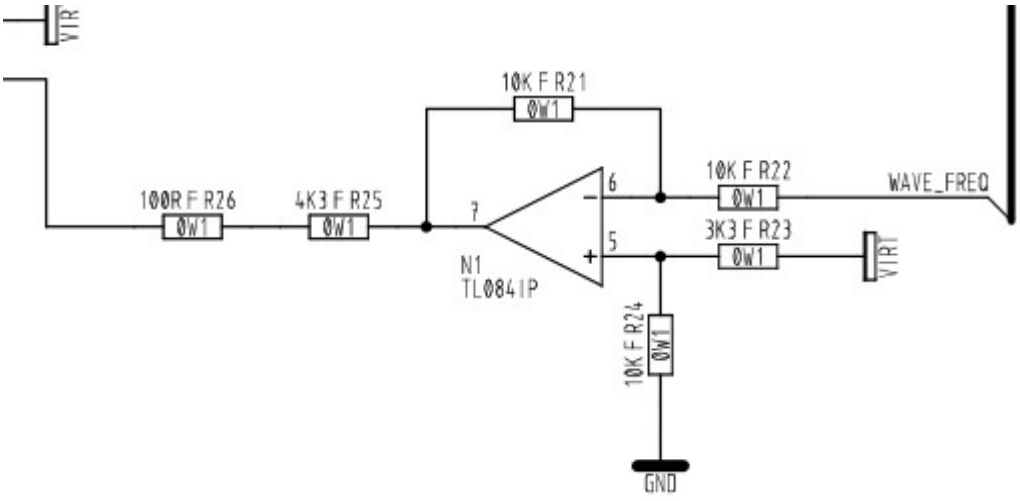


Der Signalgenerator ist mit einer Bipolarquelle von +-6V gespiesen (die Betriebsspannung wird durch ein Step-Up-Converter auf 12V umgewandelt, und eine virtuelle Masse von 6V wird durch ein Spannungsteiler und Impedanzwandler erzeugt).

Die Ausgangssignalform vom Signalgenerator kann zwei digitale Anschlüsse bestimmt werden, nämlich Rechteck, Dreieck, oder Sinus:



Ein dritter, analoger Anschluss bestimmt die Frequenz. Die analoge Spannung vom Mikrocontroller (0..3.3V) muss in ein Konstant-Strom umgewandelt werden (bezogen auf die virtuelle Masse von 6V), weil der Signalgenerator so angesteuert werden muss (siehe Datenblatt). Das wird durch ein subtrahierender Verstärker realisiert.



Somit ist am Ausgang vom Operationsverstärker eine Spannung zwischen 6V und 12V. Die Widerstände R25 und R26 bestimmen, wie viel Strom im Signalgenerator fliessen darf, was schlussendlich die Frequenz bestimmt.

4. Software

4.1. Display

4.1.1. Einführung

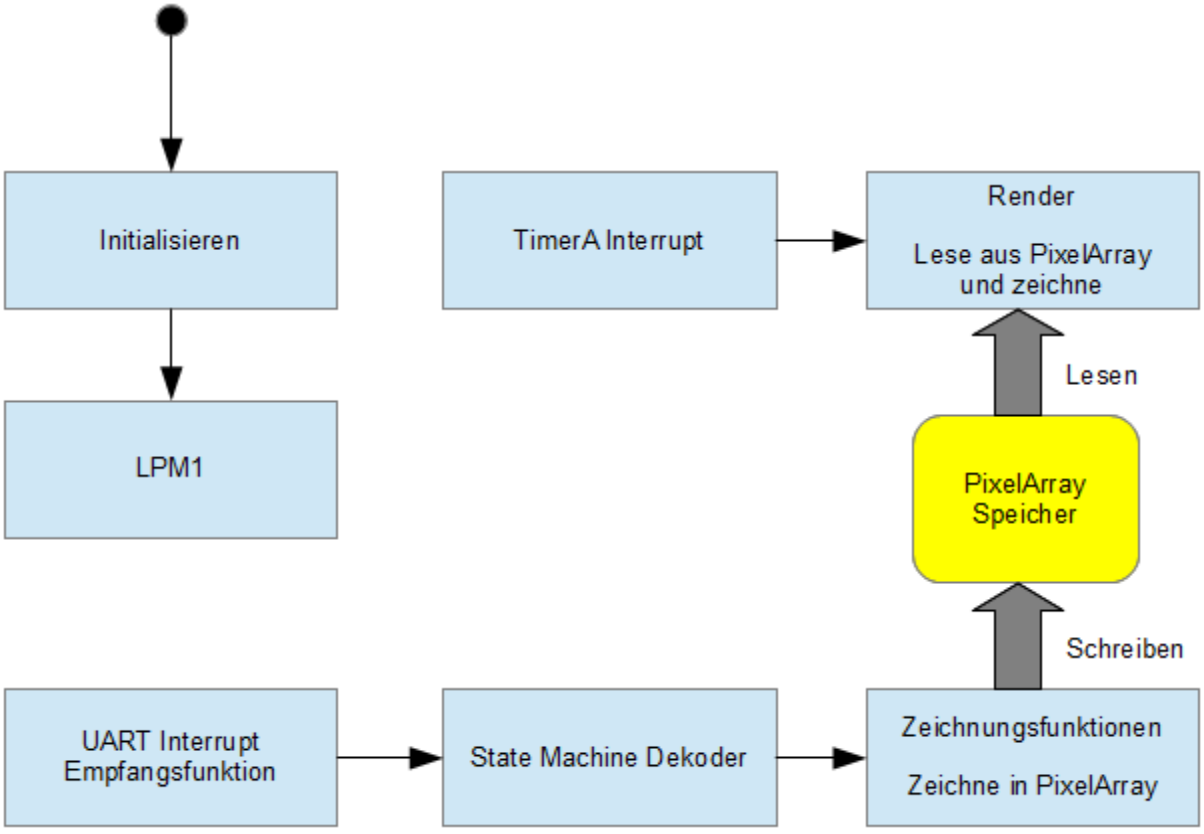
Das Display kann über UART angesteuert werden. Sie verfügt über zahlreiche Funktionen, die für das Zeichnen von verschiedenen Muster zuständig ist, und verfügt auch über Funktionen, die kontrollieren, wie die Zeichnungsbefehle mit den schon vorhandenen Pixel verlaufen (siehe *blend mode).*

Das Display ist mehr als nur eine einfache Anzeige, sie ist schon eine kleine „Grafikkarte“.

Die LEDs haben selbst keinen Speicher, deshalb müssen sie kontinuierlich und konstant immer wieder angesteuert werden. Durch viele Schieberegister und Demultiplexer war es möglich, die Ansteuerung der 768 LEDs auf nur 13 Leitungen zu reduzieren.

4.1.2. Programm Überblick

Das Programm ist wie folgt aufgebaut. Funktionen sind Hellblau und Datenspeicher sind Gelb eingezeichnet.



4.1.3. Initialisierung und Einstellungen

Obwohl ein externer Quarz eingebaut wurde, wurde entschieden, den internen Taktgenerator zu benutzen, weil er schneller ist. Er wurde auf eine Frequenz von 29 MHz eingestellt, was sich als ziemlich die höchstmögliche Frequenz erwiesen hatte.

TimerA wird benutzt ,um ein Interrupt auszulösen, der den momentanen Zustand im PixelArray an die LEDs schreibt. Durch den TimerA wird eine konstante Refresh-Rate erreicht. Der Interrupt wird ungefähr 80 mal pro Sekunde ausgelöst.

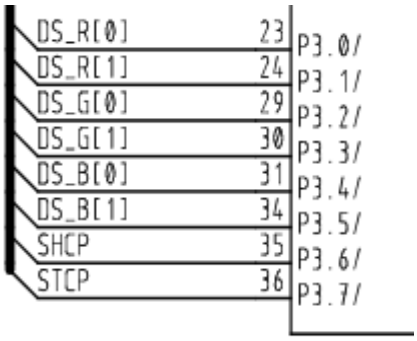
Das UART-Modul wurde auf einer Baudrate von 3‘686‘400 eingestellt, um möglichst schnell die Daten übertragen zu können.

4.1.4. Pixel Array

Alle Daten für die Pixel werden in ein *PixelArray* abgespeichert, was nichts anderes ist als ein 256-bit grosser Array.

Die Datenstruktur im Array wurde so optimiert, dass es am schnellsten ist, die Daten an die LEDs auszuschreiben. Dies wird aber auf Kosten von Speicherplatz im RAM getan, weil die Daten extrem „ausmultipliziert“ werden. Die Entscheidung für dieses Vorgehen fiel dadurch, dass das Ausgeben der LEDs sonst zu langsam gewesen wäre.

Die Verdrahtung sieht wie folgt aus:



Und wir haben die Daten im Array genau so strukturiert:

Bit	Funktion
0	Blau Rechts
1	Blau Links
2	Rot Rechts
3	Rot Links
4	Grün Links
5	Grün Rechts
6	0
7	0

Es können Daten für zwei Pixel pro Element im Array gespeichert werden. Was auch noch auffällt, ist dass Bit 4 und Bit 5 hardwaremässig vertauscht sind, deshalb müssen sie per Software wieder zurückgetauscht werden.

Das Array ist drei-dimensional und hat die Grösse 8x16x14, oder *[Column] [Row] [PWM vector]*. Die X-Dimension gibt an, wo auf der X-Achse der Pixel sich befindet. Sie ist nur 8 Stellen gross anstatt 16, weil die Daten für die linke sowohl auch für die rechte Seite des Displays gleichzeitig durch Maskierungen im gleichen Byte passen. Die Y-Dimension gibt an, wo auf der Y-Achse der Pixel sich befindet. Die Z-Dimension, oder „PWM-Vector“, gibt die Helligkeit an.

4.1.5. Refresh und PWM

Etwa 80 mal in der Sekunde wird eine Interrupt-Routine aufgerufen, damit die LEDs erneut ausgegeben werden können.

Dummerweise ist die Helligkeit der LEDs nicht proportional zur Duty-Cycle, weil unsere Augen logarithmisch sind. Zur Kompensation wurde ein Lookup-Table generiert aus der folgenden Formel:

$$delay = cycle * 2^{(-0.5 * d_i)}$$

Die neue Verzögerungszeit $delay$ hat für eine PWM-Auflösung von 14 die folgenden Werte: *255, 180, 128, 90, 64, 45, 32, 23, 16, 12, 8, 6, 4, 3, 2, 1,0*

Und entspricht damit die neue Duty-Cycle.

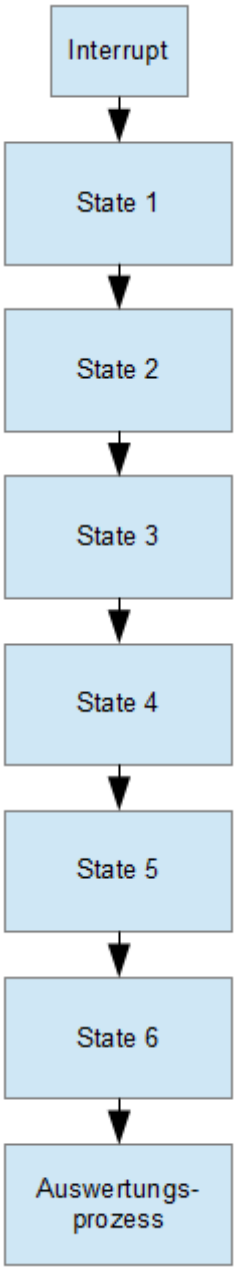
4.1.6. UART Empfang

Wird ein Byte empfangen, wird es sofort durch eine Interrupt-Routine ausgewertet. Die Auswertung erfolgt auf Basis einer *Binary Tree State-Machine*. Es wird ein Anfangszustand initialisiert, und für jedes Byte, das empfangen wird, ändert sich dieser Zustand, bis genug Daten angesammelt sind, um den Befehl ausführen zu können, wodurch sich der Zustand wieder auf den Anfangswert zurücksetzt.

Nachdem es ausgewertet worden ist, werden die empfangene Daten wieder zurückgesendet, damit der andere Mikrocontroller den nächsten Byte senden kann.

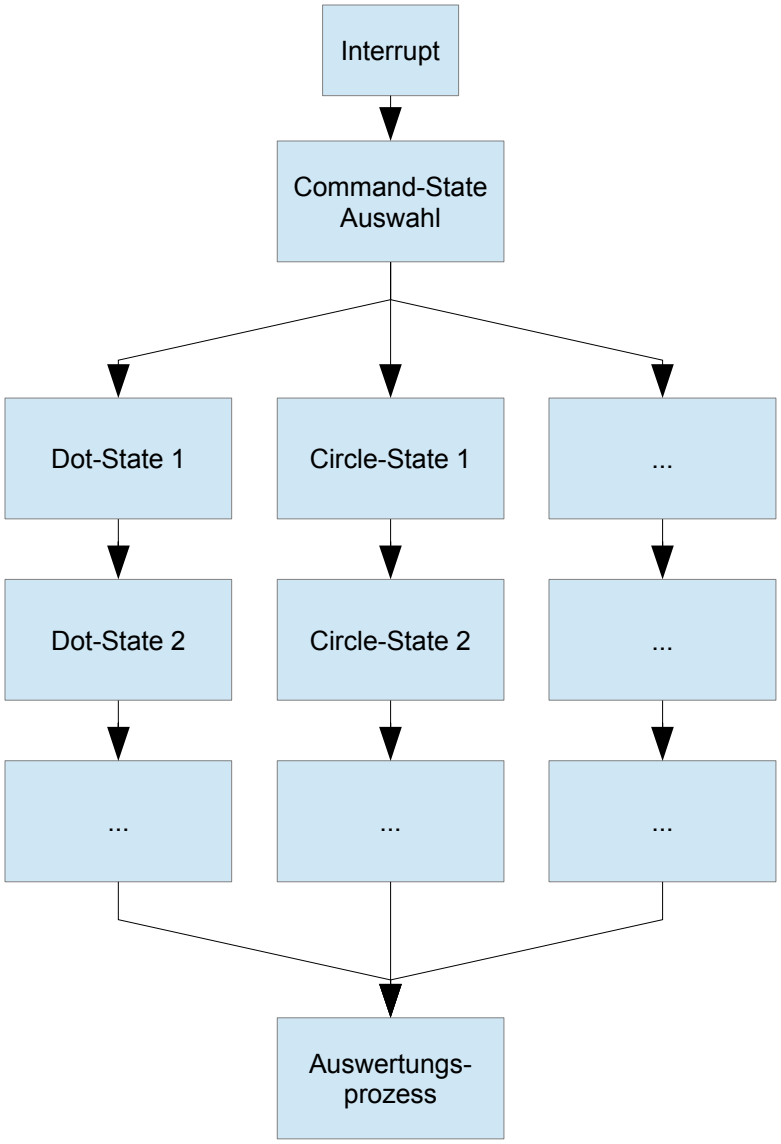
Dabei ist zu beachten, dass unerwartetes Verhalten auftreten kann, wenn während dem Auswertungsprozess weitere Daten empfangen werden.

Im folgenden Diagramm ist ein Beispiel einer normalen *State-Machine*.



Der Nachteil besteht darin, dass im schlimmsten Fall alle Zustände mit dem empfangenem Signal geprüft werden. Bei einer kleinen Anzahl von möglichen Zuständen wäre es nicht problematisch, aber beim Display sind schon eine Menge vorhanden.

Es ist möglich durch ein *Binary Tree* eine grosse Portion der geprüften Zustände auszuschliessen, und somit die Zeit, die es für die Verarbeitung braucht, im Idealfall bis zu $\sqrt{(t)}$ verkürzen zu können. Die sieht wie folgt aus:


Interrupt
Command-State Auswahl
Dot-State 1
Circle-State 1
...
Dot-State 2
Circle-State 2
...
...
...
...
Auswertungs-prozess

Obwohl die Daten extrem schnell gesendet werden, und die Empfangsroutine sowohl auch die Zeichnungsbefehle extrem optimiert wurden, konnten wir eine Refresh-Rate von nur 8 fps erreichen (2048 *dot*-Befehle pro Sekunde).

Bei der Ansteuerung ist es deshalb von Vorteil, nur die *veränderten Pixel vom letzten Zustand* zu senden, um Zeit zu sparen.

4.1.7. Blend Modes

Das Display unterstützt vier verschiedene *Blend-Modes*. Damit wird bestimmt, wie die empfangene Farben mit den schon vorhandenen Farben verrechnet werden. Der Standard-Modus ist *Replace* (Ersetzen).

- **Replace** – Dieser Modus ersetzt die vorhandene Farben mit den neuen Farben.
- **Add** – Dieser Modus addiert die vorhandene Farben mit den neuen Farben.
- **Subtract** – Dieser Modus subtrahiert die neuen Farben von den vorhandenen Farben.
- **Multiply** – Dieser Modus multipliziert die vorhandene Farben mit den neuen Farben. Dieser Modus ist gut für Maskierungen.

4.1.8. Befehlsliste

Die Befehle werden jeweils in 8-bit Teilstücke versendet. Jeder Befehl hat einen Anfangs-Byte, und darauf folgende Argumente (wenn vorhanden).

Befehl	Argumente	Kodierung (Hex)
cls	-	00
dot	x, y, rgb	01 XY RG B0
blend_colour_box	x1, y1, x2, y2, rgb	02 XY XY RG B0
blend_colour_fill_box	x1, y1, x2, y2, rgb1, rgb2, rgb3, rgb4	03 XY XY RG BR GB RG BR GB
box	x1, y1, x2, y2, rgb	04 XY XY RG B0
fill_box	x1, y1, x2, y2, rgb	05 XY XY RG B0
blend_colour_line	x1, y1, x2, y2, rgb1, rgb2	06 XY XY RG BR GB
line	x1, y1, x2, y2, rgb	07 XY XY RG B0
circle	x, y, r, rgb	08 XY rr RG B0
fill_circle	x, y, r, rgb	09 XY rr RG B0
blend_colour_fill_circle	x, y, r, rgb1, rgb2	0A XY rr RG B0
blend_mode_replace	-	0B
blend_mode_add	-	0C
blend_mode_subtract	-	0D
blend_mode_multiply	-	0E

4.2. Base

4.2.1. Einführung

Die Base ist dafür zuständig, Eingabedaten von mehreren Benutzer verarbeiten zu können, und ist dafür zuständig, die Pixel zu berechnen und an das Display zu senden.

Weiter ist sie dazu Zuständig, Musik zu generieren mit dem Signalgenerator. Dies wurde aber aus Zeitgründen nicht umgesetzt.

4.2.2. Framework

Für die Base wurde als aller erstes ein Programm entwickelt, welches die darunterliegende, meist für den Entwickler unfreundliche Hardware abstrahiert. Die ganzen Zeichnungsbefehle, UART-Kommunikation, Eingaben mit positive Flankentriggerung, virtuelle Taster, Musikausgabe, Menu, und Timer-basierte Schleifen, sowie auch andere kleine hilfreiche Funktionen, stehen dem Entwickler zu Verfügung.

In die Header-Dateien **uart.h** und **framework.h** können alle Funktionen nachgeschaut werden, wie auch in dieser Dokumentation.

4.2.2.1. Registrierung von Programmen

Will man ein neues Spiel oder Programm schreiben, muss es als Modul mit dem Framework registriert werden. Dabei übergibt man vier Funktionsnamen dem Framework. Diese werden dann zu spezifischen Zeiten vom Framework als „Callback“ aufgerufen.

Bei der Registrierung wird zusätzlich auch noch ein Eintrag in das Hauptmenü generiert, es zwischen andere Modulen ausgewählt werden kann.

Es können theoretisch eine unlimitierte Menge an Modulen registriert werden, das Array in der Datei **moduleenable.h** muss einfach angepasst werden, da Module sonst bei Überschreiten der maximalen Anzahl nicht registriert werden. Natürlich ist in der Praxis wegen Speicherplatz ein Limit vorhanden.

```
#define MAX_MODULES 12
```

Die Namen dieser vier Funktionen sind willkürlich. Die Funktionalität bleibt gleich.

- **load()** - Diese Funktion wird ein mal aufgerufen, wenn das Programm im Hauptmenü ausgewählt wird. Hier müssen alle Vorbereitungen für das persönliche Programm hin, wie zum Beispiel das Display löschen, oder die Bildwiederholungsfrequenz eingestellt werden.
- **processLoop()** - Diese Funktion ist die „Main Schleife“. Sie wird auf basis eines Timers periodisch aufgerufen. Die frequenz mit der sie aufgerufen wird kann mit der Funktion *setRefreshRate()* eingestellt werden (Bildwiederholungsfrequenz).
- **processInput()** - Diese Funktion wird immer aufgerufen, wenn eine Taster-Eingabe detektiert wird.
- **drawMenuIcon()** - Diese Funktion wird vom Menü aufgerufen, wenn sie eine „Vorschau“ vom Modul zeichnen will. Es können hier ganz normal die Zeichnugsfunktionen eingesetzt werden, um die Vorschau zu zeichnen. Anschliessend wird ein *send()* benötigt. **Die Vorschau muss zwingend im Bereich 3,3 bis 12,12 sein.**

4.2.2.2. Beispiel

Am besten nimmt man folgende Vorlage.

C-Datei

```
// ------------------------------------------------------------------------
// Vorlage (ersetze „__VORLAGE“ mit dem Namen vom Programm)
// ------------------------------------------------------------------------

// ------------------------------------------------------------------------
// Include files
// ------------------------------------------------------------------------

#include "__VORLAGE.h"
#include "framework.h"
#include "uart.h"
#include "gameenable.h"

#ifdef GAME_ENABLE___VORLAGE

static struct __VORLAGE_t __VORLAGE;

// ------------------------------------------------------------------------
// load __VORLAGE
void load__VORLAGE( unsigned short* frameBuffer, unsigned char* playerCount )
{
}

// ------------------------------------------------------------------------
// process __VORLAGE loop
void process__VORLAGELoop( void )
{
	endGame(); // remove this
}

// ------------------------------------------------------------------------
// process __VORLAGE input
void process__VORLAGEInput( void )
{
}

// ------------------------------------------------------------------------
// draws menu icon for __VORLAGE
void draw__VORLAGEMenuIcon( void )
{
}
#endif // GAME_ENABLE___VORLAGE
```

H-Datei

```
// ------------------------------------------------------------------------
// __VORLAGE (ersetze „__VORLAGE“ mit dem Namen vom Programm)
// ------------------------------------------------------------------------

#ifndef ___VORLAGE_H_
	#define ___VORLAGE_H_

// ------------------------------------------------------------------------
// Structs
// ------------------------------------------------------------------------
```

```
struct __VORLAGE_t
{
	unsigned short* frameBuffer;
	unsigned char* playerCount;
};

// ------------------------------------------------------------------------
// Function Prototypes
// ------------------------------------------------------------------------

void load__VORLAGE( unsigned short* frameBuffer, unsigned char* playerCount );
void process__VORLAGELoop( void );
void process__VORLAGEInput( void );
void draw__VORLAGEMenuIcon( void );

#endif // ___VORLAGE_H_
```

Wie gleich zu sehen ist, braucht das Framework vier Funktionen, die registriert werden müssen. In der Datei **framework.c** ist in der Funktion **initFrameWork()** folgendes zu sehen.

```
	// register user added modules
#ifdef MODULE_ENABLE_COLOUR_DEMO
	registerModule( loadColourDemo, processColourDemoLoop, processColourDemoInput,
drawColourDemoMenuIcon );
#endif
#ifdef MODULE_ENABLE_SNAKE
	registerModule( loadSnake, processSnakeLoop, processSnakeInput, drawSnakeMenuIcon );
#endif
```

Hier muss die Vorlage registriert werden:

```
	// register user added modules
#ifdef MODULE_ENABLE___VORLAGE
	registerModule( load__VORLAGE, process__VORLAGELoop, process__VORLAGEInput,
draw__VORLAGEMenuIcon );
#endif
```

*MODULE_ENABLE___*VORLAGE muss definiert werden. Dies wird in der Datei **gameenable.h** getan.

```
// ------------------------------------------------------------------------
// Comment in the modules you would like enabled
// ------------------------------------------------------------------------

#define MODULE_ENABLE_COLOUR_DEMO
#define MODULE_ENABLE_SNAKE
#define MODULE_ENABLE___VORLAGE
```

In dieser Datei ist es möglich, Module von der Kompilation auszuschliessen, indem man sie einfach nicht deklariert.

4.2.2.3. Zeichnungsfunktionen

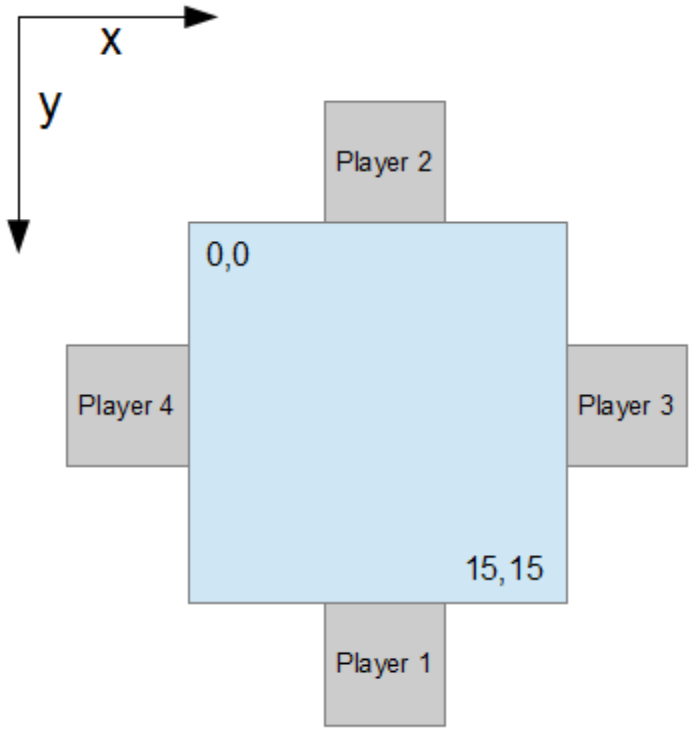
Es stehen eine grosse Anzahl an Zeichnungsfunktionen zu Verfügung. Das eigentliche *Zeichnen* wird auf dem Display ausgeführt. Die Base ist nur dazu zuständig, dem Display mitzuteilen, was es zeichnen sollte.

4.2.2.4. Anwendung von Zeichnungsfunktionen

Es sind dabei drei wichtige Punkte zu beachten.

- Der *send()* Befehl muss immer nach dem Zeichnen ausgeführt wird.
- Farben dürfen *NIE* höher als **0xEEE pro Nibble** sein. 0xF00, 0x0F0 und 0x00F dürfen **nicht** vorkommen.
- Positionen dürfen *NIE* grösser sein als **0x0F** auf beiden Achsen.

Das Display hat oben Links seine Nullstelle (von Spieler 1 aus gesehen).



Im folgendem Code-Ausschnitt werden drei Punkte mit unterschiedlichen Farben nebeneinander gezeichnet.

```
// Draw three dots
unsigned char x, y;
x=0x00; y=0x00; dot( &x, &y, &RED );
x=0x01; y=0x00; dot( &x, &y, &GREEN );
x=0x02; y=0x00; dot( &x, &y, &BLUE );
```

Diese Punkte werden aber nicht auf dem Display erscheinen, sondern werden in ein Buffer geschrieben. Erst wenn *send()* aufgerufen wird, wird das UART-Modul aktiv und beginnt, den Buffer-Inhalt zu senden.

Diese Funktionalität ist noch praktisch. Wenn zum Beispiel viele Berechnungen für jeden Pixel ausgeführt werden müssen, die Pixel aber *direkt* am Display gesendet werden, würde man eine gewisse Trägheit erkennen. Die Pixel sehen aus als ob sie langsam „gezeichnet“ werden. Wenn

sie aber zuerst in ein Buffer geschrieben werden, und erst dann gesendet werden, erkennt man diese Trägheit fast nicht.

Die drei Pixel im Beispiel oben müssen also noch gesendet werden:

```
// Draw three dots
unsigned char x, y;
x=0x00; y=0x00; dot( &x, &y, &RED );
x=0x01; y=0x00; dot( &x, &y, &GREEN );
x=0x02; y=0x00; dot( &x, &y, &BLUE );

// Send to display
send();
```

Die meisten Zeichnungsfunktionen erwarten als Argumente Pointer, was manchmal ein wenig nervend sein kann. Als Hilfe stehen in der Datei **framework.h** vordefinierte Farben, damit diese nicht immer wieder als lokale Variable definiert werden müssen.

```
// common colours
static const unsigned short BLACK      = 0x000;
static const unsigned short WHITE      = 0xEEE;
static const unsigned short RED        = 0xE00;
static const unsigned short GREEN      = 0x0E0;
static const unsigned short BLUE       = 0x00E;
static const unsigned short YELLOW     = 0xEE0;
static const unsigned short MAGENTA    = 0xE0E;
static const unsigned short LIGHTBLUE  = 0x0EE;
static const unsigned short PINK       = 0xE07;
static const unsigned short PURPLE     = 0x70E;
static const unsigned short ORANGE     = 0xE70;
static const unsigned short LIGHTGREEN = 0x7E0;
static const unsigned short BLUEGREEN  = 0x0E7;
static const unsigned short LIGHTYELLOW= 0xEE7;
```

4.2.2.5. Eingabe

Die Tastereingaben müssen immer in der Funktion **process__VORLAGEInput()** sein, und können ganz einfach abgefragt werden. Es sind für jeden Spieler sechs Taster vorhanden.

- **player1ButtonLeft()** - Wenn Spieler 1 „links“ drückt
- **player1ButtonRight()** - Wenn Spieler 1 „rechts“ drückt
- **player1ButtonUp()** - Wenn Spieler 1 „auf“ drückt
- **player1ButtonDown()** - Wenn Spieler 1 „ab“ drückt
- **player1ButtonFire()** - Wenn Spieler 1 „schiessen“ drückt
- **player1ButtonMenu()** - Wenn Spieler 1 zum Menü zurück gehen will

Natürlich können die Funktionen auch gleich für Spieler 2, 3 und 4 abgeändert werden.

Weiter besteht die Option, den Spieler als Argument weiterzugeben, damit man in einer For-Schleife gleich alle Spieler abfragen kann.

```
for( unsigned char i = 0; i != 4; i++ )
{
        if( playerButtonLeft( i ) ) doSomething();
```

```
        if( playerButtonRight( i ) ) doSomething();
        if( playerButtonUp( i ) ) doSomething();
        if( playerButtonDown( i ) ) doSomething();
        if( playerButtonFire( i ) ) doSomething();
        if( playerButtonMenu( i ) ) doSomething();
}
```

Es ist zu beachten, dass die Richtungen *up, down, left, right* lokal zur Ausrichtung vom Spieler sind, und nicht zum Display.

4.2.2.6. Referenz

4.2.2.6.1. RegisterModule

BESCHREIBUNG

Registriert ein Modul im Framework, damit es durch Callbacks aufgerufen werden kann. Bevor ein Programm für die Playtrix geschrieben werden kann, muss es registriert werden.

SYNTAX

```
void registerModule( loadFunction_cb_t loadFunction,
processLoopFunction_cb_t processLoopFunction,
processInputFunction_cb_t processInputFunction,
drawMenuIconFunction_cb_t drawMenuIconFunction );

typedef void (*loadFunction_cb_t)(unsigned short* frameBuffer, unsigned char* userData);
typedef void (*processLoopFunction_cb_t)(void);
typedef void (*processInputFunction_cb_t)(void);
typedef void (*drawMenuIconFunction_cb_t)(void);
```

ARGUMENTE

Es müssen vier Funktionen registriert werden.

LoadFunction
wird ein mal aufgerufen, wenn das Modul im Menü ausgewählt wird. Dabei wird ein pointer zu einem ***frameBuffer*** *(array von 512 Byte) übergeben, und die aktiven Spieler in* ***userData*** *übergeben. Spieler 1 ist immer aktiv, Spieler 2, 3, und 4 sind bit-maskiert in* ***userData*** *vorhanden als Bit0=Spieler 1, Bit1=Spieler2, Bit2=Spieler2.*

ProcessLoopFunction
ist die „main schleife“ vom Modul.

ProcessInputFunction
wird aufgerufen, wenn eine Eingabe festgestellt wird.

drawMenuIconFunction
wird vom Menu aufgerufen, und muss innerhalb vom Rechteck A(3,3) und B(12,12) ein „Vorschaubild“ zeichnen. send() ist nicht nötig.

4.2.2.6.2. setRefreshRate

BESCHREIBUNG

Mit dieser Funktion kann festgelegt werden, wie oft die Main-Schleife vom Modul aufgerufen wird.

SYNTAX

```
void setRefreshRate( unsigned char refresh );
```

ARGUMENTE

unsigned char *refresh*
Eine Zahl zwischen 0 und 255. Je grösser, desto schneller wird die Main-Schleife aufgerufen.

4.2.2.6.3. clearFrameBuffer

BESCHREIBUNG

Löscht den Inhalt eines Frame-Buffers (array von 512 Byte).

SYNTAX

```
void clearFrameBuffer( unsigned short* frameBuffer );
```

ARGUMENTE

unsigned short* *frameBuffer*
frameBuffer von 512 Byte länge, das gelöscht werden muss.

4.2.2.6.4. rnd

BESCHREIBUNG

Generiert eine Zufallszahl zwischen 0 und 255 und gibt sie zurück.

SYNTAX

```
unsigned char rnd( void );
```

ARGUMENTE

unsigned char *retValue*
Eine Zufallszahl zwischen 0 und 255.

4.2.2.6.5. sin

BESCHREIBUNG

Sinus, Argument in Grad. Eine Zahl zwischen -127 und 128 wird zurückgegeben.

SYNTAX

```
signed char sin( unsigned short angle );
```

ARGUMENTE

unsigned short *angle*
Winkel in Grad.

signed char *retValue*
Gibt eine Zahl zwischen -127 und 128 zurück.

4.2.2.6.6. wrap

BESCHREIBUNG

Wickelt eine Zahl zwischen 0 und einem Wert.

SYNTAX

```
void wrap( unsigned short* value, unsigned char wrap );
```

ARGUMENTE

unsigned short* *value*
Die Zahl zum verändern.

unsigned char *wrap*
Zwischen 0 und dieser Zahl wird die Zahl gewickelt.

4.2.2.6.7. sqrt

BESCHREIBUNG

Berechnet die Wurzel einer Zahl.

SYNTAX

```
unsigned char sqrt( unsigned short* value );
```

ARGUMENTE

unsigned short* *value*
Eine 16-bit Zahl.

unsigned char *retValue*

Resultat, die Wurzel von „value“.

4.2.2.6.8. playerXButtonY

BESCHREIBUNG

Rückgabe der positiven Flanke einer Tastereingabe für den Spieler „X“ vom Taster „Y“.

X kann folgende Werte haben.

- *1*
- *2*
- *3*
- *4*

Y kann folgende Werte haben.

- *Up*
- *Down*
- *Left*
- *Right*
- *Fire*
- *Menu*

SYNTAX

***unsigned char** playerXButtonY(**void**)*

ARGUMENTE

***unsigned char** retValue*
1 wenn der Taster gedrückt wurde, 0 wenn sie nicht gedrückt wurde.

4.2.2.6.9. playerButtonY

BESCHREIBUNG

Rückgabe der positiven Flanke einer Tastereingabe vom Taster „Y“.

Y kann folgende Werte haben.

- *Up*
- *Down*
- *Left*
- *Right*
- *Fire*
- *Menu*

SYNTAX

***unsigned char** playerButtonFire(**unsigned char** playerID)*

ARGUMENTE

unsigned char *playerID*
Der Spieler, der geprüft werden sollte.

unsigned char *retValue*
1 wenn der Taster gedrückt wurde, 0 wenn sie nicht gedrückt wurde.

4.2.2.6.10. cls

BESCHREIBUNG

Löscht alles, was auf dem Display vorhanden ist. Wird erst ausgeführt, wenn send() aufgerufen wird.

SYNTAX

void *cls(* ***void*** *)*

ARGUMENTE

keine.

4.2.2.6.11. dot

BESCHREIBUNG

Zeichnet einen Punkt mit einer Farbe auf dem Display. Wird erst ausgeführt, wenn send() aufgerufen wird.

SYNTAX

void *dot(* ***unsigned char***** x,* ***unsigned char***** y,* ***const unsigned short***** rgb);*

ARGUMENTE

unsigned char** x*
Die X-Koordinate des Punktes

unsigned char** y*
Die Y-Koordinate des Punktes

unsigned short** rgb*
Die Farbe des Punktes

4.2.2.6.12. blendColourBox

BESCHREIBUNG

Zeichnet ein leeres Rechteck von x1, y1 zu x2, y2. Für jede Ecke kann eine Farbe angegeben, welche zu den anderen Ecken verlauft.

SYNTAX

```
void blendColourBox(  unsigned char* x1,
                      unsigned char* y1,
                      unsigned char* x2,
                      unsigned char* y2,
                      const unsigned short* topLeftColour,
                      const unsigned short* bottomLeftColour,
                      const unsigned short* topRightColour,
                      const unsigned short* bottomRightColour
                      );
```

ARGUMENTE

unsigned char* *x1*
Die X-Koordinate der ersten Ecke des Rechtecks

unsigned char* *y1*
Die Y-Koordinate der zweiten Ecke des Rechtecks

unsigned char* *x2*
Die X-Koordinate der zweiten Ecke des Rechtecks

unsigned char* *y2*
Die Y-Koordinate der zweiten Ecke des Rechtecks

unsigned short* *topLeftColour*
Die Farbe der oberen linken Ecke

unsigned short* *bottomLeftColour*
Die Farbe der unteren linken Ecke

unsigned short* *topRightColour*
Die Farbe der oberen rechten Ecke

unsigned short* *bottomRightColour*
Die Farbe der unteren rechten Ecke

4.2.2.6.13. blendColourFillBox

BESCHREIBUNG

Zeichnet ein gefülltes Rechteck von x1, y1 zu x2, y2. Für jede Ecke kann eine Farbe angegeben, welche zu den anderen Ecken verlauft.

SYNTAX

```
void blendColourFillBox(      unsigned char* x1,
                              unsigned char* y1,
                              unsigned char* x2,
                              unsigned char* y2,
                              const unsigned short* topLeftColour,
                              const unsigned short* bottomLeftColour,
                              const unsigned short* topRightColour,
                              const unsigned short* bottomRightColour
                              );
```

ARGUMENTE

unsigned char x1*
Die X-Koordinate der ersten Ecke des Rechtecks

unsigned char y1*
Die Y-Koordinate der zweiten Ecke des Rechtecks

unsigned char x2*
Die X-Koordinate der zweiten Ecke des Rechtecks

unsigned char y2*
Die Y-Koordinate der zweiten Ecke des Rechtecks

unsigned short topLeftColour*
Die Farbe der oberen linken Ecke

unsigned short bottomLeftColour*
Die Farbe der unteren linken Ecke

unsigned short topRightColour*
Die Farbe der oberen rechten Ecke

unsigned short bottomRightColour*
Die Farbe der unteren rechten Ecke

4.2.2.6.14. box

BESCHREIBUNG

Zeichnet ein leeres Rechteck von x1, y1 zu x2, y2 mit einer Farbe.

SYNTAX

```
void box(      unsigned char* x1,
               unsigned char* y1,
               unsigned char* x2,
               unsigned char* y2,
               const unsigned short* colour,
               );
```

ARGUMENTE

unsigned char x1*

Die X-Koordinate der ersten Ecke des Rechtecks

unsigned char y1*
Die Y-Koordinate der zweiten Ecke des Rechtecks

unsigned char x2*
Die X-Koordinate der zweiten Ecke des Rechtecks

unsigned char y2*
Die Y-Koordinate der zweiten Ecke des Rechtecks

unsigned short colour*
Farbe des Rechtecks

4.2.2.6.15. fillBox

BESCHREIBUNG

Zeichnet ein gefülltes Rechteck von x1, y1 zu x2, y2 mit einer Farbe.

SYNTAX

```
void fillBox(  unsigned char* x1,
               unsigned char* y1,
               unsigned char* x2,
               unsigned char* y2,
               const unsigned short* colour
               );
```

ARGUMENTE

unsigned char x1*
Die X-Koordinate der ersten Ecke des Rechtecks

unsigned char y1*
Die Y-Koordinate der zweiten Ecke des Rechtecks

unsigned char x2*
Die X-Koordinate der zweiten Ecke des Rechtecks

unsigned char y2*
Die Y-Koordinate der zweiten Ecke des Rechtecks

unsigned short colour*
Die Farbe des Rechtecks

4.2.2.6.16. blendColourLine

BESCHREIBUNG

Zeichnet eine Linie von x1, y1 zu x2, y2. Es kann für die Anfangs- und Endposition jeweils eine Farbe angegeben werden, welche entlang der Linie ineinander verlaufen.

SYNTAX

```
void blendColourLine(  unsigned char* x1,
                       unsigned char* y1,
                       unsigned char* x2,
                       unsigned char* y2,
                       const unsigned short* colour1,
                       const unsigned short* colour2
                       );
```

ARGUMENTE

unsigned char x1*
Die X-Koordinate der Anfangsposition der Linie

unsigned char y1*
Die Y-Koordinate der Anfangsposition der Linie

unsigned char x2*
Die X-Koordinate der Endposition der Linie

unsigned char y2*
Die Y-Koordinate der Endposition der Linie

unsigned short colour1*
Die Farbe am Anfang der Linie

unsigned short colour2*
Die Farbe am Ende der Linie

4.2.2.6.17. line

BESCHREIBUNG

Zeichnet eine Linie von x1, y1 zu x2, y2 mit einer Farbe.

SYNTAX

```
void line(     unsigned char* x1,
               unsigned char* y1,
               unsigned char* x2,
               unsigned char* y2,
               const unsigned short* colour
               );
```

ARGUMENTE

unsigned char x1*
Die X-Koordinate der Anfangsposition der Linie

unsigned char y1*

Die Y-Koordinate der Anfangsposition der Linie

unsigned char x2*
Die X-Koordinate der Endposition der Linie

unsigned char y2*
Die Y-Koordinate der Endposition der Linie

unsigned short colour*
Die Farbe der Linie

4.2.2.6.18. circle

BESCHREIBUNG

Zeichnet den Umriss eines Kreises mit dem Mittelpunkt x, y und dem Radius „radius“ mit der Farbe „colour“.

SYNTAX

```
void circle(   unsigned char* x,
               unsigned char* y,
               unsigned char* radius,
               const unsigned short* colour
               );
```

ARGUMENTE

unsigned char x*
Die X-Koordinate des Mittelpunktes des Kreises

unsigned char y1*
Die Y-Koordinate des Mittelpunktes des Kreises

unsigned char radius*
Der Radius des Kreises

const unsigned short colour*
Die Farbe des Kreises

4.2.2.6.19. fillCircle

BESCHREIBUNG

Zeichnet einen gefüllten Kreis mit dem Mittelpunkt x, y und dem Radius „radius“ mit der Farbe „colour“.

SYNTAX

```
void fillCircle(      unsigned char* x,
                      unsigned char* y,
                      unsigned char* radius,
                      const unsigned short* colour
                      );
```

ARGUMENTE

unsigned char* *x*
Die X-Koordinate des Mittelpunktes des Kreises

unsigned char* *y1*
Die Y-Koordinate des Mittelpunktes des Kreises

unsigned char* *radius*
Der Radius des Kreises

const unsigned short* *colour*
Die Farbe des Kreises

4.2.2.6.20. blendColourFillCircle

BESCHREIBUNG

Zeichnet einen gefüllten Kreis mit den Mittelpunkt x, y und den Radius „radius“. Es wird eine innere Farbe und eine äussere Farbe des Kreises angegeben, welche ineinander verlaufen.

SYNTAX

```
void circle(   unsigned char* x,
               unsigned char* y,
               unsigned char* radius,
               const unsigned short* insideColour,
               const unsigned short* outsideColour
               );
```

ARGUMENTE

unsigned char* *x*
Die X-Koordinate des Mittelpunktes des Kreises

unsigned char* *y1*
Die Y-Koordinate des Mittelpunktes des Kreises

unsigned char* *radius*
Der Radius des Kreises

const unsigned short* *insideColour*
Die innere Farbe des Kreises

const unsigned short* *outsideColour*
Die äussere Farbe des Kreises

4.2.2.6.21. setBlendMode

BESCHREIBUNG

Setzt den „Verlaufs-Modus“ des Displays.

Mit dieser Funktion kann bestimmt werden, wie die schon vorhandenen Farben auf dem Display mit den neuen Farben verrechnet werden sollen. Dabei sind vier Moden vorhanden.

- ***Replace*** *– Die neuen Farben ersetzen die schon vorhandene Farben*
- ***Add*** *– Die neuen Farben werden zu den schon vorhandenen Farben addiert*
- ***Subtract*** *– Die neuen Farben werden von den vorhandenen Farben subtrahiert*
- ***Multiply*** *– Die neuen Farben werden mit den vorhandenen Farben multipliziert*

Der eingestellte Modus ***bleibt gültig*** *bis es wieder geändert wird.*

Der Standard-modus ist ***Replace****.*

SYNTAX

```
void setBlendMode( unsigned char blendMode );
```

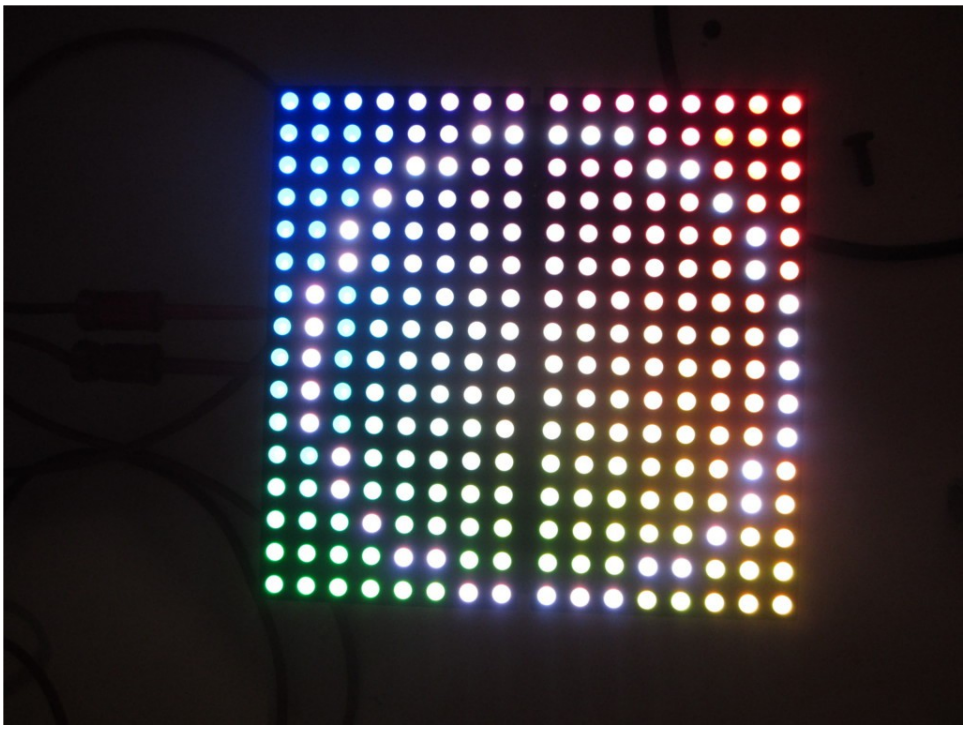
ARGUMENTE

unsigned char *blendMode*
Der Modus zum einstellen. Ein Enumerator wurde deklariert, und es können direkt die folgende Moden als Argument übergeben werden.

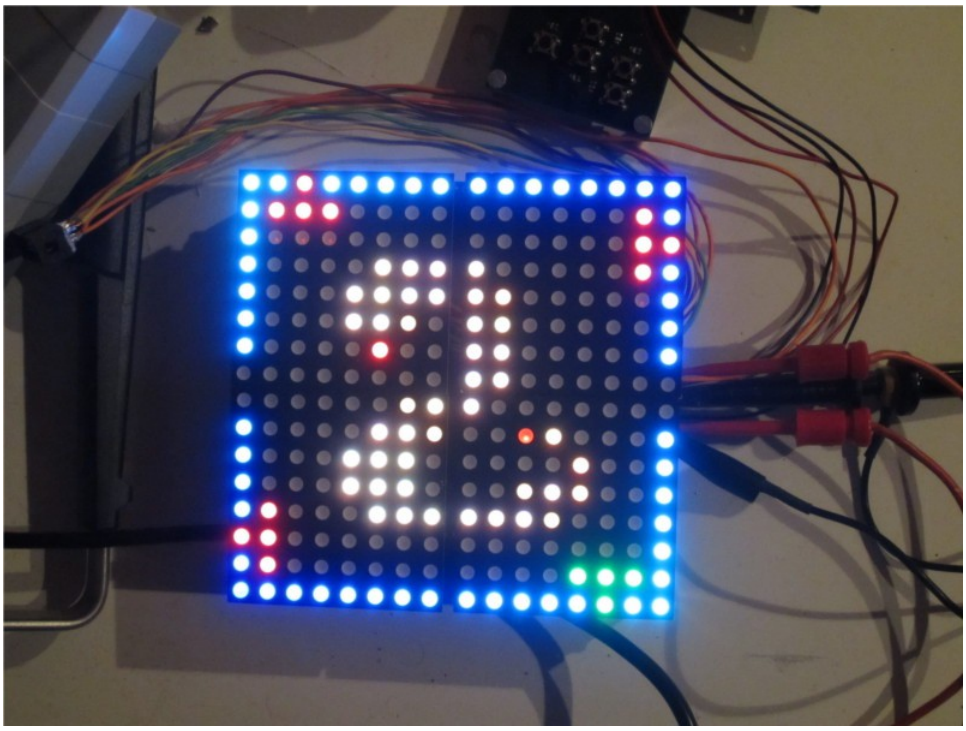
- *BLEND_MODE_REPLACE*
- *BLEND_MODE_ADD*
- *BLEND_MODE_SUBTRACT*
- *BLEND_MODE_MULTIPLY*

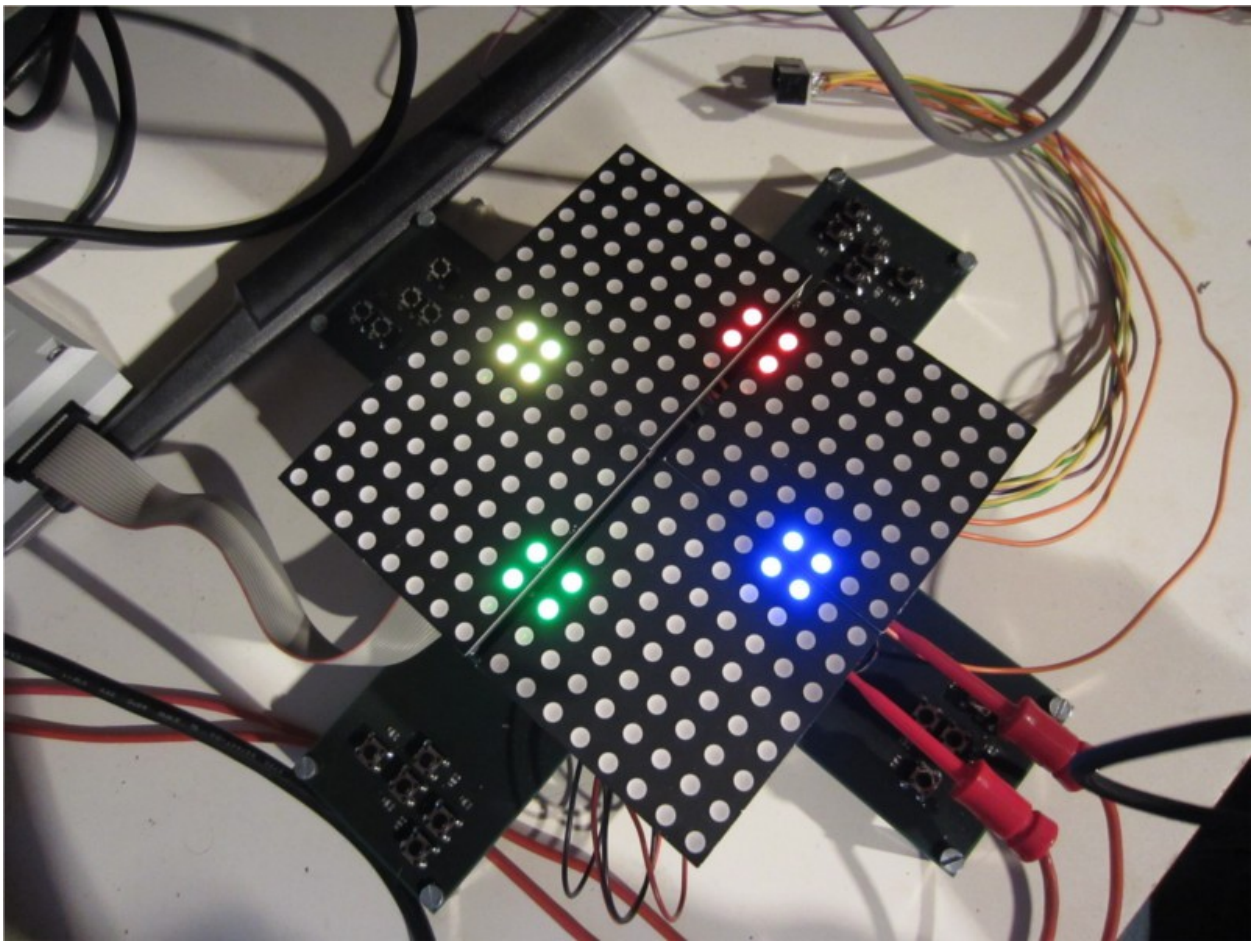
5. Das Produkt

Das Display sieht aufgebaut wie folgt aus.



Das Hauptmenü zeigt hier eine Vorschau vom Spiel „Snake“.





6. Abschluss

6.1. Schlussfolgerung

Das Projektziel ist mehr als erreicht worden. Es können alle LEDs effizient und einfach angesteuert werden, und das ganze ist sehr dynamisch und erweiterbar aufgebaut.

UART Kommunikation war keine schlechte Idee, jedoch brauchte die Dekodierung und Enkodierung der Datenpakete zu lange. Vielleicht wäre es besser

6.2. Reflexion

6.2.1. Alex Murray

Diese Arbeit hat mir viel Spass, sowohl auch Ärger bereitet. Ich habe viel dabei gelernt, von UART Kommunikation bis zu Projektmanagement.

Die Dokumentation hat viel mehr Zeit in Anspruch genommen, als ich mir vorgestellt hatte. Nächstes Mal werde ich mehr Zeit dafür einplanen.

Wie immer, hat sich Murphy in allem einschleichen können. Der Mikrocontroller ist 2 mal wegen einem Fehler beim Layout abgeraucht, und musste per Hand ersetzt werden (ich werde in Zukunft keine QFN-64 Gehäuse mehr einsetzen). Der JTAG-Stecker verursachte nach Wochenlangen, fehlerfreien Betrieb einen Kurzschluss, und verdampfte dabei auch gleich eine Versorgungsleitung, was zu vieles „Gebasteln“ mit dünnen Drähten und Epoxy-Leim führte. Ich hinterliess das Projekt auf einen Zug, der nach Olten fuhr, und holte mir eine Busse von 140.- ein, weil ich vergessen hatte, ein Billett zwischen Olten und Tecknau zu lösen (das U-ABO zählt ja nicht mehr). Ausserdem sollte man beim Programmieren keinen Kaffee trinken, weil es sonst über die Tastatur spült und den Quellcode mit unbrauchbaren Zeichen überschwemmt.

Alles in Allem aber war es erfreulich. Auch die nicht-erfreulichen Erlebnisse waren rückblickend erfreulich, denn die gehören einfach dazu.

6.2.2. Marcel Kaltenrieder

Die Projektarbeit war sehr lehrreich. Es hat Spass gemacht, das Projekt zu planen und das geplante in die Tat umzusetzen. Im allgemeinen bin ich mit dem Resultat zufrieden.

6.3. Danksagung

Wir möchten uns herzlich bei Herrn Wenk für die tolle Schulzeit bedanken.

Weiter möchten wir uns bei den vielen Fachvorgesetzten, Lehrer, und Kollegen bedanken, die uns geholfen haben: Bernhard Keser, Peter Häner, Dario Ferraro, Frank Wandeler, Aurele Fleury, und David Zingg.

Zum Schluss möchten wir uns bei der Firma Endress+Hauser Flowtec AG für die Unterstützung und Ressourcen für dieses Projekt bedanken.

6.4. Authentizitätserklärung

Playtrix, 2013

Wir erklären hiermit, dass wir diese Projektarbeit selbstständig verfasst haben. Fremde Hilfsmittel oder Hilfe durch Fachpersonen wurden vermerkt.

7. Anhang

7.1. Schema

7.2. Quellcode

Im Quellcode vom Base sind nicht alle Module vorhanden. Es wurden nur die fertigen Module in die Dokumentation kopiert.

7.2.1. Base

7.2.1.1. main.h

```
// ------------------------------------------------------------------
// Include files
// ------------------------------------------------------------------

#ifndef _MAIN_H_
	#define _MAIN_H_

// ------------------------------------------------------------------
// Include files
// ------------------------------------------------------------------

#include "common.h"
#include "init.h"
#include "uart.h"
#include "framework.h"

#endif // _MAIN_H_
```

7.2.1.2. main.c

```
// -----------------------------------------------------------------------
// LED Matrix Base
// -----------------------------------------------------------------------
// Programmed by      : Alex Murray
//                      Marcel Kaltenrieder
// -----------------------------------------------------------------------
// This program interfaces with the LED matrix module
// -----------------------------------------------------------------------

/*
-------------------------------------------------------------------------
   Pin Layout
-------------------------------------------------------------------------


                                 _________________________
                                |     MSP430F2418         |
                                |                         |
  IN/D   Player[0].btn[0] ------| P1.0             P4.0 |------ Player[3].btn[0]      IN/D
  IN/D   Player[0].btn[1] ------| P1.1             P4.1 |------ Player[3].btn[1]      IN/D
  IN/D   Player[0].btn[2] ------| P1.2             P4.2 |------ Player[3].btn[2]      IN/D
  IN/D   Player[0].btn[3] ------| P1.3             P4.3 |------ Player[3].btn[3]      IN/D
  IN/D   Player[0].btn[4] ------| P1.4             P4.4 |------ Player[3].btn[4]      IN/D
                           ------| P1.5             P4.5 |------
                           ------| P1.6             P4.6 |------
                           ------| P1.7             P4.7 |------
                                 |                         |
  IN/D   Player[1].btn[0] ------| P2.0             P5.0 |------
  IN/D   Player[1].btn[1] ------| P2.1             P5.1 |------
  IN/D   Player[1].btn[2] ------| P2.2             P5.2 |------
  IN/D   Player[1].btn[3] ------| P2.3             P5.3 |------
  IN/D   Player[1].btn[4] ------| P2.4             P5.4 |------
                           ------| P2.5             P5.5 |------
                           ------| P2.6             P5.6 |------
                           ------| P2.7             P5.7 |------
                                 |                         |
  IN/D   Player[1].btn[0] ------| P3.0             P6.0 |------
  IN/D   Player[1].btn[1] ------| P3.1             P6.1 |------
  IN/D   Player[1].btn[2] ------| P3.2             P6.2 |------
  IN/D   Player[1].btn[3] ------| P3.3             P6.3 |------
  IN/D   Player[1].btn[4] ------| P3.4             P6.4 |------
                           ------| P3.5             P6.5 |------
                     TxD   ------| P3.6             P6.6 |------
                     RxD   ------| P3.7             P6.7 |------
                                 |_________________________|


-------------------------------------------------------------------------
   Pin description
-------------------------------------------------------------------------

     Pin Name            | Description
   ----------------------+-----------------------------------------------------
     TxD                 | For serial communication
     RxD                 |
   ----------------------+-----------------------------------------------------
     Player[].btn[]      | Input buttons for each player
   ----------------------+-----------------------------------------------------
```

```
*/

// ----------------------------------------------------------------------
// Include files
// ----------------------------------------------------------------------

// main header
#include "main.h"

// ----------------------------------------------------------------------
// global variables & arrays
// ----------------------------------------------------------------------

// ----------------------------------------------------------------------
// Main entry point
// ----------------------------------------------------------------------

void main( void )
{

        // Initialise device
        initDevice();

        // start framework
        startFrameWork();
}
```

7.2.1.3. common.h

```
// ------------------------------------------------------------------------
// Common declarations
// ------------------------------------------------------------------------

#ifndef _COMMON_H_
        #define _COMMON_H_

// ------------------------------------------------------------------------
// include files
// ------------------------------------------------------------------------

#include "msp430f2418.h"

// ------------------------------------------------------------------------
// definitions
// ------------------------------------------------------------------------

// map player ports
#define MAP_PLAYER1_BUTTON          (~P1IN)&0x1F
#define MAP_PLAYER3_BUTTON          (~P2IN)&0x1F
#define MAP_PLAYER2_BUTTON          (~P3IN)&0x1F
#define MAP_PLAYER4_BUTTON          (~P4IN)&0x1F

// map virtual buttons
#define MAP_PLAYER_BUTTON_MENU      0x20
#define MAP_PLAYER_BUTTON_CLEAR     0x40

// map player buttons
#define MAP_PLAYER1_BUTTON_FIRE     0x01
#define MAP_PLAYER1_BUTTON_RIGHT    0x02
#define MAP_PLAYER1_BUTTON_DOWN     0x04
#define MAP_PLAYER1_BUTTON_LEFT     0x08
#define MAP_PLAYER1_BUTTON_UP       0x10

#define MAP_PLAYER2_BUTTON_FIRE     0x10
#define MAP_PLAYER2_BUTTON_RIGHT    0x08
#define MAP_PLAYER2_BUTTON_DOWN     0x04
#define MAP_PLAYER2_BUTTON_LEFT     0x02
#define MAP_PLAYER2_BUTTON_UP       0x01

#define MAP_PLAYER3_BUTTON_FIRE     0x10
#define MAP_PLAYER3_BUTTON_RIGHT    0x08
#define MAP_PLAYER3_BUTTON_DOWN     0x04
#define MAP_PLAYER3_BUTTON_LEFT     0x02
#define MAP_PLAYER3_BUTTON_UP       0x01

#define MAP_PLAYER4_BUTTON_FIRE     0x10
#define MAP_PLAYER4_BUTTON_RIGHT    0x08
#define MAP_PLAYER4_BUTTON_DOWN     0x04
#define MAP_PLAYER4_BUTTON_LEFT     0x02
#define MAP_PLAYER4_BUTTON_UP       0x01

// ------------------------------------------------------------------------
// global variables
// ------------------------------------------------------------------------

#endif // _COMMON_H_
```

7.2.1.4. init.h

```
// -------------------------------------------------------------------------
// Initialisations
// -------------------------------------------------------------------------

#ifndef _INIT_H_
        #define _INIT_H_

void initDevice( void );
void cfgPort1( void );
void cfgPort2( void );
void cfgPort3( void );
void cfgPort4( void );
void cfgPort5( void );
void cfgPort6( void );
void cfgSystemClock( void );
void cfgTimerA( void );
void cfgUART( void );

#endif // _INIT_H_
```

7.2.1.5. init.c

```
// -------------------------------------------------------------------------
// Initialisations
// -------------------------------------------------------------------------

// header files
#include "init.h"
#include "uart.h"
#include "framework.h"
#include "common.h"
#include "menu.h"

//
------------------------------------------------------------------------------------------------
----------------------
// call this to initialise the device
void initDevice( void )
{

	// Stop watchdog timer to prevent time out reset
	WDTCTL = WDTPW + WDTHOLD;

	// setup clock
	cfgSystemClock();

	// wait, so other micro controller definately starts up before we start sending data
	__delay_cycles( 300000 );

	// configure timers
	cfgTimerA();

	// configure ports
	cfgPort1();
	cfgPort2();
	cfgPort3();
	cfgPort4();
	cfgPort5();
	cfgPort6();

	// configure UART serial interface
	cfgUART();

	// initial values
	initUART();
	initFrameWork();
	initMenu();

	// enable global interrupts
	__bis_SR_register( GIE );
}

void cfgPort1( void )
{
	P1DIR = ~0x1F;
	P1SEL = 0x00;
}

void cfgPort2( void )
{
```

```
        P2DIR = ~0x1F;
        P2SEL = 0x00;
}

void cfgPort3( void )
{
        P3DIR = ~0x1F;
        P3SEL = 0xC0;
}

void cfgPort4( void )
{
        P4DIR = ~0x1F;
        P4DIR = 0x00;
}

void cfgPort5( void )
{
}

void cfgPort6( void )
{
}

void cfgSystemClock( void )
{

        // setup external clock (14.74560 MHz)
        BCSCTL1 &= ~0x80;                   // Turn on XT2 oscillator
        BCSCTL3 |= 0x80;                    // Select range 3-16 MHz
        BCSCTL2 |= 0x08;                    // select XT2CLK
        BCSCTL2 &= ~0x06;                   // divider to 1
}

void cfgTimerA( void )
{
        CCTL0 = CCIE;                           // CCR0 interrupt enabled
        CCR0 = 50000;
        TACTL = TASSEL_2 + MC_3 + ID_3;         // SMCLK, upmode, divide by 8
}

void cfgUART( void )
{

        // configure UART
        UCA1CTL1 |= UCSWRST;                // **Put state machine in reset**
        UCA1CTL1 |= 0xC0;                   // Select SMCLK for BRCLK
        UCA1BR0 = 0x04;                             // 14.7456 MHz divided by 0x04 = Baud
3'686'400
        UCA1BR1 = 0x00;
        UCA1CTL1 &= ~UCSWRST;               // **Initialise USCI state machine**

        // enable interrupts
        UC1IE |= UCA1RXIE;                  // Enable receive interrupt

}
```

7.2.1.6. moduleenable.h

```
// ------------------------------------------------------------------------
// Module Enable File
// ------------------------------------------------------------------------

#ifndef _MODULEENABLE_H_
	#define _MODULEENABLE_H_

#define MAX_MODULES 12

// ------------------------------------------------------------------------
// Comment in the modules you would like enabled
// ------------------------------------------------------------------------

#define MODULE_ENABLE_COLOUR_DEMO
#define MODULE_ENABLE_SNAKE
#define MODULE_ENABLE_GAME_OF_LIFE
#define MODULE_ENABLE_TRON
#define MODULE_ENABLE_TETRIS
#define MODULE_ENABLE_SPACE_INVADERS
#define MODULE_ENABLE_PONG
#define MODULE_ENABLE_BURGLER
#define MODULE_ENABLE_CAT_AND_MOUSE

// ------------------------------------------------------------------------
// Will count the commented modules so the menu knows
// ------------------------------------------------------------------------

unsigned char getModuleCount( void );

#endif // _MODULEENABLE_H_
```

7.2.1.7. moduleenable.c

```
// ------------------------------------------------------------------------
// Counts how many modules were enabled
// ------------------------------------------------------------------------

// ------------------------------------------------------------------------
// Include Files
// ------------------------------------------------------------------------

#include "moduleenable.h"

// ------------------------------------------------------------------------
// counts modules enabled
unsigned char getModuleCount( void )
{
	unsigned char moduleCount = 0;
#ifdef MODULE_ENABLE_COLOUR_DEMO
	moduleCount++;
#endif
#ifdef MODULE_ENABLE_SNAKE
	moduleCount++;
#endif
#ifdef MODULE_ENABLE_GAME_OF_LIFE
	moduleCount++;
#endif
#ifdef MODULE_ENABLE_TRON
	moduleCount++;
#endif
#ifdef MODULE_ENABLE_SPACE_INVADERS
	moduleCount++;
#endif
#ifdef MODULE_ENABLE_TETRIS
	moduleCount++;
#endif
#ifdef MODULE_ENABLE_PONG
	moduleCount++;
#endif
#ifdef MODULE_ENABLE_BURGLER
	moduleCount++;
#endif
#ifdef MODULE_ENABLE_CAT_AND_MOUSE
	moduleCount++;
#endif
	return moduleCount-1;
}
```

7.2.1.8. common.h

```
// ------------------------------------------------------------------------
// Common declarations
// ------------------------------------------------------------------------

#ifndef _COMMON_H_
	#define _COMMON_H_

// ------------------------------------------------------------------------
// include files
// ------------------------------------------------------------------------

#include "msp430f2418.h"

// ------------------------------------------------------------------------
// definitions
// ------------------------------------------------------------------------

// map player ports
#define MAP_PLAYER1_BUTTON          (~P1IN)&0x1F
#define MAP_PLAYER3_BUTTON          (~P2IN)&0x1F
#define MAP_PLAYER2_BUTTON          (~P3IN)&0x1F
#define MAP_PLAYER4_BUTTON          (~P4IN)&0x1F

// map virtual buttons
#define MAP_PLAYER_BUTTON_MENU     0x20
#define MAP_PLAYER_BUTTON_CLEAR    0x40

// map player buttons
#define MAP_PLAYER1_BUTTON_FIRE    0x01
#define MAP_PLAYER1_BUTTON_RIGHT   0x02
#define MAP_PLAYER1_BUTTON_DOWN    0x04
#define MAP_PLAYER1_BUTTON_LEFT    0x08
#define MAP_PLAYER1_BUTTON_UP      0x10

#define MAP_PLAYER2_BUTTON_FIRE    0x10
#define MAP_PLAYER2_BUTTON_RIGHT   0x08
#define MAP_PLAYER2_BUTTON_DOWN    0x04
#define MAP_PLAYER2_BUTTON_LEFT    0x02
#define MAP_PLAYER2_BUTTON_UP      0x01

#define MAP_PLAYER3_BUTTON_FIRE    0x10
#define MAP_PLAYER3_BUTTON_RIGHT   0x08
#define MAP_PLAYER3_BUTTON_DOWN    0x04
#define MAP_PLAYER3_BUTTON_LEFT    0x02
#define MAP_PLAYER3_BUTTON_UP      0x01

#define MAP_PLAYER4_BUTTON_FIRE    0x10
#define MAP_PLAYER4_BUTTON_RIGHT   0x08
#define MAP_PLAYER4_BUTTON_DOWN    0x04
#define MAP_PLAYER4_BUTTON_LEFT    0x02
#define MAP_PLAYER4_BUTTON_UP      0x01

// ------------------------------------------------------------------------
// global variables
// ------------------------------------------------------------------------

#endif // _COMMON_H_
```

7.2.1.9. uart.h

```
// ------------------------------------------------------------------------
// Serial communication
// ------------------------------------------------------------------------

#ifndef _UART_H_
	#define _UART_H_

// ------------------------------------------------------------------------
// Definitions
// ------------------------------------------------------------------------

#define UART_BUFFER_SIZE 1024

// ------------------------------------------------------------------------
// Structs
// ------------------------------------------------------------------------

// structs
struct UART_t
{
	volatile unsigned short bufferReadPtr;
	volatile unsigned short bufferWritePtr;
	volatile unsigned char buffer[ UART_BUFFER_SIZE ];
	volatile unsigned char isSending;
	volatile unsigned char timeOut;
};

// ------------------------------------------------------------------------
// Enumerators
// ------------------------------------------------------------------------

// blend modes
enum blendMode_e
{
	BLEND_MODE_REPLACE,
	BLEND_MODE_ADD,
	BLEND_MODE_SUBTRACT,
	BLEND_MODE_MULTIPLY
};

// command list
enum commandList_e
{
	CMD_CLS,
	CMD_DOT,
	CMD_BLEND_COLOUR_BOX,
	CMD_BLEND_COLOUR_FILL_BOX,
	CMD_BOX,
	CMD_FILL_BOX,
	CMD_BLEND_COLOUR_LINE,
	CMD_LINE,
	CMD_CIRCLE,
	CMD_FILL_CIRCLE,
	CMD_BLEND_COLOUR_FILL_CIRCLE,

	CMD_SET_BLEND_MODE__REPLACE,
	CMD_SET_BLEND_MODE__ADD,
	CMD_SET_BLEND_MODE__SUBTRACT,
	CMD_SET_BLEND_MODE__MULTIPLY
```

```
};

// ------------------------------------------------------------------------
// Function Prototypes
// ------------------------------------------------------------------------

void initUART( void );
unsigned char _buffer_overflow( void );
void _increase_buffer_pointer( volatile unsigned short* ptr );
void _write_to_buffer( unsigned char* data );
void UARTUpdateTimeOut( void );
void send( void );
void cls( void );
void dot( unsigned char* x, unsigned char* y, const unsigned short* rgb );
void blendColourBox( unsigned char* x1, unsigned char* y1, unsigned char* x2, unsigned char*
y2, const unsigned short* topLeftColour, const unsigned short* bottomLeftColour, const
unsigned short* topRightColour, const unsigned short* bottomRightColour );
void blendColourFillBox( unsigned char* x1, unsigned char* y1, unsigned char* x2, unsigned
char* y2, const unsigned short* topLeftColour, const unsigned short* bottomLeftColour, const
unsigned short* topRightColour, const unsigned short* bottomRightColour );
void box( unsigned char* x1, unsigned char* y1, unsigned char* x2, unsigned char* y2, const
unsigned short* colour );
void fillBox( unsigned char* x1, unsigned char* y1, unsigned char* x2, unsigned char* y2,
const unsigned short* colour );
void blendColourLine( unsigned char* x1, unsigned char* y1, unsigned char* x2, unsigned
char* y2, const unsigned short* colour1, const unsigned short* colour2 );
void line( unsigned char* x1, unsigned char* y1, unsigned char* x2, unsigned char* y2, const
unsigned short* colour );
void circle( unsigned char* x, unsigned char* y, unsigned char* radius, const unsigned
short* colour );
void fillCircle( unsigned char* x, unsigned char* y, unsigned char* radius, const unsigned
short* colour );
void blendColourFillCircle( unsigned char* x, unsigned char* y, unsigned char* radius, const
unsigned short* insideColour, const unsigned short* outsideColour );
void setBlendMode( unsigned char blendMode );

#endif // _UART_H_
```

7.2.1.10. uart.c

```
// ------------------------------------------------------------------------
// Serial communication
// ------------------------------------------------------------------------

// header files
#include "uart.h"
#include "common.h"

// structs
static struct UART_t UART;

// ------------------------------------------------------------------------
// initialise UART
void initUART( void )
{
	UART.bufferWritePtr = 0;
	UART.bufferReadPtr = 0;
	UART.isSending = 0;
	UART.timeOut = 0;
}

// ------------------------------------------------------------------------
// returns 1 on buffer overflow
unsigned char _buffer_overflow( void )
{

	// cache next write pointer, because we're using it more than once
	unsigned short ptr = UART.bufferWritePtr+1;

	// check if we're exceeding buffer size
	if( ptr != UART_BUFFER_SIZE )
	{

		// if write pointer is equal to read pointer, buffer has overflown
		if( ptr == UART.bufferReadPtr ) return 1;
	}else{

		// if after wrapping the pointer back to 0 it equals the read pointer, buffer
has overflown
		if( UART.bufferReadPtr ){}else{ return 1; }
	}
	return 0;
}

// ------------------------------------------------------------------------
// increases a buffer pointer by 1 and wraps
void _increase_buffer_pointer( volatile unsigned short* ptr )
{
	(*ptr)++;
	if( (*ptr) == UART_BUFFER_SIZE )
		(*ptr) = 0;
}

// ------------------------------------------------------------------------
// writes data into the buffer
void _write_to_buffer( unsigned char* data )
{
```

```
        // overflow? force sending
        while( _buffer_overflow() == 1 )
        {
                send();
        }

        // write to buffer
        volatile unsigned char* ptr = UART.buffer;
        ptr += UART.bufferWritePtr;
        *ptr = *data;

        // increase pointer
        _increase_buffer_pointer( &UART.bufferWritePtr );

        // return
        return;
}

// -------------------------------------------------------------------------
// called by the framework's interrupt routine
// this is used to determine if the display is taking too long to reply
void UARTUpdateTimeOut( void )
{

        // check if timeout has occured
        UART.timeOut++;
        if( UART.timeOut == 32 )
        {
                UART.timeOut = 0;

                // check if there is still data to be sent
                unsigned short temp = UART.bufferReadPtr;
                temp = ( temp != UART.bufferWritePtr );
                if( UART.isSending == 0 && temp )
                {

                        // force resend
                        UART.isSending = 1;
                        UCA1TXBUF = UART.buffer[ UART.bufferReadPtr ];

                }
        }
}

// -------------------------------------------------------------------------
// initiates sending the buffer - this causes a chain reaction
// which lasts until the buffer is empty
void send( void )
{
        unsigned short temp = UART.bufferReadPtr;
        temp = ( temp != UART.bufferWritePtr );
        if( UART.isSending == 0 && temp )
        {
                UART.isSending = 1;
                UART.timeOut = 0;
                UCA1TXBUF = UART.buffer[ UART.bufferReadPtr ];
        }
}

// -------------------------------------------------------------------------
// clear screen
void cls( void )
```

```
{

	// write clear command
	unsigned char n;
	n = CMD_CLS;		_write_to_buffer( &n );

	// return
	return;

}

// ------------------------------------------------------------------------
// dot
void dot( unsigned char* x, unsigned char* y, const unsigned short* rgb )
{

	// write dot command to buffer
	unsigned char n;
	n = CMD_DOT;		_write_to_buffer( &n );
	n = (((*x)<<4)|(*y)); _write_to_buffer( &n );
	n = ((*rgb)>>4);	_write_to_buffer( &n );
	n = ((*rgb)<<4);	_write_to_buffer( &n );

	// return
	return;
}

// ------------------------------------------------------------------------
// blend colour box
void blendColourBox( unsigned char* x1, unsigned char* y1, unsigned char* x2, unsigned char*
y2, const unsigned short* topLeftColour, const unsigned short* bottomLeftColour, const
unsigned short* topRightColour, const unsigned short* bottomRightColour )
{

	// write command to buffer
	unsigned char n;
	n = CMD_BLEND_COLOUR_BOX;				_write_to_buffer( &n );
	n = (((*x1)<<4)|(*y1));					_write_to_buffer( &n );
	n = (((*x2)<<4)|(*y2));					_write_to_buffer( &n );
	n = ((*topLeftColour)>>4);				_write_to_buffer( &n );
	n = ((*topLeftColour)<<4)|((*bottomLeftColour)>>8);	_write_to_buffer( &n );
	n = (*bottomLeftColour);				_write_to_buffer( &n );
	n = ((*topRightColour)>>4);				_write_to_buffer( &n );
	n = ((*topRightColour)<<4)|((*bottomRightColour)>>8);	_write_to_buffer( &n );
	n = (*bottomRightColour);				_write_to_buffer( &n );

	// return
	return;
}

// ------------------------------------------------------------------------
// blend colour fill box
void blendColourFillBox( unsigned char* x1, unsigned char* y1, unsigned char* x2, unsigned
char* y2, const unsigned short* topLeftColour, const unsigned short* bottomLeftColour, const
unsigned short* topRightColour, const unsigned short* bottomRightColour )
{

	// write command to buffer
	unsigned char n;
	n = CMD_BLEND_COLOUR_FILL_BOX;				_write_to_buffer( &n );
	n = (((*x1)<<4)|(*y1));					_write_to_buffer( &n );
	n = (((*x2)<<4)|(*y2));					_write_to_buffer( &n );
```

```
        n = ((*topLeftColour)>>4);                                  _write_to_buffer( &n );
        n = ((*topLeftColour)<<4)|((*bottomLeftColour)>>8);         _write_to_buffer( &n );
        n = (*bottomLeftColour);                                    _write_to_buffer( &n );
        n = ((*topRightColour)>>4);                                 _write_to_buffer( &n );
        n = ((*topRightColour)<<4)|((*bottomRightColour)>>8);       _write_to_buffer( &n );
        n = (*bottomRightColour);                                   _write_to_buffer( &n );

        // return
        return;
}

// ------------------------------------------------------------------------
// box
void box( unsigned char* x1, unsigned char* y1, unsigned char* x2, unsigned char* y2, const
unsigned short* colour )
{

        // write command to buffer
        unsigned char n;
        n = CMD_BOX;                                                _write_to_buffer( &n );
        n = (((*x1)<<4)|(*y1));                                     _write_to_buffer( &n );
        n = (((*x2)<<4)|(*y2));                                     _write_to_buffer( &n );
        n = ((*colour)>>4);                                         _write_to_buffer( &n );
        n = ((*colour)<<4);                                         _write_to_buffer( &n );

        // return
        return;
}

// ------------------------------------------------------------------------
// fill box
void fillBox( unsigned char* x1, unsigned char* y1, unsigned char* x2, unsigned char* y2,
const unsigned short* colour )
{

        // write command to buffer
        unsigned char n;
        n = CMD_FILL_BOX;                                           _write_to_buffer( &n );
        n = (((*x1)<<4)|(*y1));                                     _write_to_buffer( &n );
        n = (((*x2)<<4)|(*y2));                                     _write_to_buffer( &n );
        n = ((*colour)>>4);                                         _write_to_buffer( &n );
        n = ((*colour)<<4);                                         _write_to_buffer( &n );

        // return
        return;
}

// ------------------------------------------------------------------------
// blend colour line
void blendColourLine( unsigned char* x1, unsigned char* y1, unsigned char* x2, unsigned
char* y2, const unsigned short* colour1, const unsigned short* colour2 )
{

        // write command to buffer
        unsigned char n;
        n = CMD_BLEND_COLOUR_LINE;                                  _write_to_buffer( &n );
        n = (((*x1)<<4)|(*y1));                                     _write_to_buffer( &n );
        n = (((*x2)<<4)|(*y2));                                     _write_to_buffer( &n );
        n = ((*colour1)>>4);                                        _write_to_buffer( &n );
        n = (((*colour1)<<4)|((*colour2)>>8));                      _write_to_buffer( &n );
        n = (*colour2);                                             _write_to_buffer( &n );
```

```
	// return
	return;
}

// ------------------------------------------------------------------------
// line
void line( unsigned char* x1, unsigned char* y1, unsigned char* x2, unsigned char* y2, const
unsigned short* colour )
{

	// write command to buffer
	unsigned char n;
	n = CMD_LINE;                                          _write_to_buffer( &n );
	n = (((*x1)<<4)|(*y1));                                _write_to_buffer( &n );
	n = (((*x2)<<4)|(*y2));                                _write_to_buffer( &n );
	n = ((*colour)>>4);                                    _write_to_buffer( &n );
	n = ((*colour)<<4);                                    _write_to_buffer( &n );

	// return
	return;
}

// ------------------------------------------------------------------------
// circle
void circle( unsigned char* x, unsigned char* y, unsigned char* radius, const unsigned
short* colour )
{

	// write command to buffer
	unsigned char n;
	n = CMD_CIRCLE;                                        _write_to_buffer( &n );
	n = (((*x)<<4)|(*y));                                  _write_to_buffer( &n );
	n = (*radius);                                         _write_to_buffer( &n );
	n = ((*colour)>>4);                                    _write_to_buffer( &n );
	n = ((*colour)<<4);                                    _write_to_buffer( &n );

	// return
	return;
}

// ------------------------------------------------------------------------
// fill circle
void fillCircle( unsigned char* x, unsigned char* y, unsigned char* radius, const unsigned
short* colour )
{

	// write command to buffer
	unsigned char n;
	n = CMD_FILL_CIRCLE;                                   _write_to_buffer( &n );
	n = (((*x)<<4)|(*y));                                  _write_to_buffer( &n );
	n = (*radius);                                         _write_to_buffer( &n );
	n = ((*colour)>>4);                                    _write_to_buffer( &n );
	n = ((*colour)<<4);                                    _write_to_buffer( &n );

	// return
	return;
}

// ------------------------------------------------------------------------
// blend colour fill circle
void blendColourFillCircle( unsigned char* x, unsigned char* y, unsigned char* radius, const
unsigned short* insideColour, const unsigned short* outsideColour )
```

```
{

	// write command to buffer
	unsigned char n;
	n = CMD_BLEND_COLOUR_FILL_CIRCLE;                       _write_to_buffer( &n );
	n = (((*x)<<4)|(*y));                                   _write_to_buffer( &n );
	n = (*radius);                                          _write_to_buffer( &n );
	n = ((*insideColour)>>4);                               _write_to_buffer( &n );
	n = (((*insideColour)<<4)|((*outsideColour)>>8));       _write_to_buffer( &n );
	n = (*outsideColour);                                   _write_to_buffer( &n );

	// return
	return;
}

// ------------------------------------------------------------------------
// set blend mode
void setBlendMode( unsigned char blendMode )
{

	// determine blend mode
	unsigned char n;
	switch( blendMode )
	{
		case BLEND_MODE_REPLACE  : n = CMD_SET_BLEND_MODE__REPLACE;  break;
		case BLEND_MODE_ADD      : n = CMD_SET_BLEND_MODE__ADD;      break;
		case BLEND_MODE_SUBTRACT : n = CMD_SET_BLEND_MODE__SUBTRACT; break;
		case BLEND_MODE_MULTIPLY : n = CMD_SET_BLEND_MODE__MULTIPLY; break;
		default: return;
	}

	// write command to buffer
	_write_to_buffer( &n );

	// return
	return;
}

// ------------------------------------------------------------------------
// RxD interrupt
#pragma vector=USCIAB1RX_VECTOR
__interrupt void USCI1RX_ISR(void)
{

	// increase and wrap pointer if it's the same as sent data
	volatile unsigned char* temp = UART.buffer;
	temp += UART.bufferReadPtr;
	unsigned char buf = UCA1RXBUF;
	if( *temp == buf )
	{
		_increase_buffer_pointer( &UART.bufferReadPtr );
	}

	// send next block of data, if any
	unsigned short tempWritePtr = UART.bufferWritePtr;
	if( tempWritePtr != UART.bufferReadPtr )
	{

		// send data
		UART.timeOut = 0;
		UCA1TXBUF = UART.buffer[ UART.bufferReadPtr ];
```

```
            // buffer is empty
            }else{
                    UART.isSending = 0;
            }
    }
```

7.2.1.11. framework.h

```
// ------------------------------------------------------------------------
// Framework
// ------------------------------------------------------------------------

#ifndef _FRAMEWORK_H_
	#define _FRAMEWORK_H_

#include "gameenable.h"

// ------------------------------------------------------------------------
// global constants
// ------------------------------------------------------------------------

// common colours
static const unsigned short BLACK      = 0x000;
static const unsigned short WHITE      = 0xEEE;
static const unsigned short RED        = 0xE00;
static const unsigned short GREEN      = 0x0E0;
static const unsigned short BLUE       = 0x00E;
static const unsigned short YELLOW     = 0xEE0;
static const unsigned short MAGENTA    = 0xE0E;
static const unsigned short LIGHTBLUE  = 0x0EE;
static const unsigned short PINK       = 0xE07;
static const unsigned short PURPLE     = 0x70E;
static const unsigned short ORANGE     = 0xE70;
static const unsigned short LIGHTGREEN = 0x7E0;
static const unsigned short BLUEGREEN  = 0x0E7;
static const unsigned short LIGHTYELLOW= 0xEE7;

// states
#define ZERO 0
#define ONE 1

// sin() lookup table
static const signed char sinus[30] =
{ 0x0,0x1A,0x33,0x4A,0x5E,0x6D,0x78,0x7E,0x7E,0x78,0x6D,0x5E,0x4A,0x33,0x1A,0x0,0xE6,0xCD,0x
B6,0xA2,0x93,0x88,0x82,0x82,0x88,0x93,0xA2,0xB6,0xCD,0xE6 };

// ------------------------------------------------------------------------
// Structs
// ------------------------------------------------------------------------

// for callback registration
typedef void (*loadFunction_cb_t)(unsigned short* frameBuffer, unsigned char* userData);
typedef void (*processLoopFunction_cb_t)(void);
typedef void (*processInputFunction_cb_t)(void);
typedef void (*drawMenuIconFunction_cb_t)(void);
struct FrameWork_Registered_Games_t
{
	loadFunction_cb_t load;
	processLoopFunction_cb_t processLoop;
	processInputFunction_cb_t processInput;
	drawMenuIconFunction_cb_t drawMenuIcon;
};

// for input
struct FrameWork_Buttons_t
{
```

```
        unsigned char oldButtonState;
        unsigned char buttonState;
        unsigned char buttonPositiveEdge;
        unsigned char menuButtonTracker;
};

// Framework struct
struct FrameWork_t
{

        // input
        struct FrameWork_Buttons_t player[4];
        unsigned char menuButtonFlags;

        // game registration
        struct FrameWork_Registered_Games_t game[MAX_GAMES];
        unsigned char gamesRegistered;

        // frame rate
        volatile unsigned char updateDivider;
        volatile unsigned char updateCounter;
        volatile unsigned char updateFlag;

        // misc
        unsigned short frameBuffer[256];
        unsigned short randomSeed;
        unsigned char gameSelected;
};

// ------------------------------------------------------------------------
// Function Prototypes
// ------------------------------------------------------------------------

// framework specific functions, which shouldn't be called
// by anything else other than the framework
void initFrameWork( void );
void startFrameWork( void );
void pollPorts( void );
void frameWorkUpdateProcessLoop( void );
void frameWorkUpdateInputLoop( void );
void menuUpdateIcon( unsigned char* selected );

// used to register a game to the framework for callbacks
void registerModule( loadFunction_cb_t loadFunction, processLoopFunction_cb_t
processLoopFunction, processInputFunction_cb_t processInputFunction,
drawMenuIconFunction_cb_t drawMenuIconFunction );

// change applications
void startGame( unsigned char* gameSelected, unsigned char* playerCount );
void endGame( void );

// misc
void setRefreshRate( unsigned char refresh );
void clearFrameBuffer( unsigned short* frameBuffer );
unsigned char rnd( void );
extern inline signed char sin( unsigned short angle );
extern inline void wrap( unsigned short* value, unsigned char wrap );
unsigned char sqrt( unsigned short* value );

// player specific input
extern inline unsigned char player1ButtonFire( void );
extern inline unsigned char player1ButtonLeft( void );
```

```
extern inline unsigned char player1ButtonRight( void );
extern inline unsigned char player1ButtonUp( void );
extern inline unsigned char player1ButtonDown( void );
extern inline unsigned char player1ButtonMenu( void );
extern inline unsigned char player1ButtonClear( void );

extern inline unsigned char player2ButtonFire( void );
extern inline unsigned char player2ButtonLeft( void );
extern inline unsigned char player2ButtonRight( void );
extern inline unsigned char player2ButtonUp( void );
extern inline unsigned char player2ButtonDown( void );
extern inline unsigned char player2ButtonMenu( void );
extern inline unsigned char player2ButtonClear( void );

extern inline unsigned char player3ButtonFire( void );
extern inline unsigned char player3ButtonLeft( void );
extern inline unsigned char player3ButtonRight( void );
extern inline unsigned char player3ButtonUp( void );
extern inline unsigned char player3ButtonDown( void );
extern inline unsigned char player3ButtonMenu( void );
extern inline unsigned char player3ButtonClear( void );

extern inline unsigned char player4ButtonFire( void );
extern inline unsigned char player4ButtonLeft( void );
extern inline unsigned char player4ButtonRight( void );
extern inline unsigned char player4ButtonUp( void );
extern inline unsigned char player4ButtonDown( void );
extern inline unsigned char player4ButtonMenu( void );
extern inline unsigned char player4ButtonClear( void );

// player general input
unsigned char playerButtonFire( unsigned char playerID );
unsigned char playerButtonLeft( unsigned char playerID );
unsigned char playerButtonRight( unsigned char playerID );
unsigned char playerButtonUp( unsigned char playerID );
unsigned char playerButtonDown( unsigned char playerID );
extern inline unsigned char playerButtonMenu( unsigned char playerID );
extern inline unsigned char playerButtonClear( unsigned char playerID );

#endif // _FRAMEWORK_H_
```

7.2.1.12. framework.c

```
// ------------------------------------------------------------------------
// Framework
// ------------------------------------------------------------------------

// ------------------------------------------------------------------------
// Include Files
// ------------------------------------------------------------------------

#include "common.h"
#include "framework.h"
#include "uart.h"
#include "menu.h"
#include "startupscreen.h"
#include "gameenable.h"

// added games
#include "snake.h"
#include "colourdemo.h"
#include "gameoflife.h"
#include "tron.h"
#include "spaceinvaders.h"
#include "tetris.h"
#include "pong.h"
#include "burgler.h"
#include "catandmouse.h"

static struct FrameWork_t FrameWork;

// ------------------------------------------------------------------------
// initialise framework
void initFrameWork( void )
{

        // set initial values
        FrameWork.updateCounter = 0;
        FrameWork.updateDivider = 1;
        FrameWork.updateFlag = 0;
        FrameWork.gamesRegistered = 0;

        // register start screen
        registerModule( loadStartUpScreen, processStartUpScreenLoop,
processStartUpScreenInput, drawStartUpScreenIconDummy );

        // register menu
        registerModule( loadMenu, processMenuLoop, processMenuInput, drawMenuIconDummy );

        // register user added games
#ifdef GAME_ENABLE_COLOUR_DEMO
        registerModule( loadColourDemo, processColourDemoLoop, processColourDemoInput,
drawColourDemoMenuIcon );
#endif
#ifdef GAME_ENABLE_SNAKE
        registerModule( loadSnake, processSnakeLoop, processSnakeInput, drawSnakeMenuIcon );
#endif
#ifdef GAME_ENABLE_GAME_OF_LIFE
        registerModule( loadGameOfLife, processGameOfLifeLoop, processGameOfLifeInput,
drawGameOfLifeMenuIcon );
#endif
```

```
#ifdef GAME_ENABLE_TRON
	registerModule( loadTron, processTronLoop, processTronInput, drawTronMenuIcon );
#endif
#ifdef GAME_ENABLE_TETRIS
	registerModule( loadTetris, processTetrisLoop, processTetrisInput, drawTetrisMenuIcon
);
#endif
#ifdef GAME_ENABLE_SPACE_INVADERS
	registerModule( loadSpaceInvaders, processSpaceInvadersLoop,
processSpaceInvadersInput, drawSpaceInvadersMenuIcon );
#endif
#ifdef GAME_ENABLE_PONG
	registerModule( loadPong, processPongLoop, processPongInput, drawPongMenuIcon );
#endif
#ifdef GAME_ENABLE_BURGLER
	registerModule( loadBurgler, processBurglerLoop, processBurglerInput,
drawBurglerMenuIcon );
#endif
#ifdef GAME_ENABLE_CAT_AND_MOUSE
	registerModule( loadCatAndMouse, processCatAndMouseLoop, processCatAndMouseInput,
drawCatAndMouseMenuIcon );
#endif

	// load startup screen
	unsigned char gameSelected = 0;
	unsigned char playerCount = 0;
	startGame( &gameSelected, &playerCount );
}

// ----------------------------------------------------------------------
// main loop for entire micro controller
void startFrameWork( void )
{

	// main loop
	while( 1 )
	{

		// read input
		pollPorts();

		// update
		if( FrameWork.updateFlag ){
			frameWorkUpdateProcessLoop();
			FrameWork.updateFlag = 0;
		}

	}
}

// ----------------------------------------------------------------------
// poll all ports
void pollPorts( void )
{

	// read in new button states
	FrameWork.player[0].buttonState = MAP_PLAYER1_BUTTON;
	FrameWork.player[1].buttonState = MAP_PLAYER2_BUTTON;
	FrameWork.player[2].buttonState = MAP_PLAYER3_BUTTON;
	FrameWork.player[3].buttonState = MAP_PLAYER4_BUTTON;

	// add menu button to 6th bit
```

```
        if( (FrameWork.player[0].buttonState & (MAP_PLAYER1_BUTTON_UP |
MAP_PLAYER1_BUTTON_DOWN)) == (MAP_PLAYER1_BUTTON_UP | MAP_PLAYER1_BUTTON_DOWN) )
                FrameWork.player[0].buttonState |= MAP_PLAYER_BUTTON_MENU;
        if( (FrameWork.player[1].buttonState & (MAP_PLAYER2_BUTTON_UP |
MAP_PLAYER2_BUTTON_DOWN)) == (MAP_PLAYER2_BUTTON_UP | MAP_PLAYER2_BUTTON_DOWN) )
                FrameWork.player[1].buttonState |= MAP_PLAYER_BUTTON_MENU;
        if( (FrameWork.player[2].buttonState & (MAP_PLAYER3_BUTTON_UP |
MAP_PLAYER3_BUTTON_DOWN)) == (MAP_PLAYER3_BUTTON_UP | MAP_PLAYER3_BUTTON_DOWN) )
                FrameWork.player[2].buttonState |= MAP_PLAYER_BUTTON_MENU;
        if( (FrameWork.player[3].buttonState & (MAP_PLAYER4_BUTTON_UP |
MAP_PLAYER4_BUTTON_DOWN)) == (MAP_PLAYER4_BUTTON_UP | MAP_PLAYER4_BUTTON_DOWN) )
                FrameWork.player[3].buttonState |= MAP_PLAYER_BUTTON_MENU;

        // add clear button to 7th bit
        if( (FrameWork.player[0].buttonState & (MAP_PLAYER1_BUTTON_LEFT |
MAP_PLAYER1_BUTTON_RIGHT)) == (MAP_PLAYER1_BUTTON_LEFT | MAP_PLAYER1_BUTTON_RIGHT) )
                FrameWork.player[0].buttonState |= MAP_PLAYER_BUTTON_CLEAR;
        if( (FrameWork.player[1].buttonState & (MAP_PLAYER2_BUTTON_LEFT |
MAP_PLAYER2_BUTTON_RIGHT)) == (MAP_PLAYER2_BUTTON_LEFT | MAP_PLAYER2_BUTTON_RIGHT) )
                FrameWork.player[1].buttonState |= MAP_PLAYER_BUTTON_CLEAR;
        if( (FrameWork.player[2].buttonState & (MAP_PLAYER3_BUTTON_LEFT |
MAP_PLAYER3_BUTTON_RIGHT)) == (MAP_PLAYER3_BUTTON_LEFT | MAP_PLAYER3_BUTTON_RIGHT) )
                FrameWork.player[2].buttonState |= MAP_PLAYER_BUTTON_CLEAR;
        if( (FrameWork.player[3].buttonState & (MAP_PLAYER4_BUTTON_LEFT |
MAP_PLAYER4_BUTTON_RIGHT)) == (MAP_PLAYER4_BUTTON_LEFT | MAP_PLAYER4_BUTTON_RIGHT) )
                FrameWork.player[3].buttonState |= MAP_PLAYER_BUTTON_CLEAR;

        // process buttons
        unsigned char updateFlag = 0, i;
        for( i = 0; i != 4; i++ )
        {

                // process positive edges and save old states
                FrameWork.player[i].buttonPositiveEdge =
((~FrameWork.player[i].oldButtonState) & FrameWork.player[i].buttonState);
                FrameWork.player[i].oldButtonState = FrameWork.player[i].buttonState;
                if( FrameWork.player[i].buttonPositiveEdge ) updateFlag = 1;

                // change random seed as long as buttons are being pressed
                if( FrameWork.player[i].buttonState&0x1F )
                        FrameWork.randomSeed++;
        }

        // update any input loops
        if( updateFlag ) frameWorkUpdateInputLoop();

}

// ------------------------------------------------------------------------
// register a game with the framework
void registerModule( loadFunction_cb_t loadFunction, processLoopFunction_cb_t
processLoopFunction, processInputFunction_cb_t processInputFunction,
drawMenuIconFunction_cb_t drawMenuIconFunction )
{

        // check for free slots
        if( FrameWork.gamesRegistered == MAX_GAMES ) return;

        // register game
        FrameWork.game[ FrameWork.gamesRegistered ].load = loadFunction;
        FrameWork.game[ FrameWork.gamesRegistered ].processLoop = processLoopFunction;
        FrameWork.game[ FrameWork.gamesRegistered ].processInput = processInputFunction;
```

```
        FrameWork.game[ FrameWork.gamesRegistered ].drawMenuIcon = drawMenuIconFunction;
        FrameWork.gamesRegistered++;
}

// ------------------------------------------------------------------------
// sets the refresh rate
void setRefreshRate( unsigned char refresh )
{
        FrameWork.updateDivider = 255/refresh;
        FrameWork.updateCounter = 0;
}

// ------------------------------------------------------------------------
// clears a frame buffer
void clearFrameBuffer( unsigned short* frameBuffer )
{
        unsigned char x = 0;
        do{
                frameBuffer[x] = 0;
                x++;
        }while( x != 0 );
}

// ------------------------------------------------------------------------
// gets a random number
unsigned char rnd( void )
{
        FrameWork.randomSeed++;
        unsigned short x = FrameWork.randomSeed;
        x = (x<<7) ^ x;
        x = (unsigned short)(( 34071 - ( ( x * ( x * x * 15731 + 7881 ) + 13763 ) &
0x7FFF ))/9);
        unsigned char y = x>>8;
        unsigned char z = x;
        return ((y^z)*x)^FrameWork.randomSeed;
}

// ------------------------------------------------------------------------
// starts a game
void startGame( unsigned char* gameSelected, unsigned char* playerCount )
{
        FrameWork.game[ *gameSelected ].load( FrameWork.frameBuffer, playerCount );
        FrameWork.gameSelected = *gameSelected;
}

// ------------------------------------------------------------------------
// end the game
void endGame( void )
{

        // load the menu
        FrameWork.gameSelected = 1; unsigned char discard;
        FrameWork.game[ 1 ].load( FrameWork.frameBuffer, &discard );
}

// ------------------------------------------------------------------------
// updates menu icon
void menuUpdateIcon( unsigned char* selected )
{

        // clear icon space
        unsigned char x1=3, x2=12;
```

```
        fillBox( &x1, &x1, &x2, &x2, &BLACK );
        menuDrawLeftArrow( 0 );
        menuDrawRightArrow( 0 );

        // call draw function
        FrameWork.game[ *selected ].drawMenuIcon();
}

// ---------------------------------------------------------------------
// gets the button state of a specific player (positive edge only)
// unfortunately, I see no way to compress this, because it is dependant on global
definitions
inline unsigned char player1ButtonFire ( void ){ return
FrameWork.player[0].buttonPositiveEdge & MAP_PLAYER1_BUTTON_FIRE;  }
inline unsigned char player1ButtonLeft ( void ){ return
FrameWork.player[0].buttonPositiveEdge & MAP_PLAYER1_BUTTON_LEFT;  }
inline unsigned char player1ButtonRight( void ){ return
FrameWork.player[0].buttonPositiveEdge & MAP_PLAYER1_BUTTON_RIGHT; }
inline unsigned char player1ButtonUp   ( void ){ return
FrameWork.player[0].buttonPositiveEdge & MAP_PLAYER1_BUTTON_UP;    }
inline unsigned char player1ButtonDown ( void ){ return
FrameWork.player[0].buttonPositiveEdge & MAP_PLAYER1_BUTTON_DOWN;  }
inline unsigned char player1ButtonMenu ( void ){ return
FrameWork.player[0].buttonPositiveEdge & MAP_PLAYER_BUTTON_MENU;   }
inline unsigned char player1ButtonClear( void ){ return
FrameWork.player[0].buttonPositiveEdge & MAP_PLAYER_BUTTON_CLEAR;  }

inline unsigned char player2ButtonFire ( void ){ return
FrameWork.player[1].buttonPositiveEdge & MAP_PLAYER2_BUTTON_FIRE;  }
inline unsigned char player2ButtonLeft ( void ){ return
FrameWork.player[1].buttonPositiveEdge & MAP_PLAYER2_BUTTON_LEFT;  }
inline unsigned char player2ButtonRight( void ){ return
FrameWork.player[1].buttonPositiveEdge & MAP_PLAYER2_BUTTON_RIGHT; }
inline unsigned char player2ButtonUp   ( void ){ return
FrameWork.player[1].buttonPositiveEdge & MAP_PLAYER2_BUTTON_UP;    }
inline unsigned char player2ButtonDown ( void ){ return
FrameWork.player[1].buttonPositiveEdge & MAP_PLAYER2_BUTTON_DOWN;  }
inline unsigned char player2ButtonMenu ( void ){ return
FrameWork.player[1].buttonPositiveEdge & MAP_PLAYER_BUTTON_MENU;   }
inline unsigned char player2ButtonClear( void ){ return
FrameWork.player[1].buttonPositiveEdge & MAP_PLAYER_BUTTON_CLEAR;  }

inline unsigned char player3ButtonFire ( void ){ return
FrameWork.player[2].buttonPositiveEdge & MAP_PLAYER3_BUTTON_FIRE;  }
inline unsigned char player3ButtonLeft ( void ){ return
FrameWork.player[2].buttonPositiveEdge & MAP_PLAYER3_BUTTON_LEFT;  }
inline unsigned char player3ButtonRight( void ){ return
FrameWork.player[2].buttonPositiveEdge & MAP_PLAYER3_BUTTON_RIGHT; }
inline unsigned char player3ButtonUp   ( void ){ return
FrameWork.player[2].buttonPositiveEdge & MAP_PLAYER3_BUTTON_UP;    }
inline unsigned char player3ButtonDown ( void ){ return
FrameWork.player[2].buttonPositiveEdge & MAP_PLAYER3_BUTTON_DOWN;  }
inline unsigned char player3ButtonMenu ( void ){ return
FrameWork.player[2].buttonPositiveEdge & MAP_PLAYER_BUTTON_MENU;   }
inline unsigned char player3ButtonClear( void ){ return
FrameWork.player[2].buttonPositiveEdge & MAP_PLAYER_BUTTON_CLEAR;  }

inline unsigned char player4ButtonFire ( void ){ return
FrameWork.player[3].buttonPositiveEdge & MAP_PLAYER4_BUTTON_FIRE;  }
inline unsigned char player4ButtonLeft ( void ){ return
FrameWork.player[3].buttonPositiveEdge & MAP_PLAYER4_BUTTON_LEFT;  }
```

```
inline unsigned char player4ButtonRight( void ){ return
FrameWork.player[3].buttonPositiveEdge & MAP_PLAYER4_BUTTON_RIGHT; }
inline unsigned char player4ButtonUp   ( void ){ return
FrameWork.player[3].buttonPositiveEdge & MAP_PLAYER4_BUTTON_UP;    }
inline unsigned char player4ButtonDown ( void ){ return
FrameWork.player[3].buttonPositiveEdge & MAP_PLAYER4_BUTTON_DOWN;  }
inline unsigned char player4ButtonMenu ( void ){ return
FrameWork.player[3].buttonPositiveEdge & MAP_PLAYER_BUTTON_MENU;   }
inline unsigned char player4ButtonClear( void ){ return
FrameWork.player[3].buttonPositiveEdge & MAP_PLAYER_BUTTON_CLEAR;  }

// ------------------------------------------------------------------------
// general player input for "fire"
unsigned char playerButtonFire( unsigned char playerID )
{
        unsigned char mask;
        switch( playerID )
        {
                case 0 : mask = MAP_PLAYER1_BUTTON_FIRE; break;
                case 1 : mask = MAP_PLAYER2_BUTTON_FIRE; break;
                case 2 : mask = MAP_PLAYER3_BUTTON_FIRE; break;
                case 3 : mask = MAP_PLAYER4_BUTTON_FIRE; break;
                default: break;
        }
        return FrameWork.player[playerID].buttonPositiveEdge & mask;
}

// ------------------------------------------------------------------------
// general player input for "left"
unsigned char playerButtonLeft( unsigned char playerID )
{
        unsigned char mask;
        switch( playerID )
        {
                case 0 : mask = MAP_PLAYER1_BUTTON_LEFT; break;
                case 1 : mask = MAP_PLAYER2_BUTTON_LEFT; break;
                case 2 : mask = MAP_PLAYER3_BUTTON_LEFT; break;
                case 3 : mask = MAP_PLAYER4_BUTTON_LEFT; break;
                default: break;
        }
        return FrameWork.player[playerID].buttonPositiveEdge & mask;
}

// ------------------------------------------------------------------------
// general player input for "right"
unsigned char playerButtonRight( unsigned char playerID )
{
        unsigned char mask;
        switch( playerID )
        {
                case 0 : mask = MAP_PLAYER1_BUTTON_RIGHT; break;
                case 1 : mask = MAP_PLAYER2_BUTTON_RIGHT; break;
                case 2 : mask = MAP_PLAYER3_BUTTON_RIGHT; break;
                case 3 : mask = MAP_PLAYER4_BUTTON_RIGHT; break;
                default: break;
        }
        return FrameWork.player[playerID].buttonPositiveEdge & mask;
}

// ------------------------------------------------------------------------
// general player input for "up"
unsigned char playerButtonUp( unsigned char playerID )
```

```
{
	unsigned char mask;
	switch( playerID )
	{
		case 0 : mask = MAP_PLAYER1_BUTTON_UP; break;
		case 1 : mask = MAP_PLAYER2_BUTTON_UP; break;
		case 2 : mask = MAP_PLAYER3_BUTTON_UP; break;
		case 3 : mask = MAP_PLAYER4_BUTTON_UP; break;
		default: break;
	}
	return FrameWork.player[playerID].buttonPositiveEdge & mask;
}

// ------------------------------------------------------------------------
// general player input for "down"
unsigned char playerButtonDown( unsigned char playerID )
{
	unsigned char mask;
	switch( playerID )
	{
		case 0 : mask = MAP_PLAYER1_BUTTON_DOWN; break;
		case 1 : mask = MAP_PLAYER2_BUTTON_DOWN; break;
		case 2 : mask = MAP_PLAYER3_BUTTON_DOWN; break;
		case 3 : mask = MAP_PLAYER4_BUTTON_DOWN; break;
		default: break;
	}
	return FrameWork.player[playerID].buttonPositiveEdge & mask;
}

// ------------------------------------------------------------------------
// general player input for "menu"
inline unsigned char playerButtonMenu( unsigned char playerID )
{
	return FrameWork.player[playerID].buttonPositiveEdge & MAP_PLAYER_BUTTON_MENU;
}

// ------------------------------------------------------------------------
// general player input for "clear"
inline unsigned char playerButtonClear( unsigned char playerID )
{
	return FrameWork.player[playerID].buttonPositiveEdge & MAP_PLAYER_BUTTON_MENU;
}

// ------------------------------------------------------------------------
// sinus
inline signed char sin( unsigned short angle )
{
	angle /= 12;
	wrap( &angle, 30 );
	return sinus[ angle ];
}

// ------------------------------------------------------------------------
// wraps a value between 0 and <wrap>
inline void wrap( unsigned short* value, unsigned char wrap )
{
	while( (*value) >= wrap ){ (*value) -= wrap; }
}

// ------------------------------------------------------------------------
// square root
unsigned char sqrt( unsigned short* value )
```

```
{

        // prepare first guess
        register unsigned char result;
        register unsigned char currentBitMask;
        if( 0x4000 > *value )
        {
            result = 0;
            currentBitMask = 0x40;
        }else{
            currentBitMask = 0x80;
        }

        // loop through all 8 bits in result
        while( currentBitMask )
        {

                // add next bit
                result |= currentBitMask;

                // if squared result is larger than value, remove bit again
                if( result*result > *value ) result ^= currentBitMask;

                // select next bit
                currentBitMask >>= 1;
        }

        // return result
        return result;
}

// ----------------------------------------------------------------------
// call update loop of current game running - passes the process on to
// the current "main loop" by using a callback
// this allows expandability for more games/demos in the future
void frameWorkUpdateProcessLoop( void )
{

       // callback selected game's main loop
       FrameWork.game[ FrameWork.gameSelected ].processLoop();
}

// ----------------------------------------------------------------------
// call input loop of current running game
void frameWorkUpdateInputLoop( void )
{

       // callback selected game's input loop
       FrameWork.game[ FrameWork.gameSelected ].processInput();
}

// ----------------------------------------------------------------------
// Update interrupt
#pragma vector=TIMERA0_VECTOR
__interrupt void Timer_A( void )
{

       // call UART timeout update (for resend)
       UARTUpdateTimeOut();

       // divide update rate
       unsigned char temp = (FrameWork.updateCounter++);
```

```
            if( temp != FrameWork.updateDivider ) return;
            FrameWork.updateCounter = 0;

            // set update flag (this is caught in the main loop)
            FrameWork.updateFlag = 1;
    }
```

7.2.1.13. menu.h

```
// ------------------------------------------------------------------------
// Main Menu
// ------------------------------------------------------------------------

#ifndef _MENU_H_
	#define _MENU_H_

#include "gameenable.h"

// ------------------------------------------------------------------------
// Structs
// ------------------------------------------------------------------------

struct Menu_t
{
	unsigned char toggleArrow;
	unsigned char selected;
	unsigned char playerList;
	unsigned char gameCount;
};

// ------------------------------------------------------------------------
// Function Prototypes
// ------------------------------------------------------------------------

void initMenu( void );
void loadMenu( unsigned short* frameBuffer, unsigned char* playerCount );
void processMenuLoop( void );
void processMenuInput( void );
void drawMenuIconDummy( void );
void menuDrawFrame( void );
void menuDrawLeftArrow( unsigned char clear );
void menuDrawRightArrow( unsigned char clear );
void menuDrawStartArrow( unsigned char offset );
void menuDrawJoinArrows( unsigned char* playerList );

#endif // _MENU_H_
```

7.2.1.14. menu.c

```
// ------------------------------------------------------------------------
// Main Menu
// ------------------------------------------------------------------------

// ------------------------------------------------------------------------
// Include files
// ------------------------------------------------------------------------

#include "menu.h"
#include "uart.h"
#include "framework.h"
#include "gameenable.h"

static struct Menu_t Menu;

// ------------------------------------------------------------------------
// draws the menu frame
void menuDrawFrame( void )
{
	unsigned char x1=0, x2=15;
	blendColourBox( &x1, &x1, &x2, &x2, &LIGHTBLUE, &BLUE, &BLUE, &LIGHTBLUE );
}

// ------------------------------------------------------------------------
// draws the left and right game selection arrows
void menuDrawLeftArrow( unsigned char clear )
{
	unsigned char i;
	for( i = 0; i != 2; i++ )
	{
		unsigned char x1=i, y1=7-i, y2=8+i;
		unsigned short cA=0xEE0*clear;
		line( &x1, &y1, &x1, &y2, &cA );
	}
}
void menuDrawRightArrow( unsigned char clear )
{
	unsigned char i;
	for( i = 0; i != 2; i++ )
	{
		unsigned char x1=15-i, y1=7-i, x2=15-i, y2=8+i;
		unsigned short cA=0xEE0*clear;
		line( &x1, &y1, &x2, &y2, &cA );
	}
}

// ------------------------------------------------------------------------
// draw join arrows
void menuDrawJoinArrows( unsigned char* playerList )
{

	// player 1 (always green)
	unsigned char x1=12, y1=14, x2=14;
	line( &x1, &y1, &x2, &y1, &GREEN );
	x1=13; y1=15;
	dot( &x1, &y1, &GREEN );

	// player 2
```

```
        unsigned short cA;
        x1=1; y1=1; x2=3;
        if( (*playerList)&0x01 ) cA = 0x0E0; else cA = 0xE00;
        line( &x1, &y1, &x2, &y1, &cA );
        x1=2; y1=0;
        dot( &x1, &y1, &cA );

        // player 3
        y1=1; x1=12; x2=14;
        if( (*playerList)&0x02 ) cA = 0x0E0; else cA = 0xE00;
        line( &y1, &x1, &y1, &x2, &cA ),
        x1=0; y1=13;
        dot( &x1, &y1, &cA );

        // player 4
        y1=14; x1=1; x2=3;
        if( (*playerList)&0x04 ) cA = 0x0E0; else cA = 0xE00;
        line( &y1, &x1, &y1, &x2, &cA );
        x1=15; y1=2;
        dot( &x1, &y1, &cA );
}

// initialises some values for the menu
void initMenu( void )
{
        Menu.selected  = 2;
        Menu.gameCount = getGameCount() + 2; // startup screen and menu are 0, 1
}

// ------------------------------------------------------------------------
// load menu
void loadMenu( unsigned short* frameBuffer, unsigned char* playerCount )
{

        // set refresh rate
        setRefreshRate( 70 );

        // reset values
        Menu.toggleArrow = 0;
        Menu.playerList = 0;

        // draw current game
        cls();
        menuDrawFrame();
        menuDrawRightArrow(0);
        menuDrawLeftArrow(0);
        menuDrawJoinArrows( &Menu.playerList );
        menuUpdateIcon( &Menu.selected );
        send();
}

// ------------------------------------------------------------------------
// process menu loop
void processMenuLoop( void )
{

        // toggle
        Menu.toggleArrow = 1 - Menu.toggleArrow;

        // control left and right arrow blinking
        if( Menu.selected > 2 ) menuDrawLeftArrow( Menu.toggleArrow );
        if( Menu.selected < Menu.gameCount ) menuDrawRightArrow( Menu.toggleArrow );
```

```
        send();

}

// ------------------------------------------------------------------------
// process menu input
void processMenuInput( void )
{

        // process players joining/leaving
        if( player2ButtonFire() ){ Menu.playerList ^= 0x01;
menuDrawJoinArrows( &Menu.playerList ); send(); }
        if( player3ButtonFire() ){ Menu.playerList ^= 0x02;
menuDrawJoinArrows( &Menu.playerList ); send(); }
        if( player4ButtonFire() ){ Menu.playerList ^= 0x04;
menuDrawJoinArrows( &Menu.playerList ); send(); }

        // select previous game
        if( player1ButtonLeft() && Menu.selected > 2 )
        {
                Menu.selected--;

                // reset players joined
                Menu.playerList = 0;
                menuDrawJoinArrows( &Menu.playerList );

                // draw appropriate icon
                menuUpdateIcon( &Menu.selected );
                send();
        }

        // select next game
        if( player1ButtonRight() && Menu.selected < Menu.gameCount )
        {
                Menu.selected++;

                // reset players joined
                Menu.playerList = 0;
                menuDrawJoinArrows( &Menu.playerList );

                // draw appropriate icon
                menuUpdateIcon( &Menu.selected );
                send();
        }

        // start a game
        if( player1ButtonFire() )
        {
                startGame( &Menu.selected, &Menu.playerList );
        }
}

// ------------------------------------------------------------------------
// Menu Icon dummy, so the framework is happy
void drawMenuIconDummy( void )
{
}
```

7.2.1.15. startupscreen.h

```
// ------------------------------------------------------------------------
// Start-Up Screen
// ------------------------------------------------------------------------

#ifndef _STARTUPSCREEN_H_
	#define _STARTUPSCREEN_H_

// ------------------------------------------------------------------------
// Structs
// ------------------------------------------------------------------------

struct StartUpScreen_t
{
	unsigned char toggle;
};

// ------------------------------------------------------------------------
// Function Prototypes
// ------------------------------------------------------------------------

void drawStartArrow( void );
void drawButtonHelp( void );
void drawStartUpScreenIconDummy( void );
void loadStartUpScreen( unsigned short* frameBuffer, unsigned char* playerCount );
void processStartUpScreenLoop( void );
void processStartUpScreenInput( void );

#endif // _STARTUPSCREEN_H_
```

7.2.1.16. startupscreen.c

```
// ----------------------------------------------------------------------
// Start-Up Screen
// ----------------------------------------------------------------------

// ----------------------------------------------------------------------
// Include Files
// ----------------------------------------------------------------------

#include "startupscreen.h"
#include "framework.h"
#include "uart.h"

static struct StartUpScreen_t StartUpScreen;

// ----------------------------------------------------------------------
// draws an arrow pointing to player 1's port
void drawStartArrow( void )
{

	// draw arrow
	unsigned char i;
	for( i = 0; i != 5; i++ )
	{
		unsigned char x1=7-i, y1=15-StartUpScreen.toggle-i, x2=8+i;
		unsigned short cA=(0x00E|((i<<4)*3));
		line( &x1, &y1, &x2, &y1, &cA );
	}
}

// ----------------------------------------------------------------------
// draws help so the user knows what to press
void drawButtonHelp( void )
{

	// draw non-changing buttons
	unsigned char x1, y1, x2, y2;
	x1=0x05; y1=0x03; x2=0x06; y2=0x04; fillBox( &x1, &y1, &x2, &y2, &BLUE );
	x1=0x09;          x2=0x0A;          fillBox( &x1, &y1, &x2, &y2, &BLUE );
		  y1=0x05;          y2=0x06; fillBox( &x1, &y1, &x2, &y2, &BLUE );

	// draw changing buttons
	x1=0x07; y1=0x01; x2=0x08; y2=0x02; if( StartUpScreen.toggle ) fillBox( &x1, &y1,
&x2, &y2, &BLUE ); else fillBox( &x1, &y1, &x2, &y2, &YELLOW );
		  y1=0x03;          y2=0x04; if( StartUpScreen.toggle ) fillBox( &x1, &y1,
&x2, &y2, &BLUE ); else fillBox( &x1, &y1, &x2, &y2, &YELLOW );
}

// ----------------------------------------------------------------------
// start up screen icon dummy, to satisfy the framework
void drawStartUpScreenIconDummy( void )
{
}

// ----------------------------------------------------------------------
// load start up screen
void loadStartUpScreen( unsigned short* frameBuffer, unsigned char* playerCount )
{
```

```
        // reset values
        StartUpScreen.toggle = 0;

        // set refresh rate
        setRefreshRate( 50 );

        // prepare screen
        cls();
        send();
}

// ---------------------------------------------------------------------
// process start up screen loop
void processStartUpScreenLoop( void )
{

        // toggle
        StartUpScreen.toggle = 1 - StartUpScreen.toggle;

        // draw graphics
        cls();
        drawStartArrow();
        drawButtonHelp();
        send();
}

// ---------------------------------------------------------------------
// process start up screen input
void processStartUpScreenInput( void )
{

        // start is pressed
        if( player1ButtonMenu() ) endGame();
}
```

7.2.1.17. colourdemo.h

```
// ------------------------------------------------------------------------
// Colour Demo
// ------------------------------------------------------------------------

#ifndef _COLOURDEMO_H_
	#define _COLOURDEMO_H_

// ------------------------------------------------------------------------
// Structs
// ------------------------------------------------------------------------

struct ColourDemo_t
{
	unsigned short* frameBuffer;
	unsigned short angle;
	unsigned char startR;
	unsigned char startG;
	unsigned char startB;
};

// ------------------------------------------------------------------------
// Function Prototypes
// ------------------------------------------------------------------------

void loadColourDemo( unsigned short* frameBuffer, unsigned char* playerCount );
void processColourDemoLoop( void );
void processColourDemoInput( void );
void colourDemoDrawFrameBuffer( void );
void drawColourDemoMenuIcon( void );

#endif // _COLOURDEMO_H_
```

7.2.1.18. colourdemo.c

```
// ------------------------------------------------------------------------
// Colour Demo
// ------------------------------------------------------------------------

// ------------------------------------------------------------------------
// Include files
// ------------------------------------------------------------------------

#include "colourdemo.h"
#include "framework.h"
#include "uart.h"
#include "gameenable.h"

#ifdef GAME_ENABLE_COLOUR_DEMO

static struct ColourDemo_t ColourDemo;

// ------------------------------------------------------------------------
// load colour demo
void loadColourDemo( unsigned short* frameBuffer, unsigned char* playerCount )
{

        // get frame buffer
        ColourDemo.frameBuffer = frameBuffer;

        // clear screen
        cls();
        send();

        // starting colour
        ColourDemo.startR = 14;
        ColourDemo.startG = 0;
        ColourDemo.startB = 0;

        // set refresh rate
        setRefreshRate( 255 );
}

// ------------------------------------------------------------------------
// process colour demo loop
void processColourDemoLoop( void )
{

        // locals
        register unsigned char cx, cy;
        register signed short sx, sy;
        unsigned short colour;
        unsigned char r, g, b;

        // slowly rotate centre point
        ColourDemo.angle += 5;

        // set starting colour
        if( ColourDemo.startR == 14 && ColourDemo.startG <  14 ) ColourDemo.startG+=2;
        if( ColourDemo.startG == 14 && ColourDemo.startR >  0  ) ColourDemo.startR-=2;
        if( ColourDemo.startG == 14 && ColourDemo.startB <  14 ) ColourDemo.startB+=2;
        if( ColourDemo.startB == 14 && ColourDemo.startG >  0  ) ColourDemo.startG-=2;
        if( ColourDemo.startB == 14 && ColourDemo.startR <  14 ) ColourDemo.startR+=2;
```

```
        if( ColourDemo.startR == 14 && ColourDemo.startB >  0  ) ColourDemo.startB-=2;

        // convert angle to coordinates
        sx=60, sy=60;
        sx += (sin( ColourDemo.angle )>>2);
        sy += (sin( ColourDemo.angle+90 )>>2);
        cx = (unsigned char) sx;
        cy = (unsigned char) sy;

        // computer colours of every dot
        for( unsigned char x = 0; x != 16; x++ )
        {
                for( unsigned char y = 0; y != 16; y++ )
                {

                        // get distance
                        sx = (cx - (x<<3)); sy = (cy - (y<<3));
                        sx *= sx; sy *= sy;
                        colour = sx + sy;
                        colour = sqrt( &colour );

                        // compute colours
                        colour >>= 1;
                        r = ColourDemo.startR; g = ColourDemo.startG; b = ColourDemo.startB;
                        while( colour )
                        {
                                colour--;
                                if( r == 14 && g <  14 && b == 0 ) g++;
                                if( g == 14 && r >  0  ) r--;
                                if( g == 14 && b <  14 ) b++;
                                if( b == 14 && g >  0  ) g--;
                                if( b == 14 && r <  14 ) r++;
                                if( r == 14 && b >  0  ) b--;
                        }

                        // final colour
                        colour = ((r<<8)|(g<<4)|b);

                        // save in frame buffer
                        (*(ColourDemo.frameBuffer+y+(x<<4))) = colour;
                }
        }

        // update display
        colourDemoDrawFrameBuffer();
        send();
}

// ----------------------------------------------------------------------
// process colour demo input
void processColourDemoInput( void )
{

        // exit game
        if( player1ButtonMenu() ) endGame();
}

// ----------------------------------------------------------------------
// draws the frame buffer to the screen
void colourDemoDrawFrameBuffer()
{
        unsigned short* bufferPtr;
```

```
        for( unsigned char x = 0; x != 16; x++ )
        {
                bufferPtr = (ColourDemo.frameBuffer+(x<<4));
                for( unsigned char y = 0; y != 16; y++ )
                {
                        dot( &x, &y, bufferPtr+y );
                }
        }
}

// ----------------------------------------------------------------------
// menu icon
void drawColourDemoMenuIcon( void )
{
        unsigned char x1=3, x2=12;
        unsigned short cA=0xE00, cB=0x0E0, cC=0x00E, cD=0xEE0;
        blendColourFillBox( &x1, &x1, &x2, &x2, &cA, &cB, &cC, &cD );
}
#endif // GAME_ENABLE_COLOUR_DEMO
```

7.2.1.19. gameoflife.h

```
// ------------------------------------------------------------------------
// Game of Life
// ------------------------------------------------------------------------

#ifndef _GAMEOFLIFE_H_
	#define _GAMEOFLIFE_H_

// ------------------------------------------------------------------------
// Structs
// ------------------------------------------------------------------------

struct GameOfLife_Cursor_t
{
	unsigned char x;
	unsigned char y;
};

struct GameOfLife_Player_t
{
	struct GameOfLife_Cursor_t cursor;
	struct GameOfLife_Cursor_t oldCursor;
	unsigned char cellsPlaced;
};

struct GameOfLife_t
{
	unsigned short* frameBuffer;
	unsigned char state;
	unsigned char bufferOffset;
	struct GameOfLife_Player_t player[4];
	unsigned char* playerCount;
};

// ------------------------------------------------------------------------
// Enumerators
// ------------------------------------------------------------------------

enum GameOfLife_States_e
{
	GAMEOFLIFE_STATE_PLAY_SINGLE,
	GAMEOFLIFE_STATE_EDIT_SINGLE,
	GAMEOFLIFE_STATE_PLAY_MULTI,
	GAMEOFLIFE_STATE_WINNER,
	GAMEOFLIFE_STATE_WINNER2
};

// ------------------------------------------------------------------------
// Function Prototypes
// ------------------------------------------------------------------------

void loadGameOfLife( unsigned short* frameBuffer, unsigned char* playerCount );
void processGameOfLifeLoop( void );
void processGameOfLifeInput( void );
void drawGameOfLifeMenuIcon( void );
void randomizeFrameBuffer( void );
void gameOfLifeDrawFrameBufferNoCheck( void );
void gameOfLifeDrawFrameBuffer( void );
```

```
void gameOfLifeDrawFrameBufferCustom( const unsigned short* colour1, const unsigned short*
colour2, const unsigned short* colour3, const unsigned short* colour4 );
void computeNextCycle( void );

#endif // _GAMEOFLIFE_H_
```

7.2.1.20. gameoflife.c

```
// ------------------------------------------------------------------------
// Game of Life
// ------------------------------------------------------------------------

// ------------------------------------------------------------------------
// include files
// ------------------------------------------------------------------------

#include "gameoflife.h"
#include "framework.h"
#include "uart.h"
#include "gameenable.h"

#ifdef GAME_ENABLE_GAME_OF_LIFE

static struct GameOfLife_t GameOfLife;

// ------------------------------------------------------------------------
// load game of life
void loadGameOfLife( unsigned short* frameBuffer, unsigned char* playerCount )
{

	// get frame buffer and player count
	GameOfLife.frameBuffer = frameBuffer;
	GameOfLife.bufferOffset = 0;
	GameOfLife.playerCount = playerCount;
	clearFrameBuffer( frameBuffer );

	// set player data
	unsigned char i;
	for( i = 0; i != 4; i++ )
	{
		GameOfLife.player[i].cursor.x = 7;
		GameOfLife.player[i].cursor.y = 7;
		GameOfLife.player[i].oldCursor = GameOfLife.player[i].cursor;
		if( ((*playerCount) & (1<<(i-1))) || i == 0 )
		{
			GameOfLife.player[i].cellsPlaced = 2;
		}else{
			GameOfLife.player[i].cellsPlaced = 0;
		}
	}

	// multi-player specific settings
	if( *playerCount )
	{

		// set initial game state
		GameOfLife.state = GAMEOFLIFE_STATE_PLAY_MULTI;

		// draw starting positions for each player
		*(GameOfLife.frameBuffer+12+(7*16)) = 0x01;
		*(GameOfLife.frameBuffer+12+(8*16)) = 0x01;
		*(GameOfLife.frameBuffer+13+(7*16)) = 0x01;
		*(GameOfLife.frameBuffer+13+(8*16)) = 0x01;

		// player 2
		if( (*playerCount) & 0x01 )
```

```
		{
			*(GameOfLife.frameBuffer+2+(7*16)) = 0x04;
			*(GameOfLife.frameBuffer+2+(8*16)) = 0x04;
			*(GameOfLife.frameBuffer+3+(7*16)) = 0x04;
			*(GameOfLife.frameBuffer+3+(8*16)) = 0x04;
		}

		// player 3
		if( (*playerCount) & 0x02 )
		{
			*(GameOfLife.frameBuffer+7+(2*16)) = 0x10;
			*(GameOfLife.frameBuffer+7+(3*16)) = 0x10;
			*(GameOfLife.frameBuffer+8+(2*16)) = 0x10;
			*(GameOfLife.frameBuffer+8+(3*16)) = 0x10;
		}

		// player 4
		if( (*playerCount) & 0x04 )
		{
			*(GameOfLife.frameBuffer+7+(12*16)) = 0x40;
			*(GameOfLife.frameBuffer+7+(13*16)) = 0x40;
			*(GameOfLife.frameBuffer+8+(12*16)) = 0x40;
			*(GameOfLife.frameBuffer+8+(13*16)) = 0x40;
		}

	// single player specific settinsg
	}else{

		// set initial game state
		GameOfLife.state = GAMEOFLIFE_STATE_PLAY_SINGLE;

		// randomize frame buffer
		randomizeFrameBuffer();
	}

	// set refresh rate
	setRefreshRate( 64 );

	// initialise screen
	cls();
	gameOfLifeDrawFrameBuffer();
	send();
}

// ------------------------------------------------------------------------
// process game of life
void processGameOfLifeLoop( void )
{

	// states
	switch( GameOfLife.state )
	{

		// single player mode
		case GAMEOFLIFE_STATE_PLAY_SINGLE :

			// simulate continuously
			computeNextCycle();

			break;

		// multi player mode
```

```
                case GAMEOFLIFE_STATE_PLAY_MULTI :

                        // everything happens in the input state

                        break;

                // winner
                case GAMEOFLIFE_STATE_WINNER :

                        break;

                default : break;
        }
}

// ------------------------------------------------------------------------
// process game of life input
void processGameOfLifeInput( void )
{

        // local variables
        unsigned short* bufferPtr;
        unsigned char readMask;
        unsigned char* cursorX;
        unsigned char* cursorY;
        unsigned char* oldCursorX;
        unsigned char* oldCursorY;
        unsigned short cursorColour;
        unsigned short cursorSelectColour;
        unsigned short cellColour;
        unsigned char i;

        // state dependant input
        switch( GameOfLife.state )
        {

                // edit mode
                case GAMEOFLIFE_STATE_EDIT_SINGLE :

                        // for speed reasons, get pointers to player data
                        cursorX = &GameOfLife.player[0].cursor.x;
                        cursorY = &GameOfLife.player[0].cursor.y;
                        oldCursorX = &GameOfLife.player[0].oldCursor.x;
                        oldCursorY = &GameOfLife.player[0].oldCursor.y;

                        // move cursor with up, down, left, right
                        if( player1ButtonUp() ) (*cursorY) = (((*cursorY)-1)&0x0F);
                        if( player1ButtonDown() ) (*cursorY) = (((*cursorY)+1)&0x0F);
                        if( player1ButtonLeft() ) (*cursorX) = (((*cursorX)-1)&0x0F);
                        if( player1ButtonRight() ) (*cursorX) = (((*cursorX)+1)&0x0F);

                        // get buffer
                        bufferPtr = (GameOfLife.frameBuffer + (*cursorY) + ((*cursorX)<<4));

                        // get read mask
                        readMask = (0x01<<GameOfLife.bufferOffset);

                        // edit cells
                        if( player1ButtonFire() )
                        {

                                // update frame buffer and draw new cursor
```

```
                                if( (*bufferPtr) & readMask )
                                        (*bufferPtr) = 0x00;
                                else
                                        (*bufferPtr) = readMask;
                        }

                        // remove old cursor
                        if( (*(GameOfLife.frameBuffer + (*oldCursorY) + ((*oldCursorX)<<4))) &
readMask )
                                dot( oldCursorX, oldCursorY, &BLUE );
                        else
                                dot( oldCursorX, oldCursorY, &BLACK );
                        GameOfLife.player[0].oldCursor = GameOfLife.player[0].cursor;

                        // draw new cursor
                        if( (*bufferPtr) & readMask )
                                dot( cursorX, cursorY, &YELLOW );
                        else
                                dot( cursorX, cursorY, &WHITE );

                        // clear frame buffer with clear button
                        if( player1ButtonClear() )
                        {
                                clearFrameBuffer( GameOfLife.frameBuffer );
                                gameOfLifeDrawFrameBufferNoCheck();
                                dot( cursorX, cursorY, &WHITE );
                                send();
                        }

                        // resume simulation
                        if( player1ButtonMenu() )
                        {
                                gameOfLifeDrawFrameBufferNoCheck(); // force re-drawing of all
pixels
                                GameOfLife.state = GAMEOFLIFE_STATE_PLAY_SINGLE;
                        }

                        // send graphic changes to display
                        send();

                        break;

                // during play
                case GAMEOFLIFE_STATE_PLAY_SINGLE :

                        // switch to editing mode
                        if( player1ButtonFire() )
                        {
                                gameOfLifeDrawFrameBufferCustom( &BLUE, &RED, &GREEN,
&YELLOW ); // force re-drawing of all pixels in a different colour
                                dot( &GameOfLife.player[0].cursor.x,
&GameOfLife.player[0].cursor.y, &WHITE );
                                send();
                                GameOfLife.state = GAMEOFLIFE_STATE_EDIT_SINGLE;
                        }

                        // end game with menu button
                        if( player1ButtonMenu() ) endGame();

                        break;

                // during multi play
```

```
		case GAMEOFLIFE_STATE_PLAY_MULTI :

			// loop through each player
			for( i = 0; i != 4; i++ )
			{

				// only active players and players that are able to place
				if( (((*GameOfLife.playerCount) & (1<<(i-1))) || i == 0) &&
GameOfLife.player[i].cellsPlaced )
				{

					// for speed reasons, get pointers to player data
					cursorX = &GameOfLife.player[i].cursor.x;
					cursorY = &GameOfLife.player[i].cursor.y;
					oldCursorX = &GameOfLife.player[i].oldCursor.x;
					oldCursorY = &GameOfLife.player[i].oldCursor.y;

					// player specific actions
					switch( i )
					{
						case 0 :

							// move cursor with up, down, left,
right
							if( playerButtonUp(i) ) (*cursorY) =
(((*cursorY)-1)&0x0F);
							if( playerButtonDown(i) ) (*cursorY) =
(((*cursorY)+1)&0x0F);
							if( playerButtonLeft(i) ) (*cursorX) =
(((*cursorX)-1)&0x0F);
							if( playerButtonRight(i) ) (*cursorX) =
(((*cursorX)+1)&0x0F);


							// set colours
							cursorColour = BLUEGREEN;
							cursorSelectColour = LIGHTGREEN;
							break;
						case 1 :

							// move cursor with up, down, left,
right
							if( playerButtonUp(i) ) (*cursorY) =
(((*cursorY)+1)&0x0F);
							if( playerButtonDown(i) ) (*cursorY) =
(((*cursorY)-1)&0x0F);
							if( playerButtonLeft(i) ) (*cursorX) =
(((*cursorX)+1)&0x0F);
							if( playerButtonRight(i) ) (*cursorX) =
(((*cursorX)-1)&0x0F);


							// set colours
							cursorColour = PURPLE;
							cursorSelectColour = ORANGE;
							break;
						case 2 :

							// move cursor with up, down, left,
right
							if( playerButtonUp(i) ) (*cursorX) =
(((*cursorX)+1)&0x0F);
							if( playerButtonDown(i) ) (*cursorX) =
(((*cursorX)-1)&0x0F);
```

```
                                                        if( playerButtonLeft(i) ) (*cursorY) =
(((*cursorY)-1)&0x0F);
                                                        if( playerButtonRight(i) ) (*cursorY) =
(((*cursorY)+1)&0x0F);

                                                        // set colours
                                                        cursorColour = PINK;
                                                        cursorSelectColour = LIGHTBLUE;
                                                        break;
                                                case 3 :

                                                        // move cursor with up, down, left,
right
                                                        if( playerButtonUp(i) ) (*cursorX) =
(((*cursorX)-1)&0x0F);
                                                        if( playerButtonDown(i) ) (*cursorX) =
(((*cursorX)+1)&0x0F);
                                                        if( playerButtonLeft(i) ) (*cursorY) =
(((*cursorY)+1)&0x0F);
                                                        if( playerButtonRight(i) ) (*cursorY) =
(((*cursorY)-1)&0x0F);

                                                        // set colours
                                                        cursorColour = LIGHTYELLOW;
                                                        cursorSelectColour = WHITE;
                                                        break;
                                                default: break;
                                        }

                                        // find cell colour under old cursor
                                        cellColour = BLACK;
                                        bufferPtr = GameOfLife.frameBuffer + (*oldCursorY) +
((*oldCursorX)<<4);
                                        if( (*bufferPtr) & (0x55<<GameOfLife.bufferOffset) )
                                        {
                                                if( (*bufferPtr) &
(0x01<<GameOfLife.bufferOffset) ) cellColour = GREEN;
                                                if( (*bufferPtr) &
(0x04<<GameOfLife.bufferOffset) ) cellColour = RED;
                                                if( (*bufferPtr) &
(0x10<<GameOfLife.bufferOffset) ) cellColour = BLUE;
                                                if( (*bufferPtr) &
(0x40<<GameOfLife.bufferOffset) ) cellColour = YELLOW;
                                        }

                                        // get buffer
                                        bufferPtr = (GameOfLife.frameBuffer + (*cursorY) +
((*cursorX)<<4));

                                        // get read mask
                                        readMask = ((1<<(i<<1))<<GameOfLife.bufferOffset);

                                        // edit cells
                                        if( playerButtonFire(i) )
                                        {

                                                // decrement cells placed
                                                GameOfLife.player[i].cellsPlaced--;

                                                // update frame buffer
                                                if( (*bufferPtr) & readMask )
                                                        (*bufferPtr) &= ~readMask;
```

```
                                        else
                                                (*bufferPtr) |= readMask;
                                }

                                // remove old cursor
                                dot( oldCursorX, oldCursorY, &cellColour );
                                GameOfLife.player[i].oldCursor =
GameOfLife.player[i].cursor;

                                // draw new cursors
                                if( GameOfLife.player[i].cellsPlaced )
                                {
                                        if( (*bufferPtr) & readMask )
                                                dot( cursorX, cursorY,
&cursorSelectColour );
                                        else
                                                dot( cursorX, cursorY, &cursorColour );

                                // player has placed all cells for this round, remove
cursor
                                }else{
                                        cellColour = BLACK;
                                        if( (*bufferPtr) &
(0x55<<GameOfLife.bufferOffset) )
                                        {
                                                if( (*bufferPtr) &
(0x01<<GameOfLife.bufferOffset) ) cellColour = GREEN;
                                                if( (*bufferPtr) &
(0x04<<GameOfLife.bufferOffset) ) cellColour = RED;
                                                if( (*bufferPtr) &
(0x10<<GameOfLife.bufferOffset) ) cellColour = BLUE;
                                                if( (*bufferPtr) &
(0x40<<GameOfLife.bufferOffset) ) cellColour = YELLOW;
                                        }
                                        dot( cursorX, cursorY, &cellColour );
                                }

                        }
                }

                // check if any players have any cells left
                unsigned char x;
                for( i = 0; i != 4; i++ )
                {
                        if( GameOfLife.player[i].cellsPlaced ) break;

                // time to update
                } if( i == 4 )
                {

                        // compute next frame
                        computeNextCycle();

                        // reset player cell counters
                        for( i = 0; i != 4; i++ )
                        {
                                if( ((*GameOfLife.playerCount) & (1<<(i-1)) ) || i == 0
)
                                {

                                        // check if player has any more cells left on
the field
```

```
                                                x = 0;
                                                do{
                                                        if( (*(GameOfLife.frameBuffer+x)) &
((1<<(i<<1))<<GameOfLife.bufferOffset) ) break;
                                                        x++;
                                                }while( x != 0 );
                                                if( x != 0 ) GameOfLife.player[i].cellsPlaced =
2;
                                        }
                                }

                                // only one more player active
                                x = 0;
                                for( i = 0; i != 4; i++ )
                                {
                                        if( GameOfLife.player[i].cellsPlaced ) x++;
                                } if( x == 1 ) GameOfLife.state = GAMEOFLIFE_STATE_WINNER;
                        }

                        // update display
                        send();

                        // end game with menu button
                        if( player1ButtonMenu() ) endGame();

                        break;

                // winner state
                case GAMEOFLIFE_STATE_WINNER :

                        // end game with menu button
                        if( player1ButtonMenu() ) endGame();

                        break;

                // winner state
                case GAMEOFLIFE_STATE_WINNER2 :

                        // end game with menu button
                        if( player1ButtonMenu() ) endGame();

                        break;

                default: break;
        }
}

// ------------------------------------------------------------------------
// draws the menu icon for game of life
void drawGameOfLifeMenuIcon( void )
{
        unsigned char x, y, i;
        for( i = 0; i != 40; i++ )
        {
                x = rnd() & 0x0F;
                y = rnd() & 0x0F;
                if( x < 3 ) x = 3;
                if( x > 12 ) x = 3+(x-12);
                if( y < 3 ) y = 3;
                if( y > 12 ) y = 3+(y-12);
                dot( &x, &y, &GREEN );
        }
```

```
}

// ------------------------------------------------------------------------
// randomizes buffer
void randomizeFrameBuffer( void )
{
	unsigned char x = 0;
	do{
		*(GameOfLife.frameBuffer+x) = ((rnd() >> 4)&0x01);
		x++;
	}while( x != 0 );
}

// ------------------------------------------------------------------------
// draws entire buffer without checking for differences
void gameOfLifeDrawFrameBufferNoCheck()
{

	// render pixels
	unsigned char x, y;
	for( x = 0; x != 16; x++ )
	{
		for( y = 0; y != 16; y++ )
		{
			unsigned char buffer = (*(GameOfLife.frameBuffer+y+(x*16)));
			if( buffer & (0x01 << GameOfLife.bufferOffset ) ) dot( &x, &y,
&GREEN );
			if( buffer & (0x04 << GameOfLife.bufferOffset ) )  dot( &x, &y,
&RED );
			if( buffer & (0x10 << GameOfLife.bufferOffset ) ) dot( &x, &y,
&BLUE );
			if( buffer & (0x40 << GameOfLife.bufferOffset ) ) dot( &x, &y, &YELLOW
);
			if( (buffer & (0x55 << GameOfLife.bufferOffset)) == 0 ) dot( &x, &y,
&BLACK );
		}
	}
}

// ------------------------------------------------------------------------
// renders entire frame buffer with out checks and custom colours
void gameOfLifeDrawFrameBufferCustom( const unsigned short* colour1, const unsigned short*
colour2, const unsigned short* colour3, const unsigned short* colour4 )
{

	// render pixels
	unsigned char x, y;
	for( x = 0; x != 16; x++ )
	{
		for( y = 0; y != 16; y++ )
		{
			unsigned char buffer = (*(GameOfLife.frameBuffer+y+(x*16)));
			if( buffer & (0x01 << GameOfLife.bufferOffset ) ) dot( &x, &y, colour1
);
			if( buffer & (0x04 << GameOfLife.bufferOffset ) ) dot( &x, &y, colour2
);
			if( buffer & (0x10 << GameOfLife.bufferOffset ) ) dot( &x, &y, colour3
);
			if( buffer & (0x40 << GameOfLife.bufferOffset ) ) dot( &x, &y, colour4
);
```

```
                        if( (buffer & (0x55 << GameOfLife.bufferOffset)) == 0 ) dot( &x, &y,
&BLACK );
                }
        }
}

// ------------------------------------------------------------------------
// draws the buffer for all players
// optimized to only update the pixels that have changed
void gameOfLifeDrawFrameBuffer( void )
{

        // render pixels
        unsigned char x, y;
        unsigned char buffer;
        for( x = 0; x != 16; x++ )
        {
                for( y = 0; y != 16; y++ )
                {

                        // get buffer content
                        buffer = (*(GameOfLife.frameBuffer+y+(x<<4)));

                        // player 1 - draw new pixels
                        if( (buffer & (0x01 << GameOfLife.bufferOffset)) )
                        {
                                if( (buffer & (0x02 >> GameOfLife.bufferOffset)) == 0 )
dot( &x, &y, &GREEN );
                        }else{

                                // player 2 - draw new pixels
                                if( (buffer & (0x04 << GameOfLife.bufferOffset)) )
                                {
                                        if( (buffer & (0x08 >> GameOfLife.bufferOffset)) == 0 )
dot( &x, &y, &RED );
                                }else{

                                        // player 3 - draw new pixels
                                        if( (buffer & (0x10 << GameOfLife.bufferOffset)) )
                                        {
                                                if( (buffer & (0x20 >>
GameOfLife.bufferOffset)) == 0 ) dot( &x, &y, &BLUE );
                                        }else{

                                                // player 4 - draw new pixels
                                                if( (buffer & (0x40 <<
GameOfLife.bufferOffset)) )
                                                {
                                                        if( (buffer & (0x80 >>
GameOfLife.bufferOffset)) == 0 ) dot( &x, &y, &YELLOW );
                                                }else{

                                                        // clear old pixels
                                                        if( buffer & (0xAA >>
GameOfLife.bufferOffset) ) dot( &x, &y, &BLACK );
                                                }
                                        }
                                }
                        }

                }
        }
```

```
}

// ------------------------------------------------------------------------
// computes next cycle of evolution
void computeNextCycle( void )
{

	// local variables
	register unsigned char count;
	unsigned char readMask;
	register unsigned char i, x, y;
	unsigned short* bufferPtr;

	// loop through all cells
	for( x = 0; x != 16; x++ )
	{
		for( y = 0; y != 16; y++ )
		{

			// create read mask
			readMask = (0x55 << GameOfLife.bufferOffset);

			// count adjacent cells (general for all players)
			count = 0;
			if( (*(GameOfLife.frameBuffer + ((y+1)&0x0F) + (((x+1)&0x0F)<<4) )) &
readMask ) count++;
			if( (*(GameOfLife.frameBuffer + ((y+1)&0x0F) + (x<<4)            )) &
readMask ) count++;
			if( (*(GameOfLife.frameBuffer + ((y+1)&0x0F) + (((x-1)&0x0F)<<4) )) &
readMask ) count++;
			if( (*(GameOfLife.frameBuffer + (y&0x0F)     + (((x-1)&0x0F)<<4) )) &
readMask ) count++;
			if( (*(GameOfLife.frameBuffer + ((y-1)&0x0F) + (((x-1)&0x0F)<<4) )) &
readMask ) count++;
			if( (*(GameOfLife.frameBuffer + ((y-1)&0x0F) + (x<<4)            )) &
readMask ) count++;
			if( (*(GameOfLife.frameBuffer + ((y-1)&0x0F) + (((x+1)&0x0F)<<4) )) &
readMask ) count++;
			if( (*(GameOfLife.frameBuffer + (y&0x0F)     + (((x+1)&0x0F)<<4) )) &
readMask ) count++;

			// get buffer pointer
			bufferPtr = (GameOfLife.frameBuffer + y + (x<<4));

			// current cell is alive
			if( (*bufferPtr) & readMask )
			{

				// less than 2 neighbours or more than 3 neighbours kills it
				if( count < 2 || count > 3 )
				{
					(*bufferPtr) &= readMask;

				// cell remains alive
				}else{
					if( GameOfLife.bufferOffset ) (*bufferPtr) |=
(((*bufferPtr) & readMask)>>1); else (*bufferPtr) |= (((*bufferPtr) & readMask)<<1);
				}

			// current cell is dead
			}else{
```

```
                                // has 3 neighbours, new cell is born
                                if( count == 3 )
                                {

                                        // determine who it belongs to by counting who has the
most adjacent cells
                                        for( i = 0; i != 4; i++ )
                                        {

                                                // counts surrounding cells of current player
                                                count = 0;
                                                readMask =
(1<<(i<<1))<<GameOfLife.bufferOffset;
                                                if( (*(GameOfLife.frameBuffer + ((y+1)&0x0F) +
(((x+1)&0x0F)<<4) )) & readMask ) count++;
                                                if( (*(GameOfLife.frameBuffer + ((y+1)&0x0F) +
(x<<4)            )) & readMask ) count++;
                                                if( (*(GameOfLife.frameBuffer + ((y+1)&0x0F) +
(((x-1)&0x0F)<<4) )) & readMask ) count++;
                                                if( (*(GameOfLife.frameBuffer + (y&0x0F)     +
(((x-1)&0x0F)<<4) )) & readMask ) count++;
                                                if( (*(GameOfLife.frameBuffer + ((y-1)&0x0F) +
(((x-1)&0x0F)<<4) )) & readMask ) count++;
                                                if( (*(GameOfLife.frameBuffer + ((y-1)&0x0F) +
(x<<4)            )) & readMask ) count++;
                                                if( (*(GameOfLife.frameBuffer + ((y-1)&0x0F) +
(((x+1)&0x0F)<<4) )) & readMask ) count++;
                                                if( (*(GameOfLife.frameBuffer + (y&0x0F)     +
(((x+1)&0x0F)<<4) )) & readMask ) count++;

                                                // more than one cell and we have found our
winner
                                                if( count > 1 ) break;
                                        }

                                        // rare case where 3 players surround the same cell -
cell remains dead
                                        if( i == 4 )
                                        {
                                                (*bufferPtr) &=
(0x55<<GameOfLife.bufferOffset);

                                        // spawn cell
                                        }else{
                                                (*bufferPtr) |= ((1<<(i<<1)) <<
(1>>GameOfLife.bufferOffset));;
                                        }

                                // cell remains dead
                                }else{
                                        (*bufferPtr) &= readMask;
                                }
                        }

                }
        }

        // flip buffers
        GameOfLife.bufferOffset = 1 - GameOfLife.bufferOffset;

        // draw buffer
        gameOfLifeDrawFrameBuffer();
```

```
        send();

    }
    #endif // GAME_ENABLE_GAME_OF_LIFE
```

7.2.2. Display

7.2.2.1. main.h

```
// ------------------------------------------------------------------------
// Include files
// ------------------------------------------------------------------------

#ifndef _MAIN_H_
	#define _MAIN_H_

#include "init.h"
#include "render.h"
#include "drawUtils.h"
#include "uart.h"

#endif // _MAIN_H_
```

7.2.2.2. main.c

```
// -----------------------------------------------------------------------
// LED Matrix Driver
// -----------------------------------------------------------------------
// Programmed by      : Alex Murray
//                      Marcel Kaltenrieder
// -----------------------------------------------------------------------
// This program will render a 16x16 matrix of RGB LEDs.
//
// This module can communicate via UART with other devices.
// Check the documentation for protocol details.
// -----------------------------------------------------------------------

/*
-------------------------------------------------------------------------
   Pin Layout
-------------------------------------------------------------------------


                                 _____________________
                                |     MSP430F5172     |
                                |                     |
                            ----| P1.0           P3.0 |---- DS_R_0                OUT/D
                      TxD   ----| P1.1           P3.1 |---- DS_R_1                OUT/D
                      RxD   ----| P1.2           P3.2 |---- DS_G_0                OUT/D
                            ----| P1.3           P3.3 |---- DS_G_1                OUT/D
                            ----| P1.4           P3.4 |---- DS_B_0                OUT/D
                            ----| P1.5           P3.5 |---- DS_B_1                OUT/D
                            ----| P1.6           P3.6 |---- SHCP                  OUT/D
                            ----| P1.7           P3.7 |---- STCP                  OUT/D
                                |                     |
  OUT/D              ROW_0 ----| P2.0                |
  OUT/D              ROW_1 ----| P2.1                |
  OUT/D              ROW_2 ----| P2.2                |
  OUT/D              ROW_3 ----| P2.3                |
  OUT/D            /ROW_EN ----| P2.4                |
                            ----| P2.5                |
                            ----| P2.6                |
                            ----| P2.7                |
                                |                     |
                                |_____________________|


-------------------------------------------------------------------------
   Pin description
-------------------------------------------------------------------------

     Pin Name            | Description
    ---------------------+-------------------------------------------------------------
     TxD                 | For serial communication
     RxD                 |
    ---------------------+-------------------------------------------------------------
     ROW_0               | Selects the active row to write to.
     ROW_1               | These 4 bits are externally demultiplexed to 16 bits.
     ROW_2               |
     ROW_3               |
    ---------------------+-------------------------------------------------------------
     /ROW_EN             | When set to 1, all LEDs are deactivated
    ---------------------+-------------------------------------------------------------
     DS_R_0              | Lower byte for Red colour data to write to shift registers
     DS_R_1              | Higher byte for Red colour data to write to shift registers
```

```
    DS_G_0            | Lower byte for Green colour data to write to shift registers
    DS_G_1            | Higher byte for Green colour data to write to shift registers
    DS_B_0            | Lower byte for Blue colour data to write to shift registers
    DS_B_1            | Higher byte for Blue colour data to write to shift registers
   -------------------+-------------------------------------------------------------
    SHCP              | Serial data is read in on positive edge
    STCP              | Serial data is applied to outputs on positive edge

*/

// ----------------------------------------------------------------------
// Include files
// ----------------------------------------------------------------------

// main header
#include "common.h"
#include "main.h"

// ----------------------------------------------------------------------
// global variables & arrays
// ----------------------------------------------------------------------

// see main.h for more info
volatile unsigned char pixelArray[8][16][PWM_RESOLUTION];
unsigned char ROW_EN;

// ----------------------------------------------------------------------
// Main entry point
// ----------------------------------------------------------------------

void main( void )
{

       // Initialse device
       initDevice();

       // initialise screen
       cls();
/*
       // annoy zingg
       dot( 0x2F, 0xEEE );
       dot( 0x1E, 0xEEE );
       dot( 0x0D, 0xEEE );
       dot( 0x0C, 0xEEE );
       dot( 0x0B, 0xEEE );
       dot( 0x0A, 0xEEE );
       dot( 0x19, 0xEEE );
       dot( 0x28, 0xEEE );
       dot( 0x38, 0xEEE );
       dot( 0x48, 0xEEE );
       dot( 0x58, 0xEEE );
       dot( 0x69, 0xEEE );
       dot( 0x7A, 0xEEE );
       dot( 0x7B, 0xEEE );
       dot( 0x7C, 0xEEE );
       dot( 0x7D, 0xEEE );
       dot( 0x6E, 0xEEE );
       dot( 0x5F, 0xEEE );
       dot( 0x4F, 0xEEE );
       dot( 0x3F, 0xEEE );

       dot( 0x2A, 0x00E );
```

```
        dot( 0x5A, 0x00E );
        dot( 0x2B, 0x00E );
        dot( 0x5B, 0x00E );

        dot( 0x2D, 0x0EE );
        dot( 0x3E, 0x0EE );
        dot( 0x4E, 0x0EE );
        dot( 0x5D, 0x0EE );
*/

/*
        unsigned char x1, y1, x2, y2;
        unsigned short colour1, colour2, colour3, colour4;
        x1 = 0; y1 = 0; x2 = 15; y2 = 15;
        colour1 = 0xEE0; colour2 = 0xE00; colour3 = 0x0E0; colour4 = 0x00E;
        blendColourFillBox( &x1, &y1, &x2, &y2, &colour1, &colour2, &colour3, &colour4 );
        x1 = 7; y1 = 7; x2 = 7; colour1 = 0xEEE;
        circle( &x1, &y1, &x2, &colour1 );
*/


        // main loop
        while( 1 )
        {
        }
}

// ------------------------------------------------------------------------
// Utility functions and other stuff
// ------------------------------------------------------------------------

// returns the absolute value of a number
unsigned char absolute( signed char value )
{
        if( value & 0x80 ) return 0-value;
        return value;
}

// The question is: Will it Blend?
extern inline unsigned short blendColours( unsigned short colour1, unsigned short colour2,
unsigned char startPosition, unsigned char endPosition, unsigned char position, unsigned
char blendDistance )
{
        return ((((colour1&0xF00)>>8)*(endPosition-position)/blendDistance +
((colour2&0xF00)>>8)*(position-startPosition)/blendDistance)<<8) |
((((colour1&0x0F0)>>4)*(endPosition-position)/blendDistance +
((colour2&0x0F0)>>4)*(position-startPosition)/blendDistance)<<4) |
((colour1&0x00F)*(endPosition-position)/blendDistance + (colour2&0x00F)*(position-
startPosition)/blendDistance);
}

// returns true if the point is out of bounds of the screen
extern inline unsigned char isOffScreen( unsigned char x, unsigned char y )
{
        return (x&0xF0 || y&0xF0);
}
```

7.2.2.3. init.h

```
// -------------------------------------------------------------------------
// Initialisations
// -------------------------------------------------------------------------

#ifndef _INIT_H_
        #define _INIT_H_

// initialising functions
void initDevice( void );
void cfgPort1( void );
void cfgPort2( void );
void cfgPort3( void );
void cfgUART( void );
void cfgSystemClock( void );
void cfgTimerA( void );

#endif // _INIT_H_
```

7.2.2.4. init.c

```
// -------------------------------------------------------------------------
// Initialisations
// -------------------------------------------------------------------------

// header files
#include "common.h"
#include "init.h"
#include "uart.h"
#include "drawUtils.h"

//
------------------------------------------------------------------------------------------------
----------------------
// call this to initialise the device
void initDevice( void )
{

        // Stop watchdog timer to prevent time out reset
        WDTCTL = WDTPW + WDTHOLD;

        // setup clock
        cfgSystemClock();

        // configure timers
        cfgTimerA();

        // configure ports
        cfgPort1();
        cfgPort2();
        cfgPort3();

        // configure UART serial interface
        cfgUART();

        // initialise some variables
        UART.commandState = CMD_STATE_NOP;
        UART.commandStateGroup = CMD_STATE_NOP;
        drawUtils.blendMode = BLEND_MODE_REPLACE;

        // enable global interrupts
        __bis_SR_register(GIE);
}

void cfgPort1( void )
{
        P1SEL = 0x06;                 // select TxD and RxD
}

void cfgPort2( void )
{
        P2DIR |= 0x1F;
        P2SEL = 0xE0;
        P2OUT = 0x00;
}

void cfgPort3( void )
{
        P3DIR |= 0xFF;
        P3SEL = 0x00;
```

```
        ROW_EN = ENABLE_ROW;
}

void cfgSystemClock( void )
{

        // set up internal clock
        UCSCTL3 |= SELREF_2;                        // Set DCO FLL reference = REFO
        UCSCTL4 |= SELA_2;                          // Set ACLK = REFO

        __bis_SR_register(SCG0);                    // Disable the FLL control loop
        UCSCTL0 = 0x0000;                           // Set lowest possible DCOx, MODx
        UCSCTL1 = DCORSEL_5;                        // Select DCO range 24MHz operation
        UCSCTL2 = 0x0300;                           // Set DCO Multiplier for 30MHz

        __bic_SR_register(SCG0);                    // re-enable the FLL control loop

        // Worst-case settling time for the DCO when the DCO range bits have been
        // changed is n x 32 x 32 x f_MCLK / f_FLL_reference. See UCS chapter in 5xx
        // UG for optimization.
        // 32 x 32 x 12 MHz / 32,768 Hz = 375000 = MCLK cycles for DCO to settle
        __delay_cycles(375000);

        // Loop until XT1 & DCO fault flag is cleared
        do
        {
                UCSCTL7 &= ~(XT1LFOFFG + XT1HFOFFG + DCOFFG);
                                                    // Clear XT1,DCO fault flags
                SFRIFG1 &= ~OFIFG;                  // Clear fault flags
        }while (SFRIFG1&OFIFG);                             // Test oscillator fault flag
}

void cfgTimerA( void )
{
        TA0CCTL0 = CCIE;                            // CCR0 interrupt enabled
        TA0CCR0 = 35000;
        TA0CTL = TASSEL_2 + MC_1 + TACLR + 0x00C0;  // SMCLK, upmode, clear TAR
}

void cfgUART( void )
{

        // configure UART
        UCA0CTL1 |= UCSWRST;                        // **Put state machine in reset**
        UCA0CTL1 |= UCSSEL_2;                       // SMCLK (29088000)
        UCA0BR0 = 0x07;                                     // 0xDD for Baud 115'200, 0x07
for Baud 3'686'400
        UCA0BR1 = 0x00;
        UCA0MCTL |= UCBRS_1 + UCBRF_0;                      // Modulation UCBRSx=1, UCBRFx=0
        UCA0CTL1 &= ~UCSWRST;                       // **Initialize USCI state machine**
        UCA0IE |= UCRXIE;                           // Enable USCI_A0 RX interrupt
}
```

7.2.2.5. drawutils.h

```
// ----------------------------------------------------------------------
// Drawing Utilities
// ----------------------------------------------------------------------

#ifndef _DRAWUTILS_H_
	#define _DRAWUTILS_H_

// blend modes
enum blendMode_e
{
	BLEND_MODE_REPLACE,
	BLEND_MODE_ADD,
	BLEND_MODE_SUBTRACT,
	BLEND_MODE_MULTIPLY
};

// struct
struct drawUtils_t
{
	unsigned char blendMode;
};
extern struct drawUtils_t drawUtils;

// function prototypes
void cls( void );
void dot( unsigned char* x, unsigned char* y, unsigned short* rgb );
void blendColourBox( unsigned char* x1, unsigned char* y1, unsigned char* x2, unsigned char*
y2, unsigned short* topLeftColour, unsigned short* bottomLeftColour, unsigned short*
topRightColour, unsigned short* bottomRightColour );
void blendColourFillBox( unsigned char* x1, unsigned char* y1, unsigned char* x2, unsigned
char* y2, unsigned short* topLeftColour, unsigned short* bottomLeftColour, unsigned short*
topRightColour, unsigned short* bottomRightColour );
void box( unsigned char* x1, unsigned char* y1, unsigned char* x2, unsigned char* y2,
unsigned short* colour );
void fillBox( unsigned char* x1, unsigned char* y1, unsigned char* x2, unsigned char* y2,
unsigned short* colour );
void blendColourLine( unsigned char* x1, unsigned char* y1, unsigned char* x2, unsigned
char* y2, unsigned short* colour1, unsigned short* colour2 );
void line( unsigned char* x1, unsigned char* y1, unsigned char* x2, unsigned char* y2,
unsigned short* colour );
void _circle_draw8points( unsigned char* cx, unsigned char* cy, signed char* x, signed char*
y, unsigned short* colour );
void circle( unsigned char* x, unsigned char* y, unsigned char* radius, unsigned short*
colour );
void fillCircle( unsigned char* x, unsigned char* y, unsigned char* radius, unsigned short*
colour );
void blendColourFillCircle( unsigned char* x, unsigned char* y, unsigned char* radius,
unsigned short* insideColour, unsigned short* outsideColour );
void drawUtils_SetBlendMode( unsigned char blendMode );
extern inline void _process_blend_mode( volatile unsigned char* pixelArray, unsigned char*
blendMode, unsigned char* colour, unsigned char* pwm );

#endif // _DRAWUTILS_H_
```

7.2.2.6. drawutils.c

```
// ------------------------------------------------------------------------
// Drawing Utilities
// ------------------------------------------------------------------------

// include files
#include "common.h"
#include "drawUtils.h"

// global variables
struct drawUtils_t drawUtils;

//
-------------------------------------------------------------------------------------------------
----------------------
// Sets the blending mode. Available modes: BLEND_MODE_REPLACE *** BLEND_MODE_ADD ***
BLEND_MODE_SUBTRACT *** BLEND_MODE_MULTIPLY
void drawUtils_SetBlendMode( unsigned char blendMode )
{
	drawUtils.blendMode = blendMode;
}

//
-------------------------------------------------------------------------------------------------
----------------------
// Will draw a filled box from x1,y1 to x2,y2 with a different colour for each corner and
blend between them
void blendColourFillBox( unsigned char* x1, unsigned char* y1, unsigned char* x2, unsigned
char* y2, unsigned short* topLeftColour, unsigned short* bottomLeftColour, unsigned short*
topRightColour, unsigned short* bottomRightColour )
{

	// local variables
	unsigned char x, y;
	unsigned short finalColour;
	unsigned char blendFactorX = *x2 - *x1;
	unsigned char blendFactorY = *y2 - *y1;

	// blend entire top and bottom rows and store in array
	unsigned char blendTable[3][2];
	for( x = *x1; x <= *x2; x++ )
	{
	}

	// calculates all pixels between the four points
	for( x = *x1; x <= *x2; x++ )
	{

		// calculate x blend
		blendTable[0][0] = (((*topLeftColour&0xF00)>>8)    * (*x2-x)/blendFactorX) +
(((*topRightColour&0xF00)>>8)    * (x-*x1)/blendFactorX);
		blendTable[1][0] = (((*topLeftColour&0x0F0)>>4)    * (*x2-x)/blendFactorX) +
(((*topRightColour&0x0F0)>>4)    * (x-*x1)/blendFactorX);
		blendTable[2][0] = (( *topLeftColour&0x00F)        * (*x2-x)/blendFactorX) +
(( *topRightColour&0x00F)        * (x-*x1)/blendFactorX);
		blendTable[0][1] = (((*bottomLeftColour&0xF00)>>8) * (*x2-x)/blendFactorX) +
(((*bottomRightColour&0xF00)>>8) * (x-*x1)/blendFactorX);
		blendTable[1][1] = (((*bottomLeftColour&0x0F0)>>4) * (*x2-x)/blendFactorX) +
(((*bottomRightColour&0x0F0)>>4) * (x-*x1)/blendFactorX);
```

```
                blendTable[2][1] = (( *bottomLeftColour&0x00F)     * (*x2-x)/blendFactorX) +
(( *bottomRightColour&0x00F)     * (x-*x1)/blendFactorX);
                for( y = *y1; y <= *y2; y++ )
                {

                        // blend y coordinates
                        finalColour =  (blendTable[0][0] * (*y2-y)/blendFactorY +
(blendTable[0][1]) * (y-*y1)/blendFactorY)<<8;
                        finalColour |= (blendTable[1][0] * (*y2-y)/blendFactorY +
(blendTable[1][1]) * (y-*y1)/blendFactorY)<<4;
                        finalColour |=  blendTable[2][0] * (*y2-y)/blendFactorY +
(blendTable[2][1]) * (y-*y1)/blendFactorY;

                        // set pixel
                        dot( &x, &y, &finalColour );
                }
        }

        // return
        return;
}

//
-----------------------------------------------------------------------------------------------
----------------------
// Will draw the outline of a box from x1,y1 to x2,y2 with a different colour for each
corner and blend between them
void blendColourBox( unsigned char* x1, unsigned char* y1, unsigned char* x2, unsigned char*
y2, unsigned short* topLeftColour, unsigned short* bottomLeftColour, unsigned short*
topRightColour, unsigned short* bottomRightColour )
{

        // local variables
        unsigned char x;
        unsigned char blendFactorX = *x2 - *x1;
        unsigned char blendFactorY = *y2 - *y1;
        unsigned short finalColour;

        // blend top and bottom rows
        for( x = *x1; x <= *x2; x++ )
        {
                finalColour = blendColours( *topLeftColour, *topRightColour, *x1, *x2, x,
blendFactorX); dot( &x, y1, &finalColour );
                finalColour = blendColours( *bottomLeftColour, *bottomRightColour, *x1, *x2,
x, blendFactorX); dot( &x, y2, &finalColour );
        }

        // blend left and right columns
        for( x = *y1; x <= *y2; x++ )
        {
                finalColour = blendColours( *topLeftColour, *bottomLeftColour, *y1, *y2, x,
blendFactorY); dot( x1, &x, &finalColour );
                finalColour = blendColours( *topRightColour, *bottomRightColour, *y1, *y2, x,
blendFactorY); dot( x2, &x, &finalColour );
        }

        // return
        return;
}
```

```
//
-----------------------------------------------------------------------------------------------
----------------------
// will draw the outline of a box with one colour
void box( unsigned char* x1, unsigned char* y1, unsigned char* x2, unsigned char* y2,
unsigned short* colour )
{

        // local variables
        unsigned char x;

        // draw box
        for( x = *x1; x <= *x2; x++ )
        {
                dot( &x, y1, colour );
                dot( &x, y2, colour );
        }
        for( x = *y1; x <= *y2; x++ )
        {
                dot( x1, &x, colour );
                dot( x2, &x, colour );
        }

        // return
        return;
}

//
-----------------------------------------------------------------------------------------------
----------------------
// will draw a filled box with one colour
void fillBox( unsigned char* x1, unsigned char* y1, unsigned char* x2, unsigned char* y2,
unsigned short* colour )
{

        // local variables
        unsigned char x, y;

        // draw box
        for( x = *x1; x <= *x2; x++ )
        {
                for( y = *y1; y <= *y2; y++ )
                {
                        dot( &x, &y, colour );
                }
        }

        // return
        return;
}

//
-----------------------------------------------------------------------------------------------
----------------------
// will draw a line from point A to point B with one colour
void line( unsigned char* x1, unsigned char* y1, unsigned char* x2, unsigned char* y2,
unsigned short* colour )
{

        // get delta
        unsigned char dx = absolute(*x2-*x1);
        unsigned char dy = absolute(*y2-*y1);
```

```
        signed char sx, sy;
        if( *x1 < *x2 ) sx=1; else sx=-1;
        if( *y1 < *y2 ) sy=1; else sy=-1;
        signed char err = (signed char)(dx - dy);
        signed char e2;

        // plot line
        while(1)
        {
                dot( x1, y1, colour );
                if( *x1 == *x2 && *y1 == *y2 ) break;
                e2 = 2*err;
                if( e2 > 0-dy )
                {
                        err -= dy;
                        *x1 += sx;
                }
                if( e2 < dx )
                {
                        err += dx;
                        *y1 += sy;
                }
        }

        // return
        return;
}

//
-------------------------------------------------------------------------------------------
----------------------
// will draw a line from point A to point B and blend between two colours
void blendColourLine( unsigned char* x1, unsigned char* y1, unsigned char* x2, unsigned
char* y2, unsigned short* colour1, unsigned short* colour2 )
{

        // get delta
        unsigned char dx = absolute(*x2-*x1);
        unsigned char dy = absolute(*y2-*y1);
        signed char sx, sy;
        if( *x1 < *x2 ) sx=1; else sx=-1;
        if( *y1 < *y2 ) sy=1; else sy=-1;
        signed char err = (signed char)(dx - dy);
        signed char e2;
        unsigned short finalColour;

        // plot line
        unsigned char x=*x1, y=*y1, blendStart, blendEnd;
        while(1)
        {

                // blend colours using longest distance and draw line
                if( dx > dy )
                {
                        if( *x2 > *x1 ){ blendStart = *x1; blendEnd = *x2; }else{ blendStart =
*x2; blendEnd = *x1; }
                        finalColour = blendColours( *colour1, *colour2, blendStart, blendEnd,
x, dx);
                        dot( &x, &y, &finalColour );
                }else{
                        if( *y2 > *y1 ){ blendStart = *y1; blendEnd = *y2; }else{ blendStart =
*y2; blendEnd = *y1; }
```

```
                        finalColour = blendColours( *colour1, *colour2, blendStart, blendEnd,
y, dy);
                        dot( &x, &y, &finalColour );
                }

                // draw line
                if( x == *x2 && y == *y2 ) break;
                e2 = 2*err;
                if( e2 > 0-dy )
                {
                        err -= dy;
                        x += sx;
                }
                if( e2 < dx )
                {
                        err += dx;
                        y += sy;
                }
        }

        // return
        return;
}

//
-----------------------------------------------------------------------------------------------
----------------------
// will draw an empty circle with one colour
void circle( unsigned char* cx, unsigned char* cy, unsigned char* radius, unsigned short*
colour )
{

        // local variables
        signed char f = 1- *radius;
        signed char ddF_x = 1;
        signed char ddF_y = (-2)*(*radius);
        signed char x = 0;
        signed char y = *radius;

        // draw first 8 points
        _circle_draw8points( cx, cy, &x, &y, colour );

        // Bresenham Algorithm
        while ( x < y )
        {
                if( f >= 0 )
                {
                        y--;
                        ddF_y += 2;
                        f += ddF_y;
                }

                x++;
                ddF_x += 2;
                f += ddF_x;
                _circle_draw8points( cx, cy, &x, &y, colour );
        }

        // return
        return;
}
```

```
//
-----------------------------------------------------------------------------------------------
----------------------
// will draw a filled circle with one colour
void fillCircle( unsigned char* x, unsigned char* y, unsigned char* radius, unsigned short*
colour )
{

	// draw circles with decreasing radius
	unsigned char x1, i1;
	for( unsigned char i = *radius; i != 0; i-- )
	{
		circle( x, y, &i, colour );
		x1 = *x+1;
		i1 = i-1;
		circle( &x1, y, &i1, colour );
	}

	// return
	return;
}

//
-----------------------------------------------------------------------------------------------
----------------------
// will draw a filled circle and blend between an outer and inner colour
void blendColourFillCircle( unsigned char* x, unsigned char* y, unsigned char* radius,
unsigned short* insideColour, unsigned short* outsideColour )
{

	// draw circles with decreasing radius
	unsigned char x1, i1;
	unsigned short finalColour;
	for( unsigned char i = *radius; i != 0; i-- )
	{
		finalColour = blendColours( *insideColour, *outsideColour, 0, *radius, i,
*radius );
		circle( x, y, &i, &finalColour );
		x1 = *x+1;
		i1 = i-1;
		circle( &x1, y, &i1, &finalColour );
	}

	// dot in centre
	dot( x, y, insideColour );

	// fix two dots (observant fix)
	finalColour = blendColours( *insideColour, *outsideColour, 0, *radius, 2, *radius );
	x1 = *x-1;
	i1 = *y-1; if( isOffScreen( x1, i1 ) ){}else{ dot( &x1, &i1, &finalColour ); }
	i1 = *y+1; if( isOffScreen( x1, i1 ) ){}else{ dot( &x1, &i1, &finalColour ); }

	// return
	return;
}

void _circle_draw8points( unsigned char* cx, unsigned char* cy, signed char* x, signed char*
y, unsigned short* colour )
{
	unsigned char nx, ny;
	nx = *cx+*x; ny = *cy+*y; if( isOffScreen( nx, ny ) ){} else { dot( &nx, &ny,
colour ); }
```

```
        nx = *cx-*x; ny = *cy+*y; if( isOffScreen( nx, ny ) ){} else { dot( &nx, &ny,
colour ); }
        nx = *cx+*x; ny = *cy-*y; if( isOffScreen( nx, ny ) ){} else { dot( &nx, &ny,
colour ); }
        nx = *cx-*x; ny = *cy-*y; if( isOffScreen( nx, ny ) ){} else { dot( &nx, &ny,
colour ); }
        nx = *cx+*y; ny = *cy+*x; if( isOffScreen( nx, ny ) ){} else { dot( &nx, &ny,
colour ); }
        nx = *cx-*y; ny = *cy+*x; if( isOffScreen( nx, ny ) ){} else { dot( &nx, &ny,
colour ); }
        nx = *cx+*y; ny = *cy-*x; if( isOffScreen( nx, ny ) ){} else { dot( &nx, &ny,
colour ); }
        nx = *cx-*y; ny = *cy-*x; if( isOffScreen( nx, ny ) ){} else { dot( &nx, &ny,
colour ); }
        return;
}

//
-----------------------------------------------------------------------------------------------
----------------------
// will clear the screen
void cls( void )
{
        unsigned char x, y, i;
        for( x = 0; x != 8; x++ )
        {
                for( y = 0; y != 16; y++ )
                {
                        for( i = 0; i != PWM_RESOLUTION; i++ ) pixelArray[x][y][i] = 0x00;
                }
        }
        return;
}

//
-----------------------------------------------------------------------------------------------
----------------------
// Will set a pixel to the specified colour
void dot( unsigned char* x, unsigned char* y, unsigned short* rgb )
{

        // shift bit is set to one if x coordinate > 7 so the pixel is rendered to the other
half of the display
        unsigned char shift = (*x & 0x08) || 0;

        // get x coordinate
        unsigned char x_copy = *x & 0x07;
        unsigned char x_inverse = 7-x_copy;

        // used to extract the data
        register unsigned char i;
        unsigned char colour;
        unsigned char pwm;

        // get pointer to array, should be faster
        volatile unsigned char* pixelArrayPtr = &pixelArray[x_copy][*y][0];
        volatile unsigned char* pixelArrayInversePtr = &pixelArray[x_inverse][*y][0];

        // disable interrupts during array manipulation
        __bic_SR_register( GIE );

        // clears only the pixels we're manipulating in the array
```

```
        // only need to do this with BLEND_MODE_REPLACE
        if( drawUtils.blendMode == BLEND_MODE_REPLACE )
        {
                unsigned char clr1 = 0xF5>>shift, clr2 = 0xDF>>shift;
                for( i = 0; i != PWM_RESOLUTION; i++ )
                {
                        (*(pixelArrayPtr+i)) &= clr1;
                        (*(pixelArrayInversePtr+i)) &= clr2;
                }
        }

        // extracts blue
        pwm = (*rgb) & 0x000F;
        colour = 0x02 >> shift;
        if( drawUtils.blendMode != BLEND_MODE_REPLACE ) _process_blend_mode( pixelArrayPtr,
&drawUtils.blendMode, &colour, &pwm );
        for( i = 0; i != pwm; i++ )
        {
                (*(pixelArrayPtr+i)) |= colour;
        }

        // extracts green
        unsigned char c_rgb = (*rgb)>>4;
        pwm = c_rgb & 0x000F;
        colour = 0x20 >> shift;
        if( drawUtils.blendMode != BLEND_MODE_REPLACE )
_process_blend_mode( pixelArrayInversePtr, &drawUtils.blendMode, &colour, &pwm );
        for( i = 0; i != pwm; i++ )
        {
                (*(pixelArrayInversePtr+i)) |= colour;
        }

        // extracts red
        pwm = c_rgb >> 4;
        colour = 0x08 >> shift;
        if( drawUtils.blendMode != BLEND_MODE_REPLACE ) _process_blend_mode( pixelArrayPtr,
&drawUtils.blendMode, &colour, &pwm );
        for( i = 0; i != pwm; i++ )
        {
                (*(pixelArrayPtr+i)) |= colour;
        }

        // enable interrupts
        __bis_SR_register( GIE );

        // return
        return;
}

//
-----------------------------------------------------------------------------------------------
----------------------
// processes the colour according to the set blend mode, and changes the value of pwm
accordingly
inline void _process_blend_mode( volatile unsigned char* pixelArray, unsigned char*
blendMode, unsigned char* colour, unsigned char* pwm )
{

        // get current pwm value in array
        register unsigned char current_pwm, c;
        for( current_pwm = 0; current_pwm != PWM_RESOLUTION; current_pwm++ )
        {
```

```
                c = ( (*(pixelArray+current_pwm)) & (*colour) );
                if( c == 0 ) break;
        }

        // check which blend mode we're using
        switch( *blendMode )
        {
                // addition
                case BLEND_MODE_ADD :
                        *pwm += current_pwm;
                        if( *pwm > PWM_RESOLUTION ) *pwm = PWM_RESOLUTION;
                        break;

                // subtraction
                case BLEND_MODE_SUBTRACT :
                        *pwm -= current_pwm;
                        if( *pwm > PWM_RESOLUTION ) *pwm = PWM_RESOLUTION;
                        break;

                // multiplication
                case BLEND_MODE_MULTIPLY :
                        *pwm *= current_pwm;
                        if( *pwm > PWM_RESOLUTION ) *pwm = PWM_RESOLUTION;
                        break;

                default:break;
        }

        // return
        return;
}
```

7.2.2.7. render.h

```
// ------------------------------------------------------------------------
// Handels rendering of LEDs
// ------------------------------------------------------------------------

#ifndef _RENDER_H_
        #define _RENDER_H_

void refreshScreen( void );

#endif // _RENDER_H_
```

7.2.2.8. render.c

```
// ---------------------------------------------------------------------
// Handels rendering of LEDs
// ---------------------------------------------------------------------

// include files
#include "common.h"
#include "render.h"

//
---------------------------------------------------------------------------------------------
----------------------
// refreshes the entire screen
void refreshScreen( void )
{

	// local variables
	unsigned char x, y, currentPWMCycle;

	for( y = 0; y != 16; y++ )
	{

		// output current row
		P2OUT = y | ROW_EN;

		// pwm control
		for( currentPWMCycle = 0; currentPWMCycle != PWM_RESOLUTION; currentPWMCycle+
+ )
		{

			// delay as to make LED brightness to PWM ratio linear
			// Lookup table: 255, 180, 128, 90, 64, 45, 32, 23, 16, 12, 8, 6, 4,
3, 2, 1,0
			switch( currentPWMCycle )
			{
				case 1 : __delay_cycles( PWM_DELAY_CYCLES*0 ); break;
				case 2 : __delay_cycles( PWM_DELAY_CYCLES*1 ); break;
				case 3 : __delay_cycles( PWM_DELAY_CYCLES*2 ); break;
				case 4 : __delay_cycles( PWM_DELAY_CYCLES*3 ); break;
				case 5 : __delay_cycles( PWM_DELAY_CYCLES*4 ); break;
				case 6 : __delay_cycles( PWM_DELAY_CYCLES*6 ); break;
				case 7 : __delay_cycles( PWM_DELAY_CYCLES*8 ); break;
				case 8 : __delay_cycles( PWM_DELAY_CYCLES*12 ); break;
				case 9 : __delay_cycles( PWM_DELAY_CYCLES*16 ); break;
				case 10 : __delay_cycles( PWM_DELAY_CYCLES*23 ); break;
				case 11 : __delay_cycles( PWM_DELAY_CYCLES*32 ); break;
				case 12 : __delay_cycles( PWM_DELAY_CYCLES*45 ); break;
				case 13 : __delay_cycles( PWM_DELAY_CYCLES*64 ); break;
				case 14 : __delay_cycles( PWM_DELAY_CYCLES*90 ); break;
				default: break;
			}

			// output serial data
			for( x = 0; x != 8; x++ )
			{

				// NOTE: my god, I hope the shift registers can handle this.
It's torture I say, TORTURE
				// NOTE2: On second thought, they *were* a pain to solder. I
guess it's only fair to punish them
```

```
                                // output next 6 bits
                                P3OUT = pixelArray[x][y][currentPWMCycle];

                                // write them to shift registers
                                P3OUT |= SHIFT_REGISTER_WRITE;
                        }

                        // output shift registers to LEDs
                        P3OUT |= SHIFT_REGISTER_OUTPUT;

                }

                // reset shift registers to 0
                // stops signals from overlapping and causing half-on LEDs
                for( x = 0; x != 8; x++ )
                {
                        P3OUT = 0x00;
                        P3OUT |= SHIFT_REGISTER_WRITE;
                }
                P3OUT |= SHIFT_REGISTER_OUTPUT;

        }

        // return
        return;
}

//
-----------------------------------------------------------------------------------------------
----------------------
// for refreshing the display
#pragma vector=TIMER0_A0_VECTOR
__interrupt void TIMER0_A0_ISR( void )
{
        refreshScreen();
}
```

7.2.2.9. uart.h

```
// ------------------------------------------------------------------------
// UART handling
// ------------------------------------------------------------------------

#ifndef _UART_H_
	#define _UART_H_

// command list
enum commandList_e
{
	CMD_CLS,
	CMD_DOT,
	CMD_BLEND_COLOUR_BOX,
	CMD_BLEND_COLOUR_FILL_BOX,
	CMD_BOX,
	CMD_FILL_BOX,
	CMD_BLEND_COLOUR_LINE,
	CMD_LINE,
	CMD_CIRCLE,
	CMD_FILL_CIRCLE,
	CMD_BLEND_COLOUR_FILL_CIRCLE,

	CMD_SET_BLEND_MODE__REPLACE,
	CMD_SET_BLEND_MODE__ADD,
	CMD_SET_BLEND_MODE__SUBTRACT,
	CMD_SET_BLEND_MODE__MULTIPLY
};

// command states
enum commandState_e
{
	CMD_STATE_NOP,
	CMD_STATE_CLS,

	CMD_STATE_DOT__POSITION,
	CMD_STATE_DOT__COLOUR_MSB,
	CMD_STATE_DOT__COLOUR_LSB,

	CMD_STATE_BLEND_COLOUR_BOX__POSITION_A,
	CMD_STATE_BLEND_COLOUR_BOX__POSITION_B,
	CMD_STATE_BLEND_COLOUR_BOX__COLOUR_A,
	CMD_STATE_BLEND_COLOUR_BOX__COLOUR_AB,
	CMD_STATE_BLEND_COLOUR_BOX__COLOUR_B,
	CMD_STATE_BLEND_COLOUR_BOX__COLOUR_C,
	CMD_STATE_BLEND_COLOUR_BOX__COLOUR_CD,
	CMD_STATE_BLEND_COLOUR_BOX__COLOUR_D,

	CMD_STATE_BLEND_COLOUR_FILL_BOX__POSITION_A,
	CMD_STATE_BLEND_COLOUR_FILL_BOX__POSITION_B,
	CMD_STATE_BLEND_COLOUR_FILL_BOX__COLOUR_A,
	CMD_STATE_BLEND_COLOUR_FILL_BOX__COLOUR_AB,
	CMD_STATE_BLEND_COLOUR_FILL_BOX__COLOUR_B,
	CMD_STATE_BLEND_COLOUR_FILL_BOX__COLOUR_C,
	CMD_STATE_BLEND_COLOUR_FILL_BOX__COLOUR_CD,
	CMD_STATE_BLEND_COLOUR_FILL_BOX__COLOUR_D,

	CMD_STATE_BOX__POSITION_A,
	CMD_STATE_BOX__POSITION_B,
	CMD_STATE_BOX__COLOUR_MSB,
```

```
        CMD_STATE_BOX__COLOUR_LSB,

        CMD_STATE_FILL_BOX__POSITION_A,
        CMD_STATE_FILL_BOX__POSITION_B,
        CMD_STATE_FILL_BOX__COLOUR_MSB,
        CMD_STATE_FILL_BOX__COLOUR_LSB,

        CMD_STATE_BLEND_COLOUR_LINE__POSITION_A,
        CMD_STATE_BLEND_COLOUR_LINE__POSITION_B,
        CMD_STATE_BLEND_COLOUR_LINE__COLOUR_A,
        CMD_STATE_BLEND_COLOUR_LINE__COLOUR_AB,
        CMD_STATE_BLEND_COLOUR_LINE__COLOUR_B,

        CMD_STATE_LINE__POSITION_A,
        CMD_STATE_LINE__POSITION_B,
        CMD_STATE_LINE__COLOUR_MSB,
        CMD_STATE_LINE__COLOUR_LSB,

        CMD_STATE_CIRCLE__POSITION,
        CMD_STATE_CIRCLE__RADIUS,
        CMD_STATE_CIRCLE__COLOUR_MSB,
        CMD_STATE_CIRCLE__COLOUR_LSB,

        CMD_STATE_FILL_CIRCLE__POSITION,
        CMD_STATE_FILL_CIRCLE__RADIUS,
        CMD_STATE_FILL_CIRCLE__COLOUR_MSB,
        CMD_STATE_FILL_CIRCLE__COLOUR_LSB,

        CMD_STATE_BLEND_COLOUR_FILL_CIRCLE__POSITION,
        CMD_STATE_BLEND_COLOUR_FILL_CIRCLE__RADIUS,
        CMD_STATE_BLEND_COLOUR_FILL_CIRCLE__COLOUR_A,
        CMD_STATE_BLEND_COLOUR_FILL_CIRCLE__COLOUR_AB,
        CMD_STATE_BLEND_COLOUR_FILL_CIRCLE__COLOUR_B,

        CMD_STATE_SET_BLEND_MODE__REPLACE,
        CMD_STATE_SET_BLEND_MODE__ADD,
        CMD_STATE_SET_BLEND_MODE__SUBTRACT,
        CMD_STATE_SET_BLEND_MODE__MULTIPLY
};

// structs
struct UART_t
{

        // state machine for incomming commands
        unsigned char commandState;
        unsigned char commandStateGroup;

        // data to be set by the state machine
        // positions
        unsigned char x1;
        unsigned char y1;
        unsigned char x2;
        unsigned char y2;

        // colours
        unsigned short cA;
        unsigned short cB;
        unsigned short cC;
        unsigned short cD;
};
extern struct UART_t UART;
```

```
// function prototypes
unsigned char processCommand( void );
unsigned char error( void );

#endif // _UART_H_
```

7.2.2.10. uart.c

```
// ------------------------------------------------------------------------
// UART handling
// ------------------------------------------------------------------------

// ------------------------------------------------------------------------
// NOTES
// received data is always sent back when the command was processed
// ------------------------------------------------------------------------

// include files
#include "common.h"
#include "uart.h"
#include "drawUtils.h"

// global variables
struct UART_t UART;

//
-----------------------------------------------------------------------------------------------------
----------------------
// process received data
unsigned char processCommand( void )
{

	// extract command to execute, if ready
	if( UART.commandStateGroup == CMD_STATE_NOP )
	{
		if( UCA0RXBUF == CMD_CLS ){
			cls();
			return 1;
		}
		if( UCA0RXBUF == CMD_DOT ){
			UART.commandState = CMD_STATE_DOT__POSITION;
			UART.commandStateGroup = CMD_DOT;
			return 1;
		}
		if( UCA0RXBUF ==  CMD_BLEND_COLOUR_BOX ){
			UART.commandState = CMD_STATE_BLEND_COLOUR_BOX__POSITION_A;
			UART.commandStateGroup = CMD_BLEND_COLOUR_BOX;
			return 1;
		}
		if( UCA0RXBUF ==  CMD_BLEND_COLOUR_FILL_BOX ){
			UART.commandState = CMD_STATE_BLEND_COLOUR_FILL_BOX__POSITION_A;
			UART.commandStateGroup = CMD_BLEND_COLOUR_FILL_BOX;
			return 1;
		}
		if( UCA0RXBUF ==  CMD_BOX){
			UART.commandState = CMD_STATE_BOX__POSITION_A;
			UART.commandStateGroup = CMD_BOX;
			return 1;
		}
		if( UCA0RXBUF == CMD_FILL_BOX){
			UART.commandState = CMD_STATE_FILL_BOX__POSITION_A;
			UART.commandStateGroup = CMD_FILL_BOX;
			return 1;
		}
		if( UCA0RXBUF == CMD_BLEND_COLOUR_LINE ){
			UART.commandState = CMD_STATE_BLEND_COLOUR_LINE__POSITION_A;
			UART.commandStateGroup = CMD_BLEND_COLOUR_LINE;
```

```
                return 1;
            }
            if( UCA0RXBUF == CMD_LINE ){
                UART.commandState = CMD_STATE_LINE__POSITION_A;
                UART.commandStateGroup = CMD_LINE;
                return 1;
            }
            if( UCA0RXBUF == CMD_CIRCLE ){
                UART.commandState = CMD_STATE_CIRCLE__POSITION;
                UART.commandStateGroup = CMD_CIRCLE;
                return 1;
            }
            if( UCA0RXBUF == CMD_FILL_CIRCLE ){
                UART.commandState = CMD_STATE_FILL_CIRCLE__POSITION;
                UART.commandStateGroup = CMD_FILL_CIRCLE;
                return 1;
            }
            if( UCA0RXBUF == CMD_BLEND_COLOUR_FILL_CIRCLE ){
                UART.commandState = CMD_STATE_BLEND_COLOUR_FILL_CIRCLE__POSITION;
                UART.commandStateGroup = CMD_BLEND_COLOUR_FILL_CIRCLE;
                return 1;
            }
            if( UCA0RXBUF == CMD_SET_BLEND_MODE__REPLACE ){
                drawUtils_SetBlendMode( BLEND_MODE_REPLACE );
                return 1;
            }
            if( UCA0RXBUF == CMD_SET_BLEND_MODE__ADD ){
                drawUtils_SetBlendMode( BLEND_MODE_ADD );
                return 1;
            }
            if( UCA0RXBUF == CMD_SET_BLEND_MODE__SUBTRACT ){
                drawUtils_SetBlendMode( BLEND_MODE_SUBTRACT );
                return 1;
            }
            if( UCA0RXBUF == CMD_SET_BLEND_MODE__MULTIPLY ){
                drawUtils_SetBlendMode( BLEND_MODE_MULTIPLY );
                return 1;
            }

            // transmission or synchronization failure, reset
            return error();
        }

        //
-------------------------------------------------------------------------------------------
        // dot
        if( UART.commandStateGroup == CMD_DOT )
        {
            if( UART.commandState == CMD_STATE_DOT__POSITION ){
                UART.x1 = UCA0RXBUF>>4;
                UART.y1 = UCA0RXBUF&0x0F;
                UART.commandState = CMD_STATE_DOT__COLOUR_MSB;
                return 1;
            }
            if( UART.commandState == CMD_STATE_DOT__COLOUR_MSB ){
                UART.cA = UCA0RXBUF;
                UART.cA <<= 4;
                if( (UART.cA&0xF00) > 0xE00 ) return error(); // error check
                UART.commandState = CMD_STATE_DOT__COLOUR_LSB;
                return 1;
            }
            if( UART.commandState == CMD_STATE_DOT__COLOUR_LSB ){
```

```
                        UART.cA |= (UCA0RXBUF>>4);
                        if( (UART.cA&0x0F0)>0x0E0 || (UART.cA&0x00F)>0x00E ) return
error(); // error check
                        dot( &UART.x1, &UART.y1, &UART.cA );
                        UART.commandState = CMD_STATE_NOP;
                        UART.commandStateGroup = CMD_STATE_NOP;
                        return 1;
                }

                // transmission or synchronization failure, reset
                return error();
        }

        //
---------------------------------------------------------------------------------------------
        // blend colour box
        if( UART.commandStateGroup == CMD_BLEND_COLOUR_BOX )
        {
                if( UART.commandState == CMD_STATE_BLEND_COLOUR_BOX__POSITION_A ){
                        UART.x1 = UCA0RXBUF>>4;
                        UART.y1 = UCA0RXBUF&0x0F;
                        UART.commandState = CMD_STATE_BLEND_COLOUR_BOX__POSITION_B;
                        return 1;
                }
                if( UART.commandState == CMD_STATE_BLEND_COLOUR_BOX__POSITION_B ){
                        UART.x2 = UCA0RXBUF>>4;
                        UART.y2 = UCA0RXBUF&0x0F;
                        UART.commandState = CMD_STATE_BLEND_COLOUR_BOX__COLOUR_A;
                        return 1;
                }
                if( UART.commandState == CMD_STATE_BLEND_COLOUR_BOX__COLOUR_A ){
                        UART.cA = UCA0RXBUF;
                        UART.cA <<= 4;
                        if( (UART.cA&0xF00)>0xE00 || (UART.cA&0x0F0)>0x0E0 ) return
error(); // error check
                        UART.commandState = CMD_STATE_BLEND_COLOUR_BOX__COLOUR_AB;
                        return 1;
                }
                if( UART.commandState == CMD_STATE_BLEND_COLOUR_BOX__COLOUR_AB ){
                        UART.cA |= UCA0RXBUF>>4;
                        if( (UART.cA&0x00F)>0x00E ) return error(); // error check
                        UART.cB = UCA0RXBUF&0x0F;
                        UART.cB <<= 8;
                        if( (UART.cB&0xF00)>0xE00 ) return error(); // error check
                        UART.commandState = CMD_STATE_BLEND_COLOUR_BOX__COLOUR_B;
                        return 1;
                }
                if( UART.commandState == CMD_STATE_BLEND_COLOUR_BOX__COLOUR_B ){
                        UART.cB |= UCA0RXBUF;
                        if( (UART.cB&0x0F0)>0x0E0 || (UART.cB&0x00F)>0x00E ) return
error(); // error check
                        UART.commandState = CMD_STATE_BLEND_COLOUR_BOX__COLOUR_C;
                        return 1;
                }
                if( UART.commandState == CMD_STATE_BLEND_COLOUR_BOX__COLOUR_C ){
                        UART.cC = UCA0RXBUF;
                        UART.cC <<= 4;
                        if( (UART.cC&0xF00)>0xE00 || (UART.cC&0x0F0)>0x0E0 ) return
error(); // error check
                        UART.commandState = CMD_STATE_BLEND_COLOUR_BOX__COLOUR_CD;
                        return 1;
                }
```

```
                if( UART.commandState == CMD_STATE_BLEND_COLOUR_BOX__COLOUR_CD ){
                        UART.cC |= UCA0RXBUF>>4;
                        if( (UART.cC&0x00F)>0x00E ) return error(); // error check
                        UART.cD = UCA0RXBUF&0x0F;
                        UART.cD <<= 8;
                        if( (UART.cD&0xF00)>0xE00 ) return error(); // error check
                        UART.commandState = CMD_STATE_BLEND_COLOUR_BOX__COLOUR_D;
                        return 1;
                }
                if( UART.commandState == CMD_STATE_BLEND_COLOUR_BOX__COLOUR_D ){
                        UART.cD |= UCA0RXBUF;
                        if( (UART.cD&0x0F0)>0x0E0 || (UART.cD&0x00F)>0x00E ) return
error(); // error check
                        blendColourBox( &UART.x1, &UART.y1, &UART.x2, &UART.y2, &UART.cA,
&UART.cB, &UART.cC, &UART.cD );
                        UART.commandState = CMD_STATE_NOP;
                        UART.commandStateGroup = CMD_STATE_NOP;
                        return 1;
                }

                // transmission or synchronization failure, reset
                return error();
        }

        //
---------------------------------------------------------------------------------------------
        // blend colour fill box
        if( UART.commandStateGroup == CMD_BLEND_COLOUR_FILL_BOX )
        {
                if( UART.commandState == CMD_STATE_BLEND_COLOUR_FILL_BOX__POSITION_A ){
                        UART.x1 = UCA0RXBUF>>4;
                        UART.y1 = UCA0RXBUF&0x0F;
                        UART.commandState = CMD_STATE_BLEND_COLOUR_FILL_BOX__POSITION_B;
                        return 1;
                }
                if( UART.commandState == CMD_STATE_BLEND_COLOUR_FILL_BOX__POSITION_B ){
                        UART.x2 = UCA0RXBUF>>4;
                        UART.y2 = UCA0RXBUF&0x0F;
                        UART.commandState = CMD_STATE_BLEND_COLOUR_FILL_BOX__COLOUR_A;
                        return 1;
                }
                if( UART.commandState == CMD_STATE_BLEND_COLOUR_FILL_BOX__COLOUR_A ){
                        UART.cA = UCA0RXBUF;
                        UART.cA <<= 4;
                        UART.commandState = CMD_STATE_BLEND_COLOUR_FILL_BOX__COLOUR_AB;
                        return 1;
                }
                if( UART.commandState == CMD_STATE_BLEND_COLOUR_FILL_BOX__COLOUR_AB ){
                        UART.cA |= UCA0RXBUF>>4;
                        UART.cB = UCA0RXBUF&0x0F;
                        UART.cB <<= 8;
                        UART.commandState = CMD_STATE_BLEND_COLOUR_FILL_BOX__COLOUR_B;
                        return 1;
                }
                if( UART.commandState == CMD_STATE_BLEND_COLOUR_FILL_BOX__COLOUR_B ){
                        UART.cB |= UCA0RXBUF;
                        UART.commandState = CMD_STATE_BLEND_COLOUR_FILL_BOX__COLOUR_C;
                        return 1;
                }
                if( UART.commandState == CMD_STATE_BLEND_COLOUR_FILL_BOX__COLOUR_C ){
                        UART.cC = UCA0RXBUF;
                        UART.cC <<= 4;
```

```
                        UART.commandState = CMD_STATE_BLEND_COLOUR_FILL_BOX__COLOUR_CD;
                        return 1;
                }
                if( UART.commandState == CMD_STATE_BLEND_COLOUR_FILL_BOX__COLOUR_CD ){
                        UART.cC |= UCA0RXBUF>>4;
                        UART.cD = UCA0RXBUF&0x0F;
                        UART.cD <<= 8;
                        UART.commandState = CMD_STATE_BLEND_COLOUR_FILL_BOX__COLOUR_D;
                        return 1;
                }
                if( UART.commandState == CMD_STATE_BLEND_COLOUR_FILL_BOX__COLOUR_D ){
                        UART.cD |= UCA0RXBUF;
                        blendColourFillBox( &UART.x1, &UART.y1, &UART.x2, &UART.y2, &UART.cA,
&UART.cB, &UART.cC, &UART.cD );
                        UART.commandState = CMD_STATE_NOP;
                        UART.commandStateGroup = CMD_STATE_NOP;
                        return 1;
                }

                // transmission or synchronization failure, reset
                return error();
        }

        //
-------------------------------------------------------------------------------------------
        // box
        if( UART.commandStateGroup == CMD_BOX )
        {
                if( UART.commandState == CMD_STATE_BOX__POSITION_A ){
                        UART.x1 = UCA0RXBUF>>4;
                        UART.y1 = UCA0RXBUF&0x0F;
                        UART.commandState = CMD_STATE_BOX__POSITION_B;
                        return 1;
                }
                if( UART.commandState == CMD_STATE_BOX__POSITION_B ){
                        UART.x2 = UCA0RXBUF>>4;
                        UART.y2 = UCA0RXBUF&0x0F;
                        UART.commandState = CMD_STATE_BOX__COLOUR_MSB;
                        return 1;
                }
                if( UART.commandState == CMD_STATE_BOX__COLOUR_MSB ){
                        UART.cA = UCA0RXBUF;
                        UART.cA <<= 4;
                        UART.commandState = CMD_STATE_BOX__COLOUR_LSB;
                        return 1;
                }
                if( UART.commandState == CMD_STATE_BOX__COLOUR_LSB ){
                        UART.cA |= UCA0RXBUF>>4;
                        box( &UART.x1, &UART.y1, &UART.x2, &UART.y2, &UART.cA );
                        UART.commandState = CMD_STATE_NOP;
                        UART.commandStateGroup = CMD_STATE_NOP;
                        return 1;
                }

                // transmission or synchronization failure, reset
                return error();
        }

        //
-------------------------------------------------------------------------------------------
        // fill box
        if( UART.commandStateGroup == CMD_FILL_BOX )
```

```
	{
		if( UART.commandState == CMD_STATE_FILL_BOX__POSITION_A ){
			UART.x1 = UCA0RXBUF>>4;
			UART.y1 = UCA0RXBUF&0x0F;
			UART.commandState = CMD_STATE_FILL_BOX__POSITION_B;
			return 1;
		}
		if( UART.commandState == CMD_STATE_FILL_BOX__POSITION_B ){
			UART.x2 = UCA0RXBUF>>4;
			UART.y2 = UCA0RXBUF&0x0F;
			UART.commandState = CMD_STATE_FILL_BOX__COLOUR_MSB;
			return 1;
		}
		if( UART.commandState == CMD_STATE_FILL_BOX__COLOUR_MSB ){
			UART.cA = UCA0RXBUF;
			UART.cA <<= 4;
			UART.commandState = CMD_STATE_FILL_BOX__COLOUR_LSB;
			return 1;
		}
		if( UART.commandState == CMD_STATE_FILL_BOX__COLOUR_LSB ){
			UART.cA |= UCA0RXBUF>>4;
			fillBox( &UART.x1, &UART.y1, &UART.x2, &UART.y2, &UART.cA );
			UART.commandState = CMD_STATE_NOP;
			UART.commandStateGroup = CMD_STATE_NOP;
			return 1;
		}

		// transmission or synchronization failure, reset
		return error();
	}

	//
-----------------------------------------------------------------------------------------
	// blend colour line
	if( UART.commandStateGroup == CMD_BLEND_COLOUR_LINE )
	{
		if( UART.commandState == CMD_STATE_BLEND_COLOUR_LINE__POSITION_A ){
			UART.x1 = UCA0RXBUF>>4;
			UART.y1 = UCA0RXBUF&0x0F;
			UART.commandState = CMD_STATE_BLEND_COLOUR_LINE__POSITION_B;
			return 1;
		}
		if( UART.commandState == CMD_STATE_BLEND_COLOUR_LINE__POSITION_B ){
			UART.x2 = UCA0RXBUF>>4;
			UART.y2 = UCA0RXBUF&0x0F;
			UART.commandState = CMD_STATE_BLEND_COLOUR_LINE__COLOUR_A;
			return 1;
		}
		if( UART.commandState == CMD_STATE_BLEND_COLOUR_LINE__COLOUR_A ){
			UART.cA = UCA0RXBUF;
			UART.cA <<= 4;
			UART.commandState = CMD_STATE_BLEND_COLOUR_LINE__COLOUR_AB;
			return 1;
		}
		if( UART.commandState == CMD_STATE_BLEND_COLOUR_LINE__COLOUR_AB ){
			UART.cA |= UCA0RXBUF>>4;
			UART.cB = UCA0RXBUF&0x0F;
			UART.cB <<= 8;
			UART.commandState = CMD_STATE_BLEND_COLOUR_LINE__COLOUR_B;
			return 1;
		}
		if( UART.commandState == CMD_STATE_BLEND_COLOUR_LINE__COLOUR_B ){
```

```
                        UART.cB |= UCA0RXBUF;
                        blendColourLine( &UART.x1, &UART.y1, &UART.x2, &UART.y2, &UART.cA,
&UART.cB );
                        UART.commandState = CMD_STATE_NOP;
                        UART.commandStateGroup = CMD_STATE_NOP;
                        return 1;
                }

                // transmission or synchronization failure, reset
                return error();
        }

        //
-----------------------------------------------------------------------------------------
        // line
        if( UART.commandStateGroup == CMD_LINE )
        {
                if( UART.commandState == CMD_STATE_LINE__POSITION_A ){
                        UART.x1 = UCA0RXBUF>>4;
                        UART.y1 = UCA0RXBUF&0x0F;
                        UART.commandState = CMD_STATE_LINE__POSITION_B;
                        return 1;
                }
                if( UART.commandState == CMD_STATE_LINE__POSITION_B ){
                        UART.x2 = UCA0RXBUF>>4;
                        UART.y2 = UCA0RXBUF&0x0F;
                        UART.commandState = CMD_STATE_LINE__COLOUR_MSB;
                        return 1;
                }
                if( UART.commandState == CMD_STATE_LINE__COLOUR_MSB ){
                        UART.cA = UCA0RXBUF;
                        UART.cA <<= 4;
                        UART.commandState = CMD_STATE_LINE__COLOUR_LSB;
                        return 1;
                }
                if( UART.commandState == CMD_STATE_LINE__COLOUR_LSB ){
                        UART.cA |= UCA0RXBUF>>4;
                        line( &UART.x1, &UART.y1, &UART.x2, &UART.y2, &UART.cA );
                        UART.commandState = CMD_STATE_NOP;
                        UART.commandStateGroup = CMD_STATE_NOP;
                        return 1;
                }

                // transmission or synchronization failure, reset
                return error();
        }

        //
-----------------------------------------------------------------------------------------
        // circle
        if( UART.commandStateGroup == CMD_CIRCLE )
        {
                if( UART.commandState == CMD_STATE_CIRCLE__POSITION ){
                        UART.x1 = UCA0RXBUF>>4;
                        UART.y1 = UCA0RXBUF&0x0F;
                        UART.commandState = CMD_STATE_CIRCLE__RADIUS;
                        return 1;
                }
                if( UART.commandState == CMD_STATE_CIRCLE__RADIUS ){
                        UART.x2 = UCA0RXBUF;
                        UART.commandState = CMD_STATE_CIRCLE__COLOUR_MSB;
                        return 1;
```

```
		}
		if( UART.commandState == CMD_STATE_CIRCLE__COLOUR_MSB ){
			UART.cA = UCA0RXBUF;
			UART.cA <<= 4;
			UART.commandState = CMD_STATE_CIRCLE__COLOUR_LSB;
			return 1;
		}
		if( UART.commandState == CMD_STATE_CIRCLE__COLOUR_LSB ){
			UART.cA |= UCA0RXBUF>>4;
			circle( &UART.x1, &UART.y1, &UART.x2, &UART.cA );
			UART.commandState = CMD_STATE_NOP;
			UART.commandStateGroup = CMD_STATE_NOP;
			return 1;
		}

		// transmission or synchronization failure, reset
		return error();
	}

	//
-------------------------------------------------------------------------------------------
	// fill circle
	if( UART.commandStateGroup == CMD_FILL_CIRCLE )
	{
		if( UART.commandState == CMD_STATE_FILL_CIRCLE__POSITION ){
			UART.x1 = UCA0RXBUF>>4;
			UART.y1 = UCA0RXBUF&0x0F;
			UART.commandState = CMD_STATE_FILL_CIRCLE__RADIUS;
			return 1;
		}
		if( UART.commandState == CMD_STATE_FILL_CIRCLE__RADIUS ){
			UART.x2 = UCA0RXBUF;
			UART.commandState = CMD_STATE_FILL_CIRCLE__COLOUR_MSB;
			return 1;
		}
		if( UART.commandState == CMD_STATE_FILL_CIRCLE__COLOUR_MSB ){
			UART.cA = UCA0RXBUF;
			UART.cA <<= 4;
			UART.commandState = CMD_STATE_FILL_CIRCLE__COLOUR_LSB;
			return 1;
		}
		if( UART.commandState == CMD_STATE_FILL_CIRCLE__COLOUR_LSB ){
			UART.cA |= UCA0RXBUF>>4;
			fillCircle( &UART.x1, &UART.y1, &UART.x2, &UART.cA );
			UART.commandState = CMD_STATE_NOP;
			UART.commandStateGroup = CMD_STATE_NOP;
			return 1;
		}

		// transmission or synchronization failure, reset
		return error();
	}

	//
-------------------------------------------------------------------------------------------
	// blend colour fill circle
	if( UART.commandStateGroup == CMD_BLEND_COLOUR_FILL_CIRCLE )
	{
		if( UART.commandState == CMD_STATE_BLEND_COLOUR_FILL_CIRCLE__POSITION ){
			UART.x1 = UCA0RXBUF>>4;
			UART.y1 = UCA0RXBUF&0x0F;
			UART.commandState = CMD_STATE_BLEND_COLOUR_FILL_CIRCLE__RADIUS;
```

```
                        return 1;
                }
                if( UART.commandState == CMD_STATE_BLEND_COLOUR_FILL_CIRCLE__RADIUS ){
                        UART.x2 = UCA0RXBUF;
                        UART.commandState = CMD_STATE_BLEND_COLOUR_FILL_CIRCLE__COLOUR_A;
                        return 1;
                }
                if( UART.commandState == CMD_STATE_BLEND_COLOUR_FILL_CIRCLE__COLOUR_A ){
                        UART.cA = UCA0RXBUF;
                        UART.cA <<= 4;
                        UART.commandState = CMD_STATE_BLEND_COLOUR_FILL_CIRCLE__COLOUR_AB;
                        return 1;
                }
                if( UART.commandState == CMD_STATE_BLEND_COLOUR_FILL_CIRCLE__COLOUR_AB ){
                        UART.cA |= UCA0RXBUF>>4;
                        UART.cB = UCA0RXBUF&0x0F;
                        UART.cB <<= 8;
                        UART.commandState = CMD_STATE_BLEND_COLOUR_FILL_CIRCLE__COLOUR_B;
                        return 1;
                }
                if( UART.commandState == CMD_STATE_BLEND_COLOUR_FILL_CIRCLE__COLOUR_B ){
                        UART.cB |= UCA0RXBUF;
                        blendColourFillCircle( &UART.x1, &UART.y1, &UART.x2, &UART.cA,
&UART.cB );
                        UART.commandState = CMD_STATE_NOP;
                        UART.commandStateGroup = CMD_STATE_NOP;
                        return 1;
                }

                // transmission or synchronization failure, reset
                return error();
        }

        // transmission or synchronization failure, reset
        return error();
}

//
-----------------------------------------------------------------------------------------------
----------------------
// error, resets the state machine
unsigned char error( void )
{
        UART.commandState = CMD_STATE_NOP;
        UART.commandStateGroup = CMD_STATE_NOP;
        return 0;
}

//
-----------------------------------------------------------------------------------------------
----------------------
// Interrupt for receiving data
#pragma vector=USCI_A0_VECTOR
__interrupt void USCI_A0_ISR( void )
{

        // process received data, if any
        switch(__even_in_range(UCA0IV,4))
        {
                case 0:break;                           // Vector 0 - no interrupt
                case 2:                                 // Vector 2 - RXIFG
```

```
                        // process data and send it back
                        processCommand();
                        UCA0TXBUF = UCA0RXBUF;

                        break;

                case 4:break;                           // Vector 4 - TXIFG
        default: break;
        }
}
```