ICE Documentation

Peter "PT_" Tillema
Lucas "KryptonicDragon" Torres

December 28, 2017

Contents

1 Welcome, Links

Welcome to our introduction and tutorial to ICE Compiler. This guide will start out with the basics and continue into the more complex uses and commands of ICE. This guide is for ICE version 2.X, you can view which version of ICE you have by reading the top text line inside of the ICE program on your calculator.
ICE's Cemetech topic for questions and bug reporting is here.
The latest version of ICE will likely be available to download from here.
Every command in ICE is listed and documented, right here. Bookmark and/or print those, you will very likely need them.

2 Introduction to ICE

ICE Compiler is a C program created by the talented Peter Tillema, known on Cemetech as an administrator with the username `PT_`. ICE allows you to create advanced programs right on your 84+ CE without using TI's slow, restrictive language that is known as TI-BASIC.
The way it works is simple, you make a program on your calculator using the standard TI-OS program editor, and use ICE to compile your program into eZ80 assembly, A.K.A raw machine code for your calculator. This means that your ICE programs will run much faster than TI-BASIC programs.
ICE uses its own programming language that has a few similarities to TI-BASIC, but has a vastly more complex programming system packed with features that allow you to make the most of your calculator. Full graphics controls, raw memory editing, lots of variables, and much more.
However, by using ICE, you can put your calculator and its data at risk. The ICE language is powerful, and allows you to create more advanced programs than TI-BASIC would ever allow, but it also allows you to freeze your calculator, cause RAM resets, corrupt data, and other dangers. When you decide to use ICE, be ready for whatever mistake you may make. Make backups, try to use an emulator if possible, and thoroughly check your code.

3 Installing ICE

Putting ICE onto your calculator is very simple and can be completed in just a few steps:

1. Download the ICECompiler.zip file from the GitHub releases page (see section 1).
2. Extract the .zip file you just downloaded using a program such as Bandizip, winRAR, 7zip, or similar.
3. If using your physical calculator, send `ICE.8xp` and `ICEAPPV.8xv` to your TI-84+ CE using TI Connect CE.
4. If using CEmu, drag and drop `ICE.8xp` to the right half of the screen (labeled "RAM"), then drag and drop `ICEAPPV.8xv` to the left half of the screen (labeled "Archive"). (In both cases, make sure ICE is sent to RAM, and ICEAPPV is sent to the archive.)
5. Run ICE at least once so it can set itself up. If you are **not** using OS 5.3 or newer, you must use the `Asm(` token found in the catalog, `[2nd],[0]` before the program name to run an assembly program like ICE. Like this:

   If you **are** using OS 5.3 or newer, you can just omit the `Asm(` token and run it like any other program.

You'll likely be greeted by a message saying ICE didn't find any programs to compile, this is expected as you don't yet have ICE programs. Hit `[Clear]` to exit ICE and return to the TI homescreen. You may see a few strange graphical artifacts on your homescreen, these are just harmless artifacts, just hit `[Clear]` on the TI homescreen to clean them up.

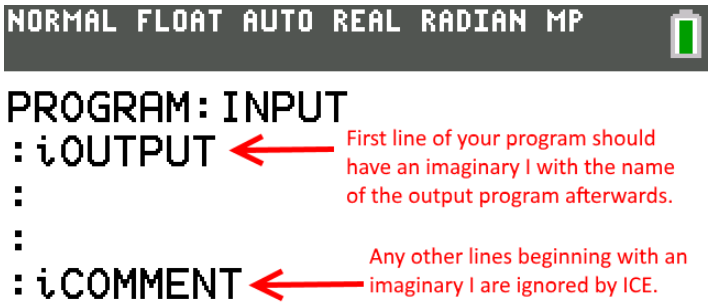
Now that you have set up ICE, you are ready to begin creating ICE programs.

4 Making an ICE program

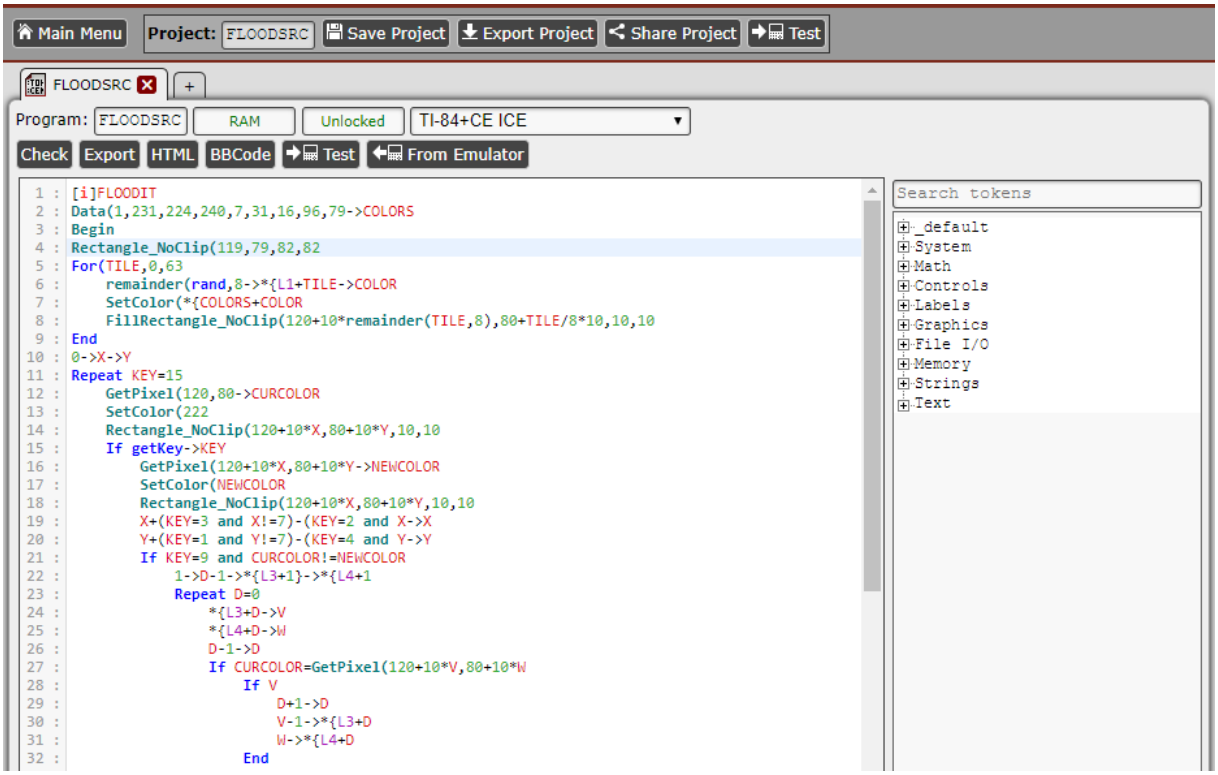
An ICE program is created like any other, from the homescreen on your calculator. Hit `[Prgm]`, `[→]`, `[→]`, `[Enter]`. Give the program a name, and hit `[Enter]`.
Right now that program doesn't do anything and ICE won't do anything to it. Because ICE compiles programs into assembly and stores the assembly into another program, you need to specify the name of the other program the compiled code will be put into.
To give your *compiled* program a name, place an imaginary *i*, `[2nd]`, `[.]`, on the very first line of your source program. This fancy *i* is called an 'Imaginary' and from now on will be referred to as *`[i]`*.
Directly after *`[i]`*, type in a name for the program this will be compiled into. It must follow the standard program naming scheme; starts with a letter, is all capital letters or numbers, no more than 8 characters.

Note: In ICE, when *`[i]`* is used at the beginning of any line other than the first, it signifies a comment line. *Everything* typed onto a comment line after *`[i]`* will be ignored by ICE and not get compiled into the final program. You may use this to create notes in your source code for yourself or to temporarily disable lines of code for debugging purposes.

An example of a comment and a name:



Editing your ICE program can be done inside your calculator's program editor, or you can instead use Cemetech's online IDE, SourceCoder3. SourceCoder3 has full ICE support, and it's much easier to write code on a computer. SourceCoder3 allows you to type in all of ICE's custom functions, graphics, and file I/O function names, and have the output automatically converted to the correct tokens. It also features syntax highlighting and indenting, giving you a nice organized look for your programming environment.

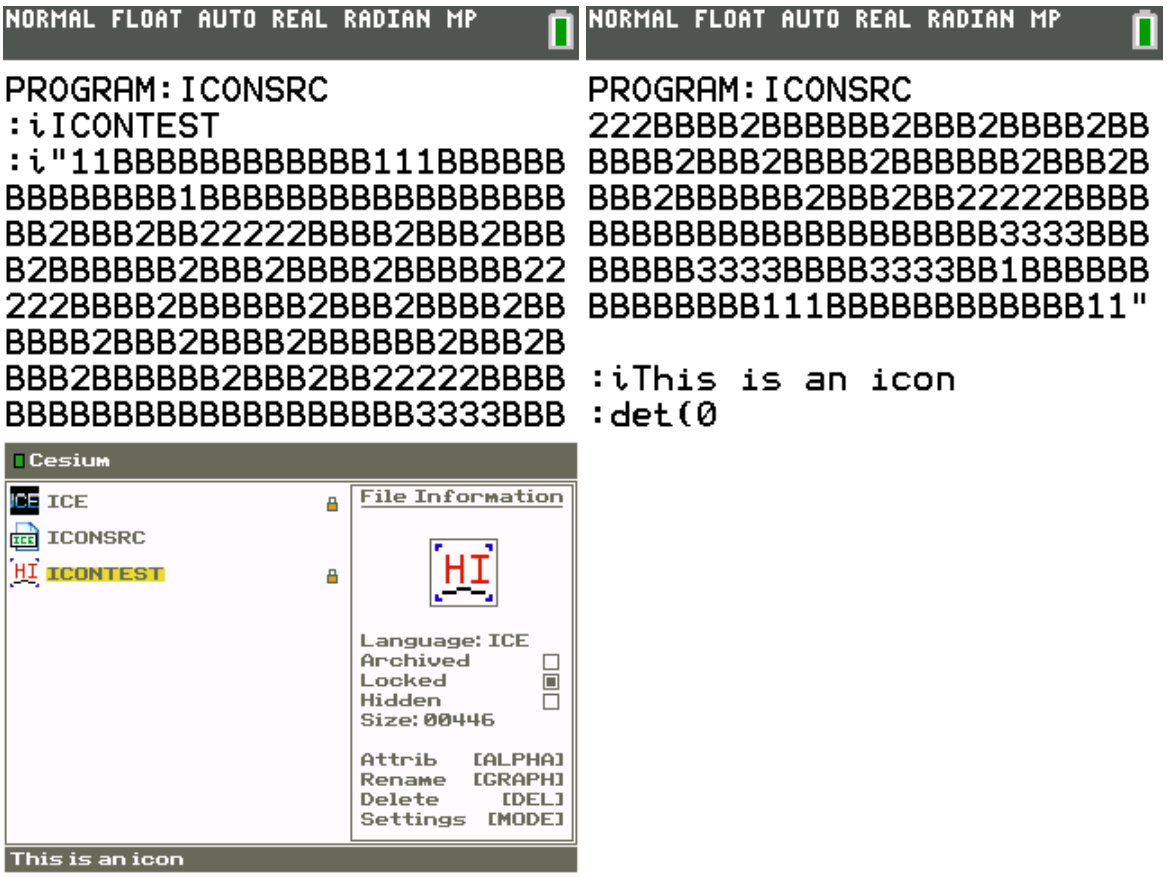



```
[i]FLOODIT
Data(1,231,224,240,7,31,16,96,79->COLORS
Begin
Rectangle_NoClip(119,79,82,82
For(TILE,0,63
    remainder(rand,8->*{L1+TILE->COLOR
    SetColor(*{COLORS+COLOR
    FillRectangle_NoClip(120+10*remainder(TILE,8),80+TILE/8*10,10,10
End
0->X->Y
Repeat KEY=15
    GetPixel(120,80->CURCOLOR
    SetColor(222
    Rectangle_NoClip(120+10*X,80+10*Y,10,10
    If getKey->KEY
        GetPixel(120+10*X,80+10*Y->NEWCOLOR
        SetColor(NEWCOLOR
        Rectangle_NoClip(120+10*X,80+10*Y,10,10
        X+(KEY=3 and X!=7)-(KEY=2 and X->X
        Y+(KEY=1 and Y!=7)-(KEY=4 and Y->Y
        If KEY=9 and CURCOLOR!=NEWCOLOR
            1->D-1->*{L3+1}->*{L4+1
            Repeat D=0
                *{L3+D->V
                *{L4+D->W
                D-1->D
                If CURCOLOR=GetPixel(120+10*V,80+10*W
                    If V
                        D+1->D
                        V-1->*{L3+D
                        W->*{L4+D
                    End
```

If you plan on using sprites and/or tilemaps, we *highly* recommend you use SourceCoder3, because it makes it much easier to paste in sprite/tilemap hex codes that you will need for graphical applications..
You can also transfer already existing programs between SourceCoder3 and your calculator.

4.1 Icon, description

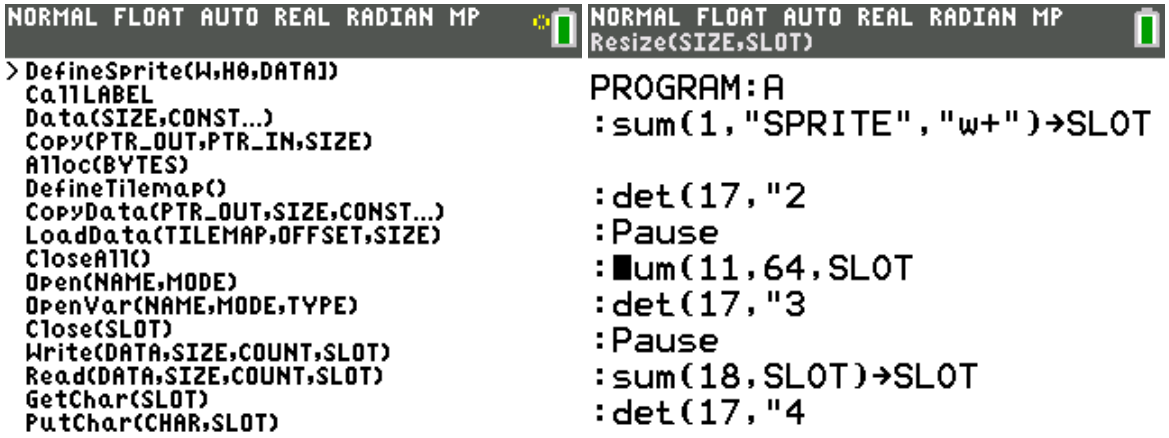
If you have used Cesium before, you might have noticed that some programs have their own icon and description. It is possible to give your programs an icon and description with ICE too!
Right after the header with the output name (the first line), put another line, starting with the imaginary *[i]* followed by a 16x16 pixel icon. This icon should begin and end with a " and consist of 256 hexadecimals.
You can make an icon with all 14 of the OS colors. Press `[VARS], [←]` to get a list with all the OS colors, and convert the number before the color to a hexadecimal.
If you are using SourceCoder3, you can switch the program type to CSE TI-BASIC, and type a ". This will open up a sprite editor. Create a new 16x16 sprite and start editing! This will automatically insert the hexadecimals, so you don't have to do that manually.
If you want to add a description too, add a new line after the icon line, that also starts with the imaginary *[i]*, and then the description, without quotes.



4.2 `[Trace]` Menu

Because a calculator can only have so many different tokens before it becomes an inefficient use of space, all of ICE's graphics functions are a `det(` token, and all of ICE's I/O functions are a `sum(` token, both with a number in it as a command index. Because all these numbers are difficult for a human to keep track of, ICE installs a useful feature into TI's program editor. If you hit `[Trace]` while inside a calculator's program editor, you will be greeted by a list of commands.
This menu includes the full names of ICE's commands as well as all of ICE's custom tokens and the syntax of each command/token. Use the left/right arrow keys to browse through the various pages of the list, and up/down arrow keys to move your cursor. Hit `[Enter]` to paste the selected command, or `[Clear]` to leave the menu without modifying anything.
Another feature that ICE implements to make it easier for you to keep track of the commands, is a "tooltip" that appears in the second line of the status bar, showing the full name and the syntax of the command. This tooltip automatically appears if you place your text caret (the black, blinking rectangle) over the beginning of a `det(` or `sum(` command.

`[Trace]` menu (left) and caret tooltip (right):



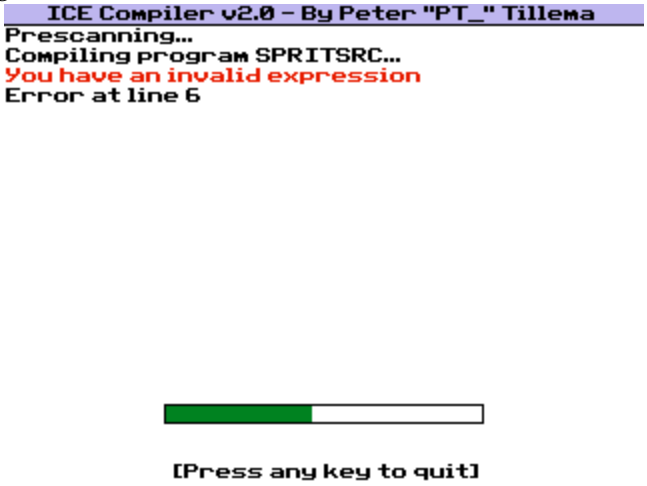
> **Note: ICE has to have been run on your calculator at least once since the last reset for these features to work.**

4.3 Compiling

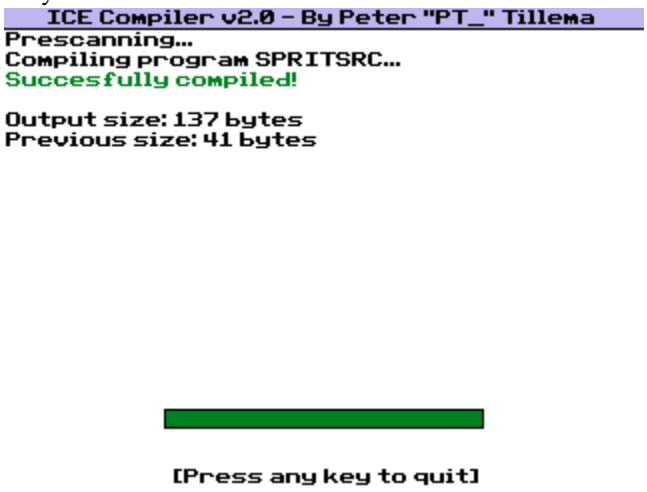
When you are ready to run your program, you need to run ICE and select the name of your source program from the list. ICE can compile program regardless of whether they are archived or in RAM.

> **Note: ICE will only show programs that have the proper *[i]* header. If you do not see your source program in the list, double check your header is correct. If you have more than 22 programs with an ICE header, a few may not be visible on screen.**

Once you hit [Enter] on your program, ICE will begin compiling. Depending on the size of your program, this may take less than a second or multiple seconds. If ICE encounters an error in your program, you will be given an error message explaining the type of error, and the line the error was encountered on. If your source program is unarchived, ICE will open the program editor and go to the line containing the error.



If there are no errors in your program, ICE will give you a "compiled successfully" message and will exit when you hit a key.



You may then run the compiled program (using the `Asm(` token if you are not using OS 5.3+). It's possible that at some point, your program will crash. When your programs have problems **after** being compiled, it is up to you to figure out the mistake you made in your program and to fix it. If you're having trouble, or are you sure that you did nothing wrong, be sure to ask for help on Cemetech (ICE's forum linked in the first section.)

5 Math & Numbers

ICE's math works just about how you would expect it to from any programming language, you have the four standard operators:

+	Addition
-	Subtraction
*	Multiplication
/	Division

You can use these operators in your math expressions and ICE will compile them in the correct order (see 5.4). ICE will also automatically simplify expressions when compiling to save space in the final program.
The next subsections go over the specifics of mathematics and numbers in ICE.

5.1 Positive Integers

ICE only supports positive integers. This means no negative numbers, no decimals, and no fractions. The most directly affected operation by this is division. Division will always return a truncated answer, meaning that anything after the decimal point (and the decimal point itself) is simply omitted (truncated). For example:

Expression	Returns	Explanation
$10/3$	3	$10/3 \approx 3.33$, truncated = 3.
$84/52$	1	$84/52 \approx 1.62$, truncated = 1.
$8/3$	2	$8/3 \approx 2.67$, truncated = 2.
$7/12$	0	$7/12 \approx 0.58$, truncated = 0.

If you try to compile a program that includes a decimal point or a fraction, you will get a "`This token/function is not implemented (yet)`" error.
Due to ICE only having positive integers, the negative sign works differently than it does normally. It works by taking 16777216 (2^{24}) and subtracting the number after the negative sign.

Note: Remember that there is a different key for a negative sign than the key for a subtraction sign.

Expression	Returns	Explanation
-85	16777131	$16777216 - 85 = 16777131$
-9000000	7777216	$16777216 - 9000000 = 7777216$
$-(2 * 8)$	16777200	$16777216 - (2 * 8) = 16777200$

5.2 3-byte Numbers

ICE uses 3 bytes, or 24 bits, to store all numbers and perform mathematical operations. This means the biggest number you can have is $2^{24} - 1$, so 16,777,215. You cannot have more than this number.

If you go over this number, you get what is known as 'rollover'. This means your result will be modulo 16777216 (the remainder when you divide by 16777216). An example of this is:

$$16777215 + 10 = 9$$

That equation went over the 3-byte limit, so it 'rolled over'(looped back around) to 0, then continued the operation. The same can happen backwards,

$$0 - 10 = 16777206$$

Later in this guide we will introduce you to some control over whether a number is stored in 3-byte, 2-byte, or 1-byte.

5.3 Not Implied Multiplication

Multiplication is never implied. Remember how in TI-BASIC you could have `10A` in your program and it would be interpreted as `10*A`? This was because TI-OS would automatically add a multiplication sign in between the 10 and A. ICE, for reasons we will cover later, doesn't do that. You must always type a multiplication sign when you want to multiply. This includes when using parentheses.

$8(12+4)$	Incorrect	$8*(12+4)$	Correct
$15A$	Incorrect	$15*A$	Correct
ABC	Incorrect	$A*B*C$	Correct
$5(AG)(7*8)$	Incorrect	$5*(A*G)*(7*8)$	Correct

5.4 Order of Operations

If you made it through middle school, you (hopefully) know of the mathematical order of operations, known as `PEMDAS`. This simple string of letters tells us the order in which we do math when we come across an expression with more than one operation in it. ICE properly follows this order. For those who don't remember, the order is:

Parentheses	Do what is inside parentheses first.
Exponents	Solve the exponents (this includes roots.)
Multiplication and **D**ivision	Both at the same time, from left to right.
Addition and **S**ubtraction	Both at the same time, from left to right.

This means that if ICE comes across $8 + (40 - 7 * 2)$, it will do $7 * 2$, then $40 -$ **14**, then $8 +$ **26**.

5.5 Modifiers

There are two little modifiers you can use with constants in your ICE programs:

1. Scientific E (ᴇ). You can paste this in by hitting `[2nd] [,]`. When this is placed directly before a constant, ICE will convert the constant from hexadecimal to decimal when compiled.
2. Pi (π). You can paste this in by hitting `[2nd] [^]`. When this is placed directly before a constant, ICE will convert that constant from binary to decimal when compiled.

5.6 Mathematical Functions

ICE supports a few of the math functions you can find in TI-OS (though a few have different syntax), they are:

Function	Explanation
`remainder(EXP1,EXP2)`	Returns the remainder when EXP1 is divided by EXP2.
`min(EXP1,EXP2)`	Returns the smaller value of EXP1 and EXP2.
`max(EXP1,EXP2)`	Returns the larger value of EXP1 and EXP2.
`mean(EXP1,EXP2)`	Returns the mean (average) value of EXP1 and EXP2.
`sqrt(EXP)`	Returns the square root of EXP.
`sin(EXP)`	Returns the sine root of `EXP`. Input `EXP` should be a value between 0 and 255 (split the circle into 256 equal sections) and will return a value between -255 and 255.
`cos(EXP)`	Returns the cosine root of `EXP`. Input `EXP` should be a value between 0 and 255 (split the circle into 256 equal sections) and will return a value between -255 and 255.

⚠ **Note: All results will be truncated.**

Examples:

Code	Returns	Explanation
`remainder(10,3)`	1	3 goes into 10 only 3 whole times, with 1 leftover.
`remainder(64,10)`	4	10 goes into 64 only 6 whole times, with 4 leftover.
`remainder(10,5)`	0	5 goes into 10 *exactly* 2 whole times, with 0 leftover.
`min(5,2)`	2	2 is *less* than 5.
`min(123,120)`	120	120 is *less* than 123.
`min(10,10)`	10	10 is *equal* to 10.
`max(5,2)`	5	5 is *greater* than 2.
`max(123,120)`	123	123 is *greater* than 120.
`max(10,10)`	10	10 is *equal* to 10.
`mean(22,20)`	21	$(22+20)/2 = 21$
`mean(64,20)`	42	$(64+20)/2 = 42$
`mean(5,8)`	6	$(5+8)/2 = 6.5$, truncated = 6.
`sqrt(25)`	5	$\sqrt{25} = 5$.
`sqrt(120)`	10	$\sqrt{120} \approx 10.954$, truncated = 10.
`sqrt(2)`	1	$\sqrt{2} \approx 1.414$, truncated = 1.
`sin(2)`	12	$255 * sin(2/256 * 2 * \pi) \approx 12.51$, truncated = 12.
`sin(120)`	49	$255 * sin(120/256 * 2 * \pi) \approx 49.75$, truncated = 49.
`sin(255)`	16777210	$255 * sin(255/256 * 2 * \pi) \approx -6.26$, truncated = -6.
`cos(2)`	254	$255 * cos(2/256 * 2 * \pi) \approx 254.69$, truncated = 254.
`cos(120)`	16776966	$255 * cos(120/256 * 2 * \pi) \approx -250.10$, truncated = -250.
`cos(255)`	254	$255 * cos(255/256 * 2 * \pi) \approx 254.92$, truncated = 254.

5.7 Binary

This subsection covers the binary number system, so if you already know how that works, feel free to skip this section. If you plan on using the more advanced features of ICE, such as bitwise operators (covered below), you'll need to have an understanding of how numbers are stored in computers as bytes and bits of binary.
In our standard decimal number system, there are ten different characters, zero through nine. When we put multiple of these characters together into one number, the order of these numbers from right to left signifies how big each place is, going by powers of ten starting at zero.
For example, in the number 5124, the farthest right number (number 4) is the smallest, having only ten to the power of zero amounts per value. If you look at the character to the left (number 2), you now have a value that has ten to the power of one, amounts. Look left again (number 1), and you have ten to the power of two amounts.
This means that the amount, 5124, is compromised of 4 amounts of one (10^0), 2 amounts of ten (10^1), 1 amount of a hundred (10^2), and 5 amounts of a thousand (10^3).
To get the amount for each place, you take the amount of characters in the number system, ten for decimal, and power it to the place the character is in going from the right.
In binary, however, there is only ones and zeroes, so only two characters. That means that each place in a binary number is an amount of a power of two. Take the following number for example:

`001101`

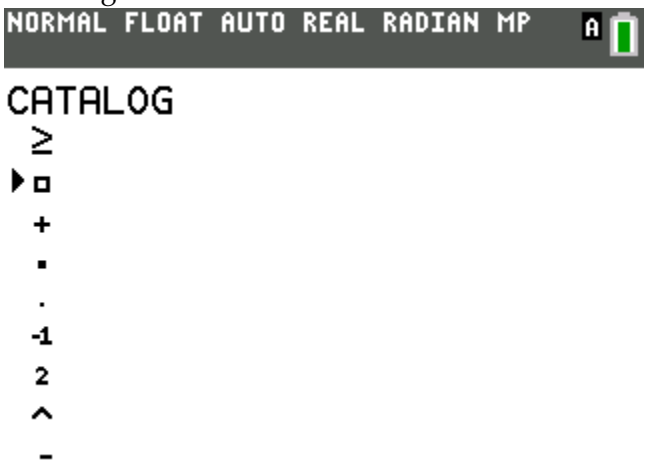
Now let's convert it into decimal so we can actually understand what value that number represents. Starting from the right-most place, 1 (the digit in the place) times 2^0 (the amount of the place) gives us 1. Now we continue to the left.
In the second place from the right, we have a 0, which means no value, so we can ignore this and continue to the left. In the third place we have a 1, so we multiply 1 by 2^2 (power of two because this is the second place after the right-most place). 1 times 4 is 4, and we add the previous values, $1 + 4$ is 5.
Look left once more and we have another 1, so we multiply 1 by 2^3 (power of three because this is the third place after the right-most place). 1 times 8 is 8, and we add the previous values, $5 + 8$ is 13.
The rest of the numbers to the left are zeroes, so we ignore them. After that tedious process, we have 13. So binary `001101` is equal to decimal 13.

5.8 Bitwise operators

Normally, all the math you do in ICE modifies bytes at a time; this makes sense, as each number is represented by one, two, or three bytes. However, there are situations in which you'd want to modify values by one bit at a time. Each byte is made up of 8 bits, each bit is either a 1 or a 0. When a bit is 1, it is considered to be "set", otherwise it is considered to be "reset". The action of changing a bit to it's opposite (changing a 1 to a 0 or a 0 to a 1) is called "flipping".
The right-most bit of a byte is bit 0, the one to the left of that is bit 1, to the left of that is bit 2, etc.
These situations are where you would want to use bitwise operators. The bitwise operators are plot style tokens, which you access from the bottom of the catalog (`[2nd], [0]`), below the inequality signs. They look like a miniature plus sign, a square outline, and a dot. The three tokens are as follows:

EXP1 + EXP2	Bitwise OR of EXP1 and EXP2
EXP1 · EXP2	Bitwise AND of EXP1 and EXP2
EXP1 ◻ EXP2	Bitwise XOR of EXP1 and EXP2

In the terminology we just gave you:
Bitwise OR is used to **set** bit(s).
Bitwise AND is used to **reset** bit(s).
Bitwise XOR is used to **flip** bit(s).
They look like this in the catalog:



If you're already familiar with binary or you read the previous subsection, you know that each bit in binary data is a power of two. The right-most bit is 2^0 (1), the bit left of that 2^1 (2), left once more is 2^2 (4), etc. This is important to know for bitwise operators, because you need to know what the values you are using look like in binary.
Say for example that you wanted to flip bit 4 of the three byte value, 4233, which looks like this:

00000000000100001001001

Because we want to flip bit **4** of 4233, we must use the bitwise XOR operator with two to the power of **4**, which looks like this:

```
4233 ◻ 16 → A
```

We used 16 after the bitwise XOR because $2^4 = 16$. The result of the A in the example is 4249, which looks like this:

00000000000100001001**1**001

It may not be very apparent right now, but bitwise operators are more useful when you're dealing with memory and pointers.

6 Variables

In ICE, you have variables just like any most other programming languages, you can read their values, store values into them, and use them in expressions. Remember that ICE still only supports 3-byte numbers.

Note: The "→" symbol in this document is the STORE character, you type this by pressing the button just left of 1, [STO→].

6.1 Multi-Letter Variables

Variables in ICE work very similarly to TI-BASIC, but they're better and you have more of them. Instead of single-letter variables in TI-BASIC such as A or X, ICE lets you have any variable name, up to 10 characters, as long as the entire name is made out of CAPITAL letters. Examples:

```
1999/32→APPLES
98*2→FRANK
42→MEANING
2*FROG→LION
```

You may still use single characters as variables if you wish. By default, all variables are stored with 3-bytes.

Note: You can use up to 85 variables!

6.2 Chained Store

In TI-BASIC, if you wanted to store the same value to multiple variables, you would have to write the initial value multiple times like this:

```
42→A
42→B
42→C
42→D
40→E
```

In ICE, you can instead chain the whole thing together on one line:

```
42→A→B→C→D-2→E
```

It saves a few bytes and makes your code less cluttered. (It also works with multi-letter variables, of course.)

6.3 Lists

Lists in ICE a very different to lists in TI-BASIC, mainly because of ICE's pointers. You can no longer store data directly into lists using `{1,2,3,4,5}→L1` and instead have to use the `Copy()` and/or `CopyData()` commands to load data into lists. `CopyData()` allows you to copy data from one place to another (hence the name), and when used with lists can be used to emulate TI-BASIC list syntax. The syntax for `CopyData()` is:

```
CopyData(DESTINATION,SIZE,DATA1,DATA2,...)
```

With DESTINATION being a memory address, and SIZE being the size in bytes for each entry, all other arguments are the data that is to be copied into the destination. Using this to store {1,2,3,4,5} into L1 would look like this:

```
CopyData(L1,3,1,2,3,4,5
```

This has one small detail you'll need to understand. The second argument in the example above, 3, says that each entry will be copied as a 3-byte number.

The size of each entry in a list is very important to how you will access data from a list. There are many different ways to access data from a list, but these are the main 5:

- `L1(X)`
- `{L1+X}`
- `***{L1+X}`
- `**{L1+X}`
- `*{L1+X}`

The first three work exactly the same way and are just different ways of typing it out, they access the Xth byte, the Xth+1 byte, and the Xth+2 byte, and return them as a single number. This means that if your list has 3-byte entries in it and you want to access one of those 3-byte entries, you want to use a multiple of 3 in your command. So let's say you previously used the example command, CopyData(L1,**3**,1,2,3,4,5), accessing your list after this would look like:

`L1(0)`	`{L1+0}`	`***{L1+0}`	**All** return 1.
`L1(3)`	`{L1+3}`	`***{L1+3}`	**All** return 2.
`L1(6)`	`{L1+6}`	`***{L1+6}`	**All** return 3.
`L1(9)`	`{L1+9}`	`***{L1+9}`	**All** return 4.
`L1(12)`	`{L1+12}`	`***{L1+12}`	**All** return 5.

Notice how the numbers you are putting inside the parentheses are **multiples of 3, starting at 0**. This is because the `L1(X)` method of accessing a list returns 3-byte numbers. If you were to use `CopyData(L1,2,1,2,3,4,5)` instead of the previous command, you would then store that data into list 1 as 2-byte entries, and then have to use:

Now notice that to access **2**-byte data from a list, you need to use curly brackets with **two** asterisks before them, and **multiples of 2, starting at 0**.

`**{L1+0}`	Returns 1.
`**{L1+2}`	Returns 2.
`**{L1+4}`	Returns 3.
`**{L1+6}`	Returns 4.
`**{L1+8}`	Returns 5.

Lastly, if you were to use `CopyData(L1,1,1,2,3,4,5)` instead of the previous command, you would then store that data into list 1 as **1**-byte entries, and then have to use:

`*{L1+0}`	Returns 1.
`*{L1+1}`	Returns 2.
`*{L1+2}`	Returns 3.
`*{L1+3}`	Returns 4.
`*{L1+4}`	Returns 5.

For that list of 1-byte data, you need to use curly brackets with **one** asterisk before them, and **multiples of 1, starting at 0**. The point here is, when storing data in 3-byte, you need to use multiples of 3 starting at 0 to access data with `L1(X)`, `{L1+X}`, or `***{L1+X}`. When storing data in 2-byte, you need to use multiples of 2 starting at 0 to access data with `**{L1+X}`. When storing data in 1-byte, you need to use multiples of 1 starting at 0 to access data with `*{L1+X}`.
If in doubt, just use 3-byte values as they will give you the largest range of values (0-16777215), but if you're an optimizing type of person, use the smallest byte sizes anywhere you can, just remember:

3-byte = 0 through 16777215
2-byte = 0 through 65535
1-byte = 0 through 255

If you're still confused on how `L1(X)` works, remember that X is the offset, in bytes, from the beginning of the list. So because `L1(X)` accesses 3-byte values, you need to use offsets of multiples of 3, otherwise you'll be reading 3 bytes in between two values.

> ⚠ **Note: Each list has a total size limit of 2000 bytes.**

6.4 Strings

In ICE, you can store a string into any of the 10 string vars (Str0 through Str9) using the standard syntax: `"THIS IS A STRING"→Str1`. You also have the `length()`, `sub()`, and `+` commands that work similarly to how they would in TI-BASIC:
`length()` returns the length, in characters, of the string you give it:

`length("ALPHABET")` returns 8.
`"SOMETHING"→Str1:length(Str1)` returns 9.

`sub()` cuts out part of a string, puts it into temporary memory, and returns a pointer to it. Proper usage would be: `sub(STRING,OFFSET,SIZE)`.

With STRING being either a string variable (Str0-Str9) or a string on it's own ("EXAMPLE"), OFFSET being the amount of characters to skip before beginning the cut, and SIZE being the length of the cut, in characters, starting from the offset. (If you wish to start cutting from the beginning of a string, you may use 0 as an offset).
With the pointer, you can read the temporary memory and put it back into a string. Remember that all strings in memory are null-terminated (simply meaning that their end is signified by a 0 byte).
+ concatenates two strings:

`"SOME"+"BODY"→Str1`	Str1 would then be "SOMEBODY".
`"BODY"+"SOME"→Str3`	Str3 would then be "BODYSOME
`Str1+" ONCE TOLD ME"→Str0`	Str0 would then be "SOMEBODY ONCE TOLD ME"

Besides using the standard syntax of `"STUFF"→Str1`, you can also place a string (text in quotes) at a part of your program, and it will be compiled into your program's code with its pointer returned. For example, when ICE reads `"PINEAPPLE"→A`, the binary data for "PINEAPPLE" will be compiled into your final program and its pointer will be stored into A. With this, you can later use `Copy()` commands to read from strings stored in your program code.
If you place a string token on its own, without storing anything to it, it will return the memory address of the string. You can use this in commands that require a pointer to a string or for other purposes. The only exception for this is when using a string in a `Disp` command.

> ⚠ **Note: Each string variable has a total size limit of 2000 bytes.**

6.4.1 Squishing hexadecimals

Sometimes you will see that ICE automatically squished the string. This means that if the string consists only of hexadecimals (0-9, A-F), it automatically replaces 2 hexadecimals with the right byte, so the hexadecimals "B" and "9" will be replaced with the byte $B9, or 185 in decimal. This is done because you can't input every single byte with TI-BASIC tokens, and it's needed for some routines, like `DefineSprite(` or `SetPalette(`. If the string is automatically squished, but you still want to use the hexadecimals, use the single apostrophe instead of the double apostrophe (or quote). This is valid because the characters of the hexadecimals are the same as the tokens. Squishing is disabled with `DefineSprite(`, so if you have a lot of sprites in a subprogram, you don't have to worry that you get tons of messages saying the string is squished.

6.5 Degree Modifier

There is a modifier you can use in ICE called the degree symbol (°). You can paste this in by hitting [2nd] [apps], [Enter] (or [1]). When this is placed *directly* before a **variable**, it will return

the memory address of where the variable is stored. Some commands require you use this, and you will see the documentation using them when required. This modifier does not work with strings or lists, only named variables.

7 Standard System Commands

The system section of the ICE commands list contains a few of the tokens available in the program editor. This section quicky goes over the commands and their use, in slightly more detail than the ICE command list gives.

7.1 getKey, getKey(X)

`getKey` works like it does in TI-BASIC, it returns the keycode of the last pressed key, or a 0 if no key was pressed since the last inquiry. However, the keycodes are different than the TI-BASIC keycodes. Your ICE download should contain an image showing the keycodes, but you may also find it here. An example of its use:

```
Repeat K=9
getKey→K
End
```

That simple code will just wait until the user hits [Enter].
`getKey` has another variation, `getKey(X)`. The difference is that `getKey(X)` returns a 1 if the key with the keycode you give it has pressed, 0 if released. It's much faster than `getKey` and it allows you to directly read the current state of a button, meaning that it's more useful for games where you need to constantly do something as the user holds down a key.
Example:

```
While getKey(2)
X+1→X
End
```

That code continues increasing the value of X for as long as the user holds down left arrow.

7.2 rand

`rand` is simple, you put it somewhere, and it gives you a pseudo-random number between 0 and 16777215. Pseudo-random means that the number is not *completely* random, because computers cannot actually generate random values, they instead take a starting number (called the seed) and apply a bunch of operations to it and spit out another number. Example:

```
rand→A
```

`A` could now be 64, it could be 7777, who knows? It's random!

7.3 randInt()

`randInt()` works basically the same as `rand`, except that you specify a range where the random number should be in. So something like `randInt(1,6)` returns a random number in between 1 and 6, like a dice.

7.4 Asm()

The `Asm()` token, found in the Catalog ([2nd] [0], you'll see it on the 7th line) directly inserts assembly code into the location of where the `Asm()` token would be in the final program. You give it the assembly code in the form of hexadecimal, all of the code inside the parentheses. This is useful if you have a particular assembly routine you'd like to use in your program, or if you just need to do something that ICE can't do.

7.5 AsmComp()

The `AsmComp()` token, found in the Catalog ([2nd], [0], you'll see it on the 8th line) tells ICE to compile another ICE program, the name of which is inside the parentheses. The subprogram should not have the *[i]* header, and will be compiled into the location of the `AsmComp()` token. This can be used if you have a large ICE program you want to split into parts or if you have a lot of sprites in your program that you'd like to avoid editing. The sub-programs can also be archived, ICE will still compile them. Example:

```
AsmComp(ZSPRITE)
```

That would compile a program called `ZSPRITE` into ICE.

7.6 Pause

The `Pause` token on it's own works how it does in TI-BASIC, it will pause the program until the user hits `[Enter]`, and then continue. You also have the option to put a number after it, which will make it pause for that amount of milliseconds (there are 1000 milliseconds in one second). Examples:

```
Pause 200
Pause 1000
Pause
```

The first line waits for 200 milliseconds ($\frac{1}{5}$ of a second), the second line waits for 1000 milliseconds (1 second), and the final line waits for the user to press `[Enter]`

7.7 prgm

If you have just a standard `prgm` token with a program name after it, your compiled program will execute the TI-BASIC program, and any errors that are encountered will be ignored. Use this if you want to run a TI-BASIC program you have from ICE. ICE execution continues as normal when the TI-BASIC program errors or ends.

7.8 Return, ReturnIf

ICE programs normally exit when execution reaches the bottom (end) of the program code, but if you need to end a program from somewhere else, you can use the `Return` token, available in the [Prgm] CTL menu from a program editor. When program execution reaches a `Return` token, the program ends as normal. There is also a `ReturnIf` token, Just place an `If` token directly after a `Return` token, with all your conditions after both tokens. This token is basically an `If` statement combined with a `Return` token. ReturnIf will evaluate the conditions you give it, and only return if the conditions return true (anything other than 0).
`Return` and `ReturnIf` also have one more use in ICE: to return from a `Call` token (see next subsection).

7.9 Goto, Lbl, Call

The `Goto` and `Lbl` tokens work the same way they do in TI-BASIC, `Lbl` defines a point in the program with a name, and `Goto` will jump to the assigned label and continue execution from there. Label names can consist of up to 10 random characters.
ICE also gives you a `Call` token, available from the [Trace] menu in the program editor. The `Call` token will go to a label, and continue executing until it hits a return token or the end of the program, then will jump back to the original `Call` token and continue from there. This means that you can have sub-routines in your program that can be used to save space if you find yourself using the same snippets of code over and over. Example:

```
rand→A
If A>1000
Call SOMETHING
End
Goto OTHER
Lbl SOMETHING
A*238→B
Return
Lbl OTHER
```

Although a very simple example, it shows the point. The example generates a random number, stores it into `A`, then checks if `A` is greater than 1000. If so, it then calls the "SOMETHING" label, from there it multiplies `A` by 238 and stores the result into `B`, and returns from the call. Then it

jumps to the "OTHER" label and continues program execution. (In the example case, the program ends because there is nothing after the "OTHER" label)

7.10 *i*

If there is a `[i]` token at the beginning of a line in your program, other than the first line, the line will be completely ignored by ICE when compiling. This means that you can comment out code for debugging purposes, or to leave yourself comments so you can understand what your own code is for.

8 Program Flow Control

Just like in any other programming language, ICE has loops and if statements that allow you to control the flow of your program. There are a few tiny differences between TI-BASIC's use and ICE's use of the control statements, so we'll quickly go over the statements:

8.1 If, Else

If statements check the condition code given to them, if the condition evaluates to false (0), the code inside the `If` statement does not execute, otherwise, the code executes. If the statement has an `Else` token before the `End` token, the code inside the `Else` statement will be executed. The difference here in ICE is that ICE does not have a `Then` token. To make a comparison to TI-BASIC, this means that all `If`'s in ICE 'automatically' have a `Then` after them, meaning that all `If` statements in ICE must be paired with an `End` token. Example:

```
rand→A
ClrHome
If A>6400
Disp "A IS GREATER THAN 6400"
Else
Disp "A IS <= 6400"
End
```

The example generates a random number and stores it into `A`, then checks if `A` is greater than 6400. If `A` is greater than 6400, the code inside the If-Else block will be executed, meaning a string is displayed to the screen. If `A` is not greater than 6400 (if it's equal to or less than 6400), the code inside the Else-End block will be executed, displaying `"A IS <= 6400"` to the screen.

8.2 Repeat

Repeat loops run the code in their block once, then evaluate the condition code, and if the condition code returns false (0), the code in their block will be run again. Each time the statement repeats, the condition code will be checked. The loop will continue indefinitely until the condition evaluates to true. Example:

```
0→A
Repeat A>1000
A+1→A
End
ClrHome
Disp A
```

The example stores 0 into A, and then continually adds 1 to A until A is more than 1000. Of course, this is an incredibly inefficient way of doing 1001→A, but it's for example purposes.

8.3 While

While is the close sibling of Repeat. The difference here is, the condition is evaluated at the beginning of the loop, instead of at the end like Repeat. Another difference is that While checks for code to be true (any number other than 0) to run the loop. You can use Repeat if you want the code inside it to be run at least once, and you can use While if you want the code inside it to run only when the condition evaluates to true (and for however long the condition evaluates to true). Example:

```
0→A
While A<16777000
rand→A
End
```

The example stores 0 into A, and then continually stores a random number into A, until the number is greater than or equal to 16777000. Once again, inefficient code but it's for example purposes.

8.4 For(

The For(token is used to repeat code a certain amount of times, while using a variable as a counter. It must be paired with an End and has two different syntax forms, the second with a minor difference. The first one is For(VAR,START,END), with VAR being the variable to use in the loop, START being an expression stating the starting value of VAR, and END being an expression stating the ending value of VAR. A For(loop will first store the starting entry in VAR and check if VAR is now greater than END. If VAR is not greater than END, the code in it's loop will be run. At the end, VAR is incremented by 1, and the greater than check is done again. The loop will continue repeating until VAR is greater than END.
The second syntax form of For(is For(VAR,START,END,INCREMENT). VAR, START, and END are the same as in the first syntax form, and the INCREMENT is an expression that will be added to VAR at the end of each loop. That allows you to increment the counter by more than just one at a time. If you use a negative **constant** (no vars, as this is done at compile time) for INCREMENT, the loop will continue until VAR is END. Example:

```
For(A,1,20
ClrHome
Disp A
Pause 200
End
```

The example will first display 1 to the screen, then pause for 200 milliseconds ($\frac{1}{5}$ of a second), and

continue through the loop, displaying the value of `A` as it increments. After it displays 20 to the screen, the program exists as the loop finishes and there is no more code after the loop.

9 Pointers

Pointers can be difficult for newcomers to programming to understand, but once you understand how to properly use them, they are an incredibly powerful tool to have in your programming toolbox.
Pointers allow you to read and write data to your calculator's RAM and archive, giving you the ability to modify almost anything in your calculator's memory. You can use their ability to cheat a high score on Calcuzap, or to simply edit the name of a program.
But before we tell you how to use them, a warning: If you're not careful with pointers and write the wrong data to the wrong address in memory, you not only risk freezing your calculator, but there is a chance of you corrupting data, breaking the OS, or worse (although unlikely), permanently damaging your calculator. Be cautious and be sure you know what you're doing when messing with pointers. Once again, if possible, use an emulator.
We'll begin with a prerequisite, memory addresses. Every single byte of RAM and flash (archive) has a number bound to it, known as a memory address. If you want to read or write from a specific byte, you need to know it's memory address. If you treat a variable as a memory address, you then have a pointer. Here is an example:

```
15958016→A
*{A}→SEC
```

In the example above, we stored the number 15958016 into A, then we used `*{A}` to read one byte from the memory address that A points to, 15958016, and stored the read value into the variable `SEC`. 15958016 is the memory address for a byte of memory that is part of the real-time clock. Reading from this value gives you the current second. When you want to read from a memory address, there are four main ways to do so:

`{VAL}`	Reads a word (3-byte) value from location VAL.
`***{VAL}`	Reads a word (3-byte) value from location VAL.
`**{VAL}`	Reads a 2-byte value from location VAL.
`*{VAL}`	Reads a 1-byte value from location VAL.

The first two are exactly the same, the first is just easier to remember while the other makes it more clear that you are reading a 3-byte value. VAL can be an expression, a variable, or a constant (you can even put pointers inside of pointers). You may recognize these from section 6.3 on lists, because when you are reading from a list, you are reading from it's memory address plus an offset.
As for uses of pointers, if you find some other documentation, you can find memory addresses that give you useful information you could use in your programs, like the date or time. Some documentation can be found on this wiki. That page contains links to different ports on the TI 84+ CE, just note that you can only access (from ICE) the ones that are memory mapped. Here is a page with ports for the real-time clock in the 84+ CE. Near the top of the page it says **Memory-mapped address:** F30000, this means that you can read from these ports as if they were memory by using the F30000 address range. Below you find a list of specific ports. If you wanted to read the current hour count, that page lists it as port 8008. You simply modify the memory mapped address to get F30008. Before you can use this in a pointer, you need to convert it from hexadecimal. Luckily, ICE has a modifier just for that, the scientific `E`. With this, you should now be able to read the current hour and store it into, for example, `A`, like this:

```
*{EF30008}→A
```

Now you can use that hour however you wish to in your program.

10 Graphics

ICE gives you the ability to use most of the C libraries, in your programs, and one of these libraries is `GraphX`. `GraphX` gives you full control of the calculator's screen, allowing you to draw sprites, lines, rectangles, triangles, text, and more.
As you can likely guess, this library is incredibly useful for creating games or any program that includes a GUI.
This section will go over the specifics of each of the commands, and provide examples.

10.1 Starting & ending the graphics canvas

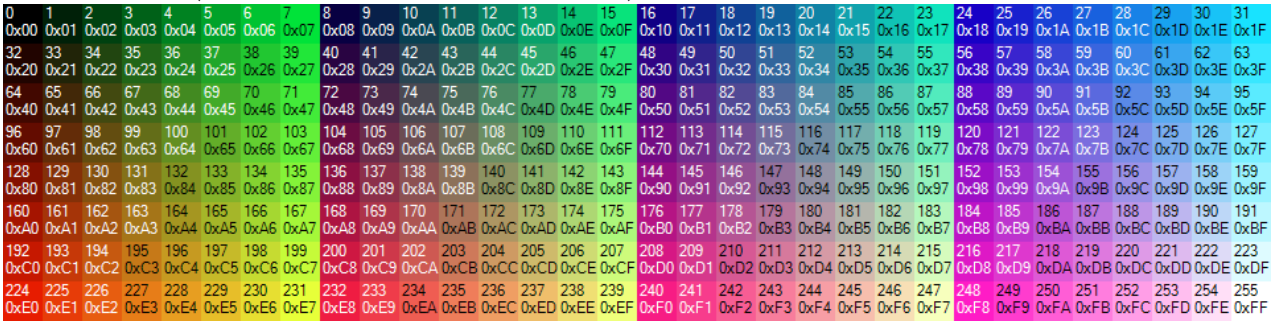
Before you can use any of the fancy commands that let you draw cookies, you need to setup the graphics canvas for use with `GraphX`. Luckily, that's incredibly easy to do. Near the beginning of you program, before **any** of the graphics functions are used, place a single `det(0` command.
When your program is executing and it hits `det(0`, the LCD is setup for use with the `GraphX` commands, and it clears the screen (sets every pixel to 0xFF (white by default)).
Because TI-OS uses a different LCD mode, you can't use the `GraphX` functions without having setup the graphics canvar with `det(0` at some point.

Before your program ends (or before it runs a TI-BASIC program that does anything graphically) you need to setup the graphics canvas for TI-OS, which is also simple. Just use `det(1` before your program ends or before you run the graphical TI-BASIC program.
If you don't end the GraphX canvas and setup for TI-OS before your program ends, TI-OS will look like this:

ICE will give you a warning if you forgot the `det(0` or `det(1`, which makes it even harder to forget. If this happens to you by accident, just turn your calculator off and back on, no need for a reset.

10.2 Color Palettes

Each color you'll use has it's own index number you use to reference the number, and `GraphX` uses an 8-bit color palette, meaning you have 256 unique colors (numbers 0 through 255). The default palette is the xLIBC palette, as shown below, with the default colors and their corresponding index number (in both decimal and hexadecimal):



You can find the full image here. You will use the index numbers in every command that asks for a color. You can set your own color pallete by using the `SetPalette(` or `det(4` command. For creating a custom palette, see section 14.1. If you wish to reset the palette back to it's default after setting your own, you can use `SetDefaultPalette(` or `det(3`.

10.3 Clipping

Most of the graphics commands in `GraphX` have two variants, the standard one and a duplicate with "_NoClip" tacked on to the end of the name. This is because the standard commands use clipping to check if what's being drawn is inside the current clip region. By default, the clip region is the entire screen (320x240 pixels). The `_NoClip` variants of commands are faster but can cause memory corruption or other strange problems if you accidentally tell one to draw something that outstreches the bounds of the screen.

You can set a custom clip region using `SetClipRegion(`. This sets a specific area to be the clipping area. Every clipped graphics command will only draw within this area, and ignoring anything outside. This is useful when you want to update a smaller part of the screen.

11 Sprites

To use sprites in ICE, you'll need to use a tool to convert from our source images into the ICE format we need. The tool is called ConvPNG, and can be downloaded from here. Make sure you download the correct version for your computer's operating system.

11.1 Creating Sprites

First of all, you have to create sprites. You can use your favorite image editor, like Adobe Photoshop, PaintDotNet, GIMP, or Microsoft Paint. The only restriction is that you need to save the image as a .PNG. Once you have created your sprites using your image editor of choice, put the image(s) (you can convert multiple images at the same time, or just one) in the same folder as ConvPNG. Then, create a new file with the following text in it:

```
#GroupICE              : ice_gfx
#Palette               : xlibc
#PNGImages             :
```

Below the last line, put all the names of the images you want to convert, without the .PNG extension. Example:

```
#GroupICE              : ice_gfx
#Palette               : xlibc
#PNGImages             :
 apple
 beer
```

In the above example, apple.png and beer.png will be converted to ICE format. Be sure to save this file as **convpng.ini**. Note the .ini extension! Double-click on the convpng file, and both the output and the palette will be written to ice_gfx.txt. If ice_gfx.txt does not appear in your directory, you may have done something wrong.
ConvPNG allows you to output the best palette too. You can do this by removing the #Palette : xlibc line. ConvPNG will then try to find the best palette, and writes that to ice_gfx.txt. This can be used in SetCustomPalette(.

11.2 Loading Sprites

If you open the output file from the previous subsection, ice_gfx.txt, you will see a width, height, and long string of hexadecimals for each sprite you converted. These are the arguments that you need to give DefineSprite(to use the sprites in your ICE program. For each sprite, place a DefineSprite(token in your ICE program and copy the corresponding line of text into the token (this is why we recommend you use SourceCoder3, it makes it significantly easier to get the sprite data into your ICE program).
In order to use the sprites you've put into your program, you need to save the pointer that

`DefineSprite(` returns, so store the result to a variable. Example:

```
DefineSprite(2,2,"00FFFF00"→A
```

This code creates a small 2 pixel wide, 2 pixel tall sprite, and stores the returned pointer into variable `A`. If we wanted to display this sprite to the screen at X-Y coordinates (20, 50), we would use:

```
det(57,A,20,50
```

`det(57` is the ICE command for `Sprite()`, which draws a clipped sprite to a given X,Y at the screen (or buffer). We can also define an 'empty' sprite, with random data in it, which can be used for creating a sprite from the screen (or buffer). To make an empty sprite, you simply define a sprite without data, like this:

```
DefineSprite(WIDTH,HEIGHT)
```

The example still returns a pointer to the empty sprite, which is required by commands that use sprites, like `GetSprite(` or `FlipSpriteC(`. Defining an empty sprite is useful for when you want to create a sprite from what's already on screen (or on buffer).

11.3 Drawing sprites

There are a couple commands available in ICE that are used to display sprites, but for these examples, we will stick to the standard command, `Sprite_NoClip(` (`det(59` in the program editor). The syntax for `Sprite_NoClip(` is:

```
det(59,PTR,X,Y)
```

`PTR` should be the pointer that was returned from `DefineSprite(` returns (see the previous section), in our case the variable `A`. `X` and `Y` are the coordinates where ICE will put the sprite.
Here is an example program to draw a mini apple sprite to the screen:

```
[i]OUTPUT
DefineSprite(4,4,"FF25E8FFE8E8E0E8E8E000E0FFE8E0FF")->A
det(0
det(59,A,20,50
Pause
det(1
```

Once you compile and run that program, you'll get a blank screen with a small apple near the top left corner. Hit enter to exit the program.
Notice how the example program uses `det(0` and `det(1`. This is because `det(0` is used to set up the graphics canvas to allow for the use of any of the graphics functions, and then `det(1` closes the graphics canvas and sets it up for TI-OS, which is what you want to do when your program ends, because it returns to TI-OS.

As you can see, drawing sprites is relatively easy and only requires a bit of effort when making and converting the sprites themselves.
You can have up to about 64K bytes worth of sprites in your ICE program, so it's likely you won't need to worry about using too many sprites. Just remember that they do take space in your program.
As for the other sprite commands available in ICE, they work quite similarly and are covered in more detail later. The similar command `Sprite` draws a *clipped* sprite to the screen, using the same syntax as `Sprite_NoClip`.
If you want to display a transparent sprite, add the following line to the `convpng.ini` file:

```
#TransparentColor :  <red>, <green>, <blue>, (<alpha>)
```

Where <red>, <green>, <blue>, and <alpha> are the RGBA values of a the color that will be treated as transparent. Then display the sprite with one of the transparent sprite commands, such as `TransparentSprite(` or `TransparentSprite_NoClip(` (they both use the same exact syntax as the `Sprite_NoClip(` command we just covered, but they are used for sprites with transparency instead).

12 Tilemaps

Games often use tilemaps, so it's important to cover them in this guide. Using tilemaps is a bit harder than sprites, but definitely not impossible. You need ConvPNG and ConvTile to convert to ICE format. Be sure to put them both in the same folder! To edit tilemaps, we will use Tiled.

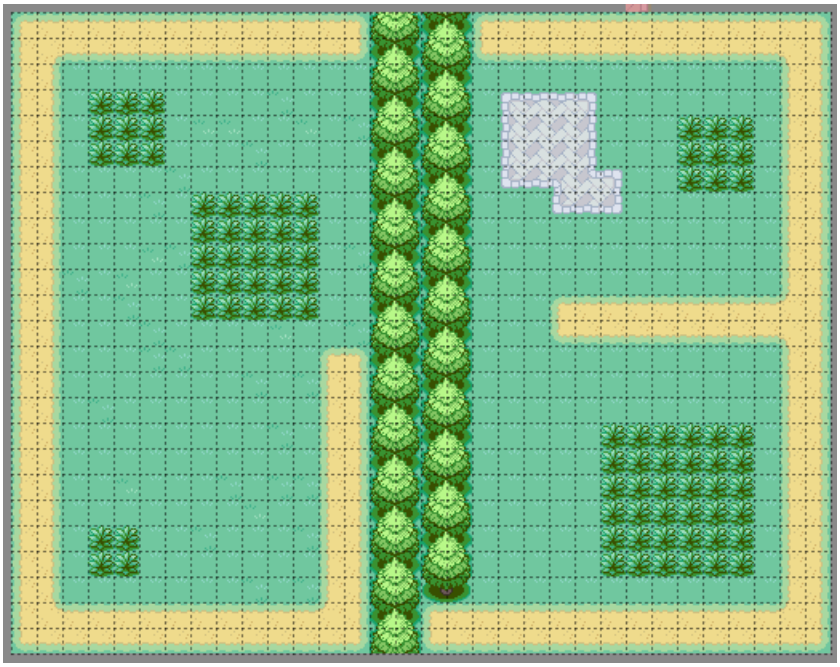
12.1 Create tilemaps

You can't use tilemaps without a tileset. This means we first need to make a tileset, consisting of tiles of equal size. Again, you can use your favorite sprite editor (tiles are basically sprites). Be sure to save the tileset as a .PNG in the same folder as the three tools! In this guide we will use this tileset:

With this tileset we are going to create the tilemap. That is why we need Tiled, a 2D level editor that will help us designing the tilemap. When you open Tiled, you need to create a tilemap, so click on the button New Map.

> ⚠ **Note: Tile layer format should be CSV!**

Change the map size and tile size to what you want, and save it in the folder where you stored the tools and tileset. But, we currently have only an empty tilemap, without any tiles. Click on File > New > New Tileset, browse for your tileset and save this tileset somewhere (you don't need it, unless you want to create more tilemaps with the same tileset). Go back to your tilemap, and start editing! Example:



When you are ready with editing, export this tilemap as a **.csv** file, and be sure to save it in the same folder again. Now we need to convert the `.csv` file to ICE format, and what is why we need `ConvTile`. Open Command Prompt, navigate to the folder, and run `convtile --ice`. The converted files will be in `tilemaps.txt`.

However, we still have one big problem. A tileset of 128x64 doesn't fit within one program! The only solution would be to store it in an appvar, and load the tileset from the appvar in ICE. That is why we need to use ConvPNG again. Create a new file with the following text in it:

```
#AppvarICE           : TILES
#PNGImages           :

#GroupICE            : tileset_gfx
#OutputPaletteArray  : false
#Tilemap             : 16,16,true
#Palette             : xlibc
#PNGImages           :
```

Put the name of the tileset under both the `#PNGImages :` lines, and save the file as `convpng.ini`, again in the same folder. Double-click on the `convpng.exe` file, and if everything went right, TILES.8xv will be created, and `tiles.txt`. You will see a line of text which looks like this:

```
"TILES",0,384
```

These are the arguments of `LoadData(`, which we will explain in the next section.

12.2 Load tilemaps

As said in the previous section, the tileset is stored in an appvar, so be sure to transfer `TILES.8xp` to your calculator. We need to 'load' the appvar data in our ICE program, and that is what the token `LoadData(` is for. Copy the data from the text file and put the token in front of it. This returns a pointer to the **table with pointers to the sprites**. If you have no idea what that means; no problem, you don't need to understand it. However, you need to store this pointer into a variable, which we will use in `DefineTilemap(`. The syntax for `DefineTilemap(` is this:

`DefineTilemap(TILE_HEIGHT,TILE_WIDTH,DRAW_HEIGHT,DRAW_WIDTH,`
`TYPE_WIDTH,TYPE_HEIGHTTILEMAP_HEIGHT,TILEMAP_WIDTH,Y,X,PTR[,"MAP_DATA"])`

The most important argument is the `PTR` one; that **must** be the variable holding the pointer from `LoadData(`. Something else doesn't work! The other arguments should be straightforward, and they should all be numbers. You can get the `"MAP_DATA"` from `tilemaps.txt`, which we already converted earlier. This function returns a pointer to the tilemap struct, which should be used in the tilemap functions.

12.3 Display tilemaps

Displaying tilemaps is as easy as display sprites! There are a few tilemap functions, and we will use one: `Tilemap_NoClip(`. The first argument should be the pointer returning from `DefineTilemap(`. The two other arguments are the coordinates where the tilemap should be displayed. If you don't understand this section, feel free to look at the code of the sample program, later in this guide.

12.4 Change tilemaps

If you have a game with multiple levels with the same tileset, you can easily change the tilemap struct such that it points to some different map data. To change it, just do this:

`"MAP_DATA"→{PTR}`

This loads a pointer to the new map data in the sprite struct. `PTR` should be the pointer returning from `DefineTilemap(`. Nothing changes of the tilemap commands, you can just use the same variable.

13 Code Snippets

13.1 randInt(

ICE doesn't support the TI-BASIC token `randInt(`. We need a small trick to simulate it. You need to notice that `randInt(1,6)` is basically the same as `1+remainder(rand,6-1+1)`. `remainder(rand,6)` returns a number in between 0 and 5, and adding 1 gives us a random number in the range `[1,6]`, which is what we needed. To make it more general:

```
randInt(A,B)=A+remainder(rand,B-A+1)
```

You'll need this code snippet routine a bunch of times in your program, so it might be useful to remember it.

13.2 Double buffering

The 84+CE is capable of having a screen buffer *and* another buffer, which is perfect for games. You can display one buffer, while drawing to the other buffer, which isn't visible. At the end of the loop, you can then just swap the LCD pointers with `SwapDraw(` to display the back buffer, and set the current visible buffer to the back buffer. It's very easy to use:

```
Begin
SetDraw(1)
While ...
...
SwapDraw
End
```

You need to tell your program to draw always at the back buffer with `SetDraw(1)`. Then your main loop starts. At the end of the main loop, you swap the pointers with `SwapDraw`. Then your program loops again, drawing to the back buffer, and displaying that buffer at the end of the loop. This has the main advantage that you don't see the screen objects being drawn. Instead, it displays all the objects at the same time!

13.3 Listing Variables

Sometimes you want to list all the available files on calc, and often that are programs, for example when you create a shell or IDE. This is pretty easy with ICE, but you need to know it, so let's see what the possibilities are. We already have `DetectVar(` and `Detect(`, so let's use them. The documentation says the first argument is `°POS` where it should be 0 if the want to find the first variable. Let's use this to create a small code snippet to find all the available programs, and open them!

```
0→POS
While DetectVar(°POS,0,5)→PROG
OpenVar(PROG,"r",5)→SLOT
...
End
```

You can use `DetectVar(` recursively, because `POS` gets automatically updated to find the next program.

⚠ **Note: The programs are found as how they are stored in the VAT, not alphabetically!**

14 Sample Game

We've made an example game for you: *The World Hardest Game*. You can grab it from SourceCoder3 here. In this documentation guide we don't explain how everything works, that takes too much space and time. The program itself has a lot of comments, and with these comments you should be able to understand it. If not, feel free to ask at Cemetech or send a message to PT_, see the next section.

15 Questions & contact

If you still have questions about anything ICE-related, feel free to post it in the right forum at Cemetech. You can also send a personal message to PT_ directly or send him an email: petertillema@solcon.nl. You can also find information and tips and tricks at ice.cemetech.net

---

Message from PT_: thanks everyone for helping, testing and giving feedback! I highly appreciate it, and I enjoy seeing new programs appearing in the ICE language. Credits to KryptonicDragon who helped me a lot finding bugs and writing this documentation. Also thanks to MateoConLechuga for helping with defining ICE syntax and for modifying his very useful tools to output ICE format. Cheers!
