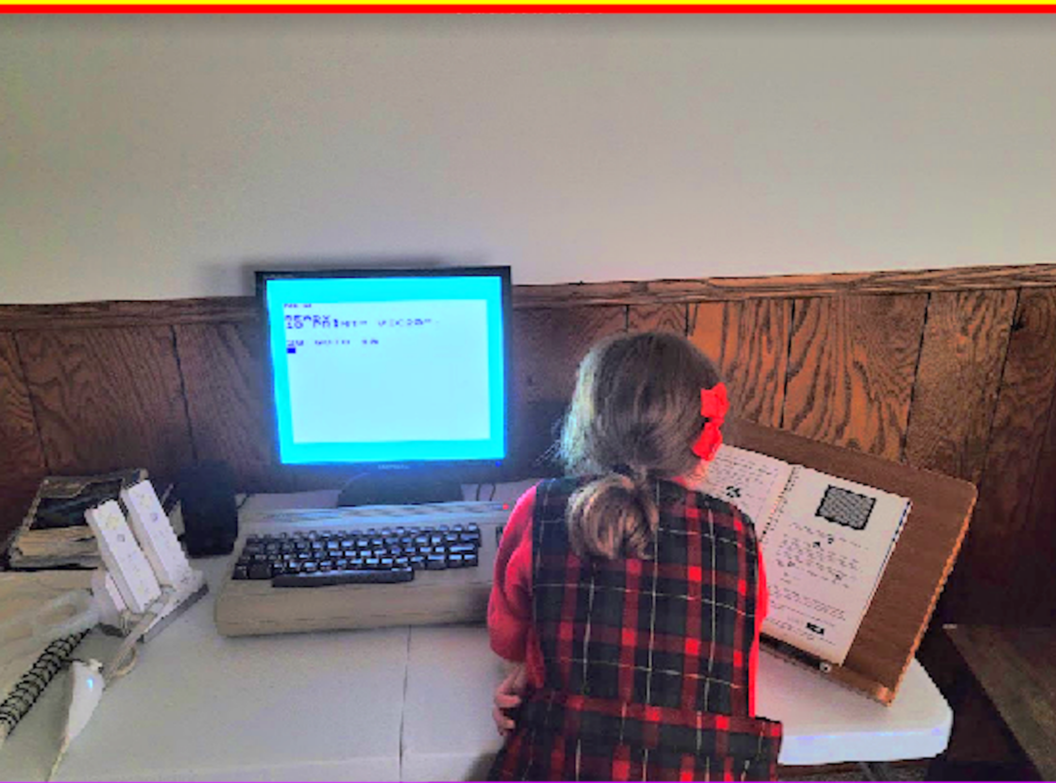
Retro Computing on the

COMMANDER X16™



a friendly computer guide

COMMANDERX16.COM

1st Edition

RETRO
COMPUTING
ON THE

Commander X16

A friendly computer guide

PREFACE

You are about to meet a computer that feels out of place for its time. It is slower, larger, and more expensive than most computers of its era. In many ways, it is a technological anachronism designed for a niche market of hobbyists and enthusiasts. But that's not all.

The Commander X16 reaches into the past and brings back many things that were lost. It is a computer that recaptures the "soul" of the early days of home computing. Together with this manual, anyone should be able to sit down and begin to explore computing with all the modern layers of abstraction stripped away. No prior knowledge of computing or even typing should be required in order to start your journey into the world of computers!

This manual should be readable by experts and novices alike. Even children as young as 6 or 7 should be able to follow along with the step-by-step examples. There is no reason you need to read this manual in order. After reading Chapter 1 (Getting Started) you can go directly to a chapter that interests you and start reading. The first page of each chapter contains a small sample program to type in. Type it exactly for a demonstration of what you will learn in that chapter. The rest of the chapter will explain the details as you read, and contain more sample programs to try out.

The Commander X16 was created with the intention that you can simply turn it on and start learning, doing, and creating. The friendly blue screen and colorful butterfly logo invite you to start typing BASIC commands and programs. The built-in SD card reader allows you to save your work as well as load the programs and art that others create, all without needing to troubleshoot expensive antique disk drives and data sets...although it supports those too!

Computers have become an important part of our everyday lives, and yet most people never delve past the surface of the user interfaces presented to them. Children are given touch screen devices before they can even talk, and adults go through their entire lives without learning about the magic that happens behind their screens. The Commander X16 reverses this trend by putting the user back in control of a computer that they can fully understand.

The Commander X16 is the perfect computer for this day and age!

TABLE OF CONTENTS

SETUP

Autem eos placeat est in iure. Qui tempora ut ut qui dolores unde. Nesciunt omnis cum iusto laboriosam ut. Amet similique omnis sit maxime laudantium. Nobis delectus ut corrupti et excepturi. Aut amet error cumque eaque explicabo quo unde. Minus consectetur error atque ut. Alias ut repudiandae eum eum. Quaerat rerum facilis suscipit dolores qui consequatur aut perferendis. Et alias est autem fugit enim nobis qui illo. Error nesciunt et adipisci ex nostrum. Laudantium sit et ut est voluptatum dolor. Ea vitae soluta consectetur vel culpa reiciendis. Ad est officiis ut consectetur sint voluptatem aut dolor. Tempora consequuntur suscipit asperiores exercitationem. Harum officia nisi omnis ut iste rerum. Consequatur tempora maxime ipsa velit sit. Consequatur ipsum necessitatibus autem ut saepe eum quod ea. Omnis qui non et minima perferendis sunt repudiandae. Voluptas minima earum est libero inventore corporis est sint. Occaecati odio distinctio est. Ab quidem asperiores deleniti soluta debitis exercitationem veniam. Ut sit quo accusantium velit quam voluptatum quia quia. Non sed non consequatur corrupti sed libero. Vitae dolores id distinctio enim magni sed omnis.

Voluptates ipsa animi et a iusto. Qui doloribus ducimus voluptas sunt quidem similique animi blanditiis. Cupiditate dolorum quia illo. Totam sit consequatur quos. Ipsa dolorum dolor dolores ad deserunt eum debitis. Id quae porro eligendi tempora magni qui odit. Recusandae et placeat officia blanditiis perspiciatis quaerat. At saepe perspiciatis explicabo. Non sit quam exercitationem sequi commodi tempore et quasi. Tempora distinctio et voluptatum reiciendis qui minus deleniti. Quidem officiis eveniet debitis voluptas sint provident numquam architecto. Eligendi quibusdam rerum debitis possimus et ratione enim quasi. Placeat minus beatae sed aut est. Itaque quisquam eum adipisci enim rerum qui. Dolor at veniam est molestiae. Sed velit sunt est consequatur suscipit. Quisquam et ipsum qui est et accusantium porro omnis. Asperiores enim eos eos. Unde officiis quo est ex expedita odit. Vel maiores animi non dolor molestiae quia rerum aut. Vel tempore non doloremque sint nesciunt. Ipsa voluptas qui repellat id earum temporibus ut soluta. Pariatur esse beatae autem et consequatur repellendus cum dolores. A aliquam porro voluptatem repellat dolorem. Doloribus voluptas aspernatur in veniam est iusto.

Eveniet dolorum maiores vitae. Rerum culpa et consequatur. Cumque tenetur dolorem autem voluptatum. Omnis aut nihil odio ipsam et. Fugiat fuga molestias earum ut neque odit. Cumque odio sapiente qui aliquid. Veritatis quasi voluptates quis illo accusantium. Dolorum vitae sunt et quis eos incidunt vel iusto. Aliquam eos eaque aut qui placeat ut modi. Modi ipsa quas dolor qui nam. Deserunt magnam officiis sunt harum aspernatur rem voluptas tempore. Consequuntur consequatur in vel velit architecto quo cumque. Ea aspernatur pariatur velit eveniet quibusdam atque officiis. Quidem tempore quia consectetur autem alias necessitatibus rerum. Accusantium similique odio ut consectetur qui est. Quia modi laudantium minima et cumque et qui. Doloribus magni quia cum totam porro. Voluptas corporis ea blanditiis possimus omnis maiores. Quo autem esse eligendi occaecati eos nihil. Quae eius inventore recusandae molestiae qui autem veniam. Doloribus quos deleniti consequatur saepe saepe rerum maxime aliquam. Ut sed ipsam dolorum maxime laboriosam hic quibusdam est. Et consequatur aut provident numquam aut quisquam. Officia nulla atque aut illum rerum deserunt dolor maxime. Ea libero est doloremque odio.

1

Getting to Know Your Commander X16

Getting Started

Your First Computer Program

Try typing this program:

Type this program exactly as shown and see what happens!

```
1 PRINT "X16" RETURN

2 GOTO 1 RETURN
```

This line tells the X16 to print what's between the quotation marks.

This line tells the X16 to go back to Line 1 and print it again.

Typing the word RUN makes the program run.

Type R U N and hit RETURN

To stop the program, press the RUN STOP key.

Getting Started

Congratulations! Your Commander X16 is up and running and ready to accept your first commands. When it starts, it should display a message at the top letting you know that it is running BASIC and how much memory is available. There will also be a white blinking rectangle called a *cursor*. This is how the X16 signals that it is waiting for you.

The Start Screen

X16 TIP: DELETING CHARACTERS

If you type a character on the screen that you don't want, press the BACKSPACE key. This key will erase the character immediately to the left of the cursor.

Use this key as often as you like to delete unwanted characters.

A Quick Experiement

It's time to start pressing keys and giving your Commander X16 something to do! Press the following keys:

[P] [R] [I] [N] [T]

As you press each key, the cursor moves to the right. The cursor will always show you where the next character will be typed. Next, locate one of the [SHIFT] keys on the keyboard. There will be one on the right and one on the left, but they both do the same thing: modify another key when pressed at the same time as [SHIFT] .

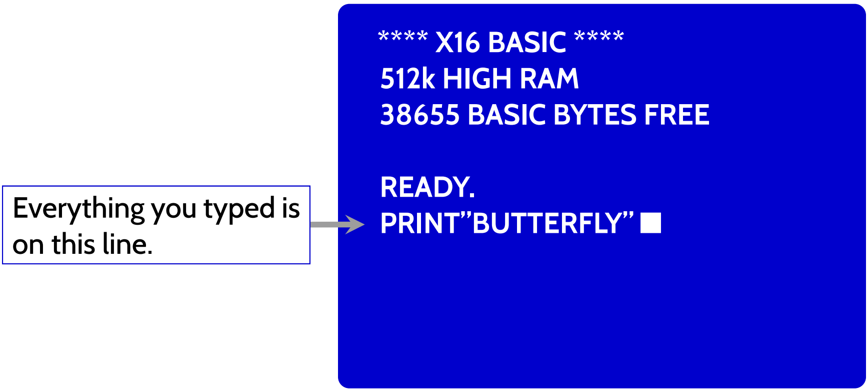
Hold down the [SHIFT] key and press the [" '] key. The screen should now look like this:

Pressing the [" '] key while holding down [SHIFT] caused the " character to by typed instead of the ' character.

Now let't type a word. Without holding down any other keys, press these keys:

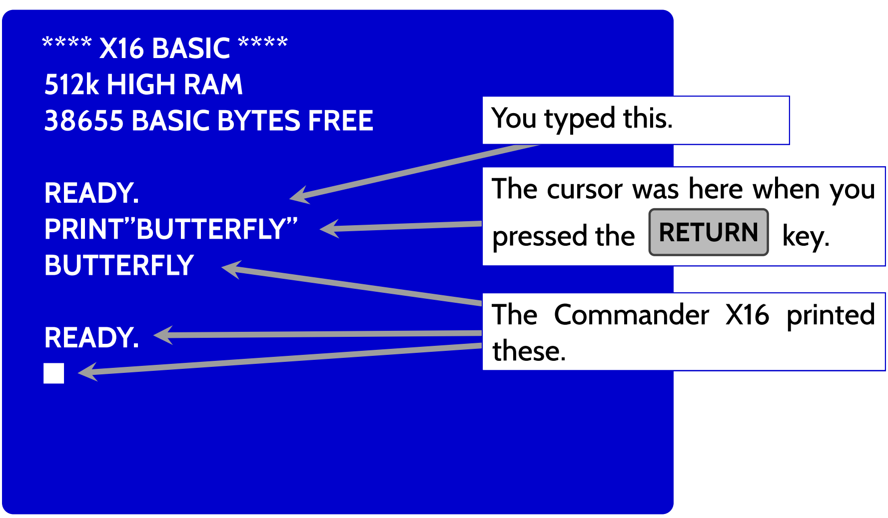
[B] [U] [T] [T] [E] [R] [F] [L] [Y]

Finally, hold down the [SHIFT] key, and press the [" '] key one more time. The screen should now show:



If something doesn't look correct, use the **BACKSPACE** key to delete characters and then you can re-type them.

Once everything looks correct, find the **RETURN** key on the keyboard and press it once. Now look at your screen.



Pressing the **RETURN** key told the X16 that you were finished typing your command. Then the X16 looked at the command you typed, saw that it was something it knows how to do, and then did it. In this case, your command told the X16 to `PRINT` a message to the screen, The X16 knew *what* to print because you told it that as well by placing your message between the quotation marks.

When the X16 finished `PRINT`ing the word `BUTTERFLY`, it let you know by displaying the `READY` message and blinking the cursor.

> **NOTE:**
>
> If you are not using an official Commander X16 keyboard, then you probably won't have a **RETURN** key, but instead have an **ENTER** key. Don't worry, they are the same thing.

Your Own Experiments

Now that you've `PRINT`ed something to the screen, try `PRINT`ing other things. Can you make the Commander X16 say `HELLO`? Can you make it say your name?

Here are some things to keep in mind:

- Make sure you spell the word `PRINT` correctly
- Put your message between quotation marks (`"`). Make sure you have one quotation mark at the beginning and one at the end
- Run your command by pressing **RETURN**
- If something isn't working as you expect, continuing reading to learn about errors

Making A Mistake On Purpose

What happens if you type something wrong? Anyone who spends any amount of time using a computer is going to mistype a command. Let's find out what happens by making a mistake *on purpose*. That way, we understand what is happening when we make a mistake *by accident*. Let's make a mistake!

Try typing our first command, but this time misspell `PRINT` by forgetting the `I` and typing `PRNT` instead:

```
READY.
PRNT"BUTTERFLY" ■
```

This is a very easy mistake to make, and at a glace you won't even notice that the command is wrong. Now press **RETURN** to run this command. You should see an error message:

The X16 lets you know that something is wrong.

```
READY.
PRNT"BUTTERFLY"

?SYNTAX ERROR
READY.
■
```

Printing `?SYNTAX ERROR` to the screen is how the X16 tells you that you typed something that it does not understand. In this case, you typed `PRNT` instead of `PRINT`.

For now, don't worry about these errors. Just do you best to you type your commands correctly before you press **RETURN**.
As you experiment with typing commands, the screen will scroll down to give you more room to type and more room for the Commander X16 to print the results of your commands. You may want to clear the screen

and bring the cursor back to the top. The Commander X16 has a built-in way to do this without even typing a command:

Hold down the SHIFT key and press the CLR HOME key.

This clears the screen immediately and places the cursor at the top of the screen.

X16 TIP: CLEARING THE SCREEN

Clearing the screen will be one of the most frequent things you do while working on your Commander X16. It is worth memorizing the SHIFT CLR HOME key combination so that you don't have to reference this manual every time you want to start with a fresh screen.

You can also clear the screen by typing the `CLS` command and pressing RETURN, but most people prefer to use SHIFT CLR HOME.

Your First Computer Program

Now that you are comfortable typing commands, it's time to write a *series* of commands to be executed at once. This is what is called a *computer program*. Let's begin.

STEP 1: Clear the screen by holding down the SHIFT key and then pressing the CLR HOME key at the same time.

STEP 2: Type N E W and press the RETURN key.

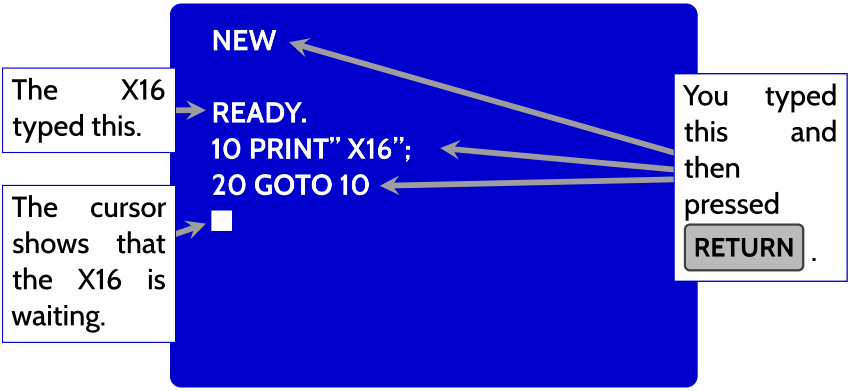
STEP 3: Type 1 0 SPACE P R I N T SPACE " X 1 6 " ; and press RETURN .

STEP 4: Type 2 0 SPACE G O T O SPACE 1 0 and press RETURN .

NOTE:

- The SPACE key is the large, wide key at the bottom of the keyboard. It should be the only key with nothing printed on it.
- The " key is simply the " ' key pressed while holding down the SHIFT key.
- the ; key is the : ; pressed while *not* holding down the SHIFT key. It is next to the " ' key.

When you are finished, the screen will look like this:



X16 TIP: EDITING MISTAKES

You can *retype a line* anytime and the Commander X16 will replace the old line with the new one. For example, if you mistyped the command `PRINT` on line 10:

```
10 PRNNT " X16";
20 GOTO 10
```

You can skip down by hitting RETURN a few times and type:

```
10 PRINT " X16";
```

Now the new line has replaced the old line in your program! If you want to make sure, type L I S T to tell the X16 print out your entire program to the screen. Replacing lines is also a quick way for you to experiment while writing programs.

Typing the line number and immediately hitting RETURN will delete the entire line from your program.

If your program looks correct, it's time to tell the X16 to *run* your program. To do this, type R U N RETURN .

The screen should be filled with `X16`:

```
X16X16X16X16X16X16X16X16X16X1
6X16X16X16X16X16X16X16X16X16X
16X16X16X16X16X16X16X16X16X16
X16X16X16X16X16X16X16X16X16X1
6X16X16X16X16X16X16X16X16X16X
16X16X16X16X16X16X16X16X16X16
X16X16X16X16X16X16X16X16X16X1
6X16X16X16X16X16X16X16X16X16X
16X16X16X16X16X16X
```

This text is scrolling up the screen because the program is continuing to add new text at the bottom. The X16 allows you to slow this down by pressing the CTRL key. Just like the SHIFT key, there are two of CTRL keys on your keyboard; one on each side. Holding CTRL tells the X16 to reduce how fast it prints to the screen. This is useful when debugging programs that move too fast for your eyes to see clearly.

With your program running, you no longer have a cursor that is waiting for you to type. To stop your program and bring back the cursor, press the RUN STOP key. This should stop the program, and display a message, and then print the `READY` prompt followed by the cursor:

```
X16X16X16X16X16X16X16X16X16X1
6X16X16X16X16X16X16X16X16X16X
16X16X16X16X16X16X16X16X16X16
X16X16X16X16X16X16X16X16X16X1
6X16X16X16X16X16X16X16X16X16X
16X16

BREAK IN 10
READY.
■
```

The word `BREAK` is how the X16 tells you that the program has stopped, and it also tells you which line it stopped at. In this case, the program *broke* at line 10. This does not mean anything is broken. It's just the word that the computer uses to let you know that it has stopped in the middle of a program.

Now that the program has stopped running, the cursor reappears to let you know that the X16 is waiting for you to tell it what to do. This allows you to change your program in some way before you run it again. It would be nice to be able to see your program printed to the screen so that you know what to change. To do this, use the `LIST` command by typing L I S T RETURN. You should now see your program on the screen so that you can make edits. When you want to run it again, simply move the cursor to a blank line and use the `RUN` command again. Don't forget to type RETURN after you type the command!

By repeating this process of writing, running, stopping, and editing your program, you can take your time to make your program run the way you want it to run. You don't have to get everything correct right away. Even the best computer programmers rarely get their programs to run correctly the first time it runs.

X16 TIP: EDITING YOUR PROGRAM

When your program is `LIST`ed out to the screen, you can edit it in place by moving the cursor to the lines you want to edit. The cursor can be moved by using the arrow keys on the keyboard. Once you are on the line you wish to edit, you can type over top of the characters that are already there or use the **BACKSPACE** key to delete them and retype the line. On each line you change, make sure to press the **RETURN** key while on the line. Otherwise, the Commander X16 will not replace the old line with the new one.

You have just been introduce to several aspects of the Commander X16 that you will use in many of the later chapters. You have:

- `PRINT`ed messages to the screen.
- Cleared the screen with the **SHIFT** and **CLR HOME** keys.
- Written your first program and created a scrolling display.
- Slowed down the program with the **CTRL** key.
- Stopped the program with the **RUN STOP** key.
- `LIST`ed the program.
- Learned ways to edit your program.

As you explore this guide, you will find yourself using these lessons often. Don't worry if there are things you don't understand. Future chapters will go into more details about what you have learned here. It's also important to know that the best way to learn is by experimenting for you yourself.

This guide is designed so that you can go directly to *any chapter* that looks interesting to you.

2

Using the Screen and Keyboard

Graphic Chracters

Colors

The X16 Keyboard

Screen Modes

Editing Text

Try typing this program:

Type this program exactly as shown and see what happens!

```
10 PRINT " SHIFT CLR HOME "
20 FOR T = 1 TO 900: NEXT
30 PRINT "your name"
40 FOR T = 1 TO 900: NEXT
50 GOTO 10
```

Type R U N and hit RETURN

To stop the program, press the RUN STOP key.

Graphic Chracters

Perspiciatis et voluptate eos et dolores placeat. Sit nisi unde perferendis ad omnis. Vel quidem ipsa aliquid. Beatae qui amet consequatur voluptatibus quaerat eos. Corrupti ab pariatur ipsa ipsam. Esse labore quasi quisquam ipsa reiciendis. Sequi ipsum numquam id sequi similique iure. Et sed illum est. Eum tempora distinctio maxime rem numquam. Rerum dolor sequi sequi beatae corporis incidunt autem. Quisquam iusto dignissimos libero hic aliquam. Excepturi ullam quibusdam accusamus et voluptatem amet. Fugit iure vel debitis dolor in omnis distinctio. In quia pariatur odit quia corrupti autem. Ipsum quo ut dolor ratione vitae. Voluptatem est explicabo sunt nemo. Voluptatibus laborum eveniet iste rerum non asperiores. Cumque velit aspernatur quidem neque tempore. Quia voluptas quis quia perferendis fugiat iure ex atque. Quia beatae quam pariatur ullam quis sunt consequuntur. Aut deserunt ad repellat aut et hic voluptatem. Numquam deleniti ut culpa. Magni eius expedita et dignissimos dolorem perspiciatis libero. Velit rem eius incidunt consequatur sed fugiat non dolor. Laudantium qui laudantium neque cumque veniam eos ducimus.

Colors

Sapiente voluptas aut accusamus. Distinctio et reiciendis ut qui aliquam fuga. Possimus cumque dolores non. Tempora facilis ratione ut cupiditate ducimus tenetur laborum vel. Eligendi sapiente maxime temporibus temporibus qui suscipit. Tempora omnis exercitationem rem perspiciatis sunt. Dolorum quisquam est excepturi est a magni consequatur. Animi non ea provident. Aut nihil non ducimus dolorem voluptatibus eius quos. Distinctio placeat cupiditate consequatur totam reprehenderit nihil est. Voluptatibus voluptatem quam veniam at et. Nobis quos voluptatibus labore autem. Mollitia sed iusto hic eius et. Nobis eaque qui id aliquam odit id. Consequatur quis et eos sint dolor eum omnis quo. Quod voluptatem consequatur odio. Dicta vero numquam minus. Aut maiores est molestiae eligendi occaecati rem suscipit eos. Et maxime aliquam voluptatem. Voluptatem sed neque est. Quia ut omnis exercitationem aperiam tenetur perferendis qui. Qui magni alias consequatur voluptatum rerum a. Exercitationem et magnam nulla odio blanditiis voluptas cum quibusdam. Quo tenetur animi autem quia alias vitae dolorem. Numquam error vero voluptatum.

The X16 Keyboard

Molestiae ad dicta praesentium et. Placeat magnam nihil est animi vel eos. Sunt consectetur nobis minima ut reiciendis hic non sed. Officiis sint voluptas non quo eos architecto. Nulla et est laboriosam voluptatem. Iure sed et ducimus nostrum est eveniet. Natus aut praesentium fugit. In quae tempora sunt autem illum perspiciatis. Amet laborum numquam aut occaecati. Quia ad ab voluptas qui autem. Qui voluptatum quibusdam est aliquam in. Quae ipsum aperiam aut saepe molestiae natus sit. Totam autem veritatis deserunt. Hic ut excepturi porro. Et ut vero voluptas iusto earum velit rerum. Assumenda enim voluptatum praesentium quam. Rerum optio iste odit. Id quia ratione quasi. Doloremque et omnis autem dolor. Vel minima numquam enim asperiores quae magni soluta a. Corrupti sint sit sunt cum sunt asperiores animi rerum. Consectetur sunt itaque ducimus soluta sed quod qui. Blanditiis alias rem ea. Doloremque nobis voluptas eius occaecati mollitia temporibus enim ut. Quia consequuntur molestias quae modi consequatur eveniet consequuntur.

Screen Modes

Minima ut quasi aliquid sapiente quo. Id veritatis ipsa vitae molestias velit modi natus et. Quia incidunt totam laboriosam nostrum sed nihil. Assumenda reiciendis molestiae quidem enim quis. Eius excepturi neque dolorem quia. Nesciunt et consequuntur expedita enim soluta in recusandae. Reprehenderit rerum ut facilis aut eius. Velit eveniet esse dolorem dolores tempore ratione tempora. Iste sequi architecto dolores repellendus rem quia sequi. Voluptas error maxime ipsam est sint saepe. Nulla similique eos dolorem esse nobis autem nam a. Aut fugit quae reprehenderit qui. Minima dolorum nihil sapiente dolorem porro minima id. Perspiciatis natus numquam voluptatum. Qui sint nemo praesentium exercitationem voluptatem esse. Et perferendis praesentium voluptatibus. Occaecati facere eligendi eos eum exercitationem. Nobis aperiam inventore laudantium eius consequatur cupiditate. Tempora dolore culpa magni ut eum sint voluptatibus. Quasi repudiandae necessitatibus repellendus cumque quia dolorum. Illo et ut qui. Ut alias et quod repellat sit nobis. Officiis doloremque quaerat vitae iste. Et nostrum pariatur dolorum iusto nulla quae ab. Et voluptatem itaque quae perspiciatis quia.

Editing Text

Ea repudiandae laboriosam omnis consequatur omnis quam nesciunt est. Hic voluptate explicabo sint pariatur mollitia. Omnis asperiores in praesentium dolor quibusdam. Odit vitae possimus ut recusandae quia sapiente ipsum non. Optio error velit eligendi. Facere itaque tenetur fugiat. Aspernatur magnam tenetur nulla aspernatur architecto. Repellat repudiandae autem sunt qui. Libero vel dolorem sed mollitia. Voluptatem eligendi minima voluptatum facere aut magnam laborum vel. Quis rem officia aliquam nisi quisquam dolor quia. Nobis magni non eos explicabo. Qui nisi in est voluptatem enim a repellendus. Id quia incidunt enim sint impedit recusandae. Ut sit ut neque. Et sit delectus id excepturi. Vero maiores libero minus. Ad accusantium sed ut vitae quia earum. Iusto fugit repudiandae aut. Illum atque et dolores. Suscipit qui culpa dolor. Voluptatibus consequuntur culpa ab vitae. Totam libero harum quia. Ipsa aut minima labore eaque eos ipsam. Nulla quae eius tempore.

3

Graphics

More Stuff

Try typing this program:

Type this program exactly as shown and see what happens!

```
NEW
10 SCREEN 128
20 FOR I=0 to 15
30 Y = I * 15
40 RECT 0,Y,319,Y+14,I
50 NEXT I
```

Type R U N and hit RETURN

To stop the program, press the RUN STOP key.

More Stuff

Minima consequuntur voluptatem nemo et qui adipisci. Voluptate voluptas nesciunt illo labore asperiores. Aut provident quisquam ipsum illum id sed. Aliquam et ipsa exercitationem corrupti iste cum. Voluptatem ducimus maiores magni. Qui ut vitae suscipit nesciunt. Quos amet nemo non aut reiciendis aut sapiente. Veniam tempore in sit rerum quam esse. Quod dolores inventore architecto autem et corporis consequatur. Blanditiis vitae ullam et nostrum. At magnam fugit rerum eaque accusantium facere atque tempora. Natus sunt odit ea accusamus nihil ut. Quos sequi veniam odit quo saepe. Ad maiores molestiae molestias provident ex. Qui perspiciatis molestiae quo nisi soluta ut ullam quasi. Consectetur est iusto in ea ea voluptate. Sunt dolor est omnis aut. Illum autem magnam vero alias animi. Delectus eos ad iste sit occaecati sit. Doloremque eum doloribus inventore autem. Illum quia necessitatibus quia iste aut consequatur. Tempore tenetur perspiciatis aperiam nemo aut est voluptatum reiciendis. Nihil tempora laboriosam quia reiciendis natus quo perferendis. Nihil repellat corrupti illum sit quo nam. Sit laudantium quos hic.

4

Sound

More Stuff

Try typing this program:

Type this program exactly as shown and see what happens!

```
10 FMINIT
20 FMINST 0,0
30 FMINST 1,0
40 FMINST 2,0
110 FMPLAY 0,"T140L6S1O5ED+ED+EO4BO5DC"
120 FMCHORD1,"O4A"
130 FMPLAY 0,"O1AO2EAO3CEA"
140 FMCHORD1,"O3B"
150 FMPLAY 0,"O1EO2EG+O3EG+B"
160 FMCHORD1,"O4C"
170 FMPLAY 0,"O1AO2EAO3EO5ED+"
180 FMPLAY 0,"O5ED+EO4BO5DC"
190 FMCHORD1,"O3A"
200 FMPLAY 0,"O1AO2EAO3CEA"
210 FMCHORD1,"O3B"
220 FMPLAY 0,"L6O1EO2EG+O3EO5CO4B"
230 FMCHORD 0,"L1 O1AO3C+A"
```

Type R U N and hit RETURN

To stop the program, press the RUN STOP key.

More Stuff

Minima consequuntur voluptatem nemo et qui adipisci. Voluptate voluptas nesciunt illo labore asperiores. Aut provident quisquam ipsum illum id sed. Aliquam et ipsa exercitationem corrupti iste cum. Voluptatem ducimus maiores magni. Qui ut vitae suscipit nesciunt. Quos amet nemo non aut reiciendis aut sapiente. Veniam tempore in sit rerum quam esse. Quod dolores inventore architecto autem et corporis consequatur. Blanditiis vitae ullam et nostrum. At magnam fugit rerum eaque accusantium facere atque tempora. Natus sunt odit ea accusamus nihil ut. Quos sequi veniam odit quo saepe. Ad maiores molestiae molestias provident ex. Qui perspiciatis molestiae quo nisi soluta ut ullam quasi. Consectetur est iusto in ea ea voluptate. Sunt dolor est omnis aut. Illum autem magnam vero alias animi. Delectus eos ad iste sit occaecati sit. Doloremque eum doloribus inventore autem. Illum quia necessitatibus quia iste aut consequatur. Tempore tenetur perspiciatis aperiam nemo aut est voluptatum reiciendis. Nihil tempora laboriosam quia reiciendis natus quo perferendis. Nihil repellat corrupti illum sit quo nam. Sit laudantium quos hic.

5

APPENDIX

Commander X16 BASIC

This manual has introduced you to the BASIC language and many of the commands, operators, and conventions. However, that is not enough in order to truly understand how to use BASIC. This appendix is a reference that aims to provide a complete documentation for Commander X16 BASIC. It will provide the rules (known as *syntax*) of the BASIC language, and concise descriptions of each BASIC command.

To make this information easier to read, it is broken up into the following sections:

1. **Variables**: describes what variables are, the different types of variables, and the allowed variable names.
2. **Operators**: describes arithmetic and logical operators.
3. **Commands**: describes the interactive commands that are used to work with programs or perform other tasks that users typically type directly into the `READY` prompt.
4. **Statements**: describes the statements that are typically used in BASIC programs, but aren't often called directly by users from the `READY` prompt.
5. **Functions**: describes the BASIC functions that return values, such as calculations and string operations.

> **NOTE:**
>
> Commands and statements are not technically different, and often these terms are used interchangeably. Commands can be used from within BASIC programs and statements can be run directly from the `READY` prompt. The reason for different labels is because many commands make little sense when used from within BASIC programs. For example, using the `NEW` command inside a BASIC program will cause the program to halt execution and be removed from memory!

Variables

Variables are values that have been given names. Programs use variables for many purposes, and they are an important part of BASIC programming. Programmers can *assign* a value to a variable, and then use that value later in their program by referring to the variable. For example:

```
10 T$ = "X16"
20 PRINT T$
```

The above BASIC program stores the value `"X16"` in a variable named `T$`, and then `PRINT`s the value of `T$` to the screen.

Variables are similar to memory addresses except for a couple of key differences. First, the programmer doesn't have to keep track of where a variable is stored in the Commander X16's memory. This job is performed by BASIC to make the programmer's job easier. Second, variables have a *type*. There are three types of variables in Commander X16 BASIC. The three types of variables are: *floating point*, *integer numeric*, and *string (alphanumeric)* variables.

Floating Point Variables

Floating point numeric variables can have any value from -10^{38} to 10^{38}, with up to nine digits of accuracy. Floating point values can hold partial values, such as 3.4, 42.7, or 0.000025. This makes them useful for a variety of mathematical uses. Floating point variables can be named with any single letter, any letter followed by a number, or with two letters[1]. For example, `A`, `A5`, or `AB`.

To assign a floating point variable, type your chosen name for the variable followed by an ttfamily = and then the value you wish to assign it:

```
A = 3.4
```

[1]There are three variable names that are *reserved* by the Commander X16 for its own use, and cannot be used for variable names in your programs. These names are `ST`, `TI`, `TI$`, and `DA$`

```
A5 = 42.7
AB = 0.000025
```

For numbers that are very large or very small, you may wish to use scientific notation to assign your variables. The Commander X16 understands scientific notation by using the letter `E` to separate the coefficient from the exponent (the base is always assumed to be 10). So to assign the value 3.7×10^{-14} to a floating point variable named `B2`, you would type:

```
B2 = 3.7E-14
```

Not only can you assign floating point variables using scientific notation, but the Commander X16 will also display values in scientific notation if they require more than nine digits.

Integer Variables

Integer numeric variables should be used whenever the number will always be a whole number, and always be between -32768 and 32767. These are numbers like $1, 5$, or -127. Integer variables take up less space in the Commander X16's memory, and doing math with integers is faster than with floating point numbers. Integer numeric variables follow the same rules as floating point variables, except they must have a `%` character at the end. For example:

```
B% = 5
C5% = -11
BC% = 1261
```

NOTE:

Sometimes when writing numbers we place a , to separate groups of three digits, such as $1,000$ or $8,006,029,545$. While this makes numbers easier for humans to read, it is not something that Commander X16 understands. When typing numbers into your programs, you should never use a , but instead type the numbers without it. So the previous numbers would be typed as 1000 and 8006029545.

String Variables

String variables are used to store characters, such as words, sentences, or any other symbol that you can type. A single string variable can store either a single character, many characters in a row, or even no characters at all!String variable names follow the same rules as floating point variables, except they must have a `$` character at the end. The value of a string variable must be enclosed in quotation marks. For example:

```
N$ = "COMMANDER X16"
B8$ = "SEVEN"
DC$ = "THE NEXT STRING HAS NO CHARACTERS IN IT"
EC$ = ""
```

Arrays

Arrays are lists of variables that all share the same name. You can specify which item, or *element*, in the list you are using by using a number. For example, if you have an array of floating point values in a variable named `AB` you can use the second value in the array by typing `AB(2)` where you would normally type a variable name or a value. You can create an array that holds any of the above types of variables, but a single a array can only hold one type of variable. So an array that was created to hold seven strings can *only* hold string variables, and will cause an error if you try to assign an integer to one of the elements.

Unlike other variables, array variables usually[2] need to be *declared* before using them. You can declare your array variable with the `DIM` statement like so:

```
DIM A(25)
```

This will tell the Commander X16 to reserve enough memory for twenty-five floating point variables. You can access these variables by *indexing* the array variable `A` when using it, like so:

```
PRINT A(14)
```

The above example prints the value of the fourteenth *element* of `A` to the screen.

Arrays can have more than one *dimension* by declaring them with more than one index. For example a two-dimensional array can be useful for storing data arranged as rows and columns. Here is how you would declare an array with 24 rows of 32 columns:

```
DIM S%(32,24)
```

The above array can store 32×24 integer values. You could even declare arrays with even higher dimensions if you have a need for it. Be warned, however, as higher dimensional arrays take up exponentially more memory so you will quickly run out.

Operators

Commander X16 BASIC uses three different types of *operators*: *arithmetic* operators, *comparison* operators, and *logical* operators.

Arithmetic Operators

Arithmetic operators are used for mathematical calculations. Here are the available arithmetic operators:

[2] see the documentation of the `DIM` statement for exceptions

- `+` addition
- `–` subtraction
- `*` multiplication
- `/` division
- `↑` raising to a power (exponentiation)

When several operators are used in the same arithmetic expression, there is an *order* in which the operations execute. First, any exponentiation operations execute. Next, any multiplication or division operations execute. Finally, any addition or subtraction operations execute. When there are two or more operations that execute at the same time, such as a multiplication followed by a division, the operations execute from left to right. Consider the following:

```
PRINT 2/4/2
```

The above code will execute and print `.25` to the screen instead of printing `1`. This is because `2/4` executes first to produce `.5`, and then `.5/2` executes to produce a final value of `.25`. If desired, you can force the order of operations by enclosing calculations inside parentheses. For example, we could reverse the order of the operations above by typing:

```
PRINT 2/(4/2)
```

Now the result is `1` because `4/2` is executed first to produce `2`, and then `2/2` executes to produce `1`.

Using parentheses is a good practice even when not necessary, because it makes the intention of the calculation obvious when reading the code. Had we used them in the original example, it would have made the execution obvious at first glance:

```
PRINT (2/4)/2
```

The above code is identical to `2/4/2`, but is easier to read.

Comparison Operators

Comparison operators are useful for determining equalities and inequalities. These are used comparing values against each other to determine if they are the same, not the same, or which is larger. The comparison operators are:

=	is equal to
<	is less than
>	is greater than
`<= or =<`	is less than or equal to
`>= or =>`	is greater than or equal to
`<> or ><`	is not equal to

Comparison operators are most often used with `IF...THEN` statements. For example:

```
A = 12
IF A > 10 THEN PRINT "GREATER THAN 10"
```

As you can see from the code above, both variables and literal values can be used with comparison operators.

Logical Operators

Logical operators are used to join together multiple comparison statements into a single statement. There are three logical operators:

`AND`	is true if both the left side and the right side are true
`OR`	is true if either the left side or the right side are true
`NOT`	is true if the right side is false

By using these logical operators, you can write complex conditions for your programs. Here's some examples:

```
IF A = B AND C = D THEN 100
IF A = B OR NOT (C = D) THEN 100
```

Notice how parentheses can be used to explicitly force the order in which logical conditions are evaluated, just like how they force the order in which arithmetic is evaluated.

Commands

Commands are instructions that you type in order to work with programs on the Commander X16 or perform other user tasks. Commands tell the Commander X16 to do things, such as `LIST` the contents of the SD card, `LOAD` a program from the SD card, or `RUN` the currently loaded program. This section contains a description of each command in alphabetical order.

BANNER

The `BANNER` command displays the Commander X16 logo and boot text, like is automatically displayed upon boot.

BOOT

The `BOOT` command loads and runs a PRG file named `AUTOBOOT.X16` from device #8 (the SD card reader). If the file is not found, nothing is done and no error is printed.

CLR

The `CLR` command clears the BASIC variables from memory. This includes variables that were assigned values while running BASIC programs as well as any BASIC assignments that were called from the `READY` prompt directly. Variables cleared with `CLR` cannot be restored with the `OLD` command.

The `CLR` command runs automatically whenever the `RUN` command is called, so that each run of a program starts with a cleared variables state. `CLR` is *not* called when the `CONT` command is run, so that a prgram can continue where it left off with the variable state in tact.

CLS

The `CLS` command clears the screen. This has the same effect as typing `PRINT CHR$(147);` or typing SHIFT + CLR HOME. This command is useful when programs and commands have cluttered up the screen, and is also useful in BASIC programs to `PRINT`ing to an empty screen.

CONT (continue)

When a program has been stopped by either using the RUN STOP key, a `STOP` statement, or an `END` statement within the program, it can be restarted by using the `CONT` command. The `CONT` command will continue executing the loaded program at the exact place from which it left off, with all the variables intact.

The `CONT` command will not always work, however. If you make any modifications to your program while it is stopped, the `CONT` command will fail and display a `CAN'T CONTINUE ERROR`. This is true even if you `LIST` the program and hit RETURN while the cursor is on a line of the program...even if you didn't make any modifications. To the X16, this is still considered a change to the program, so the only way to run it again is to start at the beginning of the program by using the `RUN` command.

DOS

This command works with the command/status channel or the directory of a Commodore DOS device and has different functionality depending on the type of argument.

- Without an argument, DOS prints the status string of the current device.
- With a string argument of `"8"` or `"9"`, it switches the current device to the given number.
- With an argument starting with `"$"`, it shows the directory of the device.
- Any other argument will be sent as a DOS command.

Examples:

```
DOS"$" : REM SHOWS DIRECTORY
DOS"S:BAD_FILE" : REM DELETES "BAD_FILE"
DOS : REM PRINTS DOS STATUS
```

HELP

The `HELP` command displays a brief summary of the current ROM build, VERA version, and PS/2 microcontroller code version. It also displays some URLs for documentation and community support. All of this information could be useful in troubleshooting issues the Commander X16.

KEYMAP

The `KEYMAP` command sets the current keyboard layout. It can be put into an `AUTOBOOT.X16` file to always set the keyboard layout on boot.

Example:

```
10 REM PROGRAM TO SET LAYOUT TO SWEDISH/SWEDEN
20 KEYMAP "SV-SE"
SAVE"AUTOBOOT.X16" :REM SAVE AS AUTOBOOT FILE
```

LIST

The `LIST` command will print the currently loaded BASIC program to the screen, either in its entirety or only the parts specified by the user. When `LIST` is used without any numbers typed after it (known as *arguments*), you will see a complete listing of the program on your screen. If the program scrolls off the screen, and you are unable to see the part that you want, you have a couple of options. First, you can use the **CTRL** key to slow down how fast lines are printed to the screen. The part you wish to see will still scroll off eventually, but you will be given a much longer time to look at it. Second, you can use the `LIST` command with arguments that will limit the listing to only the line or lines that you wish to see. When you follow the `LIST` command with a single number, the X16 will list only that line number (if it exists). If you follow `LIST` with two line numbers separated by a dash, the X16 will list all the lines from the first number to the second (including both line numbers). If you follow `LIST` with a dash followed by a single number, it lists from the beginning of the program up to and including the line number. Finally, if you follow `LIST` with a number followed by a dash, it lists from the line number until the end of the program.

Examples:

LIST	Shows entire program.
LIST 10-	Shows only from line 10 through the end.
LIST 10	Shows only line 10.
LIST -10	Shows from the beginning through line 10.
LIST 10-20	Shows lines from 10 through 20.

LOAD

The `LOAD` command is used when you want to use a program that is stored on the Commander X16's SD card[3]. Typing `LOAD` and hitting **RETURN** will find the first program on the SD card[4] and bring it into memory to be `RUN`, `LIST`ed, or edited. You can also type `LOAD` followed by a name of a file in quotes(`" "`) to specify which file to load into memory. The file name argument may be followed by a comma and a numeric value which specifies a device number. If no number is given, the X16 uses device #8, which is the SD card reader.

Examples:

[3] the `LOAD` command can also be used with other devices, but only the SD card reader ships with the Commander X16

[4] The SD card uses the FAT32 disk format, so it's complicated what makes a file considered to be the "first". It is safer to specify the name of the file when possible

LOAD	Loads the first program on the SD card into memory.
LOAD "HELLO.PRG"	Loads a program named `HELLO.PRG` from the SD card into memory.
LOAD A$	Loads a program whose name is stored in the string variable `A$` from the SD card into memory.
LOAD "HELLO.PRG",9	Loads a program named `HELLO.PRG` from the drive configured as device #9.

There are also special file names that can be loaded that perform specific tasks when used with `LOAD`:

LOAD "*",8	Loads the first program on device #8 into memory.
LOAD "$"	Loads a directory listing of the SD card into memory which can be displayed with `LIST`.

The `LOAD` command can be used with a BASIC program to load and `RUN` another program.

MENU

The `MENU` command presents the user with a menu of built-in programs stored in the X16's ROM. The user can then select a program to run, or return to BASIC.

MON

The `MON` command causes the Commander X16 to enter the machine language monitor.

NEW

The `NEW` command marks the current program and its variables as erased, but leaves them in memory. This behavior is so that both the program and its variables can be restored with the `OLD` command. The effect is that the Commander X16 is ready for a new program.

OLD

The `OLD` command recovers the BASIC program in RAM that has been previously marked erased either by using the `NEW` command, by pressing the reset button on the case, or by pressing the **CTRL** + **ALT** + **DEL** key combination on the keyboard.

POWEROFF

The `POWEROFF` command turns off the Commander X16. It is equivalent to pressing the Power button on the motherboard or case.

REBOOT

The `REBOOT` command performs a software reset of the system by calling the ROM reset vector. This performs all the boot routines in the system ROM, but does not force the hardware to reset. The `REBOOT` command *does not* clear memory, but *does* clear any BASIC program that is loaded. Because it is still in memory, however, a previously loaded BASIC program can be re-loaded by using the `OLD` command after a reboot:

```
NEW
10 PRINT "EXISTING PROGRAM"
REBOOT
```

```
**** X16 BASIC ****
512k HIGH RAM
38655 BASIC BYTES FREE
READY.
OLD
LIST
10 PRINT "EXISTING PROGRAM"
READY.
■
```

REN

The `REN` command renumbers a BASIC program while updating the line number arguments of `GOSUB`, `GOTO`, `RESTORE`, `RUN`, and `THEN`. The `REN` command takes three optional arguments:

- The line number of the first line after renumbering, default: `10`
- The value of the increment of subsequent lines, default: `10`
- The ealiest old line to start renumbering, default: `0`

Example:

```
10 PRINT "HELLO"
15 PRINT "CLEAN"
20 PRINT "THIS"
40 PRINT "UP"
```

```
REN 100,5

LIST
100 PRINT "HELLO"
105 PRINT "CLEAN"
110 PRINT "THIS"
115 PRINT "UP"
```

RESET

The `RESET` command performs a full system reset, but does not clear memory. This means that a BASIC program and its variables can be restored after a `RESET` by using the `OLD` command. Unlike the `REBOOT` command, this triggers the Commander X16's hardware reset line, which may be used by expansion ports to reset custom hardware.

NOTE:

There are multiple ways to reset a Commander X16, and each produces a slightly different result.

The first is by using the **CTRL** + **ALT** + **RESTORE** key combination. This halts the execution of any program, clears the screen, and returns the user to the `READY` prompt. It does not mark a program or its variables as erased, and so a program can be `RUN` again or `CONT`inued if desired. This is equivalent to pressing the NMI button on the motherboard.

The second is by using the `REBOOT` command. This calls the initial boot up routines, clears any BASIC program, but does not clear the memory. A previously loaded BASIC program can be restored with the `OLD` command.

The third is by using the `RESET` command. This is equivalent to pressing the reset button on the motherboard or pressing the **CTRL** + **ALT** + **DEL** key combination.

The fourth is a cold reboot, where the power to the Commander X16 is lost and then restored. This causes the current program and its variables to be completely lost and they cannot be restored with the `OLD` command.

RUN

The `RUN` command executes the program currently loaded into memory. This program could have been typed in, or it could have been loaded from the SD card with the `LOAD` command. When called, the `RUN` command will clear the BASIC variables (just like calling the `CLR` command) and begin running the program. When no number follows the `RUN` command, the program will start executing from the lowest line number in the program. Otherwise, `RUN` will start executing at the given line number, or the next lowest line number in the program.

Examples:

RUN	Starts program from lowest line number.
RUN 50	Starts program at line 50.
RUN A	UNDEFINED ERROR (`RUN` cannot be used with a variable to specify a line number).

SAVE

The `SAVE` command will store the the current program in memory to the SD card or another storage device. The `SAVE` command should be followed either by a file name in quotation marks, or a string variable that contains the desired file name[5]. The file name argument can be followed by a comma and a number or numeric variable. This number tells the Commander X16 which device to store the file on. Device number 8 is the SD card drive and is used if no number is given.

If a tape dirve is used with the Commander X16, then a second numeric argument of either `0` or `1` can be specified after the device number. If this second is a `1`, an `END-OF-TAPE` marker will be written after the program. If you are attempting to `LOAD` a program off a tape drive and this marker is read before finding the desired file, a `FILE NOT FOUND ERROR` will be displayed.

Examples:

[5] calling the `SAVE` command without any arguments is technically allowed, but doesn't do anything. On the Commodore VIC-20 and Commodore 64 this was useful for saving the current program to the current position of a tape drive with no name, but the Commander X16's default device is an SD card reader where this concept makes no sense. For historical reasons, the Commander X16 won't display an error if you run `SAVE` with no arguments, but it also won't do anything

SAVE "HELLO.PRG"	Saves the program in memory to a file on the SD card with the name `HELLO.PRG`.
SAVE A$	Saves the program in memory to a file on the SD card the name contained in the variable`A$`.
SAVE "HELLO.PRG",1	Saves the program in memory to a file on the drive configured as device #1 with the name `HELLO.PRG`.
SAVE "HELLO.PRG",1,1	Saves the program in memory to a file on the drive configured as device #1 with the name `HELLO.PRG` and writes an `END-OF-TAPE` marker after it.

VERIFY

The `VERIFY` command will compare the program in memory to a program on the SD card or other storage device. If the programs are the same, the `VERIFY` command will display an `OK` message, and if they differ it will display a `VERIFY ERROR` message. This command helps to ensure that a program is safely stored to the SD card or other storage device before the user erases it from memory[6]. When `VERIFY` is called without any arguments, it checks the program in memory against the first file on the SD card[7]. When called followed by a file name in quotation marks or a string variable containing a file name, it compares the program in memory against the given file. Just like the `LOAD` and `SAVE` commands, the `VERIFY` command can take a numeric second argument as a device number.

The `VERIFY` command can also be used if a tape drive is connected to the Commander X16 as a storage device. By `VERIFY`ing the last pro-

[6]this was far more useful in the era or tape drives and floppy disks than it is on the Commander X16

[7]this is not particularly useful, but is included behavior for historical reasons

gram on the tape, the position of the tape can be advanced to a safe section to write over. When `VERIFY` is complete, whether verification succeeds or fails, the tape will be positioned at the next available space.

Examples:

`VERIFY`	Checks the first program on the SD card.
`VERIFY A$`	Checks the program with name in variable `A$`.
`VERIFY "HELLO.PRG",1`	Checks the program on the drive configured as device #1 with the name `HELLO.PRG`.

Statements

Statements are the instructions used in BASIC on numbered lines of programs. They are used to define what it is that your program does.

BANK

The `BANK` statement sets which bank will be used when other commands and statements interpret addresses in the $A000 - $FFFF range. Because all addresses from $A000 and above are either banked "high" RAM or banked ROM, certain commands need to know *which* bank is being referred to. Specifically, `SYS`, `POKE`, and `PEEK` all need to know which bank to use when an address is given in the banked range. The `BANK` statement sets the bank for both banked RAM[8] and banked ROM, although setting the banked ROM is optional. The first argument is used to set the RAM bank, and the optional second argument sets the ROM bank. To set a bank, call the `BANK` statement followed by a numeric value from 0 through 255.

[8] RAM in bank 0 is reserved for use by the KERNAL, so it is unwise to write values into there

For example, to write some data into "high" RAM in bank 1:

```
10 BANK 1
20 POKE $A000,42
```

Then the bank can be switched, and the same address can be used to store more data, without overwriting the data in bank 1:

```
30 BANK 2
40 POKE $A000,23
50 BANK 1 REM SWITCH BACK TO BANK 1
60 PRINT PEEK($A000) REM PRINTS 42, NOT 23
```

The `BANK` statement also has some use as a command run from the `READY` prompt. It can be used to run programs that are shipped with the Commander X16 in banked ROM. For example, the CodeX16 Interactive Assembly Environment in ROM bank 7 can by run by typing the following at the `READY` prompt:

```
BANK 1,7
SYS $C000
```

BINPUT#

The `BINPUT#` statement reads a block of data from an open file and stores the data into a string variable. The `BINPUT#` statement takes 3 arguments; the device number, the string variable to store the data into, and the number of bytes to read from the file. If there are fewer bytes to be read than the specified number of bytes has been read, only the bytes available will be stored in the string variable. If the end of the file is reached, the special variable `ST` will have its bit 6 set to `1`. This means `ST AND 64` will equal `TRUE` when `BINPUT#` reads all the way to or past the end of the file.

Example:

```
10 OPEN 8,8,8,"FILE.BIN,S,R"
20 BINPUT# 8,A$,5
```

```
30 PRINT "I GOT";LEN(A$);"BYTES: ";A$
40 IF ST AND 64 THEN 60 REM END OF FILE
50 GOTO 20
60 CLOSE 8
70 PRINT "FINISHED READING"
```

BLOAD

The `BLOAD` statement loads a headerless[9] file from a device into banked RAM. If the file is too large to fit within a bank, the `BLOAD` statement will automatically continue writing the file to the next bank. This allows file resources larger than 8 kilobytes to be used without the need to break them up into smaller files. This is useful for loading resources for games and applications into "high" RAM that programs can then access during execution.

Examples:

`BLOAD "MYFILE.BIN",8,1,$A000`	Loads a file named "MYFILE.BIN" from device 8 starting in bank 1 at $A000.
`BLOAD "WHO.PCX",8,10,$B000`	Loads a file named "WHO.PCX" from device 8 starting in bank 10 at $B000.

BVERIFY

The `BVERIFY` statement compares a headerless file on the SD card or other storage device to the contents of banked RAM. As arguments, the `BVERIFY` statement takes the name of the file, the device number, the bank number, and the starting address within the bank. If the file in question extends past the end of the bank, the `BVERIFY` statement will automatically continue checking on the file on the next bank, reset-

[9] typically on Commodore computers as well as the X16, files are expected to contain a two-byte header that indicates an address where they are to be loaded into memory. A "headerless" file will not have those two bytes

ting the address to `$A000` as it changes the bank. This allows it to be used to verify files that are too large to fit inside a single bank of "high" RAM.

Examples:

BVERIFY "MYFILE.BIN",8,1,$A000	Compares a file named "MYFILE.BIN" from device 8 against the RAM in bank 1 starting at $A000.
BVERIFY "WHO.PCX",8,10,$B000	Compares a file named "WHO.PCX" from device 8 against the RAM in bank 10 starting at $B000.

BVLOAD

The `BVLOAD` statement loads a headerless file directly into the VERA's VRAM. For arguments, the `BVLOAD` statement takes the file's name, the device number where the file is stored, the bank of VRAM on the VERA (either `0` or `1`), and the address within the bank in which to load.

Examples:

BVLOAD "MYFILE.BIN",8,0,$4000	Loads a file named "MYFILE.BIN" from device 8 into VRAM at address $04000.
BVLOAD "MYFONT.BIN",8,1,$F000	Loads a file named "MYFONT.BIN" from device 8 into VRAM at address $1F000.

To load a file that has a two-byte header, see the `VLOAD` statement.

CHAR

The `CHAR` statement draws text to the screen at a given X,Y coordinate and a given color. The `CHAR` statement is only available in graphics mode, and draws the text to the bitmap graphics layer instead of the text layer. Like other graphics mode statements, the `CHAR` statement can draw in all 256 available colors.

```
10 SCREEN $80
20 CHAR 120,100,14,"COMMANDER"
30 CHAR 180,100,2,"X16"
```

CLOSE

The `CLOSE` statement completes and closes any files used by `OPEN` statements. The `CLOSE` statement takes a single argument that is the file number to be closed.

Examples:

`CLOSE 0`	Close file 0
`CLOSE 4`	Close file 4

CMD

The `CMD` statement is used to send output that would normally go to the screen to some other device instead. The other device could be a file on the SD card, a file on a disk or tape drive, a printer, a modem, or any other device supported by the Commander X16. The device must first be opened with the `OPEN` statement followed by a numerical value that will be used to reference the file or device.

Example:

OPEN 1,8,8,"NEWFILE,S,W"	OPEN a file named `NEWFIL` on the SD card
CMD 1	All normal output now goes to a file named `NEWFILE`
LIST	The `LIST`ing goes to the file, not the screen – even the word `LIST`
PRINT# 1	Direct the output back to the screen before closing the device
CLOSE 1	Close file 1

If a BASIC error occurs the data ouput is switched back to the screen, with the side effect that space characters will be send to the logical file of the selected device. This is why the `PRINT#` statement should be used prior to closing the device.

COLOR

The `COLOR` statement sets the text mode foreground color, and optionally the background color. It takes either one or two arguments, both are integers from 0 through 15. The first argument sets the color of the text, and the optional second argument sets the background color. The numbers given correspond to the first 16 colors of the VERA's palette[10].

Examples:

COLOR 2	Set the text color to red.
COLOR 5,0	Set the text color to green and the background color to black.

The `COLOR` statement only effects areas of the screen where new text

[10] If the VERA's palette has been modified, then the modified colors are used. The `COLOR` statement will not restore the default VERA palette

is placed, and will not change existing characters. This makes it convenient to use in programs for drawing diagrams and images with PETSCII characters, since you can change both foreground and background colors for each individual charater.

DATA

The `DATA` statement creates a data section of a BASIC program from which the `READ` statement will read from. The `DATA` statement is followed by a comma-separated list of values. These values can be integers, floating point numbers, or strings[11]. It is important to use the correct variable type when `READ`ing these values, otherwise a `TYPE MISMATCH` error can occur. If two commas have nothing between them, the value will be interpreted as a `0` for a number or an empty string.

Multiple `DATA` statements can be used in a program, and when one has been completely read by enough `READ` statements, the next `READ` statement will read from the next `DATA` statement. All `READ` statements in a program can be thought of as a single contiguous block of data, even if the statements are not grouped together in the program.

Examples:

```
10 READ A
20 READ B%
30 READ C$
40 PRINT A, B%,C$
50 DATA 34.2,42
60 DATA "COMMANDER X16"
```

`DATA` statements do not need to be executed, so they can slow down a program if placed before code does need to execute. Because of this, it is best to place all `DATA` statements at the end of program.

[11] String values can be specified with or without quotation marks, unless they contain a space, comma, or colon. Despite this flexibility, it is best practice to always use quotation marks for string data

DEF

The `DEF` statement defines a calculation as a named function that can be called by BASIC later. This is useful for complex calculations that a program does multiple times. The `DEF` statement is followed by the function name, which must be `FN` followed by one or two other characters that make up a legal variable name. This name is followed by a set of parentheses enclosing a legal numeric variable name. This is followed by an equals sign and the formula you want to define, using the variable in the parentheses like any other variable would be used.

Examples:

```
10 DEF FNC(R)=2*R*π : REM CIRCUMFERENCE OF A CIRCLE
20 DEF FNA(R)=π*(R↑2) : REM AREA OF A CIRCLE
30 PRINT "CIRCUMFERENCE:", FNC(10)
40 PRINT "AREA:", FNA(10)
```

Only a single variable can be defined as an argument to a function. The `DEF` statement can only be used in a BASIC program (entered with a line number), but the defined function can be used anywhere a BASIC function can normally be used.

> **NOTE:**
>
> The · character represents the number PI and is entered by pressing SHIFT + ← on the keyboard. If you are not using an official Commander X16 keyboard, then you would need to press SHIFT + ~ ‘ instead.

DIM

The `DIM` statement is used to *dimension* an array, which means to allocate enough space for the data the array will hold. An array variable needs to be `DIM`'d before using it unless it will only hold eleven or fewer elements. In all other cases, the `DIM` statement must be used.

To dimension an array variable, use `DIM` followed by the variable name. Then, the size of each dimension of the array should be given, separated by commas and surrounded by parentheses. An array can have one or more dimensions, and each dimension can be as large as needed[12]. The total number of elements in an array can be calculated by multiplying the size of each of the array's dimensions. The `DIM` statement can dimension multiple arrays at once by separating each array with a comma.

Examples:

`DIM A(16)`	An array of 16 numbers.
`DIM B$(26),C%(13)`	An array of 26 strings and an array of 13 integers.
`DIM D(32,4,4)`	A 3-dimensional array of numbers where the dimensions are 32, 4, and 4.

Once dimensioned, arrays can be used just like other variables, except that the index into each dimension must be specified:

```
10 DIM A$(3,12)
20 A$(1,1) = "ONE"
30 A$(2,6) = "TWO"
40 PRINT A$(1,1) : REM PRINTS "ONE"
50 PRINT A$(2,6) : REM PRINTS "TWO"
60 PRINT A$(1,6) : REM PRINTS NOTHING
```

Executing a `DIM` statement on the same array more than once will cause an error. It is a best practice to keep all the `DIM` statements towards the beginning of a program.

END

The `END` statement will stop a running program just as if it had run out of lines. The `CONT` command can then be used to start the program

[12] as long as all the elements fit into memory

again, starting from the line after the `END` statement. In this way, the `END` statement can be used to "pause" a program and allow the user to perform other tasks before `CONT`inuing on with the rest of the program. For details about the limitations of `CONT`inuing a program, see the documentation for the `CONT` command.

FMCHORD

The `FMCHORD` statement instructs the FM synthesis chip to begin playing multiple notes at the same time. For arguments, the `FMCHORD` statement accepts a channel and a string. Because a chord plays multiple notes at the same time, the channel argument specifies the *first* channel to use for the chord, but other channels will be used for subsequent notes. For example, if you specify a channel argument of `3` for a chord which plays 4 notes, the `FMCHORD` statement will play the notes on channels 3, 4, 5, and 6. It is important to set each of the channels to use the desired instruments with the `FMINST` statement. The string argument is used to specify which notes the `FMCHORD` statement will play. For more information on specifying notes, see the chapter on Sound.

Example:

```
REM PLAY A C MAJOR CHORD ON A PIANO
10 FMINST 0,0:FMINST 1,0:FMINST 2,0
20 FMCHORD 0,"CGE"
```

FMDRUM

The `FMDRUM` statement plays a single percussion sound from a set of percussion instruments. This set comes from the General MIDI standard Percussion set[13], which uses numbers from 25 through 87. The `FMDRUM` statement takes two arguments. The first is a channel, and the second is a drum number from the set. When this statement executes, it sets the channel to the selected drum number and plays it. The channel will retain the drum number set until it is set again with either `FMINST` or another call to `FMDRUM`.

[13] see the Drum Patch Presets table in the appendix

Examples:

FMDRUM 0,38	Play an acoustic snare on channel 0.
FMDRUM 2,50	Play a high tom on channel 2.
FMDRUM 1,55	Play a splash cymbal on channel 1.

FMFREQ

The `FMFREQ` statement plays a note on the FM synthesis chip at a given frequency. This is an alternative to playing a note with `FMNOTE`, where instead of specifying a musical note, a frequency in Hertz is specified. Like `FMNOTE` and `FMDRUM`, `FMFREQ` returns immediately and does not wait for a note to finish playing. If a Hertz value of `0` is specified, the channel is immediately silenced.

Examples:

FMFREQ 3,2600	Plays the instrument on channel 3 at 2,600hz.
FMFREQ 0,440	Equivalent to `FMNOTE 0,$4A` which plays A above middle C on channel 0.
FMFREQ 2,0	Silences channel 2.

FMINIT

The `FMINIT` statement is used to set the FM synthesis chip to a known state, and takes no arugments. It performs initializations on the YM2151 sound chip, as well as loading default patches into all 8 channels. In addition, it immediately silences the channels. This last function can be useful for silencing multiple FM channels at once, without having to call `FMNOTE` or `FMFREQ` on each one. The initializations that the `FMINIT` statement performs are called automatically when the Commander X16 boots up, so it is not necessary to call `FMINIT` directly before using other FM statements. However, it is still a good idea to call `FMINIT` before using FM statements in a program, especially if the program relies on the default patches. There's no guarantee that another program hasn't modified the state of the FM chip since boot.

FMINST

The `FMINST` statement assigns an instrument to a channel. The first argument is the channel, and the second argument is a number indicating an instrument. The Commander X16's ROM chip comes pre-loaded with 146 FM instrument patches from the General MIDI Instrument Set. These instruments and their numbers can be found in FM Instrument Patch Presets table and the Extended FM Instrument Patch Presets table in the appendix.

`FMINST 0,0`	Set channel 0 to Acoustic Grand Piano.
`FMINST 3,11`	Set channel 3 to Vibraphone.
`FMINST 7,127`	Set channel 7 to Gunshot.

FMNOTE

The `FMNOTE` statement plays a single note on the FM synthesis chip. The first argument is the channel, and the second argument specifies which note. The note argument can be any number, but is intended to be specified with hexadecimal notation. This is so that the most significant 4 bits (often called the "high nybble") represent the octave while the least significant 4 bits (the "low nybble") represent the musical note. The lowest note of any octave is **`C`**, which is represented with a `1`, and the highest note of any octave is **`B`**, which is represented with a **`C`**. A note of `0` on any octave will release the note playing on that channel, and the note values **`D`**, **`E`**, and **`F`** have no effect.

Although this may seem confusing, it is actually convenient for most uses. For example to play a "middle C" the note value `$41` would be used. The `$` tells BASIC that the value is hexadecimal, the `4` indicates the note is in the 4th octave, and `1` specifies the note "C". Here's a table of which nybble produces which note:

Nybble	$x0	$x1	$x2	$x3	$x4	$x5	$x6
Note	Release	C	C♯/D♭	D	D♯/Eb	E	F
Nybble	$x7	$x8	$x9	$xA	$xB	$xC	$xD-$xF
Note	F♯/G♭	G	G♯/A♭	A	A♯/B♭	B	no-op

Negative numbers can also be used to specify notes. These will be treated as the same note, except it will merely change an already-playing note rather than re-triggering it. This is obviously more useful with some instruments than with others, but can also be used as a clever way to create sound effects.

Examples:

```
10 FMINST 1,64 : REM LOAD SOPRANO SAX
20 FMNOTE 1,$4A : REM PLAYS CONCERT A
30 SLEEP 50 : NEXT X : REM DELAYS FOR A BIT
40 FMNOTE 1,0 : REM RELEASES THE NOTE
50 SLEEP 10 : NEXT X : REM DELAYS FOR A BIT
60 FMNOTE 1,$3A : REM PLAYS A IN THE 3RD OCTAVE
70 SLEEP 25 : NEXT X : REM SHORT DELAY
80 FMNOTE 1,-$3B : REM A# WITHOUT RETRIGGERING
90 SLEEP 25 : NEXT X : REM SHORT DELAY
100 FMNOTE 1,0 : REM RELEASES THE NOTE
```

FMPAN

The `FMPAN` statement is used to control the stereo output of an FM channel. It takes an argument for the channel, and an argument for which speaker the channel should play from. The second argument values are as follows:

Left	1
Right	2
Both	3

Examples:

FMPAN 0,3 Set channel 0 to play from both speakers.

FMPAN 3,1 Set channel 3 to play from only the left speaker.

FMPAN 7,2 Set channel 7 to play from only the right speaker.

FMPLAY

The `FMPLAY` statement plays a musical melody on a single channel. `FMPLAY` takes two arguments; a channel and a string of characters that tells the FM chip what to play. This second argument is specified in a custom macro language[14], and includes notes, releases, tempos, octaves, rests, and other musical elements. For example, the following statement will play a major scale in the key of **C**:

```
FMPLAY 0,"CDEFGAB>C"
```

Each letter indicates which note to play. Before playing the final **C** note, the > character is used to tell the channel to move up one octave. If this character wasn't included, the final **C** would play at the same octave as the first **C**.

The characters + and
ttfamily - can be placed after a note's letter to indicate sharps and flats, respectively. For example, the following will play a major scale in the key of **A**:

```
FMPLAY 0,"AB>D-DEF+A-A"
```

Because a new octave starts on each **C** note, this scale requires the > character to be placed between the **B** and **D♭** notes.

[14] for a complete guide, see the Macro Language for Music appendix

X16 TIP: RESTORING OCTAVES

You may have noticed that running these `FMPLAY` statements multiple times results in them playing in different octaves. This is because the Commander X16 remembers which octave each channel was left in. So when using > to increase the octave on one `FMPLAY` statement, the channel stays in that octave during the next `FMPLAY` statement. This can be solved in a number of ways:

- An initial octave can be specified at the beginning of the string with the `O` macro
- A < character can be placed at the end of the string to indicate that the channel should move down one octave
- The octave can be reset with the `FMINIT` statement. Just keep in mind that this resets the octaves of *all* channels, as well as silencing them and restoring them to their default instruments

FMPOKE

The `FMPOKE` statement can be used to write values directly to the registers of the FM sound chip. To understand what values to write to which registers, see the appendix on YM2551 Registers.

Using `FMPOKE`, it is possible to directly interface with the FM chip and make it do things that are not possible by using the other FM BASIC statements. For example, `FMPOKE` can be used to define a new instrument patch instead of using one of the predefined patches.

Examples:

FMPOKE $28,$4A	Set KC to A4 on channel 0.
FMPOKE $08,$00	Release channel 0.
FMPOKE $08,$78	Start note playback on channel 0 with all operators.

FMVIB

The FMVIB statement sets the speed of the FM chip's LFO, as well as the depth of the amplitude modulation or phase modulation. The first argument sets the speed from 0-255, and the second argument sets the depth from 0-127. The FMVIB statement applies to all channels, and only to instrument patches that use either amplitude or phase modulation (see the Instrument Patch Presets table in the appendix).

Example:

```
10 FMINST 0,11 : REM SET CHANNEL 0 TO VIBRAPHONE
20 FMVIB 200,60 : REM SET VIBRATO
30 FMNOTE 0,$4A : REM PLAY CONCERT A
```

FMVOL

The FMVOL statement sets a channel's volume. The first argument is the channel, and the second argument is a value from 0 through 63. The volume is maintained for the channel, even if the instrument patch is switched. Only another FMVOL statement or an FMINIT statement will cause the volume of a channel to change.

Examples:

FMVOL 0,63	Set channel 0 to full volume.
FMVOL 1,31	Set channel 1 to half volume.
FMVOL 2,0	Set channel 2 to no volume, silencing it.

FOR

The `FOR` statement is used with the `TO` statement, the `NEXT` statement, and sometimes the `STEP` statement to create a section of a program that executes a specific number of times. This repeating section of a program is commonly called a "for-loop".

The format of a for-loop is as follows:

```
FOR <loop variable> = <start> TO <end>
<code to execute multiple times ...>
NEXT <loop variable>
```

In the above example, `<loop variable>` can be any legal name for a floating point variable. Both `<start>` and `<end>` are floating point values, and both variables and constants are allowed.

For example, here is a for-loop that prints the numbers `1` through `10` to the screen:

```
FOR X = 1 TO 10
PRINT X
NEXT X
```

FRAME

The `FRAME` statement draws a rectangle frame in graphics mode in a given color. The first two arguments are the x and y coordinates for the upper left corner of the frame. The third and fourth arguments are the x and y coordinates for the lower right corder of the frame. The fifth argument is a number from 0-255 that specifies the color from the current palette.

Example:

```
10 SCREEN $80
20 FRAME 10,10,310,230,2
```

The `FRAME` statement is similar to the `RECT` statment, except that `FRAME` does not fill in the rectangle.

GET

The `GET` statement gets data from the keyboard one character at a time. The `GET` statement will read a single character from the keyboard cache and place it into a variable provided as an argument. Any character can be placed into a string variable, but only numeric characters can be placed into integer or floating point variables. When the keyboard cache is empty (no keys are currently pressed), a default value is placed into the specified variable. For string variables the default value will be the empty string (`""`), but for floating point and integer variables the default value will be `0.0` and `0`, respectively.

If there are more than one characters currently in the keyboard cache, then a single call to `GET` can retrieve them all by specifying multiple variables as arguments.

Examples:

`GET A$`	Read a single character from the keyboard into `A$`
`GET I$`	Read a single character from the keyboard into `I%`, and cause an error if that character is not numeric
`GET F`	Read a single character from the keyboard into `F`, and cause an error if that character is not numeric
`GET A$,B$`	Read two characters from the keyboard cache into `A$` and `B$`

A common use for the `GET` statement is to pause a program in a loop until the user presses a key:

```
10 PRINT "PRESS ANY KEY TO CONTINUE"
20 GET A$:IF A$="" GOTO 20
30 PRINT "THANK YOU FOR PRESSING A KEY!"
```

GET#

The GET# statement reads data from a specified logical file one character at a time. It is identical to the GET statement except that it requires a first argument to specify a logical file identifier. The logical file must first be opened with the OPEN statement. The remaining arguments work just the same as with the GET statement.

Example:

```
10 OPEN 1,1,0,"FILENAME" : REM OPEN TAPE DRIVE FILE
20 GET#1,A$ : REM READ A CHARACTER FROM THE TAPE
```

> **NOTE:**
>
> It is a popular convention when using statements and commands that end in # to place the logical file identifier directly after the command with no space. So instead of `GET# 1,A$` this example shows `GET#1,A$`.

GOSUB

The GOSUB statement transfers program execution to a specified line and remembers which line which called GOSUB. This is different than the GOTO statement which transfers program execution, but does not have a way to return control to the line which called GOTO. The GOSUB statement allows for the creation of *subroutines* (often called *functions*, *procedures*, or *methods* in other programming languages). After GOSUB is called and program execution has been transfered to a new line, the next time a RETURN statement is executed it will transfer program execution back to the line directly *after* the GOSUB statement.

Example:

```
10 PRINT "FIRST"
20 GOSUB 50
30 PRINT "THIRD"
```

```
40 END
50 PRINT "SECOND"
60 RETURN
```

The above program will print `FIRST`, `SECOND`, and `THIRD` in order. This is because the `GOSUB` statement on line 20 transfers execution to line 50, and then the `RETURN` statement on line 60 transfers execution back to line 30, which is the line *after* the `GOSUB` statement was called. Line 40 ends the program, which stops line 50 from executing again.

> ## X16 TIP: NESTING SUBROUTINES
>
> It is possible to "nest" subroutines created by `GOSUB` such that a subroutine calls a subroutine which calls a subroutine...and so on! This can be useful for creating BASIC programs with complex logic.

GOTO

The `GOTO` statement transfers program execution to the line specified. Unlike the `GOSUB` statement, the `GOTO` statement does *not* remember where it was called from, and therefore the `RETURN` statement will *not* return program execution.

Example:

```
10 PRINT "THIS WILL PRINT"
20 GOTO 40
30 PRINT "THIS WILL NOT PRINT"
40 PRINT "THIS WILL ALSO PRINT"
```

IF

The `IF` statement is how decisions are made in BASIC. The `IF` statement is followed by an expression that evaluates to either `TRUE` or `FALSE`, and the next statement executed is dependent on the outcome. The expression is followed by either a `THEN` statement or a `GOTO` statement. A `THEN` statement is followed by another statement or a line number, and a `GOTO statement` is followed by a line number. If the expression evaluates to `TRUE`, then the statement following the `THEN` statement is executed. If a line number is used, either with a `THEN` or `GOTO` statement, then the program will jump to that line number. When the expression evaluates to `FALSE`, then the line after the `IF` statement is executed.

Expressions can be either a variable or a formula. In both cases a zero is considered `FALSE`, and any non-zero value is considered `TRUE`. In most cases, the statement will be constructed from variables, comparison operators, and logical operators. See the section on operators for more details.

Example:

```
10 A=10
20 IF A=9 THEN 40
30 IF A=10 THEN 60
40 PRINT "THIS SHOULD NOT PRINT"
50 END
60 PRINT "THIS SHOULD PRINT"
```

INPUT

The `INPUT` statement asks the user of a BASIC program for data to store in a variable. The program will print an optional prompt (much like the `PRINT` statement), print a question mark (?), and then wait for the user to type something and press **RETURN**.

The optional prompt must be followed by a semicolon (;) and a variable or comma-separated list of variables. When there are multiple variables, the `INPUT` statement will stop and wait for the user to type something and hit **RETURN** for each one of the variables listed. If no prompt is given, then the semicolon (;) should not be used.

Example:

```
10 INPUT "PLEASE TYPE A NUMBER";A
20 INPUT "AND YOUR NAME";A$
30 INPUT B$
40 PRINT "BET YOU DIDN'T KNOW WHAT I WANTED!"
50 INPUT "TYPE 2 NUMBERS AND A STRING";A,B,C$
60 PRINT A,B,C$
```

INPUT#

The `INPUT#` statement works just like the `INPUT` statement, but takes the data from a previously opened file or device. The device number must be specified before the optional prompt or the variables.

LET

The `LET` statement is an optional statement used for assigning variables in a BASIC program. It is not necessary, but still exists as part of the BASIC language for compatibility purposes.

Example:

```
10 LET A=5
20 B=6
30 PRINT A+B
40 LET B=7
50 PRINT A+B
```

The above code prints `11` and `12`, respectively, showing that using `LET` on variable assignments is optional.

LINE

The `LINE` statement is used to draw a line in graphics mode. The `LINE` statement is passed the X and Y coordinates of the first point, followed by the X and Y coordinates of the second point, followed by the color of the line. The `LINE` statement can only be used in graphics mode, which must be set by calling the `SCREEN` statement with `$80`.

Examples:

Draw a red X across the screen.

```
10 SCREEN $80
20 LINE 0,0,319,239,2
30 LINE 0,239,319,0,2
```

Draw a rainbow.

```
10 SCREEN $80
20 FOR I=0 TO 255
30 LINE 159,0,I+32,239,I
40 NEXT I
```

LINPUT

The `LINPUT` statement reads input directly from the keyboard, but always stores the data as a string varible. Unlike the the `INPUT` statement, which attempts to parse the value entered into whichever variable type was supplied by the programmer, the `LINPUT` stores the data as a string just as the user typed it. This includes storing any quotation marks, commas, or colons that the user types. The `LINPUT` statement does not allow for a prompt to to be specified, so the only argument passed to the the `LINPUT` statement is the string variable used to store the user's input.

Example:

```
10 PRINT "ENTER ANY TEXT FOR LINPUT: ";
20 LINPUT A$
30 PRINT A$
40 PRINT "ENTER ANY TEXT FOR INPUT: ";
50 INPUT A$
60 PRINT A$
```

Try running the above program several times, and using the same value at both prompts. Here's some values to try:

```
"TEST"
3,4,5
11:45 AM
```

You will find that each of the above examples will have a different result when ready by `LINPUT` as opposed to `INPUT`.

LINPUT#

The `LINPUT#` works similar to the `LINPUT` statement, but instead reads the line from an open file specified by the first argument. When reading from a file, there are no "lines" to read from like there is when entering data with either the `INPUT` or `LINPUT` statements, so data is read until a carriage return character (`13`) is reached. The data is stored in a string variable supplied as the second argument. The carriage return character is not included as the input. If the end of the file is reached, the `LINPUT#` statement will set the sixth bit of the `ST` special variable. This means the end of the file can be detected with `ST AND 64`.

Example:

```
10 I=0
20 OPEN 1,8,0,"$"
30 LINPUT#1,A$,$22
40 IF ST<>0 THEN 130
50 LINPUT#1,A$,$22
60 IF I=0 THEN 90
70 PRINT "ENTRY: ";
80 GOTO 100
90 PRINT "LABEL: ";
100 PRINT CHR$($22);A$;CHR$($22)
110 I=I+1
120 IF ST=0 THEN 30
130 CLOSE 1
```

The above example parses and prints out the filenames from a directory listing.

LOCATE

The `LOCATE` statement moves the cursor in text mode, allowing a program to print text to any part of the screen. The `LOCATE` statement takes a line as the first argument, and a column as an optional second argument. Both the line and the column numbers are 1-based (the first line is `1` and the first column is `1`), the column is `1` if no column argument is given.

Examples:

`LOCATE 20`	Move the cursor to line 20 and column 1
`LOCATE 20,30`	Move the cursor to line 20 and column 30
`LOCATE 1,1`	Move the cursor to the top left corner

MOUSE

The `MOUSE` statement shows or hides the mouse cursor by passing a mode as an argument:

Mode	Description
`0`	Hides the mouse cursor
`1`	Shows the mouse cursor with the default sprite
`−1`	Shows the mouse cursor without changing the sprite

A hardware sprite[15] with the index of `0` is used to display the mouse cursor. If the `MOUSE` statement is given a `1`, it will set sprite 0's pixel data to VRAM address `$13000` and copy the default mouse cursor data to that location. If the `MOUSE` statement is given a `−1`, it will display the sprite but change neither its pixel data address nor the data that resides there. This behavior is useful for written a program that sets a custom

[15] See the chapter on Graphics

mouse cursor.

The size of the sprite will automatically be set based on the screen mode. Changing the screen mode while the mouse is displayed will automatically hide it.

Examples:

MOUSE 1	Show mouse cursor using default cursor
MOUSE -1	Show mouse cursor using existing sprite 0
MOUSE 0	Hide mouse cursor

The cursor sprite can also be changed by directly changing the VRAM where it is read from. This a program that will change the mouse cursor to be a miniature version of the default VERA palette:

```
10 MOUSE 1
20 FOR I=0 to 255
30 VPOKE 1,$3000+I,I
40 NEXT I
```

MOVSPR

The `MOVSPR` statement moves a sprite to a location on the screen. The `MOVSPR` statement requires three integer arguments. The first argument is the sprite index, the second is the location along the x-axis, and the third the location along the y-axis. The `MOVSPR` statement will position the sprite's most upper left pixel pixel at the given XY coordinates, regardless of whether that pixel is visible or not. To use a hardward sprite, see the documentation for the `SPRITE` and `SPRMEM` statements.

Example:

```
10 REM FILL A SPRITE VRAM WITH PIXEL DATA
```

```
20 FOR I=0 to 255
30 VPOKE 1,$4000+I,I
40 NEXT I
50 SPRMEM 1,1,$4000,1
60 SPRITE 1,3,0,0,1,1
70 MOVSPR 1,100,100
```

The above code will create a sprite that's a 16x16 pixel version of the default VERA palette, and then position it at coordinates (100,100).

NEXT

The `NEXT` statement is used as part of a for-loop in BASIC. Each use of the `FOR` statement will need a corresponding `NEXT` statement, and `NEXT` will never be used without `FOR`. When a program reaches the `NEXT` statement, the program goes back to the corresponding `FOR` statement and evaluates whether it needs to re-enter the loop or not.

A `NEXT` statement can take no arguments, or it can take many arguments specified as a comma-separated listed. These arguments must be loop counter variables that were created by by `FOR` loops. When no arguments are supplied, the `NEXT` statement will return control to the last `FOR` statement that was started. If loop counter arguments are supplied, the `NEXT` statement will will evaluate the variables from left to right, completing the first loop counter's for-loop before jumping to the next.

Examples:

```
FOR L=1 to 10:NEXT

FOR L=1 to 10:NEXT L

FOR L=1 to 10:FOR M=1 to 10:NEXT M,L
```

ON

The ON statement can be used to change the target line number of a GOTO or GOSUB statement. The ON statement is followed by an expression that evaluates to a number, then followed by either a GOTO or GOSUB statement, which is in turn followed by a comma-separated list of line numbers. Which line is used as the target of the GOTO or GOSUB statement depends on the result of the numerical expression[16]. For example, if the expression 1 is given, then the GOTO or GOSUB would select the first line number in the comma-separated list. If the expression 1+2 is given, the third line number in the list would be selected. If the expression evaluates to 0 or any number higher than the number of line numbers in the list, the program moves to the next line. If the expression evaluates to any number outside 0 to 255 an error occurs.

Example:

```
10 INPUT X
20 ON X GOTO 10,50,50,50
30 PRINT "NOPE!"
40 GOTO 10
50 PRINT "YUP!"
60 ON X GOTO 10,30,30
```

Try It Yourself!
Type in the above program and see if you can figure out what number needs to be entered in order to exit the program!

OPEN

The OPEN statement can be used to access various devices from within a BASIC program on the Commander X16. These devices may be the keyboard, the screen (in text mode), disk drives, and printers. The first argument to the OPEN statement is any number from 1 to 255 that will

[16]The ON statement works very similar to how switch or select works in other programming languages

be used to refer to the `OPEN`ed device from other BASIC statements. The second argument is a number that specifies which device to `OPEN`. The default devices available on the Commander X16 are:

Device #	Description	Secondary Address
0	Keyboard	(none)
1	(unused)	(none)
2	(unused)	(none)
3	Screen	0 or 1
4-5	Printer (or other IEC device)	0 = capital letters/graphic characters; 1 = capital/lowercase letters
6-30	IEC Bus devices (SD card is 8 at boot, but can be reassigned)	0 = read; 1 = write; 2-14 = data channels; 15 channel for commands

Many devices may require a third, or even a fourth argument to be passed to `OPEN`. The Secondary Address column of the above table shows some of the available values for the third argument on the various devices. For disk drives, a fourth argument specifies the name of a file.

Examples:

`OPEN 1,0`	`OPEN`s the keyboard as a device
`OPEN 3,8,0,"MYFILE"`	`OPEN`s a file on the SD card
`OPEN 4,9,15`	`OPEN`s the data channel on a disk configured as device 9

POKE

The `POKE` statement is used to write directly to the Commander X16's memory or a memory-mapped device. It is always followed by two numbers or expressions that evaluate to numbers. The first is the address in the memory map, which can be any value from 0-65535[17]. The second argument is the value to write to the specified address. Because the Commander X16 is an 8-bit computer, so each address only holds one byte of data. Therefore, any value from 0-255 is allowed.

The `POKE` statement is very powerful, since it is able to write not only to memory, but also the memory mapped hardware such as the VERA. To learn about the regions of memory available and the memory mapped hardware, see the **Memory Map** appendix.

Examples:

`POKE 2048,64`	Write `64` to address `2048`
`POKE $800,$40`	Same as above, but denoted in hexadecimal
`POKE 0,1`	Writea `1` to address `$0000`, which switches the Commander X16 to use RAM bank 1

PRINT

The `PRINT` statement is used to display text and graphic characters to the screen. When it text mode, it is the most typical way to display output of a program to a user. The `PRINT` statement can be followed by any of the following:

- **Characters inside of quotation marks -** Called *literals* because they are printed literally as they are typed in
- **Variables -** `PRINT`'s' the value the variable currently holds
- **Functions -** `PRINT`'s' the value returned by the function
- **Punctuation marks -** Provides formatting options:
  - **Comma (,) -** Advances to the next column, where each column is 10 characters wide
  - **Semicolon (;) -** Does not advance to the next line after

[17] Since the Commander X16 supports hexadecimal values, it's easier to think of this as $0000-$FFFF

Because commas have a special meaning for formatting, commas should not be used to separate PRINTing multiple literals, variables, or functions when output with a single PRINT.
Examples:

PRINT "HELLO"	PRINTs "HELLO" to the screen
PRINT "HELLO, "A$	PRINTs "HELLO, " followed by the value of A$
PRINT A+B	PRINTs The result of A+B
PRINT J	PRINTs the value of J
PRINT A,B,C,D	PRINTs the values of each variable formatted into columns

PRINT#

The PRINT# statement works nearly identically to the PRINT statement, except that it PRINTs to an open file or device instead of the screen. The first argument to the PRINT# statement must be a number used to identify the open file or device. This must be a number that was used with an OPEN statement to open the file or device. This number is followed by a comma, which is followed by the value to be printed. This can be any of the options available to the PRINT statement, including commas and semicolons used for formatting. Not every device that can be written to with PRINT# will be able to handle formatting, however.

Example:

```
10 OPEN 1,8,1,"MYFILE"
20 FOR J=1 to 10
30 FOR I=1 to J
40 PRINT#1,"*",;
50 NEXT I
60 PRINT# 1,""
70 NEXT J
```

```
80 CLOSE 1
```

The above example will write a file named "MYFILE" to the SD card, containing a pattern of asterisks. To view the file, use the Command X16's built-in text editor:

```
EDIT "MYFILE"
```

The file should contain the following pattern:

```
*
**
***
****
*****
******
*******
********
*********
**********
```

PSET

The `PSET` statement sets the color of a single pixel in graphics mode (set with `SCREEN $80`). The `PSET` statement is followed by the x and y coordinates and number from 0-255 to specify the color from the palette.

Example:

```
10 SCREEN$80
20 FOR Y=0 to 239
30 FOR X=0 to 319
40 C=INT(Y/15)*16+INT(X/15)
50 PSET X,Y,C
60 NEXT X,Y
```

The above program uses `PSET` to display the default VERA color palette to the screen. Because it calculates and draws each pixel one at a time, it takes a few minutes to complete. After all, it has 76,800 pixels to calculate! See the `RECT` statement for a much faster way to display the palette. It only has to calculate the color 256 times.

PSGCHORD

The `PSGCHORD` statement instructs the programmable sound generator to begin playing multiple notes at the same time. For arguments, the `PSGCHORD` statement accepts a channel and a string. Because a chord plays multiple notes at the same time, the channel argument specifies the *first* channel to use for the chord, but other channels will be used for subsequent notes. For example, if you specify a channel argument of `3` for a chord which plays 4 notes, the `PSGCHORD` statement will play the notes on channels 3, 4, 5, and 6. It is important to set each of the channels to use the desired waveform with the `PSGWAV` statement. The string argument is used to specify which notes the `PSGCHORD` statement will play. For more information on specifying notes, see the chapter on Sound.

Example:

```
REM PLAY A C MAJOR CHORD ON WITH A PULSE WAVEFORM
10 PSGWAV 0,63:PSGWAV 1,63:PSGWAV 2,63
20 PSGCHORD 0,"CGE"
```

PSGFREQ

The `PSGFREQ` statement plays a note on the programmable sound generator at a given frequency. This is an alternative to playing a note with `PSGNOTE`, where instead of specifying a musical note, a frequency in Hertz is specified. Like `PSGNOTE`, `PSGFREQ` returns immediately and does not wait for a note to finish playing. If a Hertz value of `0` is specified, the channel is immediately silenced.

Examples:

PSGFREQ 3,2600	Plays the waveform on channel 3 at 2,600hz.
PSGFREQ 0,440	Equivalent to `PSGNOTE 0,$4A` which plays A above middle C on channel 0.
PSGFREQ 2,0	Silences channel 2.

PSGINIT

The `PSGINIT` statement initializes the VERA's programmable sound generator (PSG). When the `PSGINIT` statement is run, it initializes the PSG, and does the following on all 16 channels:

- silences the channel
- sets the volume to 63 (the maximum)
- sets the waveform to pulse with a duty cycle of 50%

PSGNOTE

The `PSGNOTE` statement plays a single note on the programmable sound generator. The first argument is the channel, and the second argument specifies which note. The note argument can be any number, but is intended to be specified with hexadecimal notation. This is so that the most significant 4 bits (often called the "high nybble") represent the octave while the least significant 4 bits (the "low nybble") represent the musical note. The lowest note of any octave is **C**, which is represented with a `1`, and the highest note of any octave is **B**, which is represented with a **C**. A note of `0` on any octave will release the note playing on that channel, and the note values **D**, **E**, and **F** have no effect.

Although this may seem confusing, it is actually convenient for most uses. For example to play a "middle C" the note value `$41` would be used. The `$` tells BASIC that the value is hexadecimal, the `4` indicates the note is in the 4th octave, and `1` specifies the note "C". Here's a table of which nybble produces which note:

Nybble	$x0	$x1	$x2	$x3	$x4	$x5	$x6
Note	Release	C	C♯/D♭	D	D♯/Eb	E	F
Nybble	$x7	$x8	$x9	$xA	$xB	$xC	$xD-$xF
Note	F♯/G♭	G	G♯/A♭	A	A♯/B♭	B	no-op

Negative numbers can also be used to specify notes. These will be treated as the same note as the positive value.

Example:

```
10 PSGWAV 1,63 : REM PULSE WITH 50% DUTY CYCLE
20 PSGNOTE 1,$4A : REM PLAYS CONCERT A
30 SLEEP 50 : NEXT X : REM DELAYS FOR A BIT
40 PSGNOTE 1,0 : REM RELEASES THE NOTE
50 SLEEP 10 : NEXT X : REM DELAYS FOR A BIT
60 PSGNOTE 1,$3A : REM PLAYS A IN THE 3RD OCTAVE
70 SLEEP 25 : NEXT X : REM SHORT DELAY
80 PSGNOTE 1,-$3B : REM A# WITHOUT RETRIGGERING
90 SLEEP 25 : NEXT X : REM SHORT DELAY
100 PSGNOTE 1,0 : REM RELEASES THE NOTE
```

PSGPAN

The `PSGPAN` statement is used to control the stereo output of a PSG channel. It takes an argument for the channel, and an argument for which speaker the channel should play from. The second argument values are as follows:

Left	1
Right	2
Both	3

Examples:

`PSGPAN 0,3`	Set channel 0 to play from both speakers.
`PSGPAN 3,1`	Set channel 3 to play from only the left speaker.
`PSGPAN 7,2`	Set channel 7 to play from only the right speaker.

PSGPLAY

The `PSGPLAY` statement plays a musical melody on a single channel. `PSGPLAY` takes two arguments; a channel and a string of characters that tells the programmable sound generator what to play. This second argument is specified in a custom macro language[18], and includes notes, releases, tempos, octaves, rests, and other musical elements. For example, the following statement will play a major scale in the key of **C**:

```
PSGPLAY 0,"CDEFGAB>C"
```

The `PSGPLAY` statement uses the exact same macro language as the `FMPLAY` statement, so see the section on `FMPLAY` for more information.

PSGVOL

The `PSGVOL` statement sets a channel's volume. The first argument is the channel, and the second argument is a value from 0 through 63. The volume is maintained for the channel, even if the waveform is switched. Only another `PSGVOL` statement or an `PSGINIT` statement will cause the volume of a channel to change.

Examples:

[18] for a complete guide, see the Macro Language for Music appendix

PSGVOL 0,63	Set channel 0 to full volume.
PSGVOL 1,31	Set channel 1 to half volume.
PSGVOL 2,0	Set channel 2 to no volume, silencing it.

PSGWAV

The `PSGWAV` statement sets the waveform of a channel on the programmable sound generator (PSG), which results in a different timbre. The first argument is the channel, and the second argument indicates which wavform to use. There are four wavforms to choose from: pulse (square wave), sawtooth, triangle, and noise. If using the pulse waveform, the duty cycle[19] of the wavform can also be specified. Here are the values to use with the second argument for each waveform:

0-63	Pulse	Duty cycle = `(VAL+1)/128`
64-127	Sawtooth	All values have identical effect
128-191	Triangle	All values have identical effect
192-255	Noise	All values have identical effect

Examples:

PSGWAV 0,63	Set channel 0 to Pulse with a 50% duty cycle.
PSGWAV 0,31	Set channel 0 to Pulse with a 25% duty cycle.
PSGWAV 1,64	Set channel 1 to Sawtooth.
PSGWAV 2,128	Set channel 2 to Triangle.
PSGWAV 3,192	Set channel 3 to Noise.

[19] The "duty cycle" of a square wave is the percentage of the time that the wave is "high" compared to the total period of the wave

READ

The READ statement is used to get information that has been coded into the program using DATA statements. Like the INPUT statement, the READ statement is followed by a variable that matches the type of data being read. The first time a READ statement is encountered in a program, the first piece of data specified by a DATA statement is read into the variable. The next time, a READ statement is encountered, it READs the next piece of data specified by a DATA statement, and so on. If the type of the data and the type of the variable do not match, a TYPE MISMATCH ERROR will occur.

```
10 READ A$
30 READ B
40 READ C%
50 PRINT A$,B,C%
60 DATA "A VALUE"
70 DATA 27.5,42
```

RECT

The RECT statement draws a filled rectangle in graphics mode in a given color. The first two arguments are the x and y coordinates for the upper left corner of the rectangle. The third and fourth arguments are the x and y coordinates for the lower right corder of the rectangle. The fifth argument is a number from 0-255 that specifies the color from the current palette.

Example:

```
10 SCREEN $80
20 FOR Y=0 TO 15
30 FOR X=0 TO 15
40 C=Y*16+X
50 X1=X*20
60 Y1=Y*15
70 RECT X1,Y1,X1+19,Y1+14,C
80 NEXT X,Y
```

The above program displays the default VERA color palette to the graphics screen by drawing a 20x15 rectangle for each color.

The `RECT` statement is similar to the `FRAME` statement, except that `RECT` fills the rectangle with the specified color.

REM

The `REM` statement is used to leave a `REM`ark (also called a comment) in a BASIC program to help a programmer annotate sections of the program. It could help explain how a complex section of logic works, tell which variable is being used for, or even to let the programmer sign their name! A `REM` statement can be followed by anything at all, and the BASIC program will ignore it.

No program needs `REM` statements to function correctly, but they can make a huge difference when it comes to reading and modifying a BASIC program. Most programmers will use `REM` statements on any program once it has become large or complex. It's a good idea to get into the habit of leaving comments in your program to help understand how it works.

Example:

```
10 REM START OF PROGRAM
20 PRINT "ENTER A NUMBER";
30 INPUT A: REM STORE NUMBER IN A
40 PRINT "ENTER ANOTHER NUMBER";
50 INPUT B: REM STORE NUMBER IN B
60 C=A*B: REM MULTIPLY NUMBERS
70 PRINT C
```

RESTORE

The `RESTORE` statement resets where the `READ` statement will read from, causing the next `READ` to read from the first `DATA` statement again. In this way, a program can re-read data that it has already read. The `RESTORE` statement takes no arguments, and always directs the next `READ` back to the first `DATA` statement.

Example:

```
10 READ A$
20 PRINT A$
30 READ A$
40 PRINT A$
50 RESTORE
60 READ A$
70 PRINT A$
80 DATA "FIRST","SECOND"
```

Despite the above program `READ`ing three times yet only having enough data for two reads, it runs without error because `RESTORE` causes the first piece of data to be read twice.

RETURN

The `RETURN` statement is always used in conjunction with the `GOSUB` statement. When a program encounters a `RETURN` statement, the program immediately jumps to the statement after the last executed `GOSUB` statement. During execution, BASIC keeps track of which `GOSUB` statement has been called in which order. When a `RETURN` is executed, the corresponding `GOSUB` is removed from the list BASIC keeps track of. Because of this list (often called a *call stack*) that BASIC keeps track of, programmers are able to *nest* `GOSUB` statements, so that one subroutine calls another subroutine and so on. If the program ever encounters a `RETURN` statement when its list of `GOSUB`s is empty, it will cause a `RETURN WITHOUT GOSUB ERROR`.

Example:

```
10 PRINT "FIRST"
20 GOSUB 50
30 PRINT "FOURTH"
40 END
50 GOSUB 80
60 PRINT "THIRD"
```

```
70 RETURN
80 PRINT "SECOND"
90 RETURN
```

SCREEN

The `SCREEN` statement is used to select a screen mode. The screen mode determines how many characters can fit on the screen (both horizontally and vertically), whether the screen has a border, and whether bitmap graphics can be rendered in the layer behind the text. The `SCREEN` statement requires a numeric argument to tell it which screen mode to use.

The available screen modes are:

Decimal	Hexadecimal	Screen Mode
0	$00	80x60 Text
1	$01	80x30 Text
2	$02	40x60 Text
3	$03	40x30 Text
4	$04	40x15 Text
5	$05	20x30 Text
6	$06	20x15 Text
7	$07	22x23 Text /w border
8	$08	64x50 Text /w border
9	$09	64x25 Text /w border
10	$0A	32x50 Text /w border
11	$0B	32x25 Text /w border
128	$80	256 color Bitmap Graphics /w 40x30 Text

In addition to the above table, the value `-1` can be passed to `SCREEN` to toggle between modes `0` and `3`.

> **NOTE:**
>
> Other screen configurations can be used by directly setting registers in the VERA graphics module. For more details, see the chapter on Graphics.

SLEEP

The `SLEEP` statement pauses program execution for a specified time period. The `SLEEP` statement takes a single argument which is the number of VSYNC interrupts to wait before continuing program execution. These interrupts occur approximately 60 times per second, so to have a program wait for one second, an argument of `60` would be used. This unit of measurement is often called a *jiffy*. The `SLEEP` statement will wait for the specified number of jiffies, starting after the next VSYNC interrupt to occur. Using `SLEEP` with no arguments is the same as using `0` for the number of jiffies, and will cause the program to wait until the next VSYNC interrupt. This can be useful for animating graphics, since a VSYNC interrupt is how you know that the VERA graphics module is about to start drawing the next frame to the screen.

Examples:

`SLEEP 60`	Wait for about one second.
`SLEEP 600`	Wait for about ten seconds.
`SLEEP 0`	Wait until the next frame.

SPRITE

The `SPRITE` statement enables or disables one of the VERA's 128 hardware sprites, and allows you to set other sprite properties as well. The `SPRITE` statement can cause a hardware sprite to display, but it cannot tell the sprite what to look like. For that reason, the `SPRITE` statement is intended to be used alongside the `SPRMEM` statement to point a hardware sprite to a VRAM location, and some way to load data into VRAM (such as the `VLOAD` or `BVLOAD` statements).

However, there is still an easy way to test the `SPRITE` statement without loading anything into VRAM. This can be done by using the `MOUSE` statement, which will initialize hardware sprite 0 to point to VRAM data initialized to a mouse cursor:

```
10 MOUSE 1 : REM TURN ON MOUSE
20 MOUSE 0 : REM TURN OFF MOUSE
30 SPRITE 0,3 : REM DISPLAY SPRITE 0
```

The above code will cause the mouse cursor to display, even when the mouse is not active. The cursor will not be able to be moved. This works by enabling the mouse and initializing sprite 0 with the `MOUSE 1` statement, disabling the mouse and sprite 0 with `MOUSE 0`, and then redisplaying sprite 0 (without enabling the mouse) with `SPRITE 0,3`.

The first argument to the `MOUSE` statement is the hardware sprite index. With 128 hardware sprites, this number can be from 0 to 127. The second argument sets the layer the sprite renders to, including disabling the sprite completely with 0. The third argument is the palette offset used in 4 bits-per-pixel mode. The fourth argument sets how the sprite is flipped. The fifth argument is used to set width of the sprite, and the sixth sets the height. Finally, the seventh argument sets the color mode. Only the first two arguments are required. With this in mind, the sprite from the above program can be modified. For example, it can be easily flipped upside down:

```
SPRITE 0,3,0,2
```

Here is a brief summary of each argument:

Argument	Description
Index	The index of the hardware sprite (0-127)
Priority	0 - disable sprite, 1 - draw beneath both VERA layers, 2 - draw in between VERA layers, 3 - draw on top of both VERA layers
Palette Offset	0-15 in 4bbp mode, not used in 8bpp mode
Flip	0 - no flip, 1 - flipped on the x-axis, 2 - flipped on the y-axis, 3 - flipped on both x and y axis
Width	0 - 8 pixels, 1 - 16 pixels, 2 - 32 pixels, 3 - 64 pixels
Height	0 - 8 pixels, 1 - 16 pixels, 2 - 32 pixels, 3 - 64 pixels
Color Mode	0 - 4bpp, 1 - 8bpp

SPRMEM

The `SPRMEM` statement sets the VRAM address and the color mode of a given sprite. The first argument is the sprite index. The VRAM address is specified in two parts: the bank, and the address within the bank. The VERA's 128KB of video memory (or VRAM) can be thought of as two banks of 64KB. The bank value can either be a 0 to indicate the first 64KB, or a 1 to indicate the second 64KB. The address within the bank can then be specified with a 16 bit integer value. It is easiest to specify address values using hexadecimal, so an address within a bank can be any value from $0000 through $FFFF.

The final argument is the color mode. The VERA only allows sprites to use the four bits per pixel mode (4bpp) or the eight bits per pixel mode (8bpp). A value of `0` specifies 4bpp mode, and a value of `1` specifies 8bpp.

Example:

```
10 SCREEN $80 20 BVLOAD "MYSPRITE.BIN",8,1,$3000
30 SPRMEM 1,1,$3000,1
40 SPRITE 1,3,0,0,3,3
50 MOVSPR 1,160,120
```

The above program will load a sprite into VRAM bank 1 at address $3000 (sometimes specified as address $13000), set sprite 1 to use that address, and interpret the data there in 8bpp mode. The `SPRITE` statement is then use to set the rest of the sprite's attributes, and the MOVESPR statement moves the sprite's location on the screen.

STEP

The STEP statement is used with the FOR, TO, andNEXT statements in order to construct for-loops. The STEP statement is followed by a number that will be added to the loop counter variable each time the NEXT statement is executed. This number can be positive or negative. Both integer and floating point numbers are allowed. When no STEP statement is used in a for-loop, a default value of 1 is used.

Examples:

FOR I=1 TO 10 STEP 1	Count from 1 to 10.
FOR I=1 TO 10	Count from 1 to 10 (use default STEP).
FOR I=2 TO 10 STEP 2	Count to 10 by 2.
FOR I=10 TO 1 STEP -1	Count backwards from 10.
FOR I=0 TO 10 STEP 0.5	Count by 0.5.

STOP

The STOP statement will halt a program. This can be used to help debug programs by causing them to stop at a certain point, and then PRINTing the values of variables. A STOPped program can be continued by using the CONT command, which will start running the program from the line after the STOP statement.

When the STOP statement executes, it will not only stop the program, but will also print a message, BREAK IN xxxx where xxxx is the line number containing the STOP statement. This is useful when multiple STOP statements are used in a single program.

Example:

```
10 PRINT "HELLO"
20 STOP
30 PRINT "WORLD"
```

Try `CONT`inuing the above program after the `STOP` statement halts the program.

SYS

The `SYS` statement transfers control of the X16 to a machine language program in memory.

Example:

```
SYS 8192
```

In the above example, the X16 will execute a machine language program stored at address 8192 in decimal. Because the Commander X16 also supports hexadecimal arguments to BASIC statements, the same can be written as:

```
SYS $2000
```

THEN

The `THEN` statement is used with an `IF` statement to tell the program what to do when the condition of the `IF` statement is `TRUE`. A `THEN` statement can either be followed by a line number a BASIC expression such as a variable assignment or another statement. If the `THEN` statement is followed by a line number, the `THEN` statement will behave the same as a `GOTO` statement and jump program control to the specified line number. When `THEN` is followed by a BASIC expression it will execute that expression and then proceed to the next line of the program.

Examples:

```
10 A = 5
```

```
20 B = 1
30 IF A = 5 THEN B = 2
40 PRINT B
```

The above program should print `2` to the screen instead of `1`. Here is a way to accomplish the same thing by using a line number with `THEN` instead of an expression.

```
10 A = 5
20 B = 1
30 IF A <> 5 THEN 50
40 B = 2
50 PRINT B
```

TILE

The `TILE` statement can be used to place a given tile on the VERA's layer 1 tile map. This works even when the map base or map size has been changed, which makes it simple to place tiles in graphics modes. Because a text layer on the VERA is just a tile map that uses tile characters, the `TILE` statement can be used instead of using `LOCATE` and `PRINT` to place a single character anywhere on the screen. The first two arguments to the `TILE` statement are the 0-based X and Y coordinates, and the third argument is the tile number.

Example:

```
10 I=0
20 FOR Y=0 TO 15
30 FOR X=0 TO 15
40 TILE X,Y,I
50 I=I+1
60 NEXT:NEXT
70 LOCATE 17 :REM MOVE READY PROMPT DOWN
```

The above example uses the `TILE` statement to display all the characters (the default tile set) on a 16 by 16 grid.

TO

The `TO` statement is used with the `FOR` statement, the `NEXT` statement, and sometimes the `STEP` statement to create a for-loop. The `TO` statement is used to define the range of a for-loop. The number preceeding the `TO` statement will be the value of the loop variable on the first pass through the loop, and the loop will stop when the loop variable is greater than or equal to the number that follows the `TO` statement.

See the `FOR` statement for examples of using the `TO` statement.

VPOKE

The `VPOKE` statement sets a single byte of video RAM (VRAM) on the VERA's onboard memory. This allows for directly setting the data that will be interpretted as graphics and PSG sound. The VERA has a total of 131,072 bytes (128 kilobytes) of VRAM, which it exposes as 2 banks of 65,536 bytes (64 kilobytes) each. The `VPOKE` statement takes three arguments: the bank, the memory location within the bank, and the value to be stored.

Example:

```
10 FOR I=1 TO 256*64 STEP 2
20 VPOKE 1,$B000+I,0 :REM SET COLOR
30 NEXT I
```

The above BASIC program will fill the screen with black spaces one character at a time by writing directly to the area of the VERA's VRAM where the X16 stores the text mode screen data. The `VPOKE` sets the background and forground color to black, although the character data does not change.

> **NOTE:**
>
> In screen mode 0, the tile map is stored in the VERA at VRAM address $1B000, and is 128 tiles wide by 64 tiles tall.

VLOAD

The `VLOAD` statement loads a file with a two-byte header directly into the VERA's VRAM, but without loading the header. For arguments, the `VLOAD` statement takes the file's name, the device number where the file is stored, the bank of VRAM on the VERA (either `0` or `1`), and the address within the bank in which to load.

Examples:

`VLOAD "MYFILE.BIN",8,0,$4000`	Loads a file named "MYFILE.BIN" from device 8 into VRAM at address $04000.
`VLOAD "MYFONT.BIN",8,1,$F000`	Loads a file named "MYFONT.BIN" from device 8 into VRAM at address $1F000.

To load a file that does not have a two-byte header, see the `BVLOAD` statement.

WAIT

The `WAIT` statement is used to halt a program until the contents of a memory location changes in a specified way. The `WAIT` statement requires an memory address as the first argument, and a value for the second. When the value in the memory address is `AND`ed with the second argument and results in a zero, the `WAIT` statement continues to halt the program while it keeps re-checking the memory address. When the result is non-zero, the program continues. An optional third argument can be supplied that will be logically `XOR`ed to the value in memory before being `AND`ed by the second argument.

The `WAIT` statement is not very useful for most BASIC programs, and is typically only used when interfacing with hardware via memory mapped addresses.

Example:

(*When executing this program, continue to hold* **RETURN** *after typing* `RUN`)

```
10 BANK 0
20 PRINT "LET GO OF RETURN KEY"
30 WAIT $A820,16
40 PRINT "PRESS RETURN KEY"
50 WAIT $A820,16,16
60 GOTO 20
```

The above example works by reading the memory location of the X16's "joystick 0", which is a virtual joystick emulated with the keyboard". The first `WAIT` is used to detect that **RETURN** is no longer pressed (which it would be after executing the `RUN` command), and the second detects that it has been pressed again[20].

Functions

Functions are instructions that return values that can be used like variables, or even assigned to variables. Functions are often called in-line to supply arguments to statements. Functions will either return a numeric value or a string value. If the function returns a string value, its name will end with a `$`.

Because the return value is a fundamental property of a function, many of the examples in this section include the output of the example listed directly below the example, itself.

ABS

The `ABS(X)` function will return the absolute value of `X`.

[20]The use of `WAIT` here is just for an example. The `JOY` function would be a much better choice for this functionality

Example:

```
PRINT ABS(-10)
 10
```

ASC

The `ASC` function will return an integer value representing the PETSCII code[21] for the first character of string. If string is the empty string, `ASC` returns 0. The opposite of this function is the `CHR$` function.

Example:

```
PRINT ASC("A")
 65
```

ATN

The `ATN(X)` (arctangent) function will return the angle whose tangent is `X`. This is the inverse of the `TAN` function.

Example:

```
PRINT ATN(0)
 0
```

In the above example, the value 0 will be printed on the screen.

BIN$

The `BIN$` function returns a string of the binary representation (1's and 0's) of a number. The `BIN$` function will only work on numbers greater than or equal to 0, and less than 65,536. When a floating point number number is passed to `BIN$`, it will ignore the decimal part of the number and truncate the value to an integer, just as the `INT` function does. When `BIN$` is passed a number that is less than 256, it only returns 8

[21] See the PETSCII Codes table in the appendix

characters since the value can be represented by 8 bits. When a number that is 256 or greater is used, it will return 16 characters.

Examples:

```
PRINT BIN$(3)
00000011

PRINT BIN$(3.14)
00000011

PRINT BIN$(3000)
0000101110111000
```

CHR$

The `CHR$` function takes a number from 0-255 and returns the PETSCII character represted by that PETSCII code[22]. Floating point numbers will be truncated to integers, as is done when the `INT` function is used. Numbers outside that range will cause an error. The opposite of this function is the `ASC` function.

Examples:

```
PRINT CHR$(65)
A

PRINT CHR$(65.7)
A

PRINT CHR$(147) : REM CLEARS SCREEN
```

COS

The `COS(X)` (cosine) function will return the cosine of an angle `X`, measured in radians.

[22] See the PETSCII Codes table in the appendix

Example:

```
PRINT COS(π)
 -1
```

EXP

The `EXP(X)` (exponent) function will return the value of the mathematical constant `e` (2.71828183) raised to the power of `X`.

Examples:

```
PRINT EXP(5)
 148.413159

REM PRINT E, ITSELF
PRINT EXP(1)
 2.71828183
```

FNXX(X)

Any function `FNXX` (where `XX` is any legal name) that has been defined with the `DEF` statement can be called inside or outside of a BASIC program. It will execute the user-defined function and return the result.

See the `DEF` statement for details and examples.

FRE

The `FRE` function returns the number of unused (or *free*) BASIC bytes. Although the `FRE` function requires an argument (any valid numeric or string will do), this argument has no effect on the function's result.

Example:

```
10 PRINT FRE(0)
20 DIM A(4096) :REM ALLOCATE SOME MEMORY
```

```
30 PRINT FRE(0)
```

The above program will print the amount of free memory available to BASIC, allocate some of that memory, and then again print the updated amount of free memory.

HEX$

The `HEX$` function takes a numeric value and returns a string of the *hexadecimal* representation of that value. Hexadecimal is a base-16 number system, meaning that 16 different numeric values can be represented by a single character. Just like with the usual base-10 number system, or *decimal*, when the next highest value is needed you simply use more than one character to represent it. For example, in decimal when the value 10 needs to be written, two characters are used: "1" followed by "0".

In order to have 16 different characters to represent numeric values, alphabetic characters are used for values 10 through 15. Here are the characters used in hexadecimal, and their decimal equivalents:

H	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
D	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

So in order to write the value 16 in hexadecimal, two characters would need to be used: "1" followed by "0" (just like 10 in decimal).

The `HEX$` function can accept any numeric value from 0-65535. When values less than 256 are given, `HEX$` always returns exactly two characters, and when values greater than or equal to 256 are given, it always returns 4 characters. Both are padded with 0's if necessary.

Example:

```
10 FOR I=0 to 50
20 PRINT I,HEX$(I)
30 NEXT I
```

The above program will print out a table showing values in decimal followed by the same value in hexadecimal.

INT

The `INT` function returns a truncated[23] integer version of the numeric argument passed to the function. The result will always be less than or equal to the argument, including when the argument is negative.

Examples:

```
PRINT INT(3.8)
 3
```

```
PRINT INT(-3.8)
 -4
```

The `INT` function can also be used to round decimal numbers to a certain precision, such as to the nearest hundredth:

```
X=INT(X*100+0.5)/100
```

LEFT$

The `LEFT$` function takes a string and number and returns a substring containing the specified number of leftmost characters.

Example:

```
PRINT LEFT$("BASEBALL", 4)
BASE
```

[23] *Truncated* means removing all decimal places to the right of the decimal point, leaving only an integer

LEN

The `LEN` function returns the length of a string. Because a string in BASIC can never exceed a length of 255 characters, the `LEN` function will always return values between 0-255.

Example:

```
PRINT LEN("COMMANDER X16")
 13
```

LOG

The `LOG` function will return the natural log of the given numeric value. The natural log is the log to the base of `e` (see `EXP`). To convert to log base 10, simply divide by `LOG(10)`. To convert to log base 2, divide by `LOG(2)`.

Examples:

```
REM NATURAL LOG OF 8
PRINT LOG(8)
 2.07944154

REM LOG BASE 10 OF 8
PRINT LOG(8)/LOG(10)
 .903089987

REM LOG BASE 2 OF 8
PRINT LOG(8)/LOG(2)
 3
```

MID$

The `MID$` function takes a string, a start position, and an optional number of characters, and returns a substring of the original string. The substring begins with the character specified by the start position, and will be as long as is specified by the number of characters. If the number of characters is not specified, the substring will contain all characters up to the end of the original string.

Examples:

```
PRINT MID$("COMMANDER X16",4,3)
MAN

PRINT MID$("COMMANDER X16",1,3)
COM

PRINT MID$("COMMANDER X16",7,5)
DER X

PRINT MID$("COMMANDER X16",11)
X16
```

PEEK

The `PEEK` function returns the contents of the Commander X16's memory at a given address. This includes reading the values from any of the X16's memory mapped hardware devices (if they are readable) such as the VERA or the FM synthesizer chip. The address must be in the range of 0-65535, and the values returned will all be in the range of 0-255.

Example:

```
10 FOR A=0 TO 255
20 PRINT HEX$(A)+": $";
30 PRINT HEX$(PEEK(A)),;
40 NEXT A
```

The above example will print the entire contents of the zero page[24].

[24] The "zero page" is the first 256 bytes of the Commander X16's memory. This memory has special purposes on the 65C02 processor.

π

The `π` constant returns the value of pi and can be used just like any other floating point variable or literal.

```
10 INPUT "WHAT IS THE RADIUS OF THE CIRLCE";R
20 PRINT "CIRCUMFRANCE:",2*R*π
30 PRINT "AREA:",,R*R*π
```

POINTER

The `POINTER` function returns the memory address of the data structure where a BASIC variable is stored.

```
10 X$ = "COMMANDER X16"
20 PRINT HEX$(POINTER(X$))
```

POS

The `POS` function returns the current column of the text cursor. The `POS` function takes a single argument that is unused. Due to the way BASIC reads data from a program, the fastest type of variable to use is the special constant `π` (pi). Therefore, it is a common convention to always call the `POS` function as `POS(π)`.

```
10 PRINT POS(π)
20 REM ADVANCE THE COLUMN BUT NOT THE LINE
30 PRINT "COMMANDER X16";
40 PRINT POS(π)
```

RIGHT$

The `RIGHT$` function takes a string and number and returns a substring containing the specified number of rightmost characters.

Example:

```
PRINT RIGHT$("BASEBALL", 4)
```

```
BALL
```

RND

The `RND` function generates pseudo-random floating-point numbers in the range of 0 to 1 (exclusive), such as 0.153632167, 0.567453436, 0.942242351.

At power-on of the X16 computer a sequence of random numbers is generated automatically and stored. The number passed in parenthesis to the `RND` function influences the resulting values in subsequent calls to `RND`. A negative number (-1) will reseed the sequence starting point of original random generated numbers. The same negative number will result in the same sequence of random numbers. A positive number (1) will return the next random number of the current sequence. Using a zero (0) will generate a shorter sequence of random numbers, which can provide a speed improvement in returning the result (if processing times are important).

A best practice method is to initially seed the `RND` function with `-TI`, at the beginning of your program. The `TI` system variable, which stands for Time-Interval, is the elapsed time since your computer last turned on. Generally, this provides a different random sequence every time you re-run the program. When you need a random number later in your program you can then use RND(1) to provide the next random number.

Using a formula including the `RND(1)` function, can get a random value between any two numbers. Using variables to explain this formula, `LO = 10` and `HI = 40`, the formula could be `N = RND(1)*(HI-LO)+LO`. This will result in random numbers between 10 and (less than) 40. When repeated, you might see numbers like 13.3567377, 24.5913944, 16.2857004, and 39.2262697.

`X = INT(RND(1)*6)+1`	Simulate a 6-side dice roll
`X = INT(RND(1)*1000)+1`	Number from 1-1000
`X = INT(RND(1)*150)+100`	Number from 100-249

RPT$

The `RPT$` function returns a string of repeated PETSCII characters. Like the `CHR$` function, the `RPT$` function takes a PETSCII code[25], but also takes a second argument indicating a count. The string returned will be the character specified by the PETSCII code repeated as many times as the count indicates.

Example:

```
PRINT RPT$(33,10)
!!!!!!!!!!
```

SGN

The `SGN` function returns the sign of a numeric value.

Examples:

```
PRINT SGN(42)
 1

PRINT SGN(-23.96)
-1

PRINT SGN(0)
 0
```

SIN

The `SIN(X)` (sine) function will return the sine of an angle `X`, measured in radians.

Example:

```
PRINT SIN(π/2)
 1
```

[25] See the PETSCII Codes table in the appendix

SPC

The `SPC` function can be used to set a number of spaces, either printed to the screen or written to a file. It takes a numeric value for the number of spaces to use. The `SPC` function is similar to using the `RPT$` function to return a number of spaces, but not equivalent. The `SPC` function does not return a string, but rather adds the desired number of spaces directly into the output, whether it be the screen or a file. The `SPC` function will not overwrite existing text on the screen, which is fundamentally different than simply printing a string. Because of these reasons, the `SPC` function cannot be called except as part of a `PRINT` statement.

When the `SPC` function is used as the last item in a `PRINT` statement, the carriage return that would normally be inserted is suppressed, just as it is for trailing commas and semicolons.

Examples:

```
PRINT SPC(5) "HELLO"
     HELLO

10 PRINT "HELLO" SPC(5)
20 PRINT "WORLD"
RUN
HELLO     WORLD
```

SQR

The `SQR` function returns the square root of a non-zero number. If the number given is negative, the `SQR` function will return an `ILLEGAL QUANTITY ERROR`.

Examples:

```
PRINT SQR(9)
 3
PRINT SQR(25.0)
 5
PRINT SQR(48)
```

```
 6.92820323
```

STR$

The `STR$` function converts a numeric value to a string value using the same methodology as `PRINT`ing a numeric value. This means that `0` and positive numbers converted to strings will begin with a space, while negative numbers will begin with a `-`. This means that string functions such as `LEN` will work properly on the resulting string, whereas counting the digits could lead to errors in printing.

Example:

```
PRINT LEN(STR$(10))
 3
```

STRPTR

The `STRPTR` function returns the memory address of the first character of a given string variable. If the string variable passed to `STRPTR` is empty (has a length of zero), then the value returned by `STRPTR` is undefined and should not be used. To prevent using bad memory, always use the `LEN` function on a string ahead of time to make sure that it is not empty.

Example:

```
10 INPUT "ENTER A WORD";A$
20 IF LEN(A$) > 0 GOTO 40
30 END
40 P=STRPTR(A$)
50 PRINT CHR$(PEEK(P))
RUN
ENTER A WORD? THING
T
```

TAB

The `TAB` is used as part of a `PRINT` statement to advance the cursor to a given column. If the cursor is already at or beyond the given column, it simply continues printing from the current position. The `SPC` function is usually better to use than the `TAB` function when inserting spaces in output, especially when the output device is a file or a printer. The `TAB` function operates much like the comma (`,`) in a `PRINT` statement, except it allows the width of the tabulation to be set with an argument.

Example:

```
10 FOR I=0 to 10
20 PRINT I TAB(10) I*2 TAB(30) I*3
30 NEXT I
```

TAN

The `TAN(X)` (tangent) function will return the tangent of an angle `X`, measured in radians.

Example:

```
PRINT TAN(0)
 0
```

TATTR

The `TATTR` function retrieves the tile attribute byte of a given X/Y coordinate on layer 1 of the VERA. This attribute carries different information depending on which screen mode is being used. This is the same attribute byte that can be optionally set to the layer 1 tile map with with the `TILE` statement.

Example:

```
10 REM SHOW ATTRIBUTE BYTES OF BUTTERFLY LOGO
20 BANNER
30 FOR Y=1 TO 6
```

```
40 FOR X=1 TO 6
50 PRINT TATTR(X,Y),;
60 NEXT X
70 PRINT ""
80 NEXT Y
```

TDATA

The `TDATA` function retrieves the tile index of the tile used on a given X/Y coordinate of the layer 1 tile map. This is the same tile index value that can be set to the tile map with the ttfamily TILE statement.

Example:

```
10 REM SHOW TILE INDEXES OF BUTTERFLY LOGO
20 BANNER
30 FOR Y=1 TO 6
40 FOR X=1 TO 6
50 PRINT TDATA(X,Y),;
60 NEXT X
70 PRINT ""
80 NEXT Y
```

USR

The `USR` function calls a user-defined assembly routine from BASIC. It can accept any valid BASIC expression as an argument, as long as the expression resolves to a number or a string. If the expression is numeric, it places the result in a special place in memory called the Floating Point Accumulator (FACC, for short) located from `$61-$66`. From there the user-defined assembly routine can read the result of the expression as input. If the expression evaluates to a string, it places a pointer to the string located from `$54-$65`. The result of the user-defined assembly routine must be a floating point placed in the FACC.

The address of the user-defined assembly routine to be called should be stored in the address `$0311-$0312` prior to calling the `USR` function. Otherwise this address will store the address of an error handler that displays the `?ILLEGAL QUANTITY ERROR`. By using the `USR` function, any number of built-in or custom assembly routines can be called from BASIC and used as an expression.

Example:

```
10 REM SET THE USR FUNCTION TO THE
20 REM FADDH MATH LIBRARY ROUTINE
30 POKE $0311,$6F
40 POKE $0312,$FE
50 PRINT USR(4.2)
```

The above BASIC program will set the user-defined function to the `FADDH` X16 Math Library routine at address `$FE65`, which takes a number in the FACC and adds `0.5` to it. The program then calls the `FADDH` routine with `4.2` using the `USR` function. The program should print a result of `4.7`.

VAL

The `VAL` function returns a numeric value representing the characters in a string argument. Often a string variable is passed to the function, but a literal string is also valid (eg. " -540.15 "). Blank characters ("spaces") in the string are ignored. If the first non-blank character of the string is not a plus sign (+), minus sign (-), dollar sign ($), percentage sign (%) or a digit the conversion ends with a value of zero (0). These initial special characters signify the type of number to follow - Positive (+); Negative (-); Hexadecimal ($), which then validates the letters A,B,C,D,E & F; Binary Literal (%) eg. "010101"; and Numbers (0123456789). Subsequent valid characters are additional digits (or the first decimal point or E/e for Exponent). The function ends at the end of the string, or the next non-digit character for that numberic type and returns the converted result. Subsequent digits after any non-valid characters are disregarded. Other mathematical terms and arithmetic operations are ignored.

The valid range of possible numbers is from -1e+38 to 1e+38. Outside of this range the error "?OVERFLOW ERROR IN <line>" is shown and the program stops. When the argument isn't a string, the error "?TYPE MISMATCH ERROR IN <line>" would result and stop the program. When the argument is absent, the error "?SYNTAX ERROR IN <line>" is returned and stops the program.

Examples:

```
10 READ A$
20 DATA " - 120 . 64 "
30 PRINT VAL(A$)
RUN
  -120.64
```

Leading letters are invalid, but don't cause an error.

```
PRINT VAL("ABC 123")
 0
```

The binary literal string is converted to decimal number.

```
PRINT VAL("%010101")
 21
```

The exponential notation string is returned as a simplified number.

```
PRINT VAL("+352 .25 E-3 Units")
 0.35225
```

VPEEK

The `VPEEK` function gets a single byte of vidoe RAM (VRAM) from the VERA's onboard memory. The allows a BASIC program to read the data being interpreted by the VERA to produce graphics and sound. The `VPEEK` function takes two arguments: the bank and the memory location within the bank.

Example:

```
10 FOR I=0 TO 256*64-1
20 PRINT VPEEK(1,$B000+I),;
```

```
30 NEXT I
```

The above BASIC program will print out the data being used to display the text mode screen data, including the attribute data responsible for coloring the text.

BASIC Statements Table

Keyword	Type	Summary	Origin
ABS	function	Returns absolute value of a number	C64
AND	operator	Returns boolean "AND" or bitwise intersection	C64
ASC	function	Returns numeric PETSCII value from string	C64
ATN	function	Returns arctangent of a number	C64
BANK	statement	Sets the bank used to interpret addresses above $A000	X16
BIN$	function	Converts numeric to a binary string	X16
BINPUT#	command	Reads a fixed-length block of data from an open file	X16
BLOAD	command	Loads a headerless binary file from disk to a memory address	X16
BOOT	command	Loads and runs `AUTOBOOT.X16`	X16
BVERIFY	command	Verifies that a file on disk matches RAM contents	X16
BVLOAD	command	Loads a headerless binary file from disk to VRAM	X16
CHAR	command	Draws a text string in graphics mode	X16
CHR$$	function	Returns PETSCII character from numeric value	C64

CLOSE	command	Closes a logical file number	C64
CLR	command	Clears BASIC variable state	C64
CLS	command	Clears the screen	X16
CMD	command	Redirects output to non-screen device	C64
COLOR	command	Sets text fg and bg color	X16
CONT	command	Resumes execution of a BASIC program	C64
COS	function	Returns cosine of an angle in radians	C64
DA$	variable	Returns the date in YYYYMMDD format from the system clock	X16
DATA	command	Declares one or more constants	C64
DEF	command	Defines a function for use later in BASIC	C64
DIM	command	Allocates storage for an array	C64
DOS	command	Disk and SD card directory operations	X16
END	command	Terminate program execution and return to `READY.`	C64
EXP	function	Returns the inverse natural log of a number	C64
FMCHORD	command	Start or stop simultaneous notes on YM2151	X16
FMDRUM	command	Plays a drum sound on YM2151	X16
FMFREQ	command	Plays a frequency in Hz on YM2151	X16
FMINIT	command	Stops sound and reinitializes YM2151	X16

FMNOTE	command	Plays a musical note on YM2151	X16
FMPAN	command	Sets stereo panning on YM2151	X16
FMPLAY	command	Plays a series of notes on YM2151	X16
FMPOKE	command	Writes a value into a YM2151 register	X16
FMVIB	command	Controls vibrato and tremolo on YM2151	X16
FMVOL	command	Sets channel volume on YM2151	X16
FN	function	Calls a previously defined function	C64
FOR	command	Declares the start of a loop construct	C64
FRAME	command	Draws an unfilled rectangle in graphics mode	X16
FRE	function	Returns the number of unused BASIC bytes free	C64
GET	command	Polls the keyboard cache for a single keystroke	C64
GET#	command	Polls an open logical file for a single character	C64
GOSUB	command	Jumps to a BASIC subroutine	C64
GOTO	command	Branches immediately to a line number	C64
HELP	command	Displays a brief summary of online help resources	X16
HEX$	function	Converts numeric to a hexadecimal string	X16
IF	command	Tests a boolean condition and branches on result	C64

INPUT	command	Reads a line or values from the keyboard	C64
INPUT#	command	Reads lines or values from a logical file	C64
INT	function	Discards the fractional part of a number	C64
JOY	function	Reads gamepad button state	X16
KEYMAP	command	Changes the keyboard layout	X16
LEFT$	function	Returns a substring starting from the beginning of a string	C64
LEN	function	Returns the length of a string	C64
LET	command	Explicitly declares a variable	C64
LINE	command	Draws a line in graphics mode	X16
LINPUT	command	Reads a line from the keyboard	X16
LINPUT#	command	Reads a line or other delimited data from an open file	X16
LIST	command	Outputs the program listing to the screen	C64
LOAD	command	Loads a program from disk into memory	C64
LOCATE	command	Moves the text cursor to new location	X16
LOG	function	Returns the natural logarithm of a number	C64
MENU	command	Presents the user with a menu of built-in programs	X16
MID$	function	Returns a substring from the middle of a string	C64

MON	command	Enters the machine language monitor	X16
MOUSE	command	Hides or shows mouse pointer	X16
MOVSPR	command	Set the X/Y position of a sprite	X16
MX/MY/MB	variable	Reads the mouse position and button state	X16
NEW	command	Resets the state of BASIC and clears program memory	C64
NEXT	command	Declares the end of a loop construct	C64
NOT	operator	Bitwise or boolean inverse	C64
OLD	command	Undoes a NEW command or warm reset	X16
ON	command	A GOTO/GOSUB table based on a variable value	C64
OPEN	command	Opens a logical file to disk or other device	C64
OR	operator	Bitwise or boolean "OR"	C64
PEEK	function	Returns a value from a memory address	C64
π	function	Returns the constant for the value of pi	C64
POINTER	function	Returns the address of a BASIC variable	C128
POKE	command	Assigns a value to a memory address	C64
POS	function	Returns the column position of the text cursor	C64
POWEROFF	command	Returns the address of a BASIC variable	X16
PRINT	command	Prints data to the screen or other output	C64

PRINT#	command	Prints data to an open logical file	C64
PSET	command	Changes a pixel's color in graphics mode	X16
PSGCHORD	command	Starts or stops simultaneous notes on VERA PSG	X16
PSGFREQ	command	Plays a frequency in Hz on VERA PSG	X16
PSGINIT	command	Stops sound and reinitializes VERA PSG	X16
PSGNOTE	command	Plays a musical note on VERA PSG	X16
PSGPAN	command	Sets stereo panning on VERA PSG	X16
PSGPLAY	command	Plays a series of notes on VERA PSG	X16
PSGVOL	command	Sets voice volume on VERA PSG	X16
PSGWAV	command	Sets waveform on VERA PSG	X16
READ	command	Assigns the next `DATA` constant to one or more variables	C64
REBOOT	command	Performs a warm reboot on the system	X16
RECT	command	Draws a filled rectangle in graphics mode	X16
REM	command	Declares a comment	C64
REN	command	Renumbers a BASIC program	X16
RESET	command	Performs a warm reset on the system	X16
RESTORE	command	Resets the `READ` pointer to the first `DATA` constant	C64

RETURN	command	Returns from a subroutine to the statement following a GOSUB	C64
RIGHT$	function	Returns a substring from the end of a string	C64
RND	function	Returns a floating point number 0 <= n < 1	C64
RPT$	function	Returns a string of repeated characters	X16
RUN	command	Clears the variable state and starts a BASIC program	C64
SAVE	command	Saves a BASIC program from memory to disk	C64
SCREEN	command	Selects a text or graphics mode	X16
SGN	function	Returns the sign of a numeric value	C64
SIN	function	Returns the sine of an angle in radians	C64
SPC	function	Returns a string with a set number of spaces	C64
SPRITE	command	Sets attributes for a sprite including visibility	X16
SPRMEM	command	Set the VRAM address for a sprite's visual data	X16
SQR	function	Returns the square root of a numeric value	C64
ST	variable	Returns the status of certain DOS/peripheral operations	C64
STEP	keyword	Used in a `FOR` declaration to declare the iterator step	C64
STOP	command	Breaks out of a BASIC program	C64

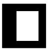
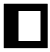
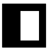
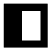
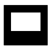
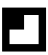
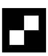
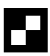
STR$	function	Converts a numeric value to a string	C64
SYS	command	Transfers control to machine language at a memory address	C64
TAB	function	Returns a string with spaces used for column alignment	C64
TAN	function	Return the tangent for an angle in radians	C64
TATTR	function	Return the layer 1 tile attributes at a given x/y coordinate	X16
TDATA	function	Return the layer 1 tile at a given x/y coordinate	X16
THEN	keyword	Control structure as part of an `IF` statement	C64
TI	variable	Returns the jiffy timer value	C64
TI$	variable	Returns the time HHMMSS from the system clock	C64
TILE	command	Changes a tile or character on the tile/text layer	X16
TO	keyword	Part of the `FOR` loop declaration syntax	C64
USR	function	Call a user-defined function in machine language	C64
VAL	function	Parse a string to return a numeric value	C64
VERIFY	command	Verify that a BASIC program was written to disk correctly	C64
VPEEK	function	Returns a value from VERA's VRAM	X16
VPOKE	command	Sets a value in VERA's VRAM	X16

VLOAD	command	Loads a file to VERA's VRAM	X16
WAIT	command	Waits for a memory location to match a condition	C64

Screen Codes

SET 1	SET 2	DEC	HEX	SET 1	SET 2	DEC	HEX
@	@	0	$00	!	!	33	$21
A	a	1	$01	"	"	34	$22
B	b	2	$02	#	#	35	$23
C	c	3	$03	$	$	36	$24
D	d	4	$04	%	%	37	$25
E	e	5	$05	&	&	38	$26
F	f	6	$06	'	'	39	$27
G	g	7	$07	(	(	40	$28
H	h	8	$08	)	)	41	$29
I	i	9	$09	*	*	42	$2a
J	j	10	$0a	+	+	43	$2b
K	k	11	$0b	,	,	44	$2c
L	l	12	$0c	-	-	45	$2d
M	m	13	$0d	.	.	46	$2e
N	n	14	$0e	/	/	47	$2f
O	o	15	$0f	0	0	48	$30
P	p	16	$10	1	1	49	$31
Q	q	17	$11	2	2	50	$32
R	r	18	$12	3	3	51	$33
S	s	19	$13	4	4	52	$34
T	t	20	$14	5	5	53	$35
U	u	21	$15	6	6	54	$36
V	v	22	$16	7	7	55	$37
W	w	23	$17	8	8	56	$38
X	x	24	$18	9	9	57	$39
Y	y	25	$19	:	:	58	$3a
Z	z	26	$1a	;	;	59	$3b
[	[	27	$1b	<	<	60	$3c
£	£	28	$1c	=	=	61	$3d
]	]	29	$1d	>	>	62	$3e
↑	↑	30	$1e	?	?	63	$3f
←	←	31	$1f			64	$40
SPACE	SPACE	32	$20		A	65	$41

SET 1	SET 2	DEC	HEX	SET 1	SET 2	DEC	HEX
	B	66	$42			91	$5b
	C	67	$43			92	$5c
	D	68	$44			93	$5d
	E	69	$45			94	$5e
	F	70	$46			95	$5f
	G	71	$47	SPACE	SPACE	96	$60
	H	72	$48			97	$61
	I	73	$49			98	$62
	J	74	$4a			99	$63
	K	75	$4b			100	$64
	L	76	$4c			101	$65
	M	77	$4d			102	$66
	N	78	$4e			103	$67
	O	79	$4f			104	$68
	P	80	$50			105	$69
	Q	81	$51			106	$6a
	R	82	$52			107	$6b
	S	83	$53			108	$6c
	T	84	$54			109	$6d
	U	85	$55			110	$6e
	V	86	$56			111	$6f
	W	87	$57			112	$70
	X	88	$58			113	$71
	Y	89	$59			114	$72
	Z	90	$5a			115	$73

SET 1	SET 2	DEC	HEX
		116	$74
		117	$75
		118	$76
		119	$77
		120	$78
		121	$79
		122	$7a
		123	$7b
		124	$7c
		125	$7d
		126	$7e
		127	$7f

PETSCII Codes

This appendix shows how characters are encoded on the Commander X16. Like the 8-bit Commodore computers that inspired it, the Commander X16 uses a modified version of the ASCII character set. This character set is commonly knows as PETSCII, as it was first widely used on the Commodore PET line of computers. Each character is assigned to an 8-bit integer value, which is how that character is represented in memory. Not all 8-bit values have characters, and not all characters are printable. This table lists each of the characters and their numbers.

CHAR	CODE	CHAR	CODE	CHAR	CODE
	0	RVS ON	18	$	36
	1	CLR HOME	19	%	37
	2	INST DEL	20	&	38
	3		21	'	39
	4		22	(	40
WHT	5		23	)	41
	6		24	*	42
	7		25	+	43
	8		26	,	44
	9		27	-	45
	10	RED	28	.	46
	11	CRSR RIGHT	29	/	47
	12	GRN	30	0	48
RETURN	13	BLU	31	1	49
LOWER CASE	14	SPACE	32	2	50
	15	!	33	3	51
	16	"	34	4	52
CRSR DOWN	17	#	35	5	53

CHAR	CODE	CHAR	CODE	CHAR	CODE
6	54	O	79		104
7	55	P	80		105
8	56	Q	81		106
9	57	R	82		107
:	58	S	83		108
;	59	T	84		109
<	60	U	85		110
=	61	V	86		111
>	62	W	87		112
?	63	X	88		113
@	64	Y	89		114
A	65	Z	90		115
B	66	[	91		116
C	67	£	92		117
D	68	]	93		118
E	69	↑	94		119
F	70	←	95		120
G	71		96		121
H	72		97		122
I	73		98		123
J	74		99		124
K	75		100		125
L	76		101		126
M	77		102		127
N	78		103		128

CHAR	CODE	CHAR	CODE	CHAR	CODE
	129		150		171
	130		151		172
	131		152		173
	132		153		174
F1	133		154		175
F2	134		155		176
F3	135	PUR	156		177
F4	136	<= CRSR	157		178
F5	137	YEL	158		179
F6	138	CYN	159		180
F7	135	SPACE	89		181
F8	140		161		182
SHIFT RETURN	141		162		183
UPPERCASE	142		163		184
	143		164		185
BLK	144		165		186
CRSR	145		166		120
RVS OFF	146		167		188
CLR HOME	147		168		189
INST DEL	148		169		190
	149		170		191

Memory Map

The Commander X16 has 512 KB of ROM and 2,088 KB (512 KB[26] + 40 KB) of RAM with up to 3.5MB of RAM or ROM available to cartridges.

Some of the ROM/RAM is always visible at certain address ranges, while the remaining ROM/RAM is banked into one of two address windows.

This is an overview of the Commander X16 memory map:

Addresses	Description
$0000-$9EFF	Fixed RAM (40 KB minus 256 bytes)
$9F00-$9FFF	I/O Area (256 bytes)
$A000-$BFFF	Banked RAM (8 KB window into one of 256 banks for a total of 2 MB)
$C000-$FFFF	Banked System ROM and Cartridge ROM/RAM (16 KB window into one of 256 banks, see below)

Banked Memory

Writing to the following zero-page addresses sets the desired RAM or ROM bank:

Address	Description
$0000	Current RAM bank (0-255)
$0001	Current ROM/Cartridge bank (ROM is 0-31, Cartridge is 32-255)

The currently set banks can also be read back from the respective memory locations. Both settings default to 0 on RESET.

[26] Developer editions of the Commander X16 typically come with 2MB of banked RAM rather than 512 MB

ROM Allocations

Here is the ROM/Cartridge bank allocation:

Bank	Name	Description
0	KERNAL	KERNAL operating system and drivers
1	KEYBD	Keyboard layout tables
2	CBDOS	The computer-based CMDR-DOS for FAT32 SD cards
3	FAT32	The FAT32 driver itself
4	BASIC	BASIC interpreter
5	MONITOR	Machine Language Monitor
6	CHARSET	PETSCII and ISO character sets (uploaded into VRAM)
7	CODEX	CodeX16 Interactive Assembly Environment / Monitor
8	GRAPH	Kernal graphics and font routines
9	DEMO	Demo routines
10	AUDIO	Audio API routines
11	UTIL	System Configuration (Date/Time, Display Preferences)
12	BANNEX	BASIC Annex (code for some added BASIC functions)
13-14	X16EDIT	The built-in text editor
13-31	-	(Currently unused)
32-255	-	Cartridge RAM/ROM

Cartridge Allocation

Cartridges can use the remaining 32-255 banks in any combination of ROM, RAM, Memory-Mapped IO, etc. This provides up to 3.5MB of additional RAM or ROM.

RAM Contents

This is the allocation of fixed RAM in the KERNAL/BASIC environment.

Addresses	Description
$0000-$00FF	Zero page
$0100-$01FF	CPU stack
$0200-$03FF	KERNAL and BASIC variables, vectors
$0400-$07FF	Available for machine code programs or custom data storage
$0800-$9EFF	BASIC program/variables; available to the user

The $0400-$07FF can be seen as the equivalent of $C000-$CFFF on a C64. A typical use would be for helper machine code called by BASIC.

Zero Page

Addresses	Description
$0000-$0001	Banking registers
$0002-$0021	16 bit registers r0-r15 for KERNAL API
$0022-$007F	Available to the user
$0080-$009C	Used by KERNAL and DOS
$009D-$00A8	Reserved for DOS/BASIC
$00A9-$00D3	Used by the Math library (and BASIC)
$00D4-$00FF	Used by BASIC

Machine code applications are free to reuse the BASIC area, and if they don't use the Math library, also that area.

RAM Banks

This is the allocation of banked RAM in the KERNAL/BASIC environment.

Bank	Description
0	Used for KERNAL/CMDR-DOS variables and buffers
1-63	Available to the user

During startup, the KERNAL activates RAM bank 1 as the default for the user.

I/O Area

This is the memory map of the I/O Area:

Addresses	Description	Speed
$9F00-$9F0F	VIA I/O controller #1	8 MHz
$9F10-$9F1F	VIA I/O controller #2	8 MHz
$9F20-$9F3F	VERA video controller	8 MHz
$9F40-$9F41	YM2151 audio controller	2 MHz
$9F42-$9F5F	Unavailable	–
$9F60-$9F7F	Expansion Card Memory Mapped IO3	8 MHz
$9F80-$9F9F	Expansion Card Memory Mapped IO4	8 MHz
$9FA0-$9FBF	Expansion Card Memory Mapped IO5	2 MHz
$9FC0-$9FDF	Expansion Card Memory Mapped IO6	2 MHz
$9FE0-$9FFF	Cartidge/Expansion Memory Mapped IO7	2 MHz

Expansion Cards and Cartridges

Expansion cards can be accessed via memory-mapped I/O (MMIO), as well as I2C. Cartridges are essentially expansion cards which are housed in an external enclosure and may contain RAM, ROM and an I2C EEPOM (for save data). Internal expansion cards may also use the RAM/ROM space, though this could cause conflicts.

While they may be uncomon, since cartridges are essentially external expansion cards in a shell, that means they can also use MMIO. This is only necessary when a cartridge includes some sort of hardware expansion and MMIO was desired (as opposed to using the I2C bus). In that case, it is recommended cartridges use the IO7 range and that range should be the last option used by expansion cards in the system.

MMIO is unneeded for cartridges which simply have RAM/ROM.

65c02 OP Codes

FM Instrument Patch Presets

† = instrument is affected by the LFO, giving it a temolo or vibrato

0	Acoustic Grand Piano	32	Acoustic Bass
1	Bright Acoustic Piano	33	Electric Bass (finger)
2	Electric Grand Piano	34	Electric Bass (picked)
3	Honky-tonk Piano	35	Fretless Bass
4	Electric Piano 1	36	Slap Bass 1
5	Electric Piano 2	37	Slap Bass 2
6	Harpsichord	38	Synth Bass 1
7	Clavinet	39	Synth Bass 2
8	Celesta	40	Violin †
9	Glockenspiel	41	Viola †
10	Music Box	42	Cello †
11	Vibraphone †	43	Contrabass †
12	Marimba	44	Tremolo Strings †
13	Xylophone	45	Pizzicato Strings
14	Tubular Bells	46	Orchestral Harp
15	Dulcimer	47	Timpani
16	Drawbar Organ †	48	String Ensemble 1 †
17	Percussive Organ †	49	String Ensemble 2 †
18	Rock Organ †	50	Synth Strings 1 †
19	Church Organ	51	Synth Strings 2 †
20	Reed Organ	52	Choir Aahs †
21	Accordion	53	Voice Doos
22	Harmonica	54	Synth Voice †
23	Bandoneon	55	Orchestra Hit
24	Acoustic Guitar (Nylon)	56	Trumpet †
25	Acoustic Guitar (Steel)	57	Trombone
26	Electric Guitar (Jazz)	58	Tuba
27	Electric Guitar (Clean)	59	Muted Trumpet †
28	Electric Guitar (Muted)	60	French Horn
29	Electric Guitar (Overdriven)	61	Brass Section
30	Electric Guitar (Distortion)	62	Synth Brass 1
31	Electric Guitar (Harmonics)	63	Synth Brass 2

64	Soprano Sax †	96	FX 1 (Raindrop)
65	Alto Sax †	97	FX 2 (Soundtrack) †
66	Tenor Sax †	98	FX 3 (Crystal)
67	Baritone Sax	99	FX 4 (Atmosphere) †
68	Oboe	100 †	FX 5 (Brightness) †
69	English Horn †	101	FX 6 (Goblin)
70	Bassoon	102	FX 7 (Echo)
71	Clarinet †	103	FX 8 (Sci-Fi) †
72	Piccolo	104	Sitar
73	Flute †	105	Banjo
74	Recorder	106	Shamisen
75	Pan Flute	107	Koto
76	Blown Bottle	108	Kalimba
77	Shakuhachi	109	Bagpipe
78	Whistle †	110	Fiddle †
79	Ocarina	111	Shanai †
80	Lead 1 (Square) †	112	Tinkle Bell
81	Lead 2 (Sawtooth) †	113	Agogo
82	Lead 3 (Triangle) †	114	Steel Drum
83	Lead 4 (Chiff+Sine) †	115	Woodblock
84	Lead 5 (Charang) †	116	Taiko Drum
85	Lead 6 (Voice) †	117	Melodic Tom
86	Lead 7 (Fifths) †	118	Synth Drum
87	Lead 8 (Solo) †	119	Reverse Cymbal
88	Pad 1 (Fantasia) †	120	Fret Noise
89	Pad 2 (Warm) †	121	Breath Noise
90	Pad 3 (Polysynth) †	122	Seashore
91	Pad 4 (Choir) †	123	Bird Tweet
92	Pad 5 (Bowed)	124	Telephone Ring
93	Pad 6 (Metallic)	125	Helicopter
94	Pad 7 (Halo) †	126	Applause †
95	Pad 8 (Sweep) †	127	Gunshot

Extended FM Instrument Patch Presets

These presets exist mainly to support playback of drum sounds, and many of them only work correctly or sound musical at certain pitches or within a small range of pitches.

128	Silent	146	Vibraslap
129	Snare Roll	147	Bongo
130	Snap	148	Maracas
131	High Q	149	Short Whistle
132	Scratch	150	Long Whistle
133	Square Click	151	Short Guiro
134	Kick	152	Long Guiro
135	Rim	153	Mute Cuica
136	Snare	154	Open Cuica
137	Clap	155	Mute Triangle
138	Tom	156	Open Triangle
139	Closed Hi-Hat	157	Jingle Bell
140	Pedal Hi-Hat	158	Bell Tree
141	Open Hi-Hat	159	Mute Surdo
142	Crash	160	Pure Sine
143	Ride Cymbal	161	Timbale
144	Splash Cymbal	162	Open Surdo
145	Tambourine		

Drum Patch Presets

These are the percussion instrument mappings for the drum number argument of the `ym_playdrum` and `ym_setdrum` API calls, and the `FMDRUM` BASIC statement.

		56	Cowbell
25	Snare Roll	57	Crash Cymbal 2
26	Finger Snap	58	Vibraslap
27	High Q	59	Ride Cymbal 2
28	Slap	60	High Bongo
29	Scratch Pull	61	Low Bongo
30	Scratch Push	62	Mute High Conga
31	Sticks	63	Open High Conga
32	Square Click	64	Low Conga
33	Metronome Bell	65	High Timbale
34	Metronome Click	66	Low Timbale
35	Acoustic Bass Drum	67	High Agogo
36	Electric Bass Drum	68	Low Agogo
37	Side Stick	69	Cabasa
38	Acoustic Snare	70	Maracas
39	Hand Clap	71	Short Whistle
40	Electric Snare	72	Long Whistle
41	Low Floor Tom	73	Short Guiro
42	Closed Hi-Hat	74	Long Guiro
43	High Floor Tom	75	Claves
44	Pedal Hi-Hat	76	High Woodblock
45	Low Tom	77	Low Woodblock
46	Open Hi-Hat	78	Mute Cuica
47	Low-Mid Tom	79	Open Cuica
48	High-Mid Tom	80	Mute Triangle
49	Crash Cymbal 1	81	Open Triangle
50	High Tom	82	Shaker
51	Ride Cymbal 1	83	Jingle Bell
52	Chinese Cymbal	84	Belltree
53	Ride Bell	85	Castanets
54	Tambourine	86	Mute Surdo
55	Splash Cymbal	87	Open Surdo

Macro Language for Music

Overview

The play commands use a string of tokens to define sequences of notes to be played on a single voice of the corresponding sound chip. Tokens cause various effects to happen, such as triggering notes, changing the playback speed, etc. In order to minimize the amount of text required to specify a sequence of sound, the player maintains an internal state for most note parameters.

Stateful Player Behavior

Playback parameters such as tempo, octave, volume, note duration, etc do not need to be specified for each note. These states are global between all voices of both the FM and PSG sound chips. The player maintains parameter state during and after playback. For instance, setting the octave to 5 in an `FMPLAY` command will result in subsequent `FMPLAY` and `PSGPLAY` statements beginning with the octave set to 5.

The player state is reset to default values whenever `FMINIT` or `PSGINIT` are used.

Parameter	Default	Equivalent Token
Tempo	120	T120
Octave	4	O4
Length	4	L4
Note Spacing	1	S4

Using Tokens

The valid tokens are: **`A–G`**, **`I`**, **`K`**, **`L`**, **`O`**, **`P`**, **`R`**, **`S`**, **`T`**, **`V`**, **`<`**, **`>`**.

Each token may be followed by optional modifiers such as numbers or symbols. Options to a token must be given in the order they are expected, and must have no spacing between them. Tokens may have spaces between them as desired. Any unknown characters are ignored.

Example:

```
FMPLAY 0,"L4" : REM DEFAULT LENGTH = QUARTER NOTE
FMPLAY 0,"A2. C+." : REM VALID
FMPLAY 0,"A.2 C.+" : REM INVALID
```

The valid command plays **A** as a dotted half, followed by **C♭** as a dotted quarter.

The invalid example would play **A** as a dotted quarter (not half) because length must come before dots. Next, it would ignore the 2 as garbage. Then it would play natural C (not sharp) as a dotted quarter. Finally, it would ignore the + as garbage, because sharp/flat must precede length and dot.

Token definitions

Musical notes

- **Synopsis: Play a musical note, optionally setting the length.**
- **Syntax: <A-G> [<+/->] [<length>] [.]**

Example:

```
FMPLAY 0,"A+2A4C.G-8."
```

On the YM2151 using channel 0, plays in the current octave an **A♯** half note followed by an **A** quarter note, followed by **C** dotted quarter note, followed by **G♭** dotted eighth note.

Lengths and dots after the note name or rest set the length just for the current note or rest. To set the default length for subsequent notes and rests, use the 'L' macro.

Rests

- **Synopsis**: Wait for a period of silence equal to the length of a note, optionally setting the length.

- **Syntax**: ‘R[<length>][.]‘

Example:

```
PSGPLAY 0,"CR2DRE"
```

On the VERA PSG using voice 0, plays in the current octave a **C** quarter note, followed by a half rest (silence), followed by a quarter **D**, followed by a quarter rest (silence), and finally a quarter **E**.
The numeral `2` in `R2` sets the length for the `R` itself but does not alter the default note length (assumed as 4 - quarter notes in this example).

Note Length

- **Synopsis**: Set the default length for notes and rests that follow
- **Syntax**: `L[<length>][.]`

Example values:

L4	quarter note (crotchet)
L16	sixteenth note (semiquaver)
L12	8th note triplets (quaver triplet)
L4.	dotted quarter note (1.5x the length)
L4..	double-dotted quarter note (1.75x the length)

Example program:

```
10 FMPLAY 0,"L4"
20 FOR I=1 TO 2
30 FMPLAY 0,"CDECL8"
40 NEXT
```

On the YM2151 using channel 0, this program, when RUN, plays in the current octave the sequence **C D E C** first as quarter notes, then as eighth notes the second time around.

Articulation

- **Synopsis**: Set the spacing between notes, from legato to extreme staccato
- **Syntax**: `S<0-7>`

`S0` indicates legato. For FMPLAY, this also means that notes after the first in a phrase don't implicitly retrigger.

`S1` is the default value, which plays a note for 7/8 of the duration of the note, and releases the note for the remaining 1/8 of the note's duration.

You can think of `S` is, out of 8, how much space is put between the notes.

Example:

```
FMPLAY 0,"L4S1CDES0CDES4CDE"
```

On the YM2151 using channel 0, plays in the current octave the sequence **`C D E`** three times, first with normal articulation, next with legato (notes all run together and without retriggering), and finally with a moderate staccato.

Explicit retrigger

- **Synopsis**: on the YM2151, when using 'S0' legato, retrigger on the next note.
- **Syntax**: `K`

Example:

```
FMPLAY 0,"S0CDEKFGA"
```

On the YM2151 using channel 0, plays in the current octave the sequence **`C D E`** using legato, only triggering on the first note, then the sequence

F G A the same way. The note **F** is triggered without needing to release the previous note early.

Octave

- **Synopsis**: Explictly set the octave number for notes that follow
- **Syntax**: `O<0-7>`

Example:

```
PSGPLAY 0,"O4AO2AO6CDE"
```

On the VERA PSG using voice 0, changes to octave 4 and plays **A** (440Hz), then switches to octave 2, and plays **A** (110Hz), then switches to octave 6 and plays the sequence **C D E**.

Octave Up

- **Synopsis**: Increases the octave by 1
- **Syntax**: `>`

If the octave would go above 7, this macro has no effect.

Example:

```
PSGPLAY 0,"O4AB>C+DE"
```

On the VERA PSG using voice 0, changes to octave 4 and plays the first five notes of the **A major** scale by switching to octave 5 starting at the **C♯**.

Octave Down

- **Synopsis**: Decreases the octave by 1
- **Syntax**: `<`

If the octave would go below 0, this macro has no effect.

Example:

```
PSGPLAY 0,"O5GF+EDC<BAG"
```

On the VERA PSG using voice 0, changes to octave 5 and plays the **`G major`** scale from the top down by switching to octave 4 starting at the **`B`**.

Tempo

- **Synopsis**: Sets the BPM, the number of quarter notes per minute
- **Syntax**: `T<1-255>`

High tempo values and short notes tend to have inaccurate lengths due to quantization error. Delays within a string do keep track of fractional frames so the overall playback length should be relatively consistent.

Low tempo values that cause delays (lengths) to exceed 255 frames will also end up being inaccurate. For very long notes, it may be better to use legato to string several together.

Example:

```
10 FMPLAY 0,"T120C4CGGAAGR"
20 FMPLAY 0,"T180C4CGGAAGR"
```

On the YM2151 using channel 0, plays in the current octave the first 7 notes of *Twinkle Twinkle Little Star*, first at 120 beats per minute, then again 1.5 times as fast at 180 beats per minute.

i

Volume

- **Synopsis**: Set the channel or voice volume
- **Syntax**: `V<0-63>`

This macro mirrors the `PSGVOL` and `FMVOL` BASIC commands for setting a channel or voice's volume. 0 is silent, 63 is maximum volume.

Example:

```
FMPLAY 0,"V40ECV45ECV50ECV55ECV60ECV63EC"
```

On the YM2151 using channel 0, starting at a moderate volume, plays the sequence **E C**, repeatedly, increasing the volume steadily each time.

Panning

- **Synopsis**: Sets the stereo output of a channel or voice to left, right, or both.
- **Syntax**: `P<1-3>`

1	Left
2	Right
3	Both

Example:

```
10 FOR I=1 TO 4
20 PSGPLAY 0,"P1CP2B+"
30 NEXT I
40 PSGPLAY 0,"P3C"
```

On the VERA PSG using voice 0, in the current octave, repeatedly plays a **C** out of the left speaker, then a **B♯** (effectively a **C** one octave higher) out of the right speaker. After 4 such loops, it plays a **C** out of both speakers.

Instrument change

- **Synopsis**: Sets the FM instrument (like FMINST) or PSG waveform (like PSGWAV)
- **Syntax**: `I<0-255>` (0-162 for FM)

Example:

```
10 FMINIT
20 FMVIB 200,15
30 FMCHORD 0,"I11CI11EI11G"
```

This program sets up appropriate vibrato/tremolo and plays a C major chord with the vibraphone patch across FM channels 0, 1, and 2.

YM2151 Registers

The YM register address space can be thought of as being divided into 3 ranges:

Range	Type	Description
$00 .. $1F	Global Values	Affect individual global parameters such as LFO frequency, noise enable, etc.
$20 .. $3F	Channel CFG	Parameters in groups of 8, one per channel. These affect the whole channel.
$40 .. $FF	Operator CFG	Parameters in groups of 32 - these map to individual operators of each voice.

Global Registers

<table>
<tr><td rowspan="2">Addr</td><td rowspan="2">Register</td><td colspan="8">Bits</td></tr>
<tr><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td rowspan="2">$01</td><td rowspan="2">Test</td><td>!</td><td>!</td><td>!</td><td>!</td><td>!</td><td>!</td><td>LR</td><td>!</td></tr>
<tr><td colspan="8">Bit 1 is the LFO reset bit. Setting it disables the LFO and holds the oscillator at 0. Clearing it enables the LFO. All other bits control various test functions and should not be written into.</td></tr>
<tr><td rowspan="2">$08</td><td rowspan="2">Key Control</td><td>.</td><td>C2</td><td>M2</td><td>C1</td><td>M1</td><td colspan="3">CHA</td></tr>
<tr><td colspan="8">Starts and Releases notes on the 8 channels. Setting/Clearing bits for M1,C1,M2,C2 controls the key state for those operators on channel CHA. NOTE: The operator order is different than the order they appear in the Operator configuration registers!</td></tr>
<tr><td>$0F</td><td>Noise Control</td><td>NE</td><td>.</td><td>.</td><td colspan="5">NFRQ</td></tr>
</table>

<table>
<tr><td></td><td></td><td colspan="7">NE = Noise Enable; NFRQ = Noise Frequency; When eabled, C2 of channel 7 will use a noise waveform instead of a sine waveform.</td></tr>
<tr><td rowspan="2">$10</td><td rowspan="2">Timer A High</td><td colspan="7">CLKA1</td></tr>
<tr><td colspan="7">Top 8 bits of Timer A period setting</td></tr>
<tr><td rowspan="2">$11</td><td rowspan="2">Timer A Low</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>.</td><td>CLKA2</td></tr>
<tr><td colspan="7">Bottom 2 bits of Timer A period setting</td></tr>
<tr><td rowspan="2">$12</td><td rowspan="2">Timer B</td><td colspan="7">CLKB</td></tr>
<tr><td colspan="7">Timer B period setting</td></tr>
<tr><td rowspan="2">$14</td><td rowspan="2">IRQ Control</td><td>CSM</td><td>.</td><td colspan="2">CLK ACK</td><td colspan="2">IRQ EN</td><td>CLK ST</td></tr>
<tr><td colspan="7">CSM: When a timer expires, trigger note key-on for all channels. For the other 3 fields, lower bit = Timer A, upper bit = Timer B. CLK ACK: clears the timer's bit in the YM_status byte and acknowledges the IRQ.</td></tr>
<tr><td rowspan="2">$18</td><td rowspan="2">LFO Freq</td><td colspan="7">LFRQ</td></tr>
<tr><td colspan="7">Sets LFO frequency. $00 = 0.008Hz $FF = 32.6Hz</td></tr>
<tr><td rowspan="3">$19</td><td rowspan="3">LFO Amplitude</td><td>0</td><td colspan="6">AMD</td></tr>
<tr><td>1</td><td colspan="6">PMD</td></tr>
<tr><td colspan="7">AMD = Amplitude Modulation Depth; PMD = Phase Modulation (vibrato) Depth; Bit 7 determines which parameter is being set when writing into this register.</td></tr>
<tr><td rowspan="2">$1B</td><td rowspan="2">LFO Waveform</td><td colspan="2">CT</td><td>.</td><td>.</td><td>.</td><td>.</td><td>W</td></tr>
<tr><td colspan="7">CT: sets output pins CT1 and CT1 high or low. (not connected to anything in X16); W: LFO Waveform: 0-4 = Saw, Square, Triange, Noise; For sawtooth: PM->//// AM->\\\\</td></tr>
</table>

LR (LFO Reset)
Register $01, bit 1

Setting this bit will disable the LFO and hold it at level 0. Clearing this bit allows the LFO to operate as normal. (See LFRQ for further info)

KON (KeyON)
Register $08

- Bits 0-2: Channel_Number
- Bits 3-6: Operator M1, C1, M2, C2 control bits:
    - 0: Releases note on operator
    - 0->1: Triggers note attack on operator
    - 1->1: No effect

Use this register to start/stop notes. Typically, all 4 operators are triggered/released together at once. Writing a value of $78+channel_number will start a note on all 4 OPs, and writing a value of $00+channel_number will stop a note on all 4 OPs.

NE (Noise Enable)
Register $0F, Bit 7

When set, the C2 operator of channel 7 will use a noise waveform instead of a sine.

NFRQ (Noise Frequency)
Register $0F, Bits 0-4

Sets the noise frequency, $00 is the lowest and $1F is the highest. NE bit must be set in order for this to have any effect. Only affects operator C2 on channel 7.

CLKA1 (Clock A, high order bits)
Register $10, Bits 0-7

This is the high-order value for Clock A (a 10-bit value).

CLKA2 (Clock A, low order bits)
Register $11, Bits 0-1

Sets the 2 low-order bits for Clock A (a 10-bit value). Timer A's period is Computed as:

```
(64*(1024-ClkA)) / PhiM ms. (PhiM = 3579.545Khz)
```

CLKB (Clock B)
Register $12, Bits 0-7

Sets the Clock B period. The period for Timer B is computed as:

```
(1024*(256-CLKB)) / PhiM ms. (PhiM = 3579.545Khz)
```

CSM
Register $14, Bit 7

When set, the YM2151 will generate a KeyON attack on all 8 channels whenever Timer A overflows.

Clock ACK
Register $14, Bits 4-5

Clear (acknowledge) IRQ status generated by Timer A and Timer B (respectively).

IRQ EN
Register $14, Bits 2-3

When set, enables IRQ generation when Timer A or Timer B (respectively) overflow. The IRQ status of the two timers is checked by reading from the YM2151_STATUS byte. Bit 0 = Timer A IRQ status, and Bit 1 = Timer B IRQ status. Note that these status bits are only active if the timer has overflowed AND has its IRQ_EN bit set.

Clock Start
Register $14, Bits 0-1

When set, these bits clear the Timer A and Timer B (respectively) counters and starts it running.

LFRQ (LFO Frequency)
Register $18, Bits 0-7

Sets the LFO frequency:

- $00 = 0.008Hz
- $FF = 32.6Hz

Note that even setting the value zero here results in a positive LFO frequency. Any channels sensitive to the LFO will still be affected by the LFO unless the `LR` bit is set in register $01 to completely disable it.

AMD (Amplitude Modulation Depth)
Register $19 Bits 0-6, Bit 7 clear

Sets the peak strength of the LFO's Amplitude Modulation effect. Note that bit 7 of the value written into $19 must be clear in order to set the AMD. If bit 7 is set, the write will be interpreted as PMD.

PMD (Phase Modulation Depth)
Register $19 Bits 0-6, Bit 7 set

Sets the peak strength of the LFO's Phase Modulation effect. Note that bit 7 of the value written into $19 must be set in order to set the PMD. If bit 7 is clear, the value is interpreted as AMD.

CT (Control pins)
Register $1B, Bits 6-7

These bits set the electrical state of the two CT pins to on/off. These pins are not connected to anything in the X16 and have no effect.

W (LFO Waveform)
Register $1B, Bits 0-1

Sets the LFO waveform:

- 0: Sawtooth
- 1: Square (50% duty cycle)
- 2: Triangle, 3: Noise

Channel CFG Registers

Register Range	Register Bits							
	7	6	5	4	3	2	1	0
$20 + channel	RL		FB			CON		
$28 + channel	.	KC						
$30 + channel	KF						.	.
$38 + channel	.	PMS			.	.	AMS	

Description	
RL	Right/Left Output Enable
FB	M1 Feedback Level
CON	Operator Connection Algorithm
KC	Key Code
KF	Key Fraction
PMS	Phase Modulation Sensitivity
AMS	Amplitude Modulation Sensitivity

RL (Right/Left output enable)
Register $20 (+ channel), Bits 6-7

Setting/Clearing these bits enables/disables audio output for the selected channel. (bit6=left, bit7=right)

FB (M1 Self-Feedback)
Register $20 (+ channel), bits 3-5

Sets the amount of self feedback on operator M1 for the selected channel. 0=none, 7=max

CON (Connection Algorithm)
Register $20 (+ channel), bits 0-2

Sets the selected channel to connect the 4 operators in one of 8 arrangements.

KC (Key Code - Note selection)
Register $28 + channel, bits 0-6

Sets the octave and semitone for the selected channel. Bits 4-6 specify the octave (0-7) and bits 0-3 specify the semitone:

0	1	2	4	5	6	8	9	A	C	D	E
C♯	D	D♯	E	F	F♯	G	G♯	A	A♯	B	C

Note that natural C is at the TOP of the selected octave, and that each 4th value is skipped. Thus if concert A (A-4, 440hz) is KC=$4A, then middle C is KC=$3E

KF (Key Fraction)
Register $30 + channel, Bits 2-7

Raises the pitch by 1/64th of a semitone * the KF value.

PMS (Phase Modulation Sensitivity)
Register $38 + channel, Bits 4-6

Sets the Phase Modulation (vibrato) sensitivity of the selected channel. The resulting vibrato depth is determined by the combination of the global PMD setting (see above) modified by each channel's PMS.

Sensitivity values: (+/- cents)

0	1	2	3	4	5	6	7
0	5	10	20	50	100	400	700

AMS (Amplitude Modulation Sensitivity)
Register $38 + channel, Bits 0-1

Sets the Amplitude Modulation sensitivity of the selected channel. Note that each operator may individually enable or disable this effect on its output by setting/clearing the AMS-Ena bit (see below). Operators acting as outputs will exhibit a tremolo effect (varying volume) and operators acting as modulators will vary their effectiveness on the timbre when enabled for amplitude modulation.

Sensitivity values: (dB)

0	1	2	3
0	23.90625	47.8125	95.625

Operator CFG Registers

Register Range	Operator	Register Bits							
		7	6	5	4	3	2	1	0
$40	M1: $40+channel M2: $48+channel C1: $50+channel C2: $58+channel	.	DT1			MUL			
$60	M1: $60+channel M2: $68+channel C1: $70+channel C2: $78+channel	.	TL						
$80	M1: $80+channel M2: $88+channel C1: $90+channel C2: $98+channel	KS		.	AR				
$A0	M1: $A0+channel M2: $A8+channel C1: $B0+channel C2: $B8+channel	AM	.	.	D1R				
$C0	M1: $C0+channel M2: $C8+channel C1: $D0+channel C2: $D8+channel	DT2		.	D2R				
$E0	M1: $E0+channel M2: $E8+channel C1: $F0+channel C2: $F8+channel	D1L				RR			

	Description
DT1	Detune Amount (fine)
MUL	Frequency Multiplier
TL	Total Level (volume attenuation) (0=max, $7F=min)
KS	Key Scaling (ADSR rate scaling)
AR	Attack Rate
AM	Amplitude Modulation Enable

D1R	Decay Rate 1 (From peak down to sustain level)
DT2	Detune Amount (coarse)
DR2	Decay Rate 2 (During sustain phase)
D1L	Decay Level 1 (Sustain level)
RR	Release Rate

Operators are arranged as follows:

name	M1	M2	C1	C2
index	0	1	2	3

These are the names used throughout this document for consistency, but they may function as either modulators or carriers, depending on which `CON` ALG is used.

The Operator Control parameters are mapped to channels/operators as follows: Register + 8op + channel. You may also choose to think of these register addresses as using bits 0-2 = channel, bits 3-4 = operator, and bits 5-7 = parameter. This reference will refer to them using the address range, e.g. $60-$7F = TL. To set TL for channel 2, operator 1, the register address would be $6A ($60 + 18 + 2).

DT1 (Detune 1 - fine detune)
Registers $40-$5F, Bits 4-6

Detunes the operator from the channel's main pitch. Values 0 and 4=no detuning. Values 1-3=detune up, 5-7 = detune down. The amount of detuning varies with pitch. It decreases as the channel's pitch increases.

MUL (Frequency Multiplier)
Registers $40-$5F, Bits 0-3

If MUL=0, it multiplies the operator's frequency by 0.5. Otherwise, the frequency is multiplied by the value in MUL (1,2,3...etc)

TL (Total Level - attenuation)
Registers $60-$7F, Bits 0-6

This is essentially "volume control" - It is an attenuation value, so $00 = maximum level and $7F is minimum level. On output operators, this is the volume output by that operator. On modulating operators, this affects the amount of modulation done to other operators.

KS (Key Scaling)
Registers $80-$9F, Bits 6-7

Controls the speed of the ADSR progression. The KS value sets four different levels of scaling. Key scaling increases along with the pitch set in KC. 0=min, 3=max

AR (Attack Rate)
Registerss $80-$9F, Bits 0-4

Sets the attack rate of the ADSR envelope. 0=slowest, $1F=fastest

AMS-Enable (Amplitude Modulation Sensitivity Enable)
Registers $A0-$BF, Bit 7

If set, the operator's output level will be affected by the LFO according to the channel's AMS setting. If clear, the operator will not be affected.

D1R (Decay Rate 1)
Registers $A0-$BF, Bits 0-4

Controls the rate at which the level falls from peak down to the sustain level (D1L). 0=none, $1F=fastest.

DT2 (Detune 2 - coarse)
Registers $C0-$DF, Bits 6-7

Sets a strong detune amount to the operator's frequency. Yamaha suggests that this is most useful for sound effects. 0=off

D2R (Decay Rate 2)
Registers $C0-$DF, Bits 0-4

Sets the Decay2 rate, which takes effect once the level has fallen from peak down to the sustain level (D1L). This rate continues until the level reaches zero or until the note is released.

0=none, $1F=fastest

D1L
Registers $E0-$FF, Bits 4-7

Sets the level at which the ADSR envelope changes decay rates from D1R to D2R. 0=minimum (no D2R), $0F=maximum (immediately at peak, which effectively disables D1R)

RR
Registers $E0-$FF, Bitst 0-3

Sets the rate at which the level drops to zero when a note is released. 0=none, $0F=fastest