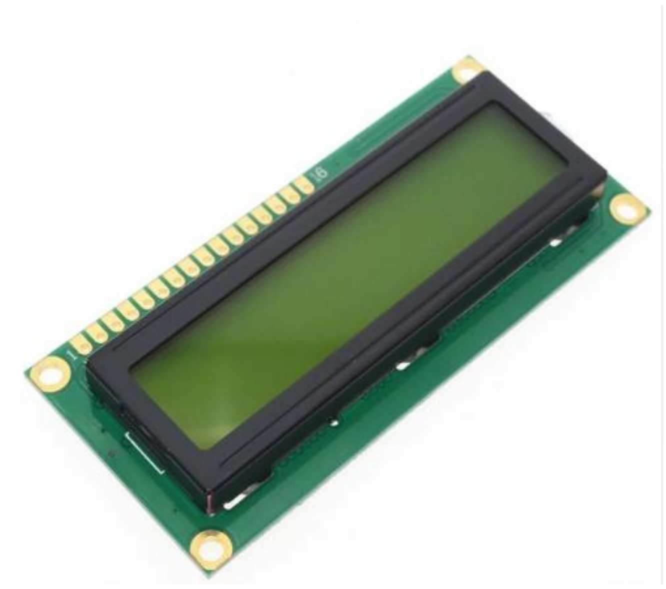
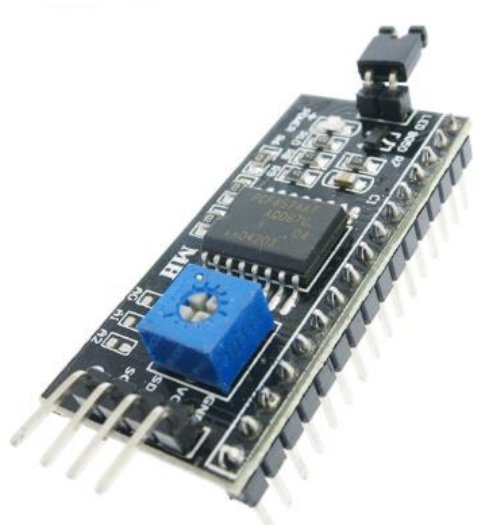
RetroLCD.com

Character Display Sprites

https://retrolcd.com/Components/LCD1602



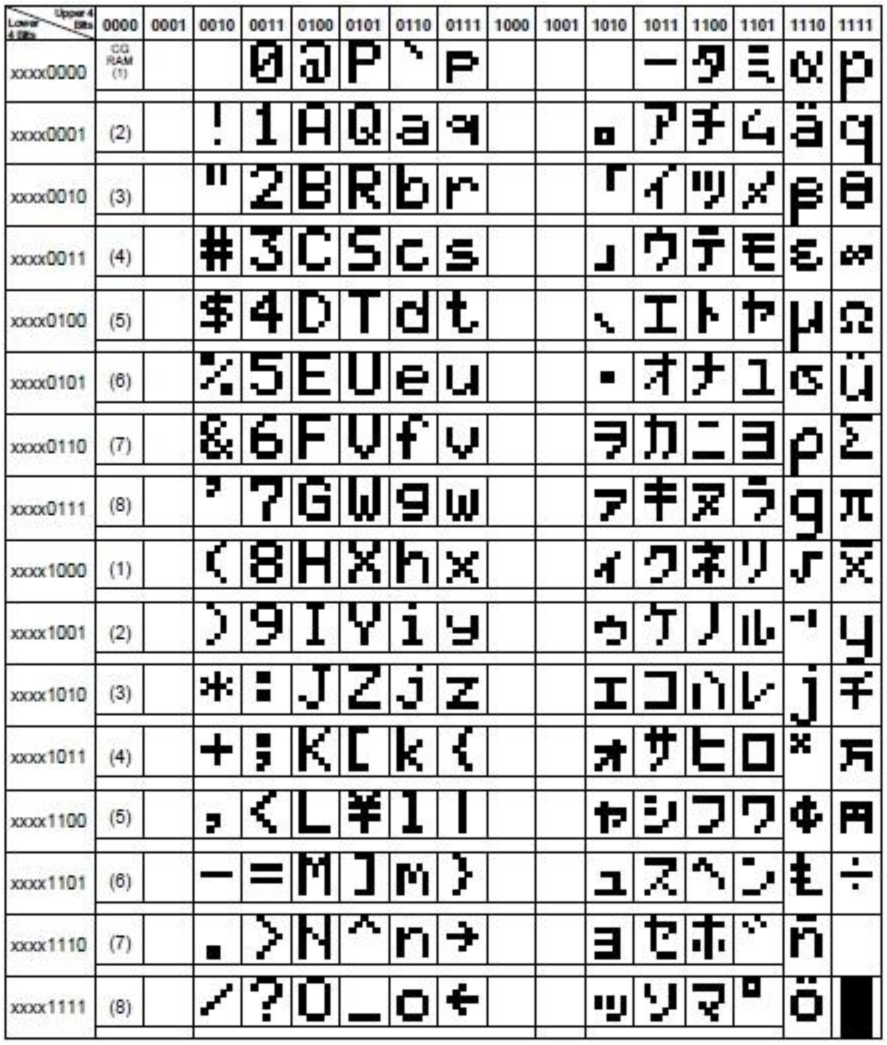
The **1602** stands for 16x2 which is 16 characters wide and 2 lines tall. The default board has 16 pins which is a lot to hook up. Fortunately, there is a corresponding controller, the **HD44780**, that goes with it.



Typically, the controllers are sold with the display and the pair go for around $2 each direct from China. With this board attached to the Character Display, you can use I2C which requires only 2 pins and power. And you can connect multiple displays.

RetroLCD.com

Built-In Characters



Upper 4 Bits / Lower 4 Bits	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
xxxx0000	CG RAM (1)			0	@	P	`	p				ー	タ	ミ	α	p
xxxx0001	(2)		!	1	A	Q	a	q			。	ア	チ	ム	ä	q
xxxx0010	(3)		"	2	B	R	b	r			「	イ	ツ	メ	β	θ
xxxx0011	(4)		#	3	C	S	c	s			」	ウ	テ	モ	ε	∞
xxxx0100	(5)		$	4	D	T	d	t			、	エ	ト	ヤ	μ	Ω
xxxx0101	(6)		%	5	E	U	e	u			・	オ	ナ	ユ	σ	ü
xxxx0110	(7)		&	6	F	V	f	v			ヲ	カ	ニ	ヨ	ρ	Σ
xxxx0111	(8)		'	7	G	W	g	w			ァ	キ	ヌ	ラ	g	π
xxxx1000	(1)		(	8	H	X	h	x			ィ	ク	ネ	リ	√	x̄
xxxx1001	(2)		)	9	I	Y	i	y			ゥ	ケ	ノ	ル	⁻¹	y
xxxx1010	(3)		*	:	J	Z	j	z			ェ	コ	ハ	レ	j	千
xxxx1011	(4)		+	;	K	[	k	{			ォ	サ	ヒ	ロ	×	万
xxxx1100	(5)		,	<	L	¥	l	|			ャ	シ	フ	ワ	¢	円
xxxx1101	(6)		-	=	M	]	m	}			ュ	ス	ヘ	ン	£	÷
xxxx1110	(7)		.	>	N	^	n	→			ョ	セ	ホ	゛	ñ	
xxxx1111	(8)		/	?	O	_	o	←			ッ	ソ	マ	゜	ö	█

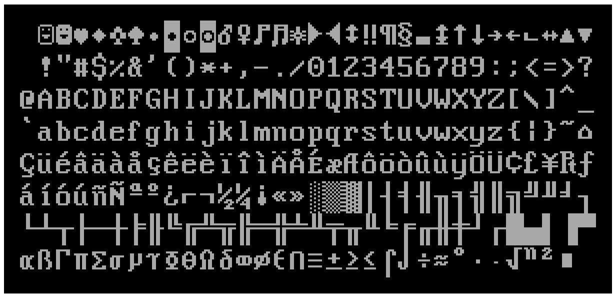
The 1602 has 128 built in characters. We use the CHAR datatype which is a signed 8-bit value and stick to the values 0-127. Generally, character 0 is the null character which terminates strings. When printing a line of text, the processor looks for the 0 so it knows when to stop reading memory. If you leave off the zero, the processor will continue to read into memory that it wasn't supposed to read.

When first learning how to print text on the display and move things around on it, it may be easier to just use the built-in characters before moving onto custom characters.

RETROLCD.com

Code Page 437

https://en.wikipedia.org/wiki/Code_page_437



This is the character set adopted on the original IBM PC. Unfortunately, the character set locked into the 1602 misses many of the very useful symbols such as those commonly used for playing cards and the faces often used in early games.

There is an interesting history of the ASCII smiley face found at

http://www.vintagecomputing.com/index.php/archives/790/the-ibm-smiley-character-turns-30

You can see from the character list of the 1602 that the first 17 characters are empty. Instead of leaving the first 32 characters in ASCII blank as they are reserved control characters, the developers decided to put in some characters that would be useful for character displays which were not interpreting those bytes for control purposes. The extended characters (128-255) were used extensively in text based user interfaces.

ASCII character 13 is still the carriage return and ASCII character 10 is still the new line character.

This original list of characters may serve as a guide for custom characters you may want to put into your own project.

Creating Your Own Custom Sprites

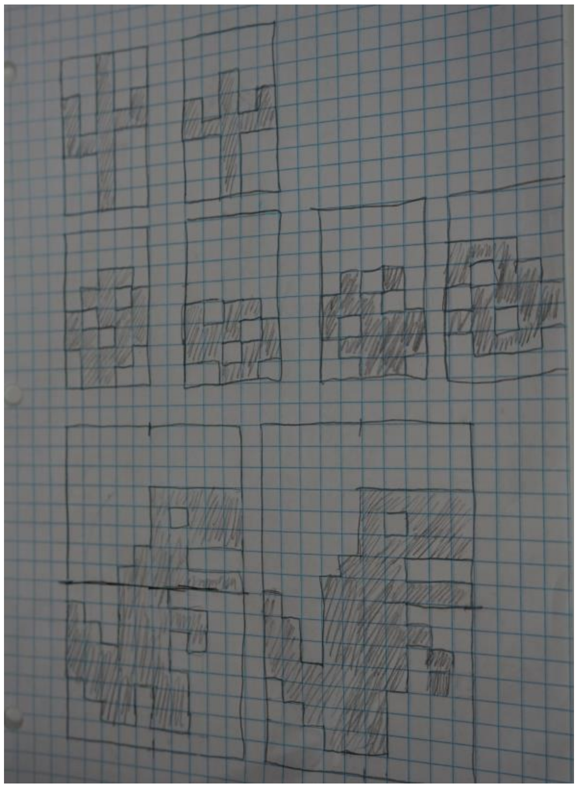
The 1602 allows you to program up to 8 custom characters in locations 0 through 7. You may recognize 0 as the null character. It is best to simply not use it as a custom character and limit yourself to memory locations 1 through 7.

The first thing you need to know is that each character is 5 pixels wide and 8 pixels tall. While you can buy graphing paper to work with, there are also online resources to generate custom graph paper to print out yourself.

https://incompetech.com/graphpaper/

The second thing to keep in mind is that the limit is 7 different custom characters displayed at once. You can change characters as many times as you want in your program.

The Dinosaur game demonstrates several ways of rendering sprites.



There are two cactus sprites. One big and one little. These never change.

In Dinosaur.ino

```
lcd.createChar(6, cacti.frame1);
lcd.createChar(7, cacti.frame2);
```

In CactiController.h

```
class CactiController {
  public:
    static byte frame1[8] = {
      0b00100,
      0b00100,
      0b10101,
      0b10111,
      0b11100,
      0b00100,
      0b00100,
      0b00100
    };

    static byte frame2[8] = {
      0b00000,
      0b00000,
      0b00100,
      0b10101,
      0b11111,
      0b00100,
      0b00100,
      0b00100
    };
};
```

This puts the two cactus sprites into slot 6 and 7. This never changes throughout the game.

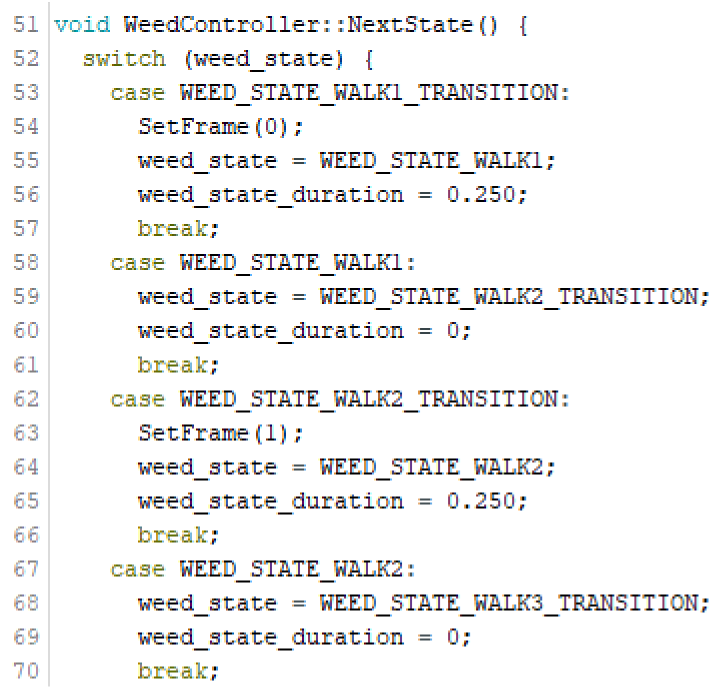
Next, we have a tumbleweed. The tumbleweed has 4 frames and uses a state machine to keep track of which frame is loaded into the display and which should be loaded next.

In WeedController.cpp

```
void WeedController::Init() {
  weed_state = WEED_STATE_WALK1_TRANSITION;
  weed_state_time = 0;
  weed_state_duration = 0;
  SetFrame(0);
}

void WeedController::Update(float s) {
  weed_state_time += s;
  if (weed_state_time >= weed_state_duration) {
    NextState();
  }
}
```

Every frame we call the Update method along with the duration of the current frame. Then, if the total time that has elapsed is more than the current state is expecting, then the weed switches to the next state.



```
void WeedController::NextState() {
  switch (weed_state) {
    case WEED_STATE_WALK1_TRANSITION:
      SetFrame(0);
      weed_state = WEED_STATE_WALK1;
      weed_state_duration = 0.250;
      break;
    case WEED_STATE_WALK1:
      weed_state = WEED_STATE_WALK2_TRANSITION;
      weed_state_duration = 0;
      break;
    case WEED_STATE_WALK2_TRANSITION:
      SetFrame(1);
      weed_state = WEED_STATE_WALK2;
      weed_state_duration = 0.250;
      break;
    case WEED_STATE_WALK2:
      weed_state = WEED_STATE_WALK3_TRANSITION;
      weed_state_duration = 0;
      break;
```

This is just a sample of the states. This could probably be simplified but it demonstrates how we use a transition state to update the frame stored in the memory of the display and then switch to another state so that we are not continually loading the frame if this method is called again. We do not want to load the current frame of animation every time we render a frame of the game. We only need to load it when it needs to change.

```
void WeedController::SetFrame(int num) {
  int j;
  switch (num) {
    case 0:
      for (j = 0; j < 8; j++) {
        frame[j] = frame1[j];
      }
      break;
    case 1:
      for (j = 0; j < 8; j++) {
        frame[j] = frame2[j];
      }
      break;
    case 2:
      for (j = 0; j < 8; j++) {
        frame[j] = frame3[j];
      }
      break;
    case 3:
      for (j = 0; j < 8; j++) {
        frame[j] = frame4[j];
      }
      break;
  }
  weed_sprite_change = true;
}
```

There is a variable in the WeedController class which holds the definition of the current frame. When we call SetFrame this variable is updated, and we set a variable that the sprite has changed. This tells the main Dinosaur program to update the display.

```
void HandleSpriteChange()
{
  if (weed.weed_sprite_change) {
    lcd.createChar(5, weed.frame);
    weed.weed_sprite_change = false;
  }
```

In Dinosaur.h, the HandleSprite change function handles checking the sprite change flag and if it's set, it stores the frame from the weed class into slot 5 of the character display and then resets the flag so it isn't continually loaded.

The Dinosaur sprite builds on this. You may have noticed that the Dinosaur is made up of 4 characters. 1 tumbleweed character + 2 cactus characters + 4 dinosaur characters = 7. Which is the maximum number of custom characters we can have at one time.

In our HandleSpriteChange function we also have

```
if (dino.dino_sprite_change) {
  // NOTE: do not use createChar(0, ...), it confuses the Ardino
  // http://forum.arduino.cc/index.php?topic=74666.0
  lcd.createChar(1, dino.dinoTL);
  lcd.createChar(2, dino.dinoTR);
  lcd.createChar(3, dino.dinoBL);
  lcd.createChar(4, dino.dinoBR);
  dino.dino_sprite_change = false;
}
```

TL = Top Left

TR = Top Right

BL = Bottom Left

BR = Bottom Right

All the sprites are defined in DinoController.h

All the characters use the same convention to make it easier to keep track of where to draw them on the display.

In our main loop function we have

```
switch (dino.dino_state) {
  case DINO_STATE_JUMP:
    PlotCHAR(player_x + 0, 0, 1);
    PlotCHAR(player_x + 1, 0, 2);
    PlotCHAR(player_x + 2, 0, 3);
    break;
  case DINO_STATE_CROUCH:
    PlotCHAR(player_x + 0, 1, 1);
    PlotCHAR(player_x + 1, 1, 2);
    PlotCHAR(player_x + 2, 1, 3);
    break;
  default:
    PlotCHAR(player_x + 0, 0, 1);
    PlotCHAR(player_x + 1, 0, 2);
    PlotCHAR(player_x + 0, 1, 3);
    PlotCHAR(player_x + 1, 1, 4);
    break;
}
```

This checks the state of the dinosaur and plots the characters that make it up in the appropriate location. When walking, there are 4 characters in use, while when jump or crouching, there are 3. Jumping and crouching use the same characters, but they are drawn on the top line when jumping and on the bottom line when crouching.

Summary

Whether you're using a character display or some other method to display graphics, the general principles will remain the same: state machines are used to transition between frames of animation, large sprites are broken up into smaller sprites, sprites are swapped in and out of memory, etc.

The original NES had a limited number of sprites that could be in memory at once just like the 1602. The NES limited developers to 64 sprites on the screen at once but only 8 per scanline. So, you could not have 9 goombas in a row.

https://megacatstudios.com/blogs/press/creating-nes-graphics

A big part of game development is understanding the limitations of the hardware you are developing on. As you become more proficient you may find ways to do things that weren't thought of before. Late NES games had much better graphics than earlier ones. While the original intent of the NES was to allow scrolling in only the vertical or horizontal direction, eventually people figured out how to do diagonal scrolling. This required working within other limitations which is why Super Mario Bros 3 put bars around the screen to hide the artifacts at the edges of the screen.