Introduction to Saturn Assembly Language

Written by
Gilbert Fernandes

Edited and Maintained by
Eric Rechlin

Authors:

Gilbert Henri Fernandes	Eric Rechlin
1 rue des Gâte Ceps	3212 Winnipeg Drive
F-92210 Saint-Cloud	Bismarck, ND 58503-0453
France	USA
gilbert.fernandes@wanadoo.fr	eric@hpcalc.org
ICQ: 78327407	ICQ: 783944

Edition 3: July 16, 2005

Preface

Assembly language is very different from most other programming languages. When learning a typical programming language, one of the first lessons will usually explain how to make a simple "Hello World" program. But with assembly language, it's not that simple. First, before any assembly language code can be written, one must understand the concepts of how the processor works and how the operating system works with it. In addition, many commands must be learned in order to do something as simple as printing to the display.

In this book you will have to go through forty pages of introductory material before even picking up your calculator. But don't worry; you'll get to start programming soon enough! After being introduced to the concepts, assembly language commands, known as instructions, will be learned in small groups, beginning with an extremely simple program.

Because machine code is binary, you will learn assembly language mnemonics that will represent this binary code. This book is aimed at owners of *all* HP 48 and HP 49 series calculators. We use the MASD syntax, because MASD is built into the 49 series and comes with Meta Kernel for the 48GX. For those without MASD, a small assembler known as HP-ASM, which is compatible with the limited subset of the MASD syntax that we use here, can be used. The syntax of this assembler is quite easy and is also supported by HP Tools on the PC. The HP syntax for assembly language, supported by Jazz and HP Tools, is very similar, so one should be able to easily switch between the two, though we will not cover the differences here.

We think many more people could learn assembly language. Coders are needed, and the main roadblock is that information about assembly language coding, particularly Saturn assembly, is too hard to get. Most good Saturn assembly language tutorials are in French, and the only English-language one is James Donnelly's "An Introduction to HP 48 System RPL and Assembly Language Programming," and although it is a great book for System RPL, its assembly language section goes by too quickly for most people. With this book, we hope to fill a gap in the documentation that already exists for the HP 48 and 49.

Though we are not what one would call experienced coders, we would like to share our knowledge. We still consider ourselves newbies as well, so this is not a definitive introduction to assembly language. If you notice any mistakes, please don't flame us, but instead email us at the addresses below. Some, like Jean-Yves Avenard, Cyrille de Brébisson, and Gerald Squelart, three of the developers of the 49, have many years of coding experience, but don't be intimidated. Everyone is a newbie at some point in time.

This book goes from the very basics of how processors work, including tutorials covering the binary and hexadecimal bases, to coding complex routines. It is intended both as an introductory manual to beginning programmers who know nothing about the low-level operation of a computer, and also as a reference manual for experienced programmers.

We spent many, many hours on this book, but if you have problems, feel free to email us and we will help as we can and add to the tutorial if needed. This online version of the book is free, so distribute it widely to all HP calculator fans!

In addition, we would like to thank the following people for their contributions to this book (in alphabetical order): Arnaud Amiel, Jean-Yves Avenard, Cyrille de Brébisson, Christoph Giesselink, Chris Griffiths, Eduardo Kalinowski, Jan Kozicki, John H Meyers, Aldiney Oliveira, Mark Power, Gerald Squelart, and Richard Steventon.

The help of Christoph Giesselink and Eduardo Kalinowski is especially appreciated, as they wrote entire sections.

Updates to this book are available at <http://www.hpcalc.org>.

Please, please, all that we ask is for you to send feedback to our email addresses:

Gilbert Fernandes <gilbert.fernandes@wanadoo.fr>
Eric Rechlin <eric@hpcalc.org>

Table of Contents

Part I: Programming Concepts

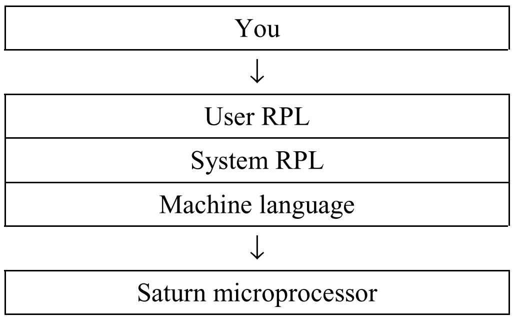
1 Introduction to User RPL and System RPL

The basic user of the HP 48 or 49 can use User RPL. It’s easy to learn, and it’s the first language you can use to program your HP.

Each time you use a User RPL command it checks the arguments, so it’s a rather safe language. You should not lose any data when using it.

If we look at the HP’s internal software, we’ll discover that each User RPL command is in fact divided into several simpler commands, each one of which does a single (usually) and simple thing. User RPL is divided internally into elementary commands, which we call System RPL. Each one of these System RPL commands ultimately ends up calling machine code, which is the only language the HP’s processor, the Saturn, can directly execute.

To help you understand, look at this:



The HP 48 gives you three development languages. User RPL is safe and easy but the slowest of all. System RPL is faster because no checks are made, so you must take care of all arguments at the beginning, but then you no longer need them as you feed data to hungry commands. Finally, we have assembly language, which produces machine language. This is the fastest code, directly executed by the processor, and of course it gives total control over the HP.

The HP 49 series has one more language at the top – HP Basic. This is essentially an algebraic form of User RPL, but it is even slower. The 49G+ and 48GII have another level at the bottom. This is because they do not have a physical Saturn processor, but instead they have an ARM processor, with an emulator running on top to emulate the Saturn. It is possible to bypass the Saturn layer and run ARM machine language directly on the ARM processor, but that is outside the scope of this tutorial.

If you want to learn System RPL, knowing User RPL will get you most of the way there. All you will need is a complete list of all System RPL commands, called entry points, inside the ROM.

Assembly language is not related to User RPL or System RPL, so you’re going to learn something really different here. Assembler doesn’t have all the commands which are available to the System RPL coder.

It is not a complex language, as its commands are simple and do very elementary things; what **may** seem complicated is that you will need to write a lot of “elementary commands” to do more complicated things than just crashing your HP. ;)

What you’ll win is the fastest code around, as well as lack of sleep, but that’s being a coder.

2 Assembly language and machine language

The Saturn processor, like any other processor, uses a series of 0's and 1's to work. We are not able, because of our inferior brains*, to understand these series.

* Mr. Avenard is good because he has a Saturn processor surgically implanted inside his brain. Another option is to eat a lot of Saturn processors, like Mika Heiskanen, the developer of Jazz and part of the CAS in the 49G, did, but it's quite difficult to eat just one, trust me! :o)

So, we will group these series of 0's and 1's together to create small packets of information. The Saturn processor has been designed to manage this binary information using mostly four-bit packets, each one called a nibble.

We will learn more about both the binary base and the hexadecimal base later, and hexadecimal is the one you will mainly use on computers. Don't worry. :)

So, machine language is a series of binary digits (0 or 1), or bits (the word resulting from the contraction of "binary" and "digit"), which are kept together to form packets of four bits, or nibbles.

Internally, the Saturn processor is only able to do very simple things, like reading some nibbles in memory, writing nibbles, and performing calculations with the nibbles read. Each elementary instruction is a kind of command to us. Assembly language coders use something called mnemonics. What are they?

"Mnemonic" is the name given to each elementary command. It's something very simple, so it reminds us what it's used for (mnemosis → memory, so "mnemonic").

We will write our programs using assembly language, which is composed of mnemonics. There is a mnemonic for every command the Saturn can use. A compiler will translate these mnemonics into binary information, and these bits will be used by the processor to obey our orders!

Remember that:

- Machine language is the *only* language the Saturn processor is able to understand. It's too complicated because it's a series a bits (0 and 1) so we don't use it directly.

- We will then translate those to mnemonics, which are simple words. One exists for each command available on the Saturn processor.

It is important to not mix "assembly language" and "machine language." The latter is the resulting form of the former after a compiler has translated it.

That is why assembly language is called a "low-level" language. In assembly language, each mnemonic corresponds to one instruction. That is why assembly language is **specific** to each processor. When we write assembly language programs, we first write mnemonics in a source file (source because it's the source of our work). Then we'll use a program called an "assembler" or "compiler" that will translate mnemonics to the machine language, a series of 0's and 1's (binary digits), which are kept together using four-bit packets on the HP.

3 The binary base and nibbles

We are going to do some math. Don't be afraid: even if you are bad at math you can do assembly language anyway. :)

As you learned, machine language is composed of series of zeroes and ones. That is because processors work internally with components that let electrons go through or not. In the computer world, 1 usually means a voltage potential exists and 0 usually means one does not exist.

So, the Saturn processor keeps the bits as four-bit packets, which are called nibbles. This is something you must *remember*. The traditional spelling is “nybble;” however, the spelling “nibble” has become more common in recent years so we will spell it with an ‘i’ for the purpose of this book.

Let’s start from something we know: the decimal base. We have ten digits so we can easily use the decimal base.

The decimal base is not the only one. Sailors use the sexagesimal base, base 60, because they use hours, minutes, and seconds.

When you type the decimal number 10 on your HP, it displays “10.” In fact, the HP internally translates it to bits, and the Saturn processor works with them, thus under the binary base. (I will teach you later that the Saturn processor is able to work with both our decimal base and its binary base, so it’s a very good processor isn’t it?!)

Decimal base means we use 10 symbols to write numbers:

0 1 2 3 4 5 6 7 8 9

Under the binary base, we only have **two** symbols:

0 1

So, zero in binary base is zero, one is one, and two?

Well, if we use the symbol “2” under the decimal base to represent the number two, we will use “10” under the binary base to represent the number two. In other words, we are going to combine zeroes and ones to represent numbers using the binary base.

The binary base is linked very closely to Boolean algebra, which will be briefly discussed later.

So, here is a small table of decimal numbers and their corresponding form in binary base:

Decimal base	Binary base
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
…	…

You must look at the way the 0’s and 1’s are combined to form numbers, and then be able to continue. We will discuss this in detail over the next few pages, as it is important to learn. It’s not something easy, so don’t feel bad if you have trouble understanding.

Now, we are going to keep bits as packets of four bits, because that’s the way the Saturn likes it.

Four packet bits are called nibbles, so a nibble is composed of four bits. Assembly language coders have some special words when they speak about bits and the different sizes used to group bits.

When four bits are packed together, it is called a “word of four bits.” If 32 bits are packed together, it becomes a “word of 32 bits.”

How many values can a packet of four bits have? It’s easy: we use a power of two to calculate:

$2^4 = 16$

So four bits packed together create a nibble, and:

A nibble can have 16 different values.

Let’s calculate another one: how many values can a word of 64 bits have?

$2^{64} = 18\ 446\ 744\ 073\ 709\ 551\ 616 \approx 1.8 * 10^{19}$ or about 18 billion billion

A lot. :-)

4 Converting from binary to decimal

We are going to examine two nibbles kept together, which is called a “byte.”

4 bits $\rightarrow$ a nibble
8 bits $\rightarrow$ a byte

We are going to give each bit a number, from right to left, and will call that the “rank” of a bit. Here is an example: I have two nibbles together (a byte) which are: 01100101, so we have:

rank:	7	6	5	4	3	2	1	0
bits:	0	1	1	0	0	1	0	1

As you see, I have started the “rank” by zero, and then I count from right to left. Each bit has a rank, a specific rank. Each bit has also something special.

The rank is used to know which “weight” the bit has. What does that mean? It means that each bit is, according to its rank, an exponent of base 2. Let me explain; look at this:

rank:	7	6	5	4	3	2	1	0
bits:	0	1	1	0	0	1	0	1
weight:	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

What I call weight is the value of the bit in the decimal base. Here is an example: what is the binary value 00100110 in decimal mode? Let’s write it:

rank:	7	6	5	4	3	2	1	0
bits:	0	0	1	0	0	1	1	0
weight:	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

Now, the “weight” of each bit can be calculated, and we’ll find the decimal value! Let’s do it:

We are going to multiply each bit by its weight, and we add each together, that is:

$(0 * 2^7) + (0 * 2^6) + (1 * 2^5) + (0 * 2^4) + (0 * 2^3) + (1 * 2^2) + (1 * 2^1) + (0 * 2^0)$

that is: $0 + 0 + 32 + 0 + 0 + 4 + 2 + 0 = 38$

Use your HP to check: press [MTH] [BASE] [BIN] and then type the number: #00100110b. Now, press [DEC] and you get:

```
#38d
```

It works! It's cool, isn't it? :-)

Now, you are able to convert a binary number to a decimal one. You write them down, and you give each one a rank, from right to left and from zero to the last one. Then, you write the "weight" of each bit, and you can calculate the decimal value.

What is the maximum value of a nibble? Two nibbles?

With two nibbles we have:

rank:	7	6	5	4	3	2	1	0
bits:	1	1	1	1	1	1	1	1
weight:	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

That is:

$(1 * 2^7) + (1 * 2^6) + (1 * 2^5) + (1 * 2^4) + (1 * 2^3) + (1 * 2^2) + (1 * 2^1) + (1 * 2^0)$

That is:

128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255

So a byte (two nibbles) can go from 0 to 255, that is 256 different values.

Remember!! 0 (zero) is a value too!

And with a nibble we have:

rank:	3	2	1	0
bits:	1	1	1	1
weight:	2^3	2^2	2^1	2^0

That is:

$(1 * 2^3) + (1 * 2^2) + (1 * 2^1) + (1 * 2^0)$

That is:

8 + 4 + 2 + 1 = 15

So with a nibble we can go from 0 to 15, that is 16 different values.

To get the decimal form of a binary integer, we multiply each bit's value by 2^{bitrank} and we add all the values.

5 Most significant bit and least significant bit

This is something important, so don't skip it.

As I have shown you, each bit has a weight. The bit that is the rightmost has a rank of zero, and the bit that is the leftmost has the highest rank (rank n-1 when we have n bits).

This means that each bit has a different weight in the final value.

The bit that is rightmost is called the Least Significant Bit, or LSB.
The bit that is leftmost is called the Most Significant Bit, or MSB.

In a nibble, the LSB and MSB are here:

rank:	3	2	1	0
	MSB			LSB

In a byte, the LSB and MSB are here:

rank:	7	6	5	4	3	2	1	0
	MSB							LSB

6 Fractional numbers

This is a little trickier.

A fractional number has two parts: the integer part before the fraction mark (a decimal point or comma, depending on region), and the fractional part after the fraction mark.

But…how can we have fractional numbers using bits?

We are going to give each bit a negative exponent weight.

Instead of:

rank:	3	2	1	0
bits:	1	1	1	1
weight:	2^3	2^2	2^1	2^0

we have for fractional numbers:

rank:	3	2	1	0
bits:	1	1	1	1
weight:	2^{-3}	2^{-2}	2^{-1}	2^{-0}

Please note: (2^{-3}) is in fact: (1 / 2^3)
(2^{-2}) is in fact: (1 / 2^2)

What is the value of 0.0110 in decimal?

rank:	3	2	1	0
bits:	0	1	1	0
weight:	2^{-3}	2^{-2}	2^{-1}	2^{-0}

That is:

$(0 * 2^{-3}) + (1 * 2^{-2}) + (1 * 2^{-1}) + (0 * 2^{-0})$

That is:

0 + 0.25 + 0.5 + 0 = 0.75

Notice, however, that going backwards is not always possible. For example, 0.6 is approximately 2^{-1} (0.5) + 2^{-4} (0.0625) + 2^{-5} (0.03125), or 0.59365, but not exactly. Most fractional numbers in the decimal base do not have exact equivalents in binary. More bits will give a better approximation, but you rarely can find an exact representation.

Lower-end modern HP calculators, such as the 30S, and TI calculators use this binary method of representing fractional numbers, though they use a large number of bits to make the approximations as accurate as reasonably possible. Higher-end HP calculators, and all the older models, use a different approach to keep numbers exact, BCD, which we will cover in chapter 12.

7 Calculating using base 2

In this section you'll learn how to add, subtract, multiply, and divide two binary values. This is going to help you understand how the Saturn works.

7.1 Adding

You will follow these rules:

0 + 0 = 0	→ no carry
0 + 1 = 1	→ no carry
1 + 0 = 1	→ no carry
1 + 1 = 0	→ carry

And here you thought you could take for granted the fact that 1 + 1 was 2!

Here is an example:

```
   0101
 + 1101
  -----
  10010
```

When I have 1 + 1, I put a zero, and I move the carry 1 to the left. I work here from right to left, adding two bits each time.

7.2 Subtracting

You will follow these rules:

0 - 0 = 0	
0 - 1 = -1	→ negative result
1 - 0 = 1	
1 - 1 = 0	

So here I need to explain to you how a negative number is represented using the binary base.

7.3 Negative numbers

There are various ways to do negative numbers. I am going to explain two that are important for you to know. You'll use either one or the other in your own programs: which one you use will depend on whether you want to let the processor do calculations for you or make it work the way *you* want, thereby ruling the beast!

7.3.1 The signed binary: the first method

We are going to use the most significant bit here (now you know why I told you about the MSB and the LSB!).

When the MSB is zero, the value is *positive*. When the MSB is one, the value is *negative*.

And we use all the other bits available to code the value.

This is called a "signed binary," and it's my favorite one (don't ask why).

If we have one nibble, a positive zero is: 0000
a negative zero is: 1000

(yes, there is a +0 and a -0 when using signed binaries!)

If you remember what I told you before, a nibble can go from 0 to 15, thus 16 possible values. Though there are still sixteen possible values, here we cannot use 0 to 15 because we use the MSB to tell if it's positive or negative. So a nibble goes:

Binary	Decimal
from 0000 to 0111 when it is positive	from positive zero to 7 when it is positive
from 1000 to 1111 when it is negative	from negative zero to -7 when it is negative

If we use a byte (two nibbles):

Binary	Decimal
from 00000000 to 01111111 when it is positive	from positive 0 to 127 when it is positive
from 10000000 to 11111111 when it is negative	from negative 0 to -127 when it is negative

Remember!! We have *two* zeroes here: a negative zero and a positive zero.

7.3.2 Two's complement: the second method

When all bits in a word (whatever its size) are inverted, we get an inverted word. We call that its "one's complement." The zeroes become ones, and the ones become zeroes. That is not difficult at all.

0101 becomes 1010
1100 becomes 0011
0010 becomes 1101

But the "two's complement" is the true complement for a word, and it is the one's complement plus one; that is:

(two's complement) = (one's complement) + 1

Example: what is the "two's complement" of 01101110 ?

First, we invert all bits, and get: 10010001

Add one and you'll get:

```
   10010001
+         1
   --------
   10010010
```

So the complement of 2 of 01101110 is 10010010

It is more complicated to handle "two's complement" binary numbers. Please pay attention; it is not complicated, but you must be awake to understand. :)

We will now learn the way the "two's complement" numbers are encoded. We will use a byte (8 bits) to learn it.

I have a start value of 0000 0000. If I add one, and again and again, it goes up…

But, what will happen if I have 0000 0000 and I remove 1?

I get:

```
0000 0000 - 1 = 1111 1111
```

As you see, when using "two's complement," the negative values start with 1 in their MSB. If the MSB is 0, the value is positive.

Interesting thing:

Positive values go from 0000 0000 to 0111 1111, or 0 to 127 in decimal.
Negative values go from 1000 0000 to 1111 1111, or -128 to -1 in decimal.

You see?

Note: here we're using 8 bits, but we could use fewer (like 4 with a nibble) or more (like 64…a lot of bits to play with).

We don't get a negative zero and a positive zero here!!

Sniff! Sniff! :'(

(Sniff! = sound of someone close to crying)

But there is something *much more important*!

We cannot do (1000 0000 - 1) !!

Why? We would go from a *negative* value to a *positive* value.

Also, we cannot do (0111 1111 + 1) because we would go from a *positive* value to a *negative* value, and adding +1 should not (unless we redefine mathematics) give a negative value from a positive value.

:-)

You must understand and know the differences between the signed binaries and the "two's complement." The first one helps one remember and understand the second one. That's what I do, and it's a good way to remember it, as the first one is very easy to remember, with its +0 and -0. :-)

Using the "two's complement" is a bit more complicated. Here is an example: how do I calculate 8 - 11 (which are decimal values) using 8 bits and "two's complement"?

8 is 0000 1000
11 is 0000 1011

To subtract 11, I'm going to add its negative value, so I have:

(8 - 11) becomes (8 + -11)

The two's complement of 0000 1011 requires inverting all bits and adding one. To show it in detail:

0000 1011 —inverting bits→ 1111 0100

Add one to get: 1111 0101

So I now add the two values:

```
      0000 1000
   +  1111 0101
      ---------
      1111 1101
```

As you see, the MSB here equals 1, so the result is known to be negative. But, I cannot use that final value directly. I have to calculate its two's complement, remember it's negative, and add a **-** sign in front of it.

The two's complement of 1111 1101 is: 0000 0011

That is finally: -3

REMEMBER!

When we use two's complement to do calculations, like the one I showed above, the final result *is not directly usable*. We have to:

1. calculate the "absolute value"
2. put the sign according to the MSB value

7.4 Multiplying

I don't like this one (because sometimes I lose myself!) but I have to. To do multiplication, we perform addition and shift bits.

When you want to multiply a value, the slowest of the slowest methods is to add the number to itself as long as needed to get the multiplied value, unless, of course, you are only multiplying by a very small amount, such as 3. This will eat up the processor's time ("a baaaaaaaaaad thing to do," my friend Mr. Avenard would say)

We can do multiplication using bit-shifting and adding values. Here is an example:

```
      0010
    x 0011
      ----
      0010 (that is 0010 * 1)
   + 0010  (that is 0010 * 1)
  + 0000   (well...)
 + 0000
  --------
  00000110
```

Well, it's not very beautiful, but it works. :-)

As you see, I have done four multiplications. As the two left bits are zero, I could simplify it like below:

```
      10
    * 11
    ----
      10
     10
    ----
     110
```

To multiply, I do:

1. I read the multiplier (the second number) from right to left.
2. When I find a bit set to 1, I copy the multiplicand (the first number).
3. Otherwise, if I find a zero, I insert zeroes. That's easy, isn't it? :)
4. Each time I look at a new bit on the second number, I shift the bits to the left (look above, the bits are written one place left at each line).
5. Finally, I add all values like we know to do.

REMEMBER:	All you have to do is look at the second number that multiplies, bit by bit from right to left, and do shifts; then, we add all values considering carry when there is one.

Although the Saturn processor is not able to do multiplication, it is able to do shifts quickly.

Shifting a value one bit to the left is like multiplying by 2. Shifting a value one bit to the right is like dividing by 2.

There is a way to optimize multiplication and division. As we have seen, each 0 found requires shifting one bit to the left. To quickly multiply a value, we find out how many zeroes are inside the number and do as many left bit shifts.

7.5 Dividing

Thanks to Eduardo Kalinowski for helping out with this section. He had originally written this using Brazilian-style long division, but I have rewritten it with American-style long division, which is hopefully more common.

OK. Here we go. First, I'd like to add that if you look at the multiplication you'll see that it is exactly like a multiplication in base 10, but much simpler, because you only have to remember four cases (0x0, 0x1, 1x0, 1x1) rather than the 100 of base 10 multiplication. Take a careful look at the process and note that you do the same thing: multiply the first number by the last digit of the second number. Then you multiply by the digit before that, and so on, and in the end you add everything. This is just like multiplication in decimal, but with only 1's and 0's. Well, division in binary is exactly the same as division in decimal, too.

Let's practice by dividing #10101b (21) by #111b (7). We should get 11 (3) with no remainder.

First, write the numbers like a normal division. I don't know how this is done in the rest of the world, but in the USA it is written like this:

```
    --------
111 | 10101
```

Then, we see how many times the divisor (111) "fits" in the first digit of the dividend. Since it can fit at most one time (base 2), we only need to check IF it fits. Well, 111 doesn't fit in "1," so we write a zero.

```
      0
    --------
111 | 10101
```

Now, I'm going to take two digits of the dividend, and see if it fits. No, it doesn't. Another zero is written:

```
      00
    --------
111 | 10101
```

Now I take the first three digits, and I see it doesn't fit again. So, another zero. I take now four digits (1010), and 111 does fit. So, I write 1 in the quotient and subtract 111 to see the partial remainder.

```
      0001
    --------
111 | 10101
      -111
      ----
        11
```

The partial remainder, 11, is obviously too small to subtract 111, so we move a digit down:

```
      0001
    --------
111 | 10101
      -111
      ----
        111
```

We now need to divide this lower number (111) by the original divisor (still 111), obtaining 1, which we move to the top, and a remainder of zero:

```
      00011
    --------
111 | 10101
      -111
      ----
        111
       -111
        ---
        000
```

Since there are no more digits to move down, we've reached the end!!! The quotient, at the top, is (ignoring leading zeroes) 11, or 3 in decimal, as we expected, and the remainder, at the bottom, is 0. Notice that we have also learned how the modulo (remainder) function works!

As you have seen (if you haven't read it yet, *do it*), it is just like regular division, but all operations are done in binary base.

8 Hexadecimal base

Unlike the decimal base, which uses base 10, and binary, which uses base 2, the hexadecimal base uses base 16. Why are we going to use base 16? As we've seen, the Saturn prefers to keep the bits in packets of four bits, which we call a nibble.

Earlier, we learned that a nibble can have up to sixteen different values. Here they are in their binary form:

Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4

Decimal	Binary	Hexadecimal
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

When we use the decimal base, we have ten symbols and combine them to represent numbers. We have also studied the binary base, which combines zeroes and ones to form numbers.

As the hexadecimal base has 16 number symbols, we have to find 16 symbols to represent them; we use the first ten decimal numbers from 0 to 9, and then we use the first 6 letters of the alphabet: A, B, C, D, E, F

And we are going to combine those digits and letters to represent numbers. What is cool here is that four bits can be represented by a *single* symbol in the hexadecimal mode. :-)

This base is useful in *all* modern computers. The eight symbol octal base can be useful for working with some older computers, but hexadecimal is all that's needed for any recent computer system.

9 Converting from hexadecimal to decimal

When we have a number in its hexadecimal form, we can give its numbers "ranks" like we did for the binary numbers. The exponents are just different (and produce bigger numbers).

Let's look at five nibbles under their hexadecimal form. Why five? Because the Saturn processor uses five nibbles to give each nibble in memory a position, five nibbles are required to have a value that can be used as an address, to get information from memory, or to write memory there.

My hex number is #ABCDE and I want to get its decimal value. I would write:

rank:	4	3	2	1	0
hex:	A	B	C	D	E
weight:	16^4	16^3	16^2	16^1	16^0

It's quite easy: just like when we were using the binary base, we are going to give each nibble a weight.

The "weight" of a nibble of rank n in hexadecimal is: 16^n

Example: what is the hexadecimal number #1AB0F in decimal?

We have here:

rank:	4	3	2	1	0
hex:	1	A	B	0	F
weight:	16^4	16^3	16^2	16^1	16^0

Now, all we have to do is multiply the nibble by 16^{rank}. Here we go:

Value = $(1 * 16^4) + (A * 16^3) + (B * 16^2) + (0 * 16^0) + (F * 16^0)$

Don't be confused because we have letters here rather than only numbers: because we are using the hexadecimal base, we will often find letters in numbers.

If you prefer, (B * 16^2) is in fact: (11 * 16^2). You can always refer back to the preceding table if you forget these equivalents.

So we calculate values, and get:

```
(1 * 65536) + (10 * 4096) + (11 * 256) + 0 + 15 = 109 327
```

So #1AB0Fh = 109 327

You can check that using your HP. Type the number 109327 and then press [MTH][BASE][R→B]

R→B is a command that will turn a real to a binary integer. You have to make sure you are using hexadecimal notation; there should be a square dot in the [HEX] menu field. If not, press [HEX] so your binary integers are shown as hexadecimal values.

Try using R→B to turn reals to hexadecimal numbers, and then you can use B→R to turn the binary integer numbers back into reals.

10 Writing groups of four bits (nibbles) using hexadecimal

So, how do we write our nibbles using hexadecimal digits? We know a nibble, with four bits, can have 16 different values, and we have 16 symbols with the hex base, so we just to have to convert the four bits to the hex form.

What if there are more than four bits? Example: I have a 12-bit binary number, and I want to convert it to hex.

The trick is to separate bits into groups of four, from right to left.

For example, let's use the number 0111 1011 0101.

To avoid a complex (and boring as math is) calculation, we won't work with all 12 bits at once, but will instead cut them into packets of four bits. Now, we are going to convert each 4-bit packet to its hexadecimal value:

0111 → 7
1011 → B
0101 → 5

(I gave you a table earlier, but as there aren't many symbols, you can write down numbers from 0 to 15, then their binary form, and then their hex value)

So I know the number:

0111 1011 0101 (binary) is #7B5h

This way we are able to easily convert very big binary numbers.

But...what if we don't have a multiple of four bits? No problem: we add enough zeroes to the left side of the number to get a multiple of four bits.

Example: I want to convert the number 011101 from binary to hexadecimal.

I'm going to separate it into groups of four bits, but obviously, there are not enough numbers:

01 1101

I will add two zeroes to the left-most group in order to have a multiple of four bits in my number:

01 1101 becomes 0001 1101

Because they are leading zeroes, the value has not changed. :-)

Now I can convert each four-bit group into its hex form:

0001 → 1
1101 → D

So we have 011101 (binary) → #1Dh

11 Is all that number crunching needed?

Of course, we have an HP, so we could use the available functions to convert from binary to hex in any way we want. But it's important to know *how* bits are encoded and used to represent numbers.

When your knowledge and programming skills increase, you will find some logical functions (described in the part after the next part) *very* useful for checking some values, using what we will call "masks."

A "mask" is a set of bits, which we will then use to check values of one or more bits.

I will tell you how a mask can be useful when we start using the HP's keyboard and reading keys from it.

When I first started learning, I didn't like bits and conversions. You may not care for learning them either, because after all, your HP can do that for you, right?

Wrong! You'll just end up more confused than you would normally have been. You're best off learning how to do it in your head, or you'll get it all wrong.

Go back and reread the previous sections if you are at all confused. Even if you are bad at math you'll be able to understand this.

12 BCD: "Binary Coded Decimal"

You may not like learning this either. But it's best to know it, especially if you want to pop reals from the stack.

The Saturn processor works with 4-bit packets called nibbles. We have learned that it stores them using a very cool format called hexadecimal. So when we want the processor to add two values, it will consider those as hexadecimal values: if we ask it to add 9 and 9, the Saturn won't give us a result of "18" but rather "12" because 18 = #12h.

But we humans (or whatever you may be ;) prefer the decimal base, and the Saturn processor has been designed so it can *also* work with decimal numbers.

I told you it's a cool processor, and quite powerful, even if it's small and so hard to swallow (gulp! I ate my second one today).

When the Saturn is asked to do calculations using base 10 numbers, it uses a special mode called BCD.

BCD means Binary Coded Decimal

A nibble, which has four bits, can have 16 values. But when we use the BCD mode, we keep our numbers inside our decimal-base 10 form.

Pay attention to the way that BCD codes numbers in the following table. Remain calm, and you'll see it's not so difficult, okay? :-)

Decimal	Nibble (four bits)	BCD
0	0000	0000
1	0001	0001
2	0010	0010
3	0011	0011
4	0100	0100
5	0101	0101
6	0110	0110
7	0111	0111
8	1000	1000
9	1001	1001
10	1010	0001 0000
11	1011	0001 0001
12	1100	0001 0010
13	1101	0001 0011
14	1110	0001 0100
15	1111	0001 0101

As you can see here, there is no change between the nibble/bits value and its value with BCD when under ten.

But, when we reach the value of *ten*, the BCD can no longer increase with a single set of four bits, so we must add another set of four bits. We will start over using the new set of four bits, just like we did on the previous set. Notice how we go from 9 to 10 and how we continue up to 15.

When using BCD, each decimal digit uses four bits. We use four bits to count from zero to 9, but "10" is written with two digits, so we have to use eight bits to code the value using BCD.

See, it's not so complicated. It may look strange, but this is not the point we have to pay attention to. :o)

13 Converting a decimal to its BCD form

When you want to convert a decimal number to its BCD form, all you have to do is read the number from left to right and write down *each* number in its binary form:

If I have the number 934, what is its BCD form?

9 → 1001
3 → 0011
4 → 0100

So 934 in BCD form is 1001 0011 0100

See? It's not difficult at all. The BCD looks complex but isn't really.

You must be cautious: the BCD binary form is not the same as the binary form.

The number I used for this example below is 934. Here you have its BCD binary form and its binary form:

BCD:	1001 0011 0100
binary:	0011 1010 0110

Don't mix the two together, or you'll be hopelessly confused.

14 Normal BCD and "Extended BCD"

The form of BCD explained earlier is the usual one, called the "Compacted BCD." There is another form of BCD called the "Extended BCD."

In the Extended BCD, instead of using four bits per digit, eight bits are used. For example, the number 11 would be coded as follows:

Compacted BCD:	0001 0001
Extended BCD:	00000001 00000001

When using Compacted BCD, values can go from 0 to 99 by using one byte (two nibbles) whereas the Extended BCD can only go from 0 to 9!

So the BCD used by the Saturn processor is not only cool, but clever. :-)

(now, look at your HP, and tell it you love it...)

15 Calculating using BCD

Storing numbers is cool, but working with them is even cooler! However, because of the way BCD codes numbers, some nibbles are not allowed.

The Saturn will help us a lot, because not only can it store BCD numbers, but it can also perform calculations with them. All we have to do is tell the processor to be in BCD (decimal) mode instead of hexadecimal and then use it.

There are *two* important things when using BCD: when a nibble gets an overflow, and when a carry appears and has to be moved.

15.1 When a nibble overflows

What will happen if I add 16 to itself using BCD?

```
  0001 0110        (16 under BCD form)
+ 0001 0110
  ---------
  0010 1100
```

Problem: we cannot have 1100, as it is not allowed when using BCD.

15.2 When a carry needs to be moved

When we add two 1 bits, we have a carry, and we must move one bit to the left and write a zero. For example, 0001 + 0001 = 0010

But, when using BCD, the carry cannot move to the left as one would like.

Most, though not all, math instructions are BCD-aware and can properly handle these problems.

16 Logical operations

This is an important part. We will meet operations like OR, AND, and others, like XOR. (Julien Meyer, or SunHP, wrote a prayer to the XOR function, as it is useful in games and graphics, and he prays to it each night. ;)

At the lowest level, processors are designed using only logical operations. I will only discuss only simple ones here, but by combining many logical operations together, processors are able to do really complex tasks.

Just think about your COS key. When you press it, the RPL operating system checks if there is an argument, then whether it's a valid one, and finally calls one or more machine language operators, which result in the processor moving bits from places to places, with logical operators working over them. Fascinating! :-)

George Boole (1815-1864) published a book in 1847 called "Analyse mathematique de la logique," or something like "Logic Analysis of Mathematics." The most important part of his work is about *logic*. Some of our words are translated into algebraic components, in order to create rules for the thinking processes of calculations. We call this Boolean algebra.

In Boolean algebra, we have two values: 0 and 1. :-)

0 is said to be "false" (the value "false")
1 is said to be "true"

Because we're speaking of logic, we have to understand the concepts of false or true inside the Boolean algebra. Each value gets one purpose: 0 is used for false, and 1 is used for true.

Electronics extensively use this concept of our beloved Boole. Get a drawing of him, along with Newton and Einstein, and put them on your wall! If you don't, you'll never be a good coder. ;)

So, we're going to use the bits 0 and 1

Zero means "false," and in electronics, it's generally found when there is no voltage.

One means "true," and it's generally found when there is voltage in the circuit.

Because we use bits here, we have two values that we are able to compute:

0 is the "complement" of 1
1 is the "complement" of 0

Because the complement is inverting the bits (I'm speaking of the "complement of 1" also called "restricted complement". It's not the "complement of 2" we used to store negative values; take a look at the first part if you are confused).

We have three basic logic operations. From *two* of those we are able to create *all* the other ones that exist (or maybe I should say "those we use and have discovered").

French name	English name	Symbol used
ET	AND	·
OU	OR	+
NON	NOT	-

When a value has been negated using NOT, a line is drawn over its name, like:

```
NOT A = $\overline{A}$
```

And that value is called “A-barre” in French. It’s called “A-bar” in English. It is sometimes written A' or ~A, but we will always use A-bar in this document.

16.1 Logical OR

It can also be called “logical adding.”

I am going to show you what is called a “truth table.” Each logical operation has one. On the left part we put letters, like A, B, C and so on, for each bit in input. On the right part, we have one S (the result) or several results, when there are several outputs.

The OR is done between *two* bits, first one is A, second one is B, and in S you have the result of the OR between the two bits.

OR Truth Table:		
A	B	S
0	0	0
0	1	1
1	0	1
1	1	1

The logical OR can also be written using the expression:

```
S = A+B
```

Note: the logical OR is *not* the addition we saw when we studied binary numbers.

```
0101 + 0011  =  1000

0101 OR 0011 =  0111
```

As you now see, OR results in 1 either if A or B are 1 or if both of them are.

16.2 Logical AND

It is also called the “logical product,” but it is NOT multiplication.

The expression is written:

```
S = A·B
```

Here is the truth table for AND:

AND Truth Table:		
A	B	S
0	0	0
0	1	0
1	0	0
1	1	1

Here the result is 1 only if both A and B are 1.

16.3 Inverting

We just invert the bit. The expression is written:

$$S = \overline{A}$$

Notice that there is a bar above the letter.

INVERT Truth Table:	
A	S
0	1
1	0

16.4 NOT OR (also called: NOR)

This is using NOT and OR together. When doing a NOR, output is set to 1 only if all inputs are set to zero.

The expression for NOR is:

$$S = \overline{A+B}$$

NOR Truth Table:		
A	B	S
0	0	1
0	1	0
1	0	0
1	1	0

16.5 AND NOT (also called: NAND)

This uses AND and NOT together. When using NAND, it's the inverse of NOR. In other words, the output is always 1 unless both entries are set to 1.

The expression for NAND is:

$$S = \overline{A \cdot B}$$

NAND Truth Table:		
A	B	S
0	0	1
0	1	1
1	0	1
1	1	0

16.6 Exclusive OR (also called: XOR)

This one is extensively used when dealing with graphics. The result is 1 if either A or B is one, but *only* if one of A or B is one. If A and B are *both* set to 1, we get zero.

The expression is:

```
A XOR B = (A OR B) AND (Ā OR B̄)
```

XOR Truth Table:		
A	B	S
0	0	0
0	1	1
1	0	1
1	1	0

It is cool because a graphic object is, when in memory, a set of bits. When we want to draw a graphic, we can XOR its bits to the actual graphic on screen. Then, if we do another XOR at the same position, it makes the graphic object disappear! :-)

We'll see that later, so be patient. :-)

17 Review: Numerical bases

Some people said they either didn't read the part on bases or read it too quickly. I understand, because I did the same when I started learning myself. But, a minimum amount of knowledge is required. To be able to recognize hexadecimal and binary forms is mandatory. When I tell you about masks and how to compose them, both binary and hexadecimal information will be required, because we'll write down bits to create our required mask, convert to hexadecimal, and then load it into C or A for using it to test another register's value.

You don't need to know how to convert values from binary to hexadecimal, and such conversion things, but knowing each bit's weight seems of importance to me.

As a quick review, there are four bits in a nibble and each has a weight. No matter how many bits there are, they are numbered from right to left.

Part II: Programming Tools

Before you can start writing code, we must discuss the tools that you'll need for assembly language development. In the beginning, the first coders had to write their assembly programs on paper, then they translated the assembly instructions to their hexadecimal form, and from there they created Code object strings.

Today we have better tools. You write your program in a string, and a compiler called an "assembler" will do the job of checking your code (after all, everyone makes mistakes sometimes) and producing the Code object.

18 Using the HP 48

The HP 48 series requires additional software for on-calculator assembly language development. You will need to install an assembler, and it is recommended that you use a replacement text editor for performance reasons. The tools listed in this section are valid for the HP 48S, 48SX, 48G, 48G+, and 48GX.

18.1 An assembler

We have tried several assemblers. Jazz is interesting and extremely powerful, but it is too big for most users. Jazz also uses a different syntax, making it less useful for a beginning trying to follow the examples in this book, which use the syntax of the more accessible assemblers. We want people using a G to be able to code, and this introduction is aimed at everyone, including those using S and G models.

Here we will use the HP-ASM 1.1 syntax. This tool is the G/GX adaptation of ASM Flash from Phong Nguyen, who, according to Jean-Yves, was 8 years old at that time of writing it. For the purpose of this document, HP-ASM is a synonym for ASM Flash.

HP-ASM is really fast. Jazz's ASS command is several times slower, and what takes half a second in HP-ASM may take several using ASS. It's not very crucial, but impressive.

You may get HP-ASM, for the HP 48G, 48G+, and 48GX, from this location:

http://www.hpcalc.org/details.php?id=1666
→ http://www.hpcalc.org/hp48/programming/asm/hpasmgx.zip

ASM Flash, for the HP 48S and 48SX, is available here:

http://www.hpcalc.org/details.php?id=1659
→ http://www.hpcalc.org/hp48/programming/asm/asmflash.zip

The hpasmgx.zip file is a ZIP file, and inside it there are two versions of HP-ASM for the G series. If you have an S or SX, you should get ASM Flash instead.

HP-ASM is available in both versions 1.0 and 1.1. Version 1.0 is 8606 bytes and works quite well. The other version, version 1.1, is 9124 bytes long. Version 1.1 has some new instructions and perhaps other things.

This library contains an assembler that will create code objects from your sources. In this tutorial version 1.1 commands are described, but 1.0 commands are very similar (→ASM instead of ASM, for example). Because the difference in size between 1.0 and 1.1 is only about 600 bytes, you should get 1.1.

If you have a GX with lots of free uncovered RAM, you can choose Jazz, which has many additional features, such as a disassembler, a debugger, and System RPL support. I won't use its syntax here, as it uses a slightly different syntax. People using G models or S models don't have enough space to use it. Because this introduction is not just for GX people but rather for everyone, I'll use HP-ASM syntax.

HP-ASM syntax is compatible with the Meta Kernel's MASD assembler, so this tutorial is valid for the Meta Kernel as well. In addition, MASD has a few *really* cool features for programmers. If you own the Meta Kernel, read its manual for more information. Meta Kernel users might also benefit from reading the 49 section below, because the 49 includes the Meta Kernel and therefore the documentation for the 49 is largely valid.

18.2 HP-ASM commands

Here are the HP-ASM commands in its library menu:

ASM: Takes a string as an argument and returns an assembled object (either a code object or a string with hex codes)

ED: Not an editor, but if you make a mistake, by pressing ED, the source string is edited, and you are moved to the beginning of the line where an error was found. When you use ED, it calls the standard editor, the one given by the HP. ED can be modified so it calls the editor you want if you know a little System RPL, but that's beyond the scope of this book.

OPT: Allows options to be selected. When a menu key is pressed, the menu key contents toggle from one option to another.

CODE/HEXA:	If CODE the assembler produces a Code object; if HEXA it pushes a string that contains the hexadecimal values corresponding to the generated code.
0-15/1-16:	When using 0-15, nibbles are numbered from 0 to 15, otherwise from 1-16. This will affect the use of the P register, like `P= n`, `CPEX`, `C=P`, `P=C`, and the bit mnemonics, like `CBIT`, `ABIT`, and tests involving `P`, and `HS`. With assembly language, we start counting at 0, so 0-15 is strongly preferred, meaning the first nibble of a register is number #0h. Advice: keep it set to 0-15 all the time.
HP/PC:	This affects which operator symbols are used. The HP has extended ASCII characters for comparisons that are not compatible with a PC's extended ASCII. Choosing HP uses these HP characters, and choosing PC uses PC-compatible two-character standard ASCII symbols: #, >= and <=. For example, the "not equal to" sign is "=" with a slash through it when HP is set and the pound sign (#) when PC is set.
JMP/UNJ:	When JMP is chosen, the assembler checks all jumps and calculates them. If you select UNJ, then the jumps to labels are not calculated, and thus not verified. It's a good idea to leave this checked. Only select UNJ it if you are compiling a small piece of code extracted from a larger project with the intent of calculating the bytes it takes without having the assembler error out with a bad jump.
ON/OFF:	When compiling a very big program, you can turn the display off and the compilation will run about 13% faster. When OFF is selected, the screen will turn off while assembling; ON keeps it on. It's a good idea to keep the screen off when assembling large sources.
EXIT:	Exits the OPT menu.

18.3 Assembling code with HP-ASM

The Code objects produced by HP-ASM (or any other assembler, like Jazz, J-ASM, ASM Flash, and MASD) are fast because the HP's Saturn processor directly executes them.

We don't code using bits because that would be too difficult. Assembly language uses mnemonics, with one mnemonic for each instruction available on the processor. An assembler like HP-ASM translates these mnemonics into machine language, which is what is found in hexadecimal form inside the Code objects.

The HP 48's RPL cannot control what happens when we use Code objects, so we must code well, or otherwise drain our batteries recovering programs we lost (or for some of us, being happy to have one or more RAM cards).

HP-ASM syntax is not HP's syntax, but rather the one used by French programmers for some time. It's like other tools, such as DEV and TC, so it's a part of history. :-)

You will put your code inside a string. Because the internal HP editor is not very fast (and misuses memory like I have never seen) it is recommended that you use an external editor. MiniWriter will work very well on G models.

There must be one instruction per line inside the string that you give to the assembler. You can add comments using the percent symbol (%). Everything that follows a `%` until the carriage return is then considered a comment.

The last symbol in the text file should be the "at" sign (@); the assembler will then know this is the end of the source. This symbol is produced by [ALPHA][Right-Shift][ENTER].

The assembler is case-sensitive, which means that "Label" is different than "LABEL". When you name your labels, either write them all uppercase or all lowercase. Choose a style of writing, and stick to it to avoid confusion.

To go to the next line, do [Right-Shift][.]. (The state of the Alpha mode doesn't matter, as it's the same either way.)

18.4 Additional HP-ASM features

This is a summarized version of the English documentation included with ASM Flash.

18.4.1 The linker

This is something really cool. If you divide your source code inside several parts, you can ask the compiler to compile each part and then produce the final Code object. If you have two sources, ONE and TWO, you just put inside of your program:

```
'ONE
'TWO
@
```

The ' symbol is used to place the contents of a file in the HOME directory (or the current directory perhaps?) into your source.

If an error is found, it tells where.

18.4.2 Directives

A directive is the '!' symbol followed by an OPT value; if you want your code to be compiled with the HP option ON, you just put inside of it:

```
!HP
```

By simply inserting a '!' followed by a name from the OPT menu, you can temporarily turn on or off any of the options from your source code string. It can be done anywhere inside of your source: some parts can use PC comparison operators, and other parts HP ones, if you want.

18.4.3 Macros

Those are not really macros like coders are used to, but they're cool anyway. You can compile code and reuse it already compiled, or insert a GROB or any kind of data inside of one of your programs. We will need to write a program to create macro-able objects, and it should be useful when we start the graphics part. See below for that program.

18.5 A fast text editor

The editor built into the HP 48 series is not very good at editing big objects, so you should use a replacement editor to edit your ASM sources.

If you have a 48GX with the Meta Kernel, you already have a good text editor similar to StringWriter built-into the Meta Kernel environment. Otherwise, you will need to install a text editor separately.

StringWriter (24 KB) is good but is too big for a G, so you may want to use a smaller one, like MiniWriter (5 KB). Because it is smaller, MiniWriter will easily fit on both G and S models.

If you have a GX with plenty of RAM, StringWriter has many more features, but the font is bigger. Those two editors have a cool option: by pressing [MTH] it will give you a list of all lines that start with a *, and HP-ASM considers those lines as labels. To be able to jump to any label of a source is very, very…mandatory :-)

In addition, TED (12 KB) has many of the same features as StringWriter but it uses the small font.

StringWriter can be downloaded here:

http://www.hpcalc.org/details.php?id=129
→ http://www.hpcalc.org/hp48/apps/editors/strwrt44.zip

MiniWriter can be downloaded here:

http://www.hpcalc.org/details.php?id=114
→ http://www.hpcalc.org/hp48/apps/editors/minwrt12.zip

TED can be downloaded here:

http://www.hpcalc.org/details.php?id=131
→ http://www.hpcalc.org/hp48/apps/editors/ted32.zip

Note: Both MiniWriter and TED use the small font, and they need the UFL. This is a library that contains fonts so each program doesn't have to have its own inside, wasting memory space. Neither MiniWriter nor TED comes with the UFL so it will have to be downloaded separately from:

http://www.hpcalc.org/details.php?id=162
→ http://www.hpcalc.org/hp48/utils/fonts/ufl102.zip

You'll need to download both MiniWriter (or TED) *and* the UFL library to your HP, or the editor won't work. In the case of MiniWriter, you don't need the library called "strwadp.lib." Send the following files to your HP:

ufl1.lib (found inside the "ufl102.zip" archive)
AND EITHER
minwrt12 (found inside the "minwrt12.zip" archive)
OR
ted.lib (found inside the "ted32.zip" archive)

I personally prefer StringWriter because the bigger font is easier to read. If you have a G or an S, where it is important to have the least memory used, use MiniWriter.

19 Using the HP 49

The HP 49 series has everything you need to get started built-in. The 49G and 49G+ include the MASD assembler from the Meta Kernel, which is compatible with the HP-ASM syntax used in this book. They also have a fast text editor, also from the Meta Kernel, which is one of the best text editors for the HP 48 series.

The 48GII, though a part of the 49 series, does not have the MASD assembler, nor does it have any third-party assembler software for it; at the present time it is therefore not recommended for assembly language programming. It does, however, have the disassembler and some other useful development tools, so it may perhaps be useful to developers if somebody ports an assembler to it.

19.1 The assembler

The 49 series has a hidden development library, library 256. This library does not appear in the library menu, and it is not attached by default, so until it is attached you will not be able to run the commands by typing their names.

To enter the development library, simply type:

```
256 MENU
```

You can then press NXT a few times to see all the available commands. To attach the development library, so you can run its commands by simply typing their names, type:

```
256 ATTACH
```

If you set system flag -86, the development library will automatically attach itself after every warmstart. You can set system flag -86 with the following command:

```
-86 SF
```

You can run the MASD assembler with the ASM command. If you get an error when assembling, a list of error codes will be placed on the stack. You can use the ER command to display the errors and jump to the relevant sections of the code that caused the errors. If you attach library 257, you can also use the ASM2 command, which calls ASM, and if there is an error, ER is automatically called. It is essentially identical to the User RPL program `<< IFERR ASM THEN ER END >>`. There are many other useful tools in the development library, most of which are explained in the next section.

19.2 Development library commands.

In additional to an assembler, the 49 series has a large number of other commands that could be useful to programmers. These are in the same library 256 that contains the assembler.

The development library has gradually gained more features as the HP 49's ROM has evolved, so it is recommended that you upgrade your calculator to the latest ROM to ensure you have complete functionality.

Here are the commands in the development library of ROM 2.00:

→H	Converts an object on stack level 1 to its hexadecimal representation.
H→	Converts a hexadecimal string on stack level 1 to the object that it represents.
→A	Returns the memory address of the object on stack level 1.
A→	Returns the object pointed to by the memory address on stack level 1. Warning: dangerous!
A→H	Converts a memory address on stack level 1 to its hexadecimal string equivalent.
H→A	Uses the first 5 digits of a hexadecimal string on stack level 1 to generate a memory address.
→CD	Converts a hexadecimal string on stack level 1 to a code object.
CD→	Converts a code object on stack level 1 to its hexadecimal representation.

S →H	Converts an ASCII string on stack level 1 to its hexadecimal string equivalent.
H→S	Converts a hexadecimal string on stack level 1 to its ASCII string equivalent.
→LST	Converts a composite object on stack level 1 to a list of its components.
→ALG	Converts a composite object on stack level 1 to an algebraic object.
→PRG	Converts a composite object on stack level 1 to an executable program object.
COMP →	Converts a composite object on stack level 1 to an algebraic object.
→RAM	Creates a new copy of the object on stack level 1 in memory, even when it is a ROM object. Useful for disassembling built-in ROM commands.
SREV	Reverses the string on stack level 1.
POKE	Pokes raw hexadecimal values into memory, using the starting address on stack level 2 and the hexadecimal string on stack level 1. **Warning: dangerous!**
PEEK	Reads raw hexadecimal values from memory, using the starting address on stack level 2 and the number of nibbles to read on stack level 1, both as user binary integers.
APEEK	Reads the address contained at the address on stack level 1.
R~SB	Converts a real number on stack level 1 to a system binary integer and vice versa. Also accepts integers.
SB~B	Converts a system binary integer on stack level 1 to a user binary integer and vice versa.
LR~R	Converts a long real number on stack level 1 to a real number and vice versa. Also accepts integers.
S~N	Converts a string on stack level 1 to a name and vice versa.
LC~C	Converts a long complex number on stack level 1 to a complex number and vice versa.
ASM→	Disassembles the Code object on stack level 1, returning its string equivalent. To recompile this with ASM, system flag -71 (show/hide addresses) must be clear.
BetaTesting	Returns a list of the people who beta-tested the HP 49 as a string.
CRLIB	Creates libraries. The specifics of this are outside the scope of this document, but more information is available elsewhere.
CRC	Generates a CRC for the given string, library, or backup as a system binary integer.
MAKESTR	Generated a string of the length given on stack level 1, repeating the string “ABCDEFG\n” as many times as necessary to reach this length.
SERIAL	Returns a string containing the internal serial number, which is different from the serial number on the back of the calculator, in the format “HP49 Serial Number: COYWWXXXXX”.
ASM	Assembles the string contained on stack level 1, using the MASD assembler. Accepts both System RPL and assembly language.
ER	Parses an error code list on stack level 1 and shows the errors in the string on stack level 2.
→S2	System RPL decompiler. Takes an object on stack level 1 and returns a string containing its System RPL source code equivalent, suitable for recompiling with ASM.
XLIB~	Takes a library number from stack level 2 and a command number from stack level 1 and creates an XLIB object representing that command, or vice versa. Input numbers can be integers, reals, or binary integers.
ARM→	Disassembles the ARM Code object on level 1 to its ARM source code equivalent, or disassembles the ARM code beginning at the memory address on stack level 2 and ending at the memory address on stack level 1. Note that this is ARM code, for the 49G+ only, and is not the same as the Saturn code covered by this book.
POKEARM	Pokes raw hexadecimal values into ARM memory, using the starting address on stack level 2 and the hexadecimal string on stack level 1. **Warning: dangerous!**
PEEKARM	Reads raw hexadecimal values from ARM memory, using the starting address on stack level 2 and the number of nibbles to read on stack level 1, both as user binary integers.

19.3 A text editor

Although the 49 series already has a fast-efficient text editor built in, there is still some room for improvement. A tool called Emacs, intended to replicate some of the functionality of the famous Unix text editor by the same name, was written by Carsten Dominik and Peter Geelhoed, enhancing the built-in text editor. It can be downloaded from the following location:

http://www.hpcalc.org/details.php?id=3940
→ http://www.hpcalc.org/hp49/apps/editors/emacs211a.zip

While most users will be just fine with the built-in text editor, Emacs is well-worth the memory needed for people who want something a bit more powerful. Emacs adds command completion, decompiling of ROM entries directly from the editor for easily examining the 49’s internal code, keyboard macros, regular expressions, bookmarks, and much more.

19.4 Assembling code with MASD

We don't code using bits because that would be too difficult. Assembly language uses mnemonics, with one mnemonic for each instruction available on the processor. An assembler like HP-ASM translates these mnemonics into machine language, which is what is found in hexadecimal form inside the Code objects. Because we have full control over the calculator when writing assembly language, we must code well, or a Memory Clear error is likely.

You will put your code inside a string. Although HP-ASM (on the 48 series) and all the examples in this document only use one instruction per line, with MASD you can put as many instructions in each line as you want. You can add comments using the percent symbol (%). The percent symbol is produced by [ALPHA][Left-Shift][1]. Everything that follows a `%` until the end of the line is then considered a comment.

The last symbol in the text file must be the "at" sign (@) on a new line by itself; the assembler will then know this is the end of the source. This symbol is produced by [ALPHA][Right-Shift][ENTER].

The assembler is case-sensitive, which means that "Label" is different than "LABEL". When you name your labels, it might help to write them either all uppercase or all lowercase. Choose a style of writing, and stick to it to avoid confusion.

To go to the next line, do [Right-Shift][.]. (The state of the Alpha mode doesn't matter, as it's the same either way.)

19.5 MASD flag settings

The development library uses a few flags to configure its behavior. They are:

-71	MASD disassembler does not show extra memory addresses in assembly code when set. This must be set in order to be able to re-assemble the code after disassembling it.
-86	Automatically attaches the development library upon a warmstart when set.
-92	MASD assembler defaults to System RPL mode instead of assembly language mode when set.

For the purpose of this book, it is recommended that you set flags -71 and -86 but leave flag -92 clear.

19.6 Additional MASD features

19.6.1 The linker

This is something really cool. If you divide your source code inside several parts, you can ask the compiler to compile each part and then produce the final Code object. If you have two sources, ONE and TWO, you just put inside of your program:

```
'ONE
'TWO
@
```

The ' symbol is used to place the contents of a file in the current directory, or any other directory up the hierarchy to the HOME directory, into your source. You can also specify the full path to files, either in the HOME hierarchy or a port, like the following example:

```
'H/PROGS/MATH/ONE
'2/TWO
@
```

Linked files can link to more files, but you are limited to a total of 64 links, and linked files cannot be deeper than 16 levels into a path hierarchy.

19.6.2 Directives

A directive is the '!' symbol followed by a text value. Directives are used to change MASD settings on-the-fly while compiling source code. You don't need to use any directives to get started, but they can be useful at times. Full documentation of the directives is in the MASD documentation, but a brief overview is given below. Some of these only apply to the 49G+ with ROM 2.00 or higher.

!1-16 or !0-15	Numbers nibbles from 1-16 or 0-15. See the HP-ASM options documentation above for details; the default is 0-15.
!CODE or !NO CODE	Determines whether to produce a Code object with the prologue $02DCC or a regular object without the prologue.
!DBGON or !DBGOFF	Determines whether to generate code from the DISP and DISPKEY macros for debugging.
!RPL or !ASM or !ARM	Switches to System RPL, Saturn assembly, or ARM assembly mode, respectively.
!STAT	Displays or updates the compilations statistics.
!DBGINF	Tells MASD to generate debugging information in a specially formatted string, placed on the stack after the compiled object.
!ABSOLUT *address*	Switches to absolute addressing mode, beginning at the given address.
!ABSADR *address*	When in absolute addressing mode, adds null instructions in order to continue the code at the given address.
!EVEN	When in absolute addressing mode, gives an error when compiling if not on an even address.
!FL=1.*n* or !FL=0.*n*	Sets or clears compilation directive flag *n*, respectively.
!?FL=1.*n* or !?FL=0.*n*	Compiles the rest of the line if compilation directive flag *n* is set or clear, respectively.
!ADR	Generates a list of all constants and labels used in the source rather than a compiled object.
!PATH+ *path*	Adds the given path to the automatic search path for the linker.
!JAZZ or !MASD	Determines whether to use Jazz- or MASD-style local variables, respectively.

By simply entering a directive on a line by itself, you can temporarily turn on or off any of the options from your source code string. Although many directives are used only at the top of the source code, some of the directives can be done anywhere inside of your source, depending on your needs.

19.6.3 Entry point table

Just like you use mnemonics to represent the numbers that form machine language operations, you can use mnemonics to represent the addresses of commonly used routines in the calculator's ROM. MASD allows you to install an entry point table that stores the mnemonics, automatically looking up the right memory address when compiling your source code. The entry point table is quite large (around 90KB), but tools are included to generate your own smaller one, containing only those entry points that you need, if you want.

You can download the software from:

http://www.hpcalc.org/details.php?id=3245
→ http://www.hpcalc.org/hp49/pc/entries/extable.zip

The file you need, containing all supported entries in the HP 49 series, is "extable.hp". Simply install this in your 49 in any port to use it. When installed, it also automatically attaches the development library, described above, for your convenience.

19.6.4 Advanced MASD features

MASD has a number of advanced capabilities, such as the ability to put code into blocks surrounded by curly braces ({ and }), and the ability to put multiple instructions on one line. With MASD, you can also use a dot (.) between an instruction and the field it affects instead of a space to keep code more clear when multiple instructions are on a line. For example, `C=0 W` can be written as `C=0.W`.

You can also eliminate the space between an operation and a constant (for example, `D0=D0+ 5` can be written as `D0=D0+5`), and you can eliminate the assigned register to the left of the equal sign and the equal sign itself if the first register on the right side of the equal sign is the same (in other words, `D0=D0+5` can be further simplified as `D0+5`).

To maintain compatibility with less-advanced assemblers, we will avoid using these features in this book, but they are covered in detail in the MASD documentation, available at the address below:

http://www.hpcalc.org/details.php?id=2986
→ http://www.hpcalc.org/hp49/docs/programming/masddocs.zip

19.7 Additional tools

There are additional tools available to make a programmer's life easier.

One excellent tool is Nosy. Nosy allows you to navigate around the ROM code, examining anything you want. You can give it a command name and see the code that makes that command work, jumping from System RPL commands to the assembly language code underneath. You can even search the ROM for examples of how a certain routine is used. One of the best ways to learn assembly language is to look at examples of existing code, and there's no better place to look than the masterpiece of efficient code that forms the 49's ROM.

Nosy was written by Jurjen N.E. Bos and can be downloaded at the following location:

http://www.hpcalc.org/details.php?id=4323
→ http://www.hpcalc.org/hp49/programming/misc/nosy.zip

Another library with plenty of useful utilities is OT49, written by Wolfgang Rautenberg. OT49 isn't specifically designed for assembly language programmers, and it is, in fact, aimed more at User RPL and System RPL programmers, but it provides a number of tools that anybody will find handy. With OT49, you can easily examine the functionality of keys, toggle settings, compress and decompress objects, quickly purges objects, quickly create libraries, and more.

OT49 can be downloaded from the following address:

http://www.hpcalc.org/details.php?id=3397
→ http://www.hpcalc.org/hp49/utils/interface/ot49.zip

For programmers who are interested in trying a couple incomplete but still useful programs, Pierre Tardy has written a primitive debugger and a replacement disassembler for the 49.

The debugger, Debug49 is an improvement over the built-in DISP macro, with the addition of several more screens for viewing a limited subset of registers and the next few instructions and for viewing memory, and the ability to step through a program. Although development appears to have stopped, it is still very functional, and the full source code is available under the GPL, so anybody can take over where the author left off. Debug49 can be downloaded from:

http://www.hpcalc.org/details.php?id=4715
→ http://www.hpcalc.org/hp49/programming/misc/debug49.zip

The disassembler, xASM->, was designed to provide easier-to-read output from assembly language disassembly, and at the same time it is considerably faster than the built-in assembler. It uses the compact MASD mnemonics when possible and outputs macros and entry point mnemonics for clearer disassembly. Source code isn't available for xASM->, but it is largely complete and still could be useful. It can be downloaded from:

http://www.hpcalc.org/details.php?id=4456
→ http://www.hpcalc.org/hp49/programming/misc/xasmto.zip

20 Using a PC

While this book is primarily aimed at programming in assembly language on a calculator, we know there are many who would prefer to do development on a PC. We will not discuss the PC-based tools in detail here, but we will go through a brief overview of the available software.

In the beginning, all PC-based development was done with a series of programs for DOS-based computers called HP Tools, released on Joe Horn's Goodies Disk 4, which used the so-called "HP syntax" for mnemonics. Because this book uses the HP-ASM/MASD syntax, you would need to use the 3.0 series of HP Tools, written by Hewlett-Packard and Jean-Yves Avenard and available in both source code and binary formats, at the following locations:

http://www.hpcalc.org/details.php?id=4263
→ http://www.hpcalc.org/hp48/pc/programming/hptools-src-3.0.8.tar.gz

http://www.hpcalc.org/details.php?id=4264
→ http://www.hpcalc.org/hp48/pc/programming/hptools-3.0.8-win32.zip

Documentation for HP Tools is included in the source code archive. Although only binaries for 32-bit Windows are shown above, the source code should also compile on any Unix-compatible system, including Linux and Mac OS X.

For Windows-based computers, there is also a much more user-friendly system, Debug4x, written by Hewlett-Packard, Cyrille de Brébisson, and William Graves.

Debug4x includes all you need to develop in assembly language and more. It takes around 45MB of disk space to fully install, and binaries and source code are available at the following locations:

http://www.hpcalc.org/details.php?id=5441
→ http://www.hpcalc.org/hp49/pc/programming/debug4x.zip

http://www.hpcalc.org/details.php?id=5455
→ http://www.hpcalc.org/hp49/pc/programming/debug4xsource.zip

It includes a project manager, editor, debugger and extensive help. Most importantly, it comes with emulators for the 48 and 49, and even has limited support for the 49G+. Because there is so much functionality, a high-resolution screen is recommended – even at 1024x768 resolution, it will feel quite cramped.

The assembler uses the MASD syntax, just like we use in this book. The only difference is that your code must start with CODEM and finish with ENDCODE instead of @. You can then generate code for the 48 and the 49 without even owning either!

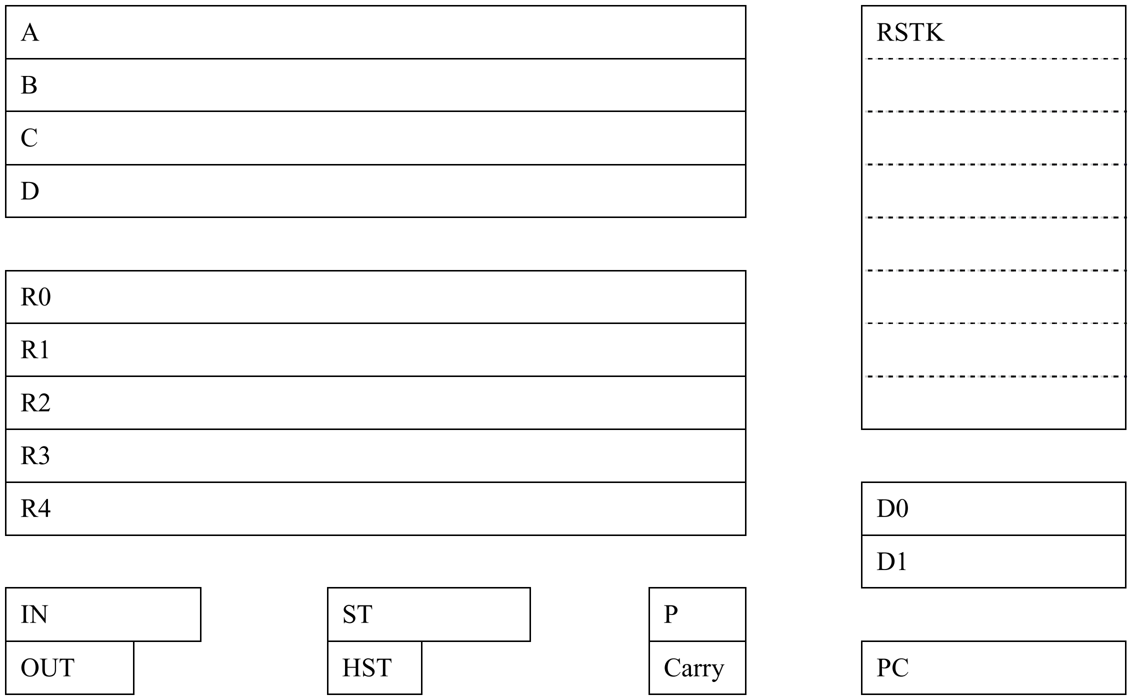
The strongest point of Debug4x is its debugger. You can easily set breakpoints in your code and check the content of the registers at those points. You can then go step by step through your code and watch the register values, add some new break points while running, and even edit the content of the registers. If you frequently make silly mistakes (and we all do!) like forgetting to clear a register before loading a new value, it is an invaluable feature.

Another good feature is that because you use an emulator you don't have to worry about losing the contents of your memory after a crash, and the code can run significantly faster. Unfortunately, being an emulator it is not the real thing, and it has sometimes (but very rarely) deceived some coders so a final test on a real calculator is always needed.

Part III: The Saturn Processor

21 Saturn registers

Here is a diagram of what is inside the Saturn processor:



As you can see above, there are several parts, each inside its own box. Each part has a name, like D0, D1, RSTK, etc. Note that this diagram is not to scale, but most of it is close.

These are internal components of the Saturn processor, and they are used to make the processor do what you want.

First, where is the processor inside the HP? Not only is it connected to the memory (working upon information stored in memory), but it's connected to the keyboard and the buzzer as well!

So we have two important parts: first, the processor, which is also called the CPU, meaning Central Processing Unit; and second, the memory.

A simple explanation is that an assembly program reads information from memory (or an input device), modifies it with simple operations (typically logical or algebraic), and then writes back to memory or to an output device.

So the work of the Saturn processor is for it to run instructions, which are programs, working with data in memory.

21.1 Processor registers

Inside the processor there are small areas that, like memory, are able to store information in the form of nibbles. There are three kinds of registers: some are called "pointers," some are simply called "registers," and others are called "save registers."

21.1.1 Pointer registers

The Saturn processor has two pointer registers: D0 and D1.

Each pointer register can store 20 bits. If we divide that by 4 (to get four-bit packets, or nibbles) we get 5. This means that, hardware speaking, the Saturn has two pointer registers, and each can receive *five* nibbles to "point" to something in memory.

For example, if I want to read a nibble stored at the address ABCDE, I will put that address into D0 or D1 using this assembly instruction:

`D0= ABCDE` or `D1= ABCDE`

Now if I read nibbles, I will read them at the memory address ABCDE.

Thus an address is something that has five nibbles, so the Saturn processor can "point" to any nibble in RAM or ROM from #00000h to #FFFFFh, so we have 1 048 576 nibbles (one nibble being four bits). We will learn more about memory later.

There is another pointer register called PC. PC means "Program Counter." It is a very important register: it contains the address in memory of the current instruction being executed. The processor uses it to know *where* it is, and mostly, where it's going. :-)

21.1.2 Counting bytes, KB

Let's divide that value, 1 048 576, by 1024. Why?

In computers and most devices that have memory, we need to say how much "space" is available.

1. You know that four bits form one nibble.
2. Two nibbles together form one byte

The *byte* is a kind of base unit. As time goes by, our computers get more and more memory, so we use exponents to quantify those increasingly huge numbers, but the principle is the same.

If we multiply 2 by 2, and so on, we get these values:

2^1	2
2^2	4
2^3	8
2^4	16
2^5	32
2^6	64
2^7	128
2^8	256
2^9	512
2^{10}	1024

1024 is a very important number for us, because we use the binary base.

The prefix "kilo" means 1000 times the value it precedes. For example, a kilogram is 1000 grams. However, when referring to memory, it's 1024 rather than 1000!

This is *important*: A kilobyte is *not* 1000 times the byte size but rather 1024 times the byte size

So, we have the "byte" (two nibbles). From the byte we can construct the kilobyte, abbreviated KB, which is 1024 bytes.

When you run BYTES, using [Left-Shift][VAR][BYTES] (on the 48) or [Left-Shift][PRG][MEM][BYTES] (on the 49), the HP gives the size in bytes on stack level 2 and the CRC of the object on stack level 1. A CRC is also called a "checksum". It is the final result of a mathematical operation performed on the object. Because the CRC is small, many different objects could calculate to the same CRC, but the chances of this happening is small enough that the CRC is a reliable way to compare whether two objects that appear the same are indeed the same.

To find out how many KB an object uses, run the BYTES command and then divide by 1024 to get the number of KB.

When you have a 48G with 32 KB, you have 32 * 1024 bytes of RAM, which is 32768 bytes. By multiplying that figure by two we can find how many nibbles of RAM are in a 48G: 65536.

21.1.3 Addressable space

How much memory space is the Saturn able to access? We know the processor has two pointer registers: D0 and D1, and each can have a five-nibble value inside. Five nibbles can have 1 048 576 different values. Because the HP's Saturn works with nibbles, we can access 1 048 576 nibbles, or 524 288 bytes. Divide that figure by 1024, and what do you get? Good! 512 KB.

So the Saturn, at any moment of its execution, can *only* manage 512 KB of memory. (Later we'll learn there are tricks to manage more than 512 KB.)

This is called the addressable space.

21.1.4 Working registers

Those registers are sometimes called calculation registers and are where the processor does its work. If you want to add a value from one area of memory to another, you have to:

- read first and second data (if there's a second one) into a working register
- work with it
- do what you want with resulting data

So if you want to add or subtract values, shift bits, perform logical operations, and so on, you have to get the value into a working register. On the Saturn these working registers are called A, B, C, and D.

So we have four working registers. Each register is 64 bits, or 16 nibbles, wide.

You see, the Saturn processor is able to work with 64 bits in a single step, inside a working register! It's a good processor, even though as time flies by, it won't be such a good one in ten years. :-(

21.1.5 Save registers

There are five save registers: R0, R1, R2, R3, and R4.

Those registers are used to keep data inside the processor, freeing one or more working registers so they can get new data. If you want to free working space, it's better to use a save register by moving nibbles from a working register to a save register than to write the nibbles somewhere else in memory. This is because what is kept *inside* the processor is done *faster* than any exchange with memory outside the processor.

Of course, every save register has the same size as any of the working registers: 64 bits, or 16 nibbles. This means that each save *any* of the working registers.

So now we've learned:

Chapter 22: The fields of working registers

- A, B, C, and D are the working registers; nibbles are worked upon inside of those registers.
- We can use one of the five save registers to save one or several working registers for a short period of time (it's a temporary area).
- D0 and D1 are used to point to any address that the Saturn processor can have access to using five nibbles.
- PC is also a register pointer, but it only points to the current instruction (it's updated by the processor as it runs machine language code).
- When either D0 or D1 "points" to an area of memory, we can read or write nibbles from the address pointed to or from a working register.

22 The fields of working registers

We know working registers can have nibbles inside of them. Their size is 64 bits, or 16 nibbles.

Oftentimes we don't need to work with all 16 nibbles of a register. Perhaps we want to work with only five nibbles, two nibbles, or maybe even only one nibble.

What we call a *field* is a part of a working register. If the register is 16 nibbles wide, we can, for example, call a five-nibble wide section a "field" of the working register.

The Saturn processor already has predefined fields, so it's easier for us to work.

Before we begin learning about fields, I want to tell you about a strange thing with the Saturn processor: nibble inversion. *Stay awake for this part*!! :-)

22.1 Nibble inversion

This is something that will be complicated for some. The Saturn inverts the order of the nibbles when reading from or writing to memory. What I call reading nibbles is moving nibbles from an address in memory to a working register, and writing being moving nibbles from a working register to an address in memory.

Working registers can store up to 16 nibbles. Each nibble inside the register will have a number, as you will learn in the diagrams below. Nibble number zero is the first one (the LSB nibble), and nibble number #Fh is the last one (the MSB nibble). The nibbles are numbered from right to left.

Here are the 16 nibbles of any working register:

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0

Note that they are numbered from right to left, so the "start" of the register is the rightmost nibble, and the "end" (nibble #Fh) is the leftmost nibble.

Although *we* (except Arabic or Hebrew friends, and Chinese/Japanese writers) read from left to right, the Saturn starts moving nibbles from the LSB nibbles (nibble number 0, then 1, and so on).

Therefore, the Saturn processor "reads" and "writes" from right to left. For the Saturn, the first nibble is the rightmost. We call this a "little endian" architecture, and *x*86 PCs work the same way. The Macintosh (both 68000 and PowerPC 970) and the SPARC, on the other hand, use a "big endian" architecture, where the MSB is on the left and the LSB is on the right. Some UNIX systems, such as those on the Alpha and PowerPC architectures, and also the ARM architecture, can be switched to operate either way.

Let's consider our register C. It's 16 nibbles wide, so we can imagine we have C like below:

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Now, think of this in memory:

Displacement	Address in memory	Nibbles
0	00100	1B01C
+5	00105	022F0
+10	0010A	00033

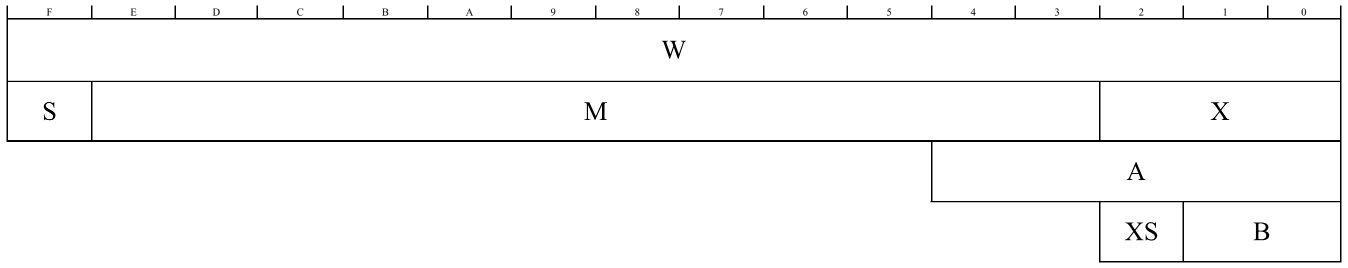
If I read 15 nibbles into the C register, starting at address 00100, C will contain the following values:

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
0	3	3	0	0	0	0	F	2	2	0	C	1	0	B	1

The nibble number #Fh has not been modified, but you can see that the Saturn loads nibbles from *right* to *left*. If we have 1B01C 022F0 00033 in memory we get 33000 0F220 C10B1 in our working register. (Notice that I removed nibble #Fh, as I am only working with 15 nibbles for this example). Remember that we read the contents of the register from *right* to *left*.

22.2 Available fields

The drawing below is extremely important, so you should *learn* it and be able to reproduce it from memory. I learned it the first time I saw it, following another coder's advice, preventing me from having to search for this information all the time. To make it simple for you: *learn this*! :o)



Throughout this document, we may refer to fields like B(A), which means field A of register B. Sometimes programmers write this as B[A], B.A or Ba, so don't be confused if you see different notation elsewhere.

22.2.1 The W field

This field is easy: it's the *whole* register of 64-bits.

If we give a number to each nibble, we start from 0 and go to 15, because we have 16 nibbles. W is all 16 nibbles:

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0

←—————————— W ——————————→

W means **word** (sometimes called "**wide**")

When we want an instruction to work with all 16 nibbles of a working register, we use the W field.

22.2.2 The A field

This is the most used field of all. It's 5 nibbles wide, and of course:

A means **address**

It looks like this:

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0

←——— A ———→

Please *remember* that the first nibble is 0, not 1. So when we think of the A field, we have five nibbles: the first one is 0, the second one is 1, the third one is 2, and so on through 4.

Don't forget it's from right to left.

22.2.3 The B field

This is another very useful field: it's the byte one! :-)

As you know, two nibbles form one byte, so the B field is two nibbles wide:

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0

← B →

Do you still remember the MSB and LSB? We can say that the B field is a part of (or is) the LSB of the working register :-)

22.2.4 The X field

This field is 3 nibbles wide. It's very useful to use with another component of the processor: OUT.

OUT will be used to check a key or to produce a sound using the HP's buzzer. OUT will get three nibbles of data, so the X field will be used with OUT. Very useful, and for other things too…believe me. ;-)

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0

←— X —→

X = eXponent (see M field below)

22.2.5 The XS field

The XS field is the third nibble of the register, that is, the MSB of the X register.

XS = eXponent Sign

22.2.6 The M field

This is the Mantissa field, nibbles 3 to 14

This field comes from the way the HP-71 handled numbers with its Saturn processor: S (described below) was used for the sign, M for the Mantissa, and X for the eXponent. This is still used when pushing real numbers onto the stack.

Now you know why those letters are there. :-)

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0

←—————— M ——————→

22.2.7 The S field

The S field is the last nibble of the working register, number 15.

22.2.8 Field summary

So, keep this in mind:

- working registers can be divided into fields
- we will often reduce an instruction to a field
- the fields here have *fixed* sizes

Remember, the sizes are:

W	64 bits or 16 nibbles	Whole register
S	4 bits or 1 nibble	Sign
M	48 bits or 12 nibbles	Mantissa
X	12 bits or 3 nibbles	eXponent
A	20 bits or 5 nibbles	Address
XS	4 bits or 1 nibble	eXponent Sign
B	8 bits or 2 nibbles	Byte

23 The P register and WP

We have seven defined fields, which are all described above. There is one field, WP, whose size can be *customized* by the coder, using the P register.

First I will explain P. This register can only store *one* nibble, but it is very special. Its value will go from 0 to F.

1. We can use P as a field. If P contains the value *n*, then the P field is the nibble number *n* of the working register. Example: if P=2, then using the P field reads or writes *one* nibble, nibble number 2 (that is, the *third* nibble of the register, because the first one is 0, second one is 1, etc.)

2. P defines the size of WP. WP is Wide-P. Its size in nibbles goes from the first nibble to nibble *n*. Example: if P=5, then WP is nibbles 0 to 5, thus we can work with 6 nibbles using WP.

WP is useful for reading information with variable sizes, like reading a binary integer.

Remember:

- When you want to read one nibble that is not the first one of the register, we load its position number into P, and then we read it. For example, if we want to read nibble number 8, we will do P=8 and then read one nibble there using the P field (I'll show you how later).
- P defines the size of the WP field, from 0 to *n*, with *n* being the value of P. If you want to read or write *m* nibbles you do `P=(m-1)` and then use WP as the field.

Now, something really, really important about the value of P.

When your program starts, P=0, and when your program ends, you must be sure P is always set to 0. If you modified its value, you'll have to put it back to 0.

Why? When you load data inside a register, P defines where (which nibble) the data will start loading.

There is an instruction to load nibbles inside the C or A registers, but there are none to load nibbles inside B or D. If you want to load nibbles into B or D, you'll first have to load nibbles into C or A and then copy those nibbles into B or D.

If P is set to 0, and you load #A3Bh inside C, you get:

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	B	3	A

But if P is set to 2, and you load #A3Bh inside C:

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	B	3	A	0	0

And if P is set to E, and you load #A3Bh:

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
3	A	0	0	0	0	0	0	0	0	0	0	0	0	0	B

WHAT?

Yes...the 3 nibbles have been loaded, starting *at* nibble E, and because we load 3 nibbles, two are loaded in #Eh and #Fh, and then the last one is loaded at the beginning of the work register. The numbers are going in a circle!

This is explained in more detail later.

24 User-defined fields

The "Saturn+" architecture in the "Apple" platform calculators with an ARM processor emulating the Saturn (the 49G+ and 48GII at the time of writing) adds additional features to the Saturn architecture, including the ability to have user-defined fields. The developer can create a mask of 64 bits and assign it to a user-defined field. Any operations on this user-defined field will only affect the bits defined by the mask. These fields are called F1 through F7.

To use these fields, load a mask into C, like the following:

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
F	F	C	0	0	0	0	0	0	0	0	0	0	3	F	F

Now call one of the seven "set field" instructions, such as `SETFLD7`. Field F7 will now only affect the non-null bits defined above.

25 Circular handling of numbers inside the processor

*** IMPORTANT STUFF * READ TWICE IF NEEDED ***

When we work with *fields*, we fix a limit on the number of nibbles that will be used.

The B field works with two nibbles. Every instruction, like adding a value, will work on *only* those two nibbles, and won't affect the remaining 14 nibbles of the register.

If a register contains #FFh in its B field size, what will happen if I, for example, add 1? Remember that we are using the B field so we cannot go from #FFh to #100h, because that would be using 3 nibbles.

What will our lovely Saturn friend do? It does this:

#FFh + 1 = #00h (when working on a B field) *and* something called the “carry” will be set to 1.

The carry is a flag; flags are indicators inside the processor that you can test. They’re described later.

If I were working with the A field, and A contained #00000h, removing 1 would return #FFFFFh inside the A field, and the carry flag would be set to 1.

If you have #FEAh inside the X field of a register and add #20h, you’ll get #00Bh in the X field of the register.

Notice that when you are working on a register and perform an operation, like adding or removing a value, if there is a loss of information because we “go around” inside a field (like #FFh + 1 gives #00h as result when using the B field) the processor sets the carry bit to one to warn you.

There is another thing that can lead to loss information: shifting bits.

I will describe that when I start giving you instructions. I am going to give you a complete description of *every* instruction, how it is coded, how many processor cycles it uses, and so on.

If we do a summary about what we have learned so far about the Saturn processor we have:

- Four working registers: A, B, C, and D
- Five save registers: R0, R1, R2, R3, and R4
- P, which gives you access to *any* nibble of a register, *and* defines the WP field size
- WP, which goes from nibble 0 to nibble (*n*-1) with *n* being the value stored in P
- D0 and D1, which are pointer registers, and allow you to point to a nibble in RAM/ROM (from #00000h to #FFFFFh)
- PC, which contains the current instruction being executed.

Now we must learn about RSTK, ST, HS, OUT, and IN.

Let’s go :-)

26 RSTK

RSTK means “Return STacK.”

First, what is a “stack?” Hopefully, as HP users, we already use a stack when using our HP. The stack is used *a lot* in computers because of memory structure. This will change when 3D chips are introduced, and when HP will have eaten Intel and Microsoft, but let’s be patient until then, and keep that secret. ;)

The RSTK is a LIFO Stack. LIFO means Last In First Out.

Like our HP stack, if you enter a number onto RSTK, the first one to be popped out will be the last one that was put in.

To put it another way, it’s like pushing coins into a tube and removing them from the same hole they were put into. Another kind of stack is FIFO, or First In First Out. This is like a tube where we put coins on one side and we wait for them at the other hole: the first ones pushed into the tube will be the first ones to be recovered.

This RSTK, being LIFO, is similar to our HP’s RPL stack. However, it’s limited in size: there are *eight* levels available, and each level has a *fixed* size: 5 nibbles.

This is used by subroutines. When you call a subroutine from an assembly language program, the processor records where we are, stores it into RSTK, and jumps to the subroutine. When the subroutine wants to go back to where it was called from, it pops the “return address” from the return stack and jumps to it. Because it is an address, we only need to have five nibbles per level on the RSTK stack.

Whenever an address is popped from the stack, the top level is replaced with zeroes. Also, the stack has something called "circular behavior", so after a stack overrun you can always read the last eight addresses. For example, suppose you write 1, then 2, then 3, 4, 5, 6, 7, 8, and 9 to RSTK. You can now read 9, 8, 7, 6, 5, 4, 3, 2, and 0. In other words, you read the eight remaining levels, plus a zero, as zeroes filled the stack as it dropped.

So the RSTK contains eight 5-nibble addresses. The interrupt system of the HP uses two levels when called, and sometimes a third level is used, so unless we don't allow *any* interruptions, we will only be allowed to use *five* levels.

RSTK is useful for quickly storing temporary information. We'll discuss this more later.

27 Flags: ST, HST, and Carry

Flags are *special* bits that can have two values: 1 or 0. When a flag is *set*, it has a value of 1. When a flag is *clear*, it has a value of 0.

27.1 Carry

This bit is a flag in itself. Each time an operation (like adding or subtracting, or when we get an overflow* on a register) has a carry, the carry bit is set to tell us that either a carry or an overflow has occurred.

* overflow is explained above. When we work with a field of a working register, if adding or removing a value makes us "go around" the carry is set to 1. If field A of C contains #FFFFFh, adding 1 to field A will put #00000h in field A of C, and the carry will be set to 1.

Every time we perform a test, if the test is *true*, the carry is set (set to 1), and if the test is *false*, the carry is cleared (set to 0).

This means that when we perform a test, the processor will always either set or clear the carry. Next the carry bit is tested to determine what to do next.

27.2 ST (Status Bits)

The ST flags are composed of 4 nibbles. Therefore, we have 16 bits available, with each one being a flag.

As a coder, you are allowed to use flags 0 to 11. Flags 12 through 15 have special meanings and can be dangerous if misused.

Here are the meanings of bits 12 through 15:

bit 12: Forced wakeup request (overrides DeepSleep)
bit 13: Set if an interrupt has occurred
bit 14: Set if an interrupt is pending
bit 15: Set to 1 if interrupts are enabled or 0 if interrupts are disabled

ST bits 0 through 11 may be given whatever values you want. You can later test them, with each one being a flag with the meaning you give it. :)

The Saturn handles bits 12-15. We'll use bit 15 sometimes, but you should not mess with the others.

27.3 HST (Hardware Status Bits)

This flag has one nibble, or four bits. The Saturn uses these four bits to handle special hardware events. The four bits are:

bit 0: XM (eXternal Module missing)
bit 1: SB (Sticky Bit)
bit 2: SR (Service Request)
bit 3: MP (Module Pulled)

There is only one bit that can be set by us on the Saturn: XM. We'll use an instruction called RTNSXM (ReTurN and Set XM). The Saturn+ on the ARM-based calculators lets us set or clear any of the bits.

You'll find the SB to be useful: when we shift bits around a register, if a bit goes from the MSB to LSB, or from LSB to MSB, the Saturn will turn the SB to 1 to warn us there has been a loss of information during the bit shift. This will happen if bit 1 of the LSB goes around and gets into the MSB nibble. As it is the sticky bit, it will remain as 1 until manually set back to 0.

28 OUT and IN

The Saturn processor is able to exchange information with the keyboard and produce sounds. Two registers inside the processor are thus used: OUT and IN. They have specific sizes:

OUT is 3 nibbles wide (12 bits)
IN is 4 nibbles wide (16 bits)

When we want to test a key, we'll put a value (from a table of values) into a register like C, and send it to the OUT register. Then electrical power will be given to lines in the keyboard (from 1 to all), and if a key is pressed, the result is put into the IN register. We'll read that value, as it will tell use whether a key was pressed, and if so, which one. :-)

29 How the processor works: life of an instruction

How does the Saturn execute an instruction?

We know the PC register (Program Counter) is used by the processor to know "where" it is. The processor will always update this counter so it knows where to look for the next instruction to execute.

This is the cycle the processor uses to execute an instruction:

1. The PC register tells the processor the location of the next instruction to execute.
2. This address is sent on the bus of the Saturn processor. The bus is a line that links the processor, the RAM, the ROM, the input/output components, etc., together.
3. The Saturn receives the instruction to execute in an internal register. It has to be decoded; this is mandatory, as the processor needs to decode the instruction using specialized circuits.
4. Once the circuits in charge of decoding the instruction have done their work of decoding, they give the instruction to the unit in charge of sequencing commands inside the processor. This unit uses a quartz crystal-generated frequency to execute commands. The quartz runs at 32 kHz and is multiplied to generate the clock frequency. The HP 48S/SX have a ~2 MHz frequency, and the HP 48G/G+/GX and 49G have a ~4 MHz frequency. The HP 48GII and 49G+ have a variable (up to 48 MHz or 75 MHz, respectively) frequency that emulates the Saturn at a variable speed, often in the 8 MHz or 12 MHz (respectively) equivalent range, but they are much faster when running native code.
5. The unit in charge of sequencing commands sends orders by calling and using the ALU (Arithmetical Logical Unit). This unit is in charge of the logical operations (complementation, logical OR/AND/NOT, etc.) and arithmetic operations (subtracting, adding, etc.).
6. The result is stored in one of the processor's registers. PC is incremented to the next instruction.

And we repeat.

This introduces the concept of:

30 What is a processor cycle?

When we ask the processor to do something, even if it looks simple to us (like adding 1 to a register's value) it is more complex for the processor. The instruction has to be fetched, then decoded, then passed to the unit in charge of executing it, and so on.

So each of these steps is divided into elementary operations, and each one is done in one cycle. Some instructions will require 11 cycles, for example, but others will require more or less.

Each cycle is a "state," or an impulse coming from the processor's quartz crystal.

Consider the HP 48G, whose processor runs at about 4 MHz. Each second, four million impulses are generated and each instruction takes a predetermined number of cycles to be completed.

Note that the cycle counts given for instructions in this book are not the exact cycle counts, as it is virtually impossible to get exact counts. That would need an oscilloscope, but these should be close enough. These cycle counts are from the Meta Kernel documentation (thanks, Jean-Yves!) and should be close to the correct figures. Instructions that are only on the 49G+ (the Saturn+ instructions) do not have cycle counts listed; because these run ARM code, they therefore do not necessarily follow Saturn rules for cycle counts anyway.

In addition, the cycle counts may vary depending on whether the instruction is at an odd or an even memory address! Some instructions are given with a fractional cycle count. With each of these instructions, if it is at an even address round down (take the FLOOR of the number), and if it is at an odd address round up (take the CEIL of the number). Also, if two counts are given and are separated by a comma, add the FLOOR of the second number if it is reading from an even address or add the CEIL of the second number if it is reading from an odd address. Because your code will be at a random address, equally likely to be odd or even, you can safely use the fractional cycle counts to get a very close approximation of execution time.

31 Review: Calculations

An important thing here is that we have a limit on how many digits we can use. It's not only because we have 16 nibbles (64-bits) in each working register, but also because we will very often use a field which is smaller than W to do calculations.

Because a byte is two nibbles, the greatest value it can have is #FFh.

The use of negative numbers is not very easy, but two's complement is available so we can use it. It's the same for fractional numbers; the first book I used to learn assembly language explained (as far as I remember) that assembly didn't let us use fractional numbers. In fact, we can either use negative exponents, or simply use a fixed decimal position, and other tricks, like Sophie explained (yes, there are girls that code using assembly language on the HP as well!).

The circular handling of numbers must also be understood. Because each field has a limited number of nibbles, when we add or remove constants the value inside can go through the #0h value, setting the carry flag. This is useful for loops: both wait loops that do nothing and loops that are used to repeat pieces of code.

If the X field is used, there are three nibbles to code values. This means one can use values from #000h to #FFFh. What will happen if one adds #1h to #FFFh? The value goes to #000h and the carry bit is set. It also works with #000h - #1h.

When your values go from one number to another and cross #0h, the carry bit is set to warn you an overflow or underflow occurred.

If you have trouble with shifts, just remember the following:

Shifting a bit to the right divides by 2.
Shifting one nibble to the right divides by 16.

Shifting a bit to the left multiplies by 2.
Shifting one nibble to the left multiplies by 16.

Or even easier:

Divide field f by 2:	`CSRB f`
Divide field f by 16:	`CSR f`
Multiply field f by 2:	`C=C+C f`
Multiply field f by 16:	`CSL f`

Don't worry if you don't know what this means yet. Here instructions have been given for use with the C register, so if you want to use another register you have to change the first letter, but we will go into detail on that in the next section. As you can see, to multiply a register (or a part of it) by two, you add it to itself.

Sorry if you didn't understand it the first time. :)

Part IV: Saturn Instruction Set

32 Entry points and other notes

Some of the instructions that we will cover in this book call code in the calculator's ROM. The code in the ROM is located at memory addresses called "entry points". While some of these entry points are the same in all calculators, many have different addresses in the HP 48 than in the HP 49. Some, known as unsupported entry points, may have different addresses in each ROM version, but it is best to avoid using these for compatibility reasons.

If you have an entry point table installed in your calculator, MASD on the HP 48 with Meta Kernel and the HP 49 allows you to enter entry point names instead of their addresses, and MASD will automatically convert the name to the address. All examples in this book will show the entry point name in the source code, but the address will always be listed in a comment following the name, because not everyone will be able to use the table.

Some of the instructions listed in this section have a plus sign (+) after the opcode. These instructions are only available on the ARM-based calculators, as they are only in the Saturn+ instruction set. These instructions also do not have execution times listed for them. Do not try to use these instructions on regular Saturn-based calculators.

33 Working register instructions

33.1 Loading a value inside a register

It is very useful to be able to load a value inside a working register. There are only two instructions available, and only the A and C registers can be loaded with a value.

The instruction LA (Load in A) is used to load a value into the A register, and the instruction LC is used to load a value into the C register.

For example, to load #ABCDE inside the C register, write:

```
    LC ABCDE
```

After the code is run, the C register will look like this, with each nibble moved into the register:

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	E	D	C	B	A

As we have seen, the register is read from right to left, and LC and LA work this way as well. Note that if the register was not 0, which is usually the case, it will keep its content past the last loaded nibble. We will go through examples of this later.

As the working registers C and A are 16 nibbles wide, we are unable to load more than 16 nibbles into either one.

Let's see what happens. Put an empty string onto the stack and edit it. Write this:

```
    LC 1234567890ABCDEF1
    @
```

You can get the @ by pressing [ALPHA][Right-shift][ENTER]. The @ must be included to indicate the end of the source file.

Now, type ASM (or →ASM if you are using HP-ASM 1.0) to compile the file.

You will get an error because you tried to load too many nibbles.

If you are using HP-ASM on the 48, you will have the following on the stack:

```
3:  "LC 12345678...
2:            <0h>
1:  "LC 12345678...
```

The string on level 3 is the original string, level 2 contains a system binary of the line where the error is, and level 1 contains the line that has the error.

Here we can use the ED command of HP-ASM to edit the source, and it will move the editor to the line of the error; but ED will use the HP's internal editor. :P

It is possible to customize ED to use any editor, but that requires a little knowledge of library hacking and System RPL programming.

Until you customize ED, you'll probably want to drop two levels and edit the string with your favorite editor when you get an error.

If you are using MASD on the 48 with Meta Kernel or the 49, you will have the following on the stack:

```
2:  "LC 12345678...
1:  {{ ¤ 10108h ...
```

The string on level 2 is the original string, and level 1 contains the error list. Here you can use the ER command to edit the source and jump to the error.

You can use `LC` or `LA` to load from 1 to 16 nibbles inside A or C. There are no instructions to load a value into B or D, so we will have to first load a value into A or C and then move it into B or D.

An important thing to consider when using `LA` or `LC` is the value of P. The value of P, when you will start a program, will be zero. You can change its value, but when you quit, you have to restore it to zero.

If P=0, when you load something into A or C it's loaded from nibble 0 of the register. If P=4, loading will start at nibble 4.

For example, this is what happens if we load ABCDE into C when P=4:

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	E	D	C	B	A	0	0	0	0

So if you modify P's value (and you will, if you need to read the nibble *n* of a register or use WP) you have to pay attention to `LA` and `LC` instructions.

Also, loading a value overwrites any value previously stored in a register. Suppose C contains:

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
3	A	8	B	B	3	8	2	0	0	0	A	F	0	F	C

After loading three zeroes into it, C will contain:

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
3	A	8	B	B	3	8	2	0	0	0	A	F	0	0	0

The nibbles are not "moved" inside the register when you load a value, so any previous value is overwritten.

I have also shown you there is a circular loading of nibbles into registers. If we give P the value of E and load 6 nibbles using `LC ABCDEF` we will get:

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
B	A	0	0	0	0	0	0	0	0	0	0	F	E	D	C

Each time an instruction is listed, five facts are given:
1. The name of the mnemonic (here "`LA`" and "`LC`").
2. How it's coded in memory – the opcode – in hexadecimal.
3. Which fields can be used or are used.
4. Whether the carry bit is affected. If yes, the carry bit will always be set to 1 or 0 after the operation.
5. The approximate number of cycles the processor needs to complete the instruction.

33.1.1 LA, LC

Mnemonic	**Opcode**	**Fields**	**Carry**	**Cycles**
`LA` *`n`*`..1`	8082 *n* 1...*n*	according to P value	no	7.5+1.5*n*
`LC` *`n`*`..1`	3 *n* 1...*n*	according to P value	no	3+1.5*n*

1 is the first nibble being loaded and *n* the last one. You have examples of how nibbles are loaded above. `LA` and `LC` do not modify the carry, even if a circular loading appears.

As you see here, `LC` is faster than `LA`. It's coded with fewer nibbles in memory. It's not very important, unless some part of your code needs to go really fast (like everything linked to display, usually). If you need speed, use `LC` rather than `LA`.

n is the number of nibbles being moved. This piece of code:

```
    LC 02A2C
```

will move 5 nibbles into the C register, and needs 3+1.5*5 cycles, or 10.5 cycles.

33.1.2 Setting a register to zero

Here we are going to learn how to set a register, from 1 to 16 nibbles, to zero. The first letter will be the letter of the register, or A, B, C, or D. Then, the equal sign (=) and 0 of course!!

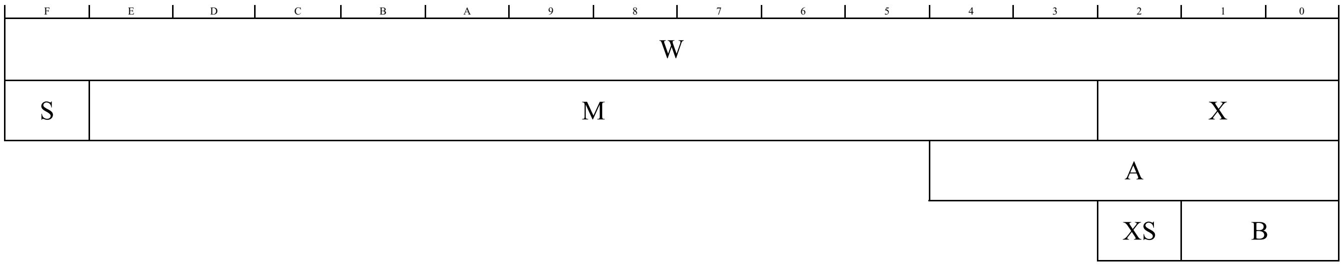
To clear the whole C register (all nibbles), simply write:

```
    C=0 W
```

:-)

Remember that W is *word* (or *wide*). It is the field that affects the whole register, or 16 nibbles!

Here are the fields, which were described in section 22.2:



So if we now want to put all zeroes into the X field of register B, we will use:

```
    B=0 X
```

Mnemonic	**Opcode**	**Fields**	**Carry**	**Cycles**
`A=0 f`	D0 Ab0	A only P,WP,XS,S,M,B,W	no no	8 4.5+*n*
`B=0 f`	D1 Ab1	A only P,WP,XS,S,M,B,W	no no	8 4.5+*n*
`C=0 f`	D2 Ab2	A only P,WP,XS,S,M,B,W	no no	8 4.5+*n*
`D=0 f`	D3 Ab3	A only P,WP,XS,S,M,B,W	no no	8 4.5+*n*

What is f? It's the field! :-)

The A field is treated separately than others, because the A field is used so often in the processor that it has specific hex forms.

Also, you will see that the "opcode" column usually lists hex numbers, but sometimes it has other letters. You must use a table to know which letter is to be used, according to the field.

Example: I want to do `A=0 S`. The instruction is Ab0, and by using the table below you can find the value of b:

Field	**a values**	**b values**
P	0	8
WP	1	9
XS	2	A
X	3	B
S	4	C
M	5	D
B	6	E
W	7	F

Because there is a "b" you need to use the "b" column. Look for S, and you'll find "C," so you know the instruction to do `A=0 S` is "AC0."

For the new 49G+ operations that work with user-defined fields, we will use "f". They are shown in the following table:

Field	**f values**
F1	8
F2	9
F3	A
F4	B
F5	C
F6	D
F7	E

You will not need this information unless you intend to code a disassembler or an assembler. But now if you want to create one you have the info.

As usual, "*n*" is the number of nibbles being cleared to 0.

Let's code! :)

We are going to code something that sets the entire C register to zero and then quits. HP-ASM wants one instruction per line, so each time you write a complete instruction, you will have to move to the next line, like below:

```
    C=0 W
    A=DAT0 A
    D0=D0+ 5
    PC=(A)
    @
```

As discussed before, MASD allows multiple instructions on a line, but all code presented here will only have one per line for maximum compatibility.

I know, there are instructions here you don't know yet, but let's first compile it and run it, okay? If you don't make any mistakes, you'll get a Code object. :-)

Press [EVAL], and it will run! The W field of the C register will clear, and then the code will end.

Now, I need to explain the last 3 instructions.

```
    A=DAT0 A
```

You know that the Saturn processor has two pointer registers: D0 and DA. (It has PC too, but we won't use it here.) If D0 points to an area somewhere, then DAT0 is the data there. It's the same for D1. If D1 points to #ABCDE, we will use DAT1 to get nibbles from there.

Let's say there are 5 nibbles:

Address	**Nibbles**
#80120h	02A2C00005

If we want to read, say, three nibbles from #80120h, and put them in A, we will code this:

```
    D1= 80120
    A=DAT1 X
```

First, we make D1 point to #80120h, and then we read three nibbles (field X) into A.

If we want to read 10 nibbles using C and D0, we do this:

```
    D0= 80120
    C=DAT0 10
```

(You can put a number in the field location, which is cool :-)

Now, let's explain these instructions:

```
    A=DAT0 A
    D0=D0+ 5
    PC=(A)
```

When a program (your program) is given control, some registers and some pointers are used by the HP:

D1 points to the first level of the stack
D0 points to the next object to execute
Field A of register B points to the return stack of the RPL
Field A of register D contains the free memory in 5-nibble blocks

When your program starts, those values are used. Sometimes we will have to "save" some register values, because if we lose B or D0, we'll get into trouble (the HP will halt, or if you're unlucky, crash).

D0 points to the next RPL object to execute. When we want to quit, we have to read the address of the next RPL object to run into A, that is:

```
   A=DAT0 A
```

So now, A contains the address (5 nibbles read from where D0 points to, A being 5 nibbles wide).

Now that we have read it, we have to move D0 5 nibbles later. It is important to increment D0 to the next object. We are going to execute the current one, so the next instruction that will read from D0 must read from the good zone. :-) That's why we increment D0 by 5 nibbles: we have read 5 nibbles, so we must move 5 nibbles later:

```
   D0=D0+ 5
```

The last instruction jumps to that address, directly using the PC pointer. The Saturn processor, when it needs to know "where" to continue, will read from PC. So we make PC point to A, and the execution continues. Remember that the following code is the usual way to "return to RPL":

```
   A=DAT0 A
   D0=D0+ 5
   PC=(A)
```

You will use this to quit your programs most of the time. :-)

Other small pieces of code will be given as we learn more Saturn instructions, like code to drop, swap, and so on.

33.1.3 Changing the value of a bit

We can change the value of any bit inside the first four nibbles of registers A or C (not B or D!). As you have learned, when a bit is equal to 1 it's set and when equal to 0 it's clear.

We will use the `ABIT` (for register A) and `CBIT` (for register C) instructions. When we want to set a bit, we use:

```
   ABIT=1 n
   CBIT=1 n
```

and to clear a bit:

```
   ABIT=0 n
   CBIT=0 n
```

where *n* is the number of the bit set to 1 or 0, and *n* may have values from 0 to 15 (number of the bit).

NOTE: Some people don't like to start counting from 0, and prefer to count from 1. HP-ASM and MASD let you choose, as described in their overviews earlier in this book. We will always use 0-15 here, so if you choose 1-16, you'll have to adapt your sources.

Here are the mnemonics:

Mnemonic	Opcode	Fields	Carry	Cycles
`ABIT=0 n`	8084*n*	bit number *n*	no	7.5
`ABIT=1 n`	8085*n*	bit number *n*	no	7.5
`CBIT=0 n`	8088*n*	bit number *n*	no	7.5
`CBIT=1 n`	8089*n*	bit number *n*	no	7.5

Example: I want to turn on the alarm indicator when I launch my code. How would I do this?

First, you have to know that indicators are just bits stored in a certain area of RAM. By giving bits values of 1 we can turn on indicators, and by giving them values of 0 we can turn them off.

I will explain that now, so you can learn about these bits and special values.

At the address #0010Bh (entry point ANNCTRL) there are two nibbles (one byte) for controlling the annunciators:

Bit number	**Function**
0	left shift indicator (1=on, 0=off)
1	right shift indicator (1=on, 0=off)
2	alpha indicator (1=on, 0=off)
3	alarm indicator (1=on, 0=off)
4	busy indicator (1=on, 0=off)
5	transmit indicator (1=on, 0=off)
6	reserved for future use
7	show indicators? (1=on, 0=off)

So if I want to turn on the alarm indicator, I have to read the byte at #0010B, set its bit number 3 to 1, and then leave.

Let's do it!! :-)

First, we need to use D0 or D1 to point there. But the HP uses D0 and D1! D0 points to the next RPL object to execute, and D1 points to the first level of stack.

So we must "save" the value of D1 (or D0) and restore it before leaving.

I'm going to use C to read the byte and A to save the value of D1, so I can have D1 point somewhere else.

To move the value of D1, we're going to use an instruction we have not seen yet: `AD1EX`. This instruction moves the 5 nibbles in D1 to A, and the 5 nibbles of A to D1. After using it, the previous value of D1 will be kept in A. This instruction is used to exchange the contents of D1 with A and is explained later in this document. :-)

First, move the value of D1 to A:

```
AD1EX
```

Then, point D1 to #0010Bh, and read one byte there:

```
D1= 0010B
C=DAT1 B
```

Why B? B is the first byte, so it reads the first nibble of the memory at the location referred to by D1 and puts it in C. Now the value of bit 3 in C must be changed. We can use the instruction we have just learned:

```
CBIT=1 3
```

Write it back to memory:

```
DAT1=C B
```

(Instructions for reading and writing data are explained later, but I think you now understand how it works.)

Then give D1 its old value:

```
D1=A
```

and quit to RPL the usual way:

```
A=DAT0 A
D0=D0+ 5
PC=(A)
```

So the code to turn on the alarm indicator and return to RPL, is:

```
AD1EX
D1= 0010B     % Entry point name is =ANNCTRL
C=DAT1 B
CBIT=1 3
DAT1=C B
D1=A
A=DAT0 A
D0=D0+ 5
PC=(A)
@
```

Now you can compile it and run it. Of course, it goes very quickly. You'll have to watch very carefully, and maybe run it a few times, but you should see the alarm indicator flash on briefly.

33.1.4 Exchanging two registers

We can exchange two register values. We can either exchange the whole register or only a smaller field of it.

Mnemonic	Opcode	Fields	Carry	Cycles
`ABEX f`	DC AbC	A P,WP,XS,S,M,B,W	no no	8 4.5+*n*
`BCEX f`	DD AbD	A P,WP,XS,S,M,B,W	no no	8 4.5+*n*
`ACEX f`	DE AbE	A P,WP,XS,S,M,B,W	no no	8 4.5+*n*
`CDEX f`	DF AbF	A P,WP,XS,S,M,B,W	no no	8 4.5+*n*

The number *n* is the number of nibbles being exchanged between one register and the other.

NOTE: ACEX is the same thing as CAEX of course. HP-ASM and MASD want you, when you have 2 registers to exchange, to put them in alphabetical order. This means you must write `ACEX f` rather than `CAEX f`, as it will not recognize `CAEX f`.

We know we cannot load a value into D because there is no LD, so we can load nibbles into C or A and then exchange them with C, like:

```
LC 124
CDEX X
```

This exchanges 3 nibbles between C and D, thus "moving" #124h into D. If you want to exchange two registers, you will use the W field, of course. :)

33.1.5 Copying a register into another

Not only can we exchange two registers, but we can also directly overwrite nibbles from one to another one. Here are the instructions to do that:

Mnemonic	Opcode	Fields	Carry	Cycles
`A=B f`	D4 Ab4 80B084f +	A P,WP,XS,S,M,B,W F1-F7	no no no	8 4.5+*n*
`B=A f`	D5 Ab5 80B081f +	A P,WP,XS,S,M,B,W F1-F7	no no no	8 4.5+*n*
`C=B f`	D6 Ab6 80B086f +	A P,WP,XS,S,M,B,W F1-F7	no no no	8 4.5+*n*
`C=D f`	D7 Ab7 80B08Df +	A P,WP,XS,S,M,B,W F1-F7	no no no	8 4.5+*n*
`A=C f`	D8 Ab8 80B088f +	A P,WP,XS,S,M,B,W F1-F7	no no no	8 4.5+*n*
`B=C f`	D9 Ab9 80B089f +	A P,WP,XS,S,M,B,W F1-F7	no no no	8 4.5+*n*
`C=A f`	DA AbA 80B082f +	A P,WP,XS,S,M,B,W F1-F7	no no no	8 4.5+*n*
`D=A f`	80B083a + 80B083f +	A,P,WP,XS,S,M,B,W F1-F7	no no	
`D=B f`	80B087a + 80B087f +	A,P,WP,XS,S,M,B,W F1-F7	no no	
`D=C f`	DB AdB 80B08Bf +	A P,WP,XS,S,M,B,W F1-F7	no no no	8 4.5+*n*
`A=D f`	80B08Ca + 80B08Cf +	A,P,WP,XS,S,M,B,W F1-F7	no no	
`B=D f`	80B08Da + 80B08Df +	A,P,WP,XS,S,M,B,W F1-F7	no no	

That's a lot, isn't it? :-)

If you look closely, you'll see that on the regular Saturn, register D can only be changed with C.

```
           A
           |\
           | \
           |  \
       D----C---B
```

This diagram is valid for all instructions on A, B, C and D. **Remember it.** B can be used with A or C, A can be used with B or C, and C can be used with A, B, or D, so D can only be used with C.

On the Saturn+, the missing instructions have been added, so any of the above registers can be copied into any of the other above registers. These instructions, as noted earlier, are indicated with a plus sign (+) to the right of the opcode.

33.2 Working with registers

33.2.1 Incrementing a register

This is the same as adding 1 to a register. Here, the field will specify how many nibbles are used. For example, if I have #FFh in C and I do `C=C+1 B` I will get #00h in C and the carry set to 1 (because an overflow occurred). If I use the X field instead, I will get #100h.

Mnemonic	Opcode	Fields	Carry	Cycles
`A=A+1 f`	E4 Ba4 818f00 +	A P,WP,XS,S,M,B,W F1-F7	yes yes no	8 4.5+*n*
`B=B+1 f`	E5 Ba5 818f10 +	A P,WP,XS,S,M,B,W F1-F7	yes yes no	8 4.5+*n*
`C=C+1 f`	E6 Ba6 818f20 +	A P,WP,XS,S,M,B,W F1-F7	yes yes no	8 4.5+*n*
`D=D+1 f`	E7 Ba7 818f30 +	A P,WP,XS,S,M,B,W F1-F7	yes yes no	8 4.5+*n*

Example: I want to do a kind of "wait" loop. All I have to do is choose one register, like C. Now, I can choose a field, though not one that's too big, like W. Let's try A. We are going to load the value #FFFFFh into A, subtract one, and loop the subtraction until we reach 0.

First, for the purpose of this exercise it's better to do:

```
C=0 A
C=C-1 A
```

than:

```
LC FFFFF
```

Why?

The second method will need 7 nibbles to be coded (34FFFFF) and 8 cycles

The first method will only need 4 nibbles (D2CE) but more cycles: 14!

As you see, bigger code is sometimes faster, and if you want to save space, you may actually need more cycles.

So as we want a small code (although if you need to go fast, choose smart coding, and don't misuse memory) we're going to use:

```
C=0 A
C=C-1 A
```

Now, if we want to loop, we need a *label*. A label is a line which starts with * so the compiler will know we can "branch" to that label.

In this part, we're also going to use a new instruction:

```
GONC
```

`GONC` means: Go if No Carry

Here is the code:

```
C=0 A
C=C-1 A
*LOOP
C=C-1 A
GONC LOOP
A=DAT0 A
D0=D0+ 5
PC=(A)
@
```

First, we need to have #FFFFFh in C, so we turn C to 0 using field A, and then subtract 1 to get: #FFFFFh.

I also inserted a label called "LOOP." A label first has a star (*) and then its name. I called it "LOOP" but you can call it whatever you want.

Inside the loop, all we have to do is remove 1 from C(A).

C(A) means "field A of C"

The instruction GONC will branch, or go to, LOOP until the carry is set to 1. When will it be? When C(A) reaches zero, the next `C=C-1 A` will put C(A) from #00000h to #FFFFFh and the carry will be set to 1. Because the carry is set to 1, `GONC` will no longer jump to LOOP.

Then the program ends.

Try the source above and see. It waits and then quits.

You can use X or B if you want a loop that lasts less time.

We can write it another way: instead of checking whether the carry is set (that is, whether a negative overflow occurred) we could have checked whether C(A) was equal to zero, and loop until it's equal to zero, but the carry loop gives smaller and faster code.

33.2.2 Something special with the carry in loops

When you use the carry in loops, you have to pay attention to the fact that the loop will repeat until an overflow or underflow occurs; in my example, when C(A) reaches zero, it loops again, because the carry is not set to 1.

Take a look at this code:

```
LC F
*LOOP
C=C-1 P
GONC LOOP
```

The LOOP is executed F + 1 times. That's 16 times, not 15, because when C(P) reaches zero, it loops, and the carry only sets as the underflow occurs, when #0h becomes #Fh.

If we want a piece of code to loop four times, if we use the carry we will then load 3, not 4, inside the register that will be used as a counter:

```
LA 3
*LOOP
```

(Insert the code that has to be repeated four times here)

```
A=A-1 P
GONC LOOP
```

If this is difficult for you, ask for help in the comp.sys.hp48 newsgroup. Everything must always be clear. If you need help, just ask. Learning well will make you code well. ;)

33.2.3 Adding two registers

Here we add the contents of a register to another one, involving as many nibbles as the field uses. The carry will be affected, of course. For example, adding #Eh to #3h using the P field (with P=0) will set the carry bit.

Mnemonic	Opcode	Fields	Carry	Cycles
`A=A+B f`	C0	A	yes	8
	Aa0	P,WP,XS,S,M,B,W	yes	4.5+*n*
	80B184f +	F1-F7	no	
`B=B+C f`	C1	A	yes	8
	Aa1	P,WP,XS,S,M,B,W	yes	4.5+*n*
	80B189f +	F1-F7	no	
`C=C+A f`	C2	A	yes	8
	Aa2	P,WP,XS,S,M,B,W	yes	4.5+*n*
	80B182f +	F1-F7	no	
`D=D+C f`	C3	A	yes	8
	Aa3	P,WP,XS,S,M,B,W	yes	4.5+*n*
	80B18Bf +	F1-F7	no	
`B=B+A f`	C8	A	yes	8
	Aa8	P,WP,XS,S,M,B,W	yes	4.5+*n*
	80B181f +	F1-F7	no	
`C=C+B f`	C9	A	yes	8
	Aa9	P,WP,XS,S,M,B,W	yes	4.5+*n*
	80B186f +	F1-F7	no	
`A=A+C f`	CA	A	yes	8
	AaA	P,WP,XS,S,M,B,W	yes	4.5+*n*
	80B188f +	F1-F7	no	
`C=C+D f`	CB	A	yes	8
	AaB	P,WP,XS,S,M,B,W	yes	4.5+*n*
	80B18Ef +	F1-F7	no	
`D=D+A f`	80B183a +	A,P,WP,XS,S,M,B,W	no	
	80B183f +	F1-F7	no	
`D=D+B f`	80B187a +	A,P,WP,XS,S,M,B,W	no	
	80B187f +	F1-F7	no	
`A=A+D f`	80B18Ca +	A,P,WP,XS,S,M,B,W	no	
	80B18Cf +	F1-F7	no	
`B=B+D f`	80B18Da +	A,P,WP,XS,S,M,B,W	no	
	80B18Df +	F1-F7	no	

Notice that instructions such as `A=B+A f` are not listed above. This is because, in this specific example, it is identical to `A=A+B f`. When an opcode can be represented by multiple instructions, we will only list the instruction with the registers in alphabetical order in this book. You can always use the alternate instructions with both HP-ASM and MASD, however.

33.2.4 Adding a register to itself

This will be used to multiply the contents of a register, or, if we use a field other than W, a part of the register. Like before, the carry will be affected if there is an overflow when adding the register to itself on field f:

Mnemonic	Opcode	Fields	Carry	Cycles
`A=A+A f`	C4 Aa4 80B180f +	A P,WP,XS,S,M,B,W F1-F7	yes yes no	8 4.5+*n*
`B=B+B f`	C5 Aa5 80B185f +	A P,WP,XS,S,M,B,W F1-F7	yes yes no	8 4.5+*n*
`C=C+C f`	C6 Aa6 80B18Af +	A P,WP,XS,S,M,B,W F1-F7	yes yes no	8 4.5+*n*
`D=D+D f`	C7 Aa7 80B18Ff +	A P,WP,XS,S,M,B,W F1-F7	yes yes no	8 4.5+*n*

33.2.5 Adding a constant to a register

The instructions below let you add a constant to a register's contents using field f. Here we cannot use all fields we would like: we cannot use S, XS, WP or P in decimal mode.

THIS IS BECAUSE OF A BUG IN THE SATURN

The Saturn has a bug when running in decimal mode: **every** instruction that increments or decrements a register with a constant greater than **one** in fields S, XS, WP, or P works with the *entire* register rather than the specified field only. In hexadecimal mode, however, everything works as expected.

In other words, if a nibble overflows, setting the carry bit, the carry propagates to *all* the bits of the register. For example, suppose your A register looks like this:

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
F	F	F	F	F	F	F	F	F	F	F	F	F	E	F	2

If you perform the instruction `A=A+4 XS` you get this as a result:

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	2	F	3

Another example:

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
F	F	F	F	F	F	F	F	F	F	F	1	F	E	F	F

If you perform the instruction `A=A+4 XS` you get this as a result:

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
F	F	F	F	F	F	F	F	F	F	F	2	0	2	F	F

SOLUTION:

If you are working with *one* nibble using field S, XS, WP (with P=0), or P and want to add *more* than 1, you **must** add only 1 as many times as needed.

Example:

```
A=A+2 S
```

will not work! If the carry is set, you will lose the contents of the register!

```
A=A+1 S
A=A+1 S
```

is just fine. :) If the carry is set because of an overflow, everything will be fine.

This also true for:

```
A=A-2 S
```

which will not work either.

```
A=A-1 S
A=A-1 S
```

is okay!

REMEMBER:	When working with a field that deals with *one* nibble, if you want to add or subtract *more* than 1, you'll have to do several +1's or -1's to avoid the carry bug.

Note that this bug has been fixed in the Saturn+ emulated processor in the ARM-based calculators. However, it is best to pay attention to this anyway if you want to create backwards-compatible code.

When you add 3 to a constant, using `A=A+3 f`, if you want to know how it's coded you have to remove 1 from the constant to find its value in the hex form. Thus, `A=A+3 A` will be coded 818F02 because we have a 2 instead of a 3, because it's c+1. This only matters if you are reading the hex form of instructions that add a constant to a register.

When you code, and write, for example, `A=A+3 A`, the assembler will do all the work, encoding "818F02," so you aren't bothered by that (c+1).

A trick from Gerald Squelart: To set a value in a one-nibble field, you may do:

```
A=0 S
A=A+2 S
```

This works because there's no carry.

Mnemonic	**Opcode**	**Fields**	**Carry**	**Cycles**

`A=A+c f`	818F0(c-1)
	818a0(c-1)
	818f0(c-1) +
		A
		W,M,X,B
		F1-F7
			yes
			yes
			no
				8+*n*
				8+*n*

`B=B+c f`	818F1(c-1)
	818a1(c-1)
	818f1(c-1) +
		A
		W,M,X,B
		F1-F7
			yes
			yes
			no
				8+*n*
				8+*n*

`C=C+c f`	818F2(c-1)
	818a2(c-1)
	818f2(c-1) +
		A
		W,M,X,B
		F1-F7
			yes
			yes
			no
				8+*n*
				8+*n*

`D=D+c f`	818F3(c-1)
	818a3(c-1)
	818f3(c-1) +
		A
		W,M,X,B
		F1-F7
			yes
			yes
			no
				8+*n*
				8+*n*

n = number of nibbles in the object that is being added to.

2 < c < 16

If you only want to add 1, it's coded different from here (see section 33.2.1), but when you write your programs, the assembler automatically chooses the right instruction. :-)

In the beginning coders had to encode something+1 or something+c with c between 2 or 16. Today, with assemblers, we just add any value from 1 to 16, and the assembler does the work for us.

Here's a cool trick!

You know each nibble has a "weight," so you can alter P and add a value to a specific nibble, doing much more than simply adding a constant c. The nibble #2h (the third nibble in the register) has a weight of 16^2, so each time 1 is added to that nibble, 16^2, or 256, is added to the value.

So, sometimes (but not always) it's better to add a value to nibble *n* of a register rather than adding a big value to the whole register.

33.2.6 Decrementing a register

This is the same as removing one, so it's easy.

Mnemonic	Opcode		Fields	Carry	Cycles
`A=A-1 f`	CC		A	yes	8
	AaC		P,WP,XS,S,M,B,W	yes	4.5+*n*
	818f80	+	F1-F7	no	
`B=B-1 f`	CD		A	yes	8
	AaD		P,WP,XS,S,M,B,W	yes	4.5+*n*
	818f90	+	F1-F7	no	
`C=C-1 f`	CE		A	yes	8
	AaE		P,WP,XS,S,M,B,W	yes	4.5+*n*
	818fA0	+	F1-F7	no	
`D=D-1 f`	CF		A	yes	8
	AaF		P,WP,XS,S,M,B,W	yes	4.5+*n*
	818fB0	+	F1-F7	no	

The carry bit will be set if we try to remove 1 from any field that is equal to zero.

33.2.7 Subtracting two registers

Well...it's subtracting one register from another.

Mnemonic	Opcode		Fields	Carry	Cycles
`A=A-B f`	E0		A	yes	8
	Ba0		P,WP,XS,S,M,B,W	yes	4.5+*n*
	80B284f	+	F1-F7	no	
`B=B-C f`	E1		A	yes	8
	Ba1		P,WP,XS,S,M,B,W	yes	4.5+*n*
	80B289f	+	F1-F7	no	

Mnemonic	Opcode	Fields	Carry	Cycles
C=C-A f	E2 Ba2 80B282f +	A P,WP,XS,S,M,B,W F1-F7	yes yes no	8 4.5+*n*
D=D-C f	E3 Ba3 80B28Bf +	A P,WP,XS,S,M,B,W F1-F7	yes yes no	8 4.5+*n*
B=B-A f	E8 Ba8 80B281f +	A P,WP,XS,S,M,B,W F1-F7	yes yes no	8 4.5+*n*
C=C-B f	E9 Ba9 80B286f +	A P,WP,XS,S,M,B,W F1-F7	yes yes no	8 4.5+*n*
A=A-C f	EA BaA 80B288f +	A P,WP,XS,S,M,B,W F1-F7	yes yes no	8 4.5+*n*
C=C-D f	EB BaB 80B28Ef +	A P,WP,XS,S,M,B,W F1-F7	yes yes no	8 4.5+*n*
A=B-A f	EC BaC	A P,WP,XS,S,M,B,W	yes yes	8 4.5+*n*
B=C-B f	ED BaD	A P,WP,XS,S,M,B,W	yes yes	8 4.5+*n*
C=A-C f	EE BaE	A P,WP,XS,S,M,B,W	yes yes	8 4.5+*n*
D=C-D f	EF BaF	A P,WP,XS,S,M,B,W	yes yes	8 4.5+*n*
D=D-A f	80B283a + 80B283f +	A, P,WP,XS,S,M,B,W F1-F7	no no	
D=D-B f	80B287a + 80B287f +	A, P,WP,XS,S,M,B,W F1-F7	no no	
A=A-D f	80B28Ca + 80B28Cf +	A, P,WP,XS,S,M,B,W F1-F7	no no	
B=B-D f	80B28Da + 80B28Df +	A, P,WP,XS,S,M,B,W F1-F7	no no	

As always, *n* is the number of nibbles and the carry will be set if there is an underflow.

33.2.8 Multiplying two registers

These multiplication instructions are new in the ARM-based calculators and are therefore not available in the HP 48 series or the 49G. On those calculators you will have to do it by hand as per section 7.4. You can also call the routine =MUL# (at address #03991h on both the 48 and 49), which will multiply A(A) by C(A) and put the product into B(A).

Mnemonic	Opcode	Fields	Carry	Cycles
B=B*A f	80B381a + 80B381f +	A,P,WP,XS,S,M,B,W F1-F7	no no	
C=C*A f	80B382a + 80B382f +	A,P,WP,XS,S,M,B,W F1-F7	no no	
D=D*A f	80B383a + 80B383f +	A, P,WP,XS,S,M,B,W F1-F7	no no	
A=A*B f	80B384a + 80B384f +	A,P,WP,XS,S,M,B,W F1-F7	no no	
B=B*B f	80B385a + 80B385f +	A,P,WP,XS,S,M,B,W F1-F7	no no	

Mnemonic	Opcode	Fields	Carry	Cycles
`C=C*B f`	80B386a +	A,P,WP,XS,S,M,B,W	no	
	80B386f +	F1-F7	no	
`D=D*B f`	80B387a +	A, P,WP,XS,S,M,B,W	no	
	80B387f +	F1-F7	no	
`A=A*C f`	80B388a +	A,P,WP,XS,S,M,B,W	no	
	80B388f +	F1-F7	no	
`B=B*C f`	80B389a +	A,P,WP,XS,S,M,B,W	no	
	80B389f +	F1-F7	no	
`A=A*A f`	80B380a +	A,P,WP,XS,S,M,B,W	no	
	80B380f +	F1-F7	no	
`C=C*C f`	80B38Aa +	A,P,WP,XS,S,M,B,W	no	
	80B38Af +	F1-F7	no	
`D=D*C f`	80B38Ba +	A,P,WP,XS,S,M,B,W	no	
	80B38Bf +	F1-F7	no	
`A=A*D f`	80B38Ca +	A, P,WP,XS,S,M,B,W	no	
	80B38Cf +	F1-F7	no	
`B=B*D f`	80B38Da +	A, P,WP,XS,S,M,B,W	no	
	80B38Df +	F1-F7	no	
`C=C*D f`	80B38Ea +	A,P,WP,XS,S,M,B,W	no	
	80B38Ef +	F1-F7	no	
`D=D*D f`	80B38Fa +	A,P,WP,XS,S,M,B,W	no	
	80B38Ff +	F1-F7	no	

33.2.9 Dividing two registers

These division instructions are also new in the ARM-based calculators and are therefore not available in the HP 48 series or the 49G. On those calculators you will have to do it by hand as per section 7.5. You can also call the routine =IntDiv (at address #03F24h on both the 48 and 49), which will divide A(A) by C(A) and put the dividend into C(A).

Mnemonic	Opcode	Fields	Carry	Cycles
`B=B/A f`	80B481a +	A,P,WP,XS,S,M,B,W	no	
	80B481f +	F1-F7	no	
`C=C/A f`	80B482a +	A,P,WP,XS,S,M,B,W	no	
	80B482f +	F1-F7	no	
`D=D/A f`	80B483a +	A, P,WP,XS,S,M,B,W	no	
	80B483f +	F1-F7	no	
`A=A/B f`	80B484a +	A,P,WP,XS,S,M,B,W	no	
	80B484f +	F1-F7	no	
`B=1 f`	80B485a +	A,P,WP,XS,S,M,B,W	no	
	80B485f +	F1-F7	no	
`C=C/B f`	80B486a +	A,P,WP,XS,S,M,B,W	no	
	80B486f +	F1-F7	no	
`D=D/B f`	80B487a +	A, P,WP,XS,S,M,B,W	no	
	80B487f +	F1-F7	no	
`A=A/C f`	80B488a +	A,P,WP,XS,S,M,B,W	no	
	80B488f +	F1-F7	no	
`B=B/C f`	80B489a +	A,P,WP,XS,S,M,B,W	no	
	80B489f +	F1-F7	no	
`A=1 f`	80B480a +	A,P,WP,XS,S,M,B,W	no	
	80B480f +	F1-F7	no	
`C=1 f`	80B48Aa +	A,P,WP,XS,S,M,B,W	no	
	80B48Af +	F1-F7	no	
`D=D/C f`	80B48Ba +	A,P,WP,XS,S,M,B,W	no	
	80B48Bf +	F1-F7	no	

Mnemonic	Opcode	Fields	Carry	Cycles
`A=A/D f`	80B48Ca + 80B48Cf +	A, P,WP,XS,S,M,B,W F1-F7	no no	
`B=B/D f`	80B48Da + 80B48Df +	A, P,WP,XS,S,M,B,W F1-F7	no no	
`C=C/D f`	80B48Ea + 80B48Ef +	A,P,WP,XS,S,M,B,W F1-F7	no no	
`D=1 f`	80B48Fa + 80B48Ff +	A,P,WP,XS,S,M,B,W F1-F7	no no	

Notice that I have listed more useful mnemonics for some of the instructions.

33.2.10 Modulo of two registers

These modulo (remainder after division) instructions are also new in the ARM-based calculators and are therefore not available in the HP 48 series or the 49G. On those calculators you will have to do it by hand as per section 7.5. You can also call the routine =IntDiv (at address #03F24h on both the 48 and 49), which will divide A(A) by C(A) and put the remainder into A(A).

Mnemonic	Opcode	Fields	Carry	Cycles
`B=B%A f`	80B581a + 80B581f +	A,P,WP,XS,S,M,B,W F1-F7	no no	
`C=C%A f`	80B582a + 80B582f +	A,P,WP,XS,S,M,B,W F1-F7	no no	
`D=D%A f`	80B583a + 80B583f +	A, P,WP,XS,S,M,B,W F1-F7	no no	
`A=A%B f`	80B584a + 80B584f +	A,P,WP,XS,S,M,B,W F1-F7	no no	
`B=B%B f`	80B585a + 80B585f +	A,P,WP,XS,S,M,B,W F1-F7	no no	
`C=C%B f`	80B586a + 80B586f +	A,P,WP,XS,S,M,B,W F1-F7	no no	
`D=D%B f`	80B587a + 80B587f +	A, P,WP,XS,S,M,B,W F1-F7	no no	
`A=A%C f`	80B588a + 80B588f +	A,P,WP,XS,S,M,B,W F1-F7	no no	
`B=B%C f`	80B589a + 80B589f +	A,P,WP,XS,S,M,B,W F1-F7	no no	
`A=A%A f`	80B580a + 80B580f +	A,P,WP,XS,S,M,B,W F1-F7	no no	
`C=C%C f`	80B58Aa + 80B58Af +	A,P,WP,XS,S,M,B,W F1-F7	no no	
`D=D%C f`	80B58Ba + 80B58Bf +	A,P,WP,XS,S,M,B,W F1-F7	no no	
`A=A%D f`	80B58Ca + 80B58Cf +	A, P,WP,XS,S,M,B,W F1-F7	no no	
`B=B%D f`	80B58Da + 80B58Df +	A, P,WP,XS,S,M,B,W F1-F7	no no	
`C=C%D f`	80B58Ea + 80B58Ef +	A,P,WP,XS,S,M,B,W F1-F7	no no	
`D=D%D f`	80B58Fa + 80B58Ff +	A,P,WP,XS,S,M,B,W F1-F7	no no	

The instructions `A=A%A`, `B=B%B`, `C=C%C`, and `D=D%D` don't seem particularly useful, but they are supported by the calculator so I am listing them anyway.

33.2.11 Subtracting a constant

Mnemonic	Opcode		Fields	Carry	Cycles
`A=A-c f`	818F8(c-1)		A	yes	8+*n*
	818a8(c-1)		W,M,X,B	yes	8+*n*
	818f8(c-1)	+	F1-F7	no	
`B=B-c f`	818F9(c-1)		A	yes	8+*n*
	818a9(c-1)		W,M,X,B	yes	8+*n*
	818f9(c-1)	+	F1-F7	no	
`C=C-c f`	818FA(c-1)		A	yes	8+*n*
	818aA(c-1)		W,M,X,B	yes	8+*n*
	818fA(c-1)	+	F1-F7	no	
`D=D-c f`	818FB(c-1)		A	yes	8+*n*
	818aB(c-1)		W,M,X,B	yes	8+*n*
	818fB(c-1)	+	F1-F7	no	

We cannot use all fields here because of the carry bug. If you want to subtract c with c > 1, you'll **only** be able to use 1, repeated as many times as needed, to avoid the bug.

You may use the same trick here that was given before, so you can work with a specific nibble by modifying P. By using the weight of each nibble, you can subtract a big value.

33.2.12 One's Complement

NOTE:	The result you will get using one's complement will depend on the *mode* the Saturn is under: decimal or hexadecimal. The instructions to change this mode are explained below. Remember that when your program starts, the processor will *always* be in hexadecimal mode, and when it quits, it *must* also be in hexadecimal mode.

Mnemonic	Opcode		Fields	Carry	Cycles
`A=-A-1 f`	FC		A	yes	8
	BbC		P,WP,XS,S,M,B,W	yes	4.5+*n*
	80B680f	+	F1-F7	no	
`B=-B-1 f`	FD		A	yes	8
	BbD		P,WP,XS,S,M,B,W	yes	4.5+*n*
	80B685f	+	F1-F7	no	
`C=-C-1 f`	FE		A	yes	8
	BbE		P,WP,XS,S,M,B,W	yes	4.5+*n*
	80B68Af	+	F1-F7	no	
`D=-D-1 f`	FF		A	yes	8
	BbF		P,WP,XS,S,M,B,W	yes	4.5+*n*
	80B68Ff	+	F1-F7	no	
`B=-A-1 f`	80B681a	+	A,P,WP,XS,S,M,B,W	no	
	80B681f	+	F1-F7	no	
`C=-A-1 f`	80B682a	+	A,P,WP,XS,S,M,B,W	no	
	80B682f	+	F1-F7	no	
`D=-D-1 f`	80B683a	+	A,P,WP,XS,S,M,B,W	no	
	80B683f	+	F1-F7	no	
`A=-B-1 f`	80B684a	+	A,P,WP,XS,S,M,B,W	no	
	80B684f	+	F1-F7	no	
`C=-B-1 f`	80B686a	+	A,P,WP,XS,S,M,B,W	no	
	80B686f	+	F1-F7	no	
`D=-B-1 f`	80B687a	+	A,P,WP,XS,S,M,B,W	no	
	80B687f	+	F1-F7	no	

Mnemonic	Opcode		Fields	Carry	Cycles
`A=-C-1 f`	80B688a 80B688f	+ +	A,P,WP,XS,S,M,B,W F1-F7	no no	
`A=-B-1 f`	80B689a 80B689f	+ +	A,P,WP,XS,S,M,B,W F1-F7	no no	
`D=-C-1 f`	80B68Ba 80B68Bf	+ +	A,P,WP,XS,S,M,B,W F1-F7	no no	
`A=-D-1 f`	80B68Ca 80B68Cf	+ +	A,P,WP,XS,S,M,B,W F1-F7	no no	
`B=-D-1 f`	80B68Da 80B68Df	+ +	A,P,WP,XS,S,M,B,W F1-F7	no no	
`C=-D-1 f`	80B68Ea 80B68Ef	+ +	A,P,WP,XS,S,M,B,W F1-F7	no no	

If the register (and thus the field) you are working with equals zero, then doing a `r=-r-1 f` (with r being the register) sets the carry.

An alternate syntax for `r=-r-1 f` in MASD is `r=~r f`. In other words, instead of `A=-A-1 B` you can write `A=~A B`. You can type ~ with [ALPHA][Right-Shift][1].

33.2.13 Two's Complement

As before, the result here depends on the mode the Saturn is running under. All the bits are here inverted:

Mnemonic	Opcode		Fields	Carry	Cycles
`A=-A f`	F8 Bb8 80B780f	 +	A P,WP,XS,S,M,B,W F1-F7	yes yes no	8 4.5+*n*
`B=-B f`	F9 Bb9 80B785f	 +	A P,WP,XS,S,M,B,W F1-F7	yes yes no	8 4.5+*n*
`C=-C f`	FA BbA 80B78Af	 +	A P,WP,XS,S,M,B,W F1-F7	yes yes no	8 4.5+*n*
`D=-D f`	FB BbB 80B78Ff	 +	A P,WP,XS,S,M,B,W F1-F7	yes yes no	8 4.5+*n*
`B=-A f`	80B781a 80B781f	+ +	A,P,WP,XS,S,M,B,W F1-F7	no no	
`C=-A f`	80B782a 80B782f	+ +	A,P,WP,XS,S,M,B,W F1-F7	no no	
`D=-D f`	80B783a 80B783f	+ +	A,P,WP,XS,S,M,B,W F1-F7	no no	
`A=-B f`	80B784a 80B784f	+ +	A,P,WP,XS,S,M,B,W F1-F7	no no	
`C=-B f`	80B786a 80B786f	+ +	A,P,WP,XS,S,M,B,W F1-F7	no no	
`D=-B f`	80B787a 80B787f	+ +	A,P,WP,XS,S,M,B,W F1-F7	no no	
`A=-C f`	80B788a 80B788f	+ +	A,P,WP,XS,S,M,B,W F1-F7	no no	
`A=-B f`	80B789a 80B789f	+ +	A,P,WP,XS,S,M,B,W F1-F7	no no	
`D=-C f`	80B78Ba 80B78Bf	+ +	A,P,WP,XS,S,M,B,W F1-F7	no no	

Mnemonic	Opcode	Fields	Carry	Cycles
`A=-D f`	80B78Ca +	A,P,WP,XS,S,M,B,W	no	
	80B78Cf +	F1-F7	no	
`B=-D f`	80B78Da +	A,P,WP,XS,S,M,B,W	no	
	80B78Df +	F1-F7	no	
`C=-D f`	80B78Ea +	A,P,WP,XS,S,M,B,W	no	
	80B78Ef +	F1-F7	no	

33.2.14 Logical operations

33.2.14.1 Logical OR (also called "logical sum")

With logical OR, the "!" symbol is used. The OR is done bit per bit with field f on the register used. The result is placed on the register *to the left* of the equal sign:

Mnemonic	Opcode	Fields	Carry	Cycles
`A=A!B f`	0EF8	A	no	11
	0Ea8	P,WP,XS,S,M,B,W	no	6+*n*
	0Ef8 +	F1-F7	no	
`B=B!C f`	0EF9	A	no	11
	0Ea9	P,WP,XS,S,M,B,W	no	6+*n*
	0Ef9 +	F1-F7	no	
`C=C!A f`	0EFA	A	no	11
	0EaA	P,WP,XS,S,M,B,W	no	6+*n*
	0EfA +	F1-F7	no	
`D=D!C f`	0EFB	A	no	11
	0EaB	P,WP,XS,S,M,B,W	no	6+*n*
	0EfB +	F1-F7	no	
`B=B!A f`	0EFC	A	no	11
	0EaC	P,WP,XS,S,M,B,W	no	6+*n*
	0EfC +	F1-F7	no	
`C=C!B f`	0EFD	A	no	11
	0EaD	P,WP,XS,S,M,B,W	no	6+*n*
	0EfD +	F1-F7	no	
`A=A!C f`	0EFE	A	no	11
	0EaE	P,WP,XS,S,M,B,W	no	6+*n*
	0EfE +	F1-F7	no	
`C=C!D f`	0EFF	A	no	11
	0EaF	P,WP,XS,S,M,B,W	no	6+*n*
	0EfF +	F1-F7	no	

33.2.14.2 Logical AND (also called "logical product")

The symbol used is "&".

Mnemonic	Opcode	Fields	Carry	Cycles
`A=A&B f`	0EF0	A	no	11
	0Ea0	P,WP,XS,S,M,B,W	no	6+*n*
	0Ef0 +	F1-F7	no	
`B=B&C f`	0EF1	A	no	11
	0Ea1	P,WP,XS,S,M,B,W	no	6+*n*
	0Ef1 +	F1-F7	no	
`C=C&A f`	0EF2	A	no	11
	0Ea2	P,WP,XS,S,M,B,W	no	6+*n*
	0Ef2 +	F1-F7	no	

Mnemonic	Opcode	Fields	Carry	Cycles
`D=D&C f`	0EF3 0Ea3 0Ef3 +	A P,WP,XS,S,M,B,W F1-F7	no no no	11 6+*n*
`B=B&A f`	0EF4 0Ea4 0Ef4 +	A P,WP,XS,S,M,B,W F1-F7	no no no	11 6+*n*
`C=C&B f`	0EF5 0Ea5 0Ef5 +	A P,WP,XS,S,M,B,W F1-F7	no no no	11 6+*n*
`A=A&C f`	0EF6 0Ea6 0Ef6 +	A P,WP,XS,S,M,B,W F1-F7	no no no	11 6+*n*
`C=C&D f`	0EF7 0Ea7 0Ef7 +	A P,WP,XS,S,M,B,W F1-F7	no no no	11 6+*n*

33.2.14.3 Logical XOR

With logical XOR, the ^ (caret) symbol is used. These XOR instructions are new in the ARM-based calculators and are therefore not available in the HP 48 series or the 49G. On those calculators you will have to do it by hand as per section 16.6.

Mnemonic	Opcode	Fields	Carry	Cycles
`B=B^A f`	80BA81a + 80BA81f +	A,P,WP,XS,S,M,B,W F1-F7	no no	
`C=C^A f`	80BA82a + 80BA82f +	A,P,WP,XS,S,M,B,W F1-F7	no no	
`D=D^A f`	80BA83a + 80BA83f +	A, P,WP,XS,S,M,B,W F1-F7	no no	
`A=A^B f`	80BA84a + 80BA84f +	A,P,WP,XS,S,M,B,W F1-F7	no no	
`B=B^B f`	80BA85a + 80BA85f +	A,P,WP,XS,S,M,B,W F1-F7	no no	
`C=C^B f`	80BA86a + 80BA86f +	A,P,WP,XS,S,M,B,W F1-F7	no no	
`D=D^B f`	80BA87a + 80BA87f +	A, P,WP,XS,S,M,B,W F1-F7	no no	
`A=A^C f`	80BA88a + 80BA88f +	A,P,WP,XS,S,M,B,W F1-F7	no no	
`B=B^C f`	80BA89a + 80BA89f +	A,P,WP,XS,S,M,B,W F1-F7	no no	
`A=A^A f`	80BA80a + 80BA80f +	A,P,WP,XS,S,M,B,W F1-F7	no no	
`C=C^C f`	80BA8Aa + 80BA8Af +	A,P,WP,XS,S,M,B,W F1-F7	no no	
`D=D^C f`	80BA8Ba + 80BA8Bf +	A,P,WP,XS,S,M,B,W F1-F7	no no	
`A=A^D f`	80BA8Ca + 80BA8Cf +	A, P,WP,XS,S,M,B,W F1-F7	no no	
`B=B^D f`	80BA8Da + 80BA8Df +	A, P,WP,XS,S,M,B,W F1-F7	no no	
`C=C^D f`	80BA8Ea + 80BA8Ef +	A,P,WP,XS,S,M,B,W F1-F7	no no	
`D=D^D f`	80BA8Fa + 80BA8Ff +	A,P,WP,XS,S,M,B,W F1-F7	no no	

33.2.15 Shifting one nibble left or right

There are four instructions to shift one nibble to the left, which multiplies the value by 16, and four others to shift one nibble to the right, which divides the value by 16.

When we shift to the left, all nibbles are moved one nibble to the left and the last nibble (number #Fh) is lost. Nibble number #0h receives a null nibble (value is #0h).

When we shift to the right, all nibbles are moved one nibble to the right and the last nibble (number #Fh) receives a null nibble. The first nibble (number #0h) is lost.

First we write the register's letter, then S (for Shift), and finally the letter of the direction, with R for right and L for left.

For example, the mnemonic to move the C register right is C + S + R, so `CSR f`.

Mnemonic	**Opcode**	**Fields**	**Carry**	**Cycles**
`ASL f`	F0 Bb0	A P,WP,XS,S,M,B,W	no no	8 4.5+*n*
`BSL f`	F1 Bb1	A P,WP,XS,S,M,B,W	no no	8 4.5+*n*
`CSL f`	F2 Bb2	A P,WP,XS,S,M,B,W	no no	8 4.5+*n*
`DSL f`	F3 Bb3	A P,WP,XS,S,M,B,W	no no	8 4.5+*n*
`ASR f`	F4 Bb4	A P,WP,XS,S,M,B,W	no no	8 4.5+*n*
`BSR f`	F5 Bb5	A P,WP,XS,S,M,B,W	no no	8 4.5+*n*
`CSR f`	F6 Bb6	A P,WP,XS,S,M,B,W	no no	8 4.5+*n*
`DSR f`	F7 Bb7	A P,WP,XS,S,M,B,W	no no	8 4.5+*n*

Here the carry isn't affected, *but* the SB (Sticky Bit) will be. If the nibble lost (on the left or right, depending on the direction we are shifting bits to) has a value other than zero, SB is *set*. It means there has been an important (at least, to be noticed) loss of value.

33.2.16 Rotating one nibble left or right

To rotate a nibble, we first write the letter of the register, then the letter S (for Shift), then the direction (R or L) and finally the letter C (Circling).

We thus have:

Mnemonic	**Opcode**	**Fields**	**Carry**	**Cycles**
`ASLC`	810	All	no	22.5
`BSLC`	811	All	no	22.5
`CSLC`	812	All	no	22.5
`DSLC`	813	All	no	22.5
`ASRC`	814	All	no	22.5
`BSRC`	815	All	no	22.5
`CSRC`	816	All	no	22.5
`DSRC`	817	All	no	22.5

When we rotate to the right, the contents of nibble number #1h move to nibble #0h and nibble #0h moves to nibble #Fh.

To give you an example, this could be the content of a register before rotating to the right:

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
3	A	8	B	B	3	8	2	0	0	0	A	F	0	F	C

This is what the register would look like after rotating to the right:

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
C	3	A	8	B	B	3	8	2	0	0	0	A	F	0	F

Here, there is no loss of information because nibble #0h has moved to the other side of the register, becoming nibble #Fh.

The Sticky Bit will only be set if the bits of the nibble number #0h (which goes to nibble number #Fh) are null.

33.2.17 Shifting one bit to the right

Shifting one bit to the right is like dividing by 2, as you should know. To form the mnemonics, simply put the letter of the register, then “S” for Shifting, then the direction (R for right) and then B (for Bit).

Mnemonic	Opcode	Fields	Carry	Cycles
`ASRB`	81C	All	no	21.5
`BSRB`	81D	All	no	21.5
`CSRB`	81E	All	no	21.5
`DSRB`	81F	All	no	21.5

Here, all 64 bits are moved to the right. Even though the carry bit is not modified, if bit number #0h is lost and was equal to 1, then the Sticky Bit is set.

Note that there are no instructions to shift one bit to the left.

33.2.17.1 Shifting one bit to the right within a field

This is like the previous commands, but here we use a field, so the shifting only occurs inside the field selected. All we have to do is add the letter of the field, like `ASRB A`.

Mnemonic	Opcode	Fields	Carry	Cycles
`ASRB f`	819F0 819a0 819f0 +	A P,WP,XS,S,M,B F1-F7	no no no	13.5 8.5+*n*
`BSRB f`	819F1 819a1 819f1 +	A P,WP,XS,S,M,B F1-F7	no no no	13.5 8.5+*n*
`CSRB f`	819F2 819a2 819f2 +	A P,WP,XS,S,M,B F1-F7	no no no	13.5 8.5+*n*
`DSRB f`	819F3 819a3 819f3 +	A P,WP,XS,S,M,B F1-F7	no no no	13.5 8.5+*n*

These instructions divide the value of field f by 2.

The carry is not modified here, but if the bit lost on the right of the field used is equal to 1, then the SB (Sticky Bit) is set. The bit inserted on the left is 0.

Note that there are no instructions to shift one bit in a field to the left, either, on the original Saturn.

33.2.18 Shifting to the left

These left bit-shifting instructions shift the register before the less-than sign left by *n* bits, where *n* is the value in the register after the less-than sign. These are new in the ARM-based calculators and are therefore not available in the HP 48 series or the 49G.

Mnemonic	Opcode		Fields	Carry	Cycles
`B=B<A f`	80B881a	+	A,P,WP,XS,S,M,B,W	no	
	80B881f	+	F1-F7	no	
`C=C<A f`	80B882a	+	A,P,WP,XS,S,M,B,W	no	
	80B882f	+	F1-F7	no	
`D=D<A f`	80B883a	+	A, P,WP,XS,S,M,B,W	no	
	80B883f	+	F1-F7	no	
`A=A<B f`	80B884a	+	A,P,WP,XS,S,M,B,W	no	
	80B884f	+	F1-F7	no	
`B=B<B f`	80B885a	+	A,P,WP,XS,S,M,B,W	no	
	80B885f	+	F1-F7	no	
`C=C<B f`	80B886a	+	A,P,WP,XS,S,M,B,W	no	
	80B886f	+	F1-F7	no	
`D=D<B f`	80B887a	+	A, P,WP,XS,S,M,B,W	no	
	80B887f	+	F1-F7	no	
`A=A<C f`	80B888a	+	A,P,WP,XS,S,M,B,W	no	
	80B888f	+	F1-F7	no	
`B=B<C f`	80B889a	+	A,P,WP,XS,S,M,B,W	no	
	80B889f	+	F1-F7	no	
`A=A<A f`	80B880a	+	A,P,WP,XS,S,M,B,W	no	
	80B880f	+	F1-F7	no	
`C=C<C f`	80B88Aa	+	A,P,WP,XS,S,M,B,W	no	
	80B88Af	+	F1-F7	no	
`D=D<C f`	80B88Ba	+	A,P,WP,XS,S,M,B,W	no	
	80B88Bf	+	F1-F7	no	
`A=A<D f`	80B88Ca	+	A, P,WP,XS,S,M,B,W	no	
	80B88Cf	+	F1-F7	no	
`B=B<D f`	80B88Da	+	A, P,WP,XS,S,M,B,W	no	
	80B88Df	+	F1-F7	no	
`C=C<D f`	80B88Ea	+	A,P,WP,XS,S,M,B,W	no	
	80B88Ef	+	F1-F7	no	
`D=D<D f`	80B88Fa	+	A,P,WP,XS,S,M,B,W	no	
	80B88Ff	+	F1-F7	no	

33.2.19 Shifting to the right

These right bit-shifting instructions shift the register before the greater-than sign right by *n* bits, where *n* is the value in the register after the greater-than sign. These are new in the ARM-based calculators and are therefore not available in the HP 48 series or the 49G.

Mnemonic	Opcode		Fields	Carry	Cycles
`B=B>A f`	80B981a	+	A,P,WP,XS,S,M,B,W	no	
	80B981f	+	F1-F7	no	
`C=C>A f`	80B982a	+	A,P,WP,XS,S,M,B,W	no	
	80B982f	+	F1-F7	no	
`D=D>A f`	80B983a	+	A, P,WP,XS,S,M,B,W	no	
	80B983f	+	F1-F7	no	
`A=A>B f`	80B984a	+	A,P,WP,XS,S,M,B,W	no	
	80B984f	+	F1-F7	no	

Mnemonic	Opcode	Fields	Carry	Cycles
`B=B>B f`	80B985a + 80B985f +	A,P,WP,XS,S,M,B,W F1-F7	no no	
`C=C>B f`	80B986a + 80B986f +	A,P,WP,XS,S,M,B,W F1-F7	no no	
`D=D>B f`	80B987a + 80B987f +	A, P,WP,XS,S,M,B,W F1-F7	no no	
`A=A>C f`	80B988a + 80B988f +	A,P,WP,XS,S,M,B,W F1-F7	no no	
`B=B>C f`	80B989a + 80B989f +	A,P,WP,XS,S,M,B,W F1-F7	no no	
`A=A>A f`	80B980a + 80B980f +	A,P,WP,XS,S,M,B,W F1-F7	no no	
`C=C>C f`	80B98Aa + 80B98Af +	A,P,WP,XS,S,M,B,W F1-F7	no no	
`D=D>C f`	80B98Ba + 80B98Bf +	A,P,WP,XS,S,M,B,W F1-F7	no no	
`A=A>D f`	80B98Ca + 80B98Cf +	A, P,WP,XS,S,M,B,W F1-F7	no no	
`B=B>D f`	80B98Da + 80B98Df +	A, P,WP,XS,S,M,B,W F1-F7	no no	
`C=C>D f`	80B98Ea + 80B98Ef +	A,P,WP,XS,S,M,B,W F1-F7	no no	
`D=D>D f`	80B98Fa + 80B98Ff +	A,P,WP,XS,S,M,B,W F1-F7	no no	

34 Save registers

You are going to discover here that registers A and C are really important for the processor. :)

Only A and C can work with the save registers, of which there are five: R0, R1, R2, R3 and R4.

With these mnemonics, the register on the *left* receives the value of the register on the *right* (which is **unmodified**).

34.1 Saving a working register into a save register

Mnemonic	Opcode	Fields	Carry	Cycles
`R0=A`	100	All	no	20.5
`R1=A`	101	All	no	20.5
`R2=A`	102	All	no	20.5
`R3=A`	103	All	no	20.5
`R4=A`	104	All	no	20.5
`R0=C`	108	All	no	20.5
`R1=C`	109	All	no	20.5
`R2=C`	10A	All	no	20.5
`R3=C`	10B	All	no	20.5
`R4=C`	10C	All	no	20.5

34.2 Saving a working register into a save register within a field

Mnemonic	Opcode	Fields	Carry	Cycles
`R0=A f`	81AF00 81Aa00 81Af00 +	A P,WP,XS,S,M,B,W F1-F7	no no no	14 9+*n*
`R1=A f`	81AF01 81Aa01 81Af01 +	A P,WP,XS,S,M,B,W F1-F7	no no no	14 9+*n*
`R2=A f`	81AF02 81Aa02 81Af02 +	A P,WP,XS,S,M,B,W F1-F7	no no no	14 9+*n*
`R3=A f`	81AF03 81Aa03 81Af03 +	A P,WP,XS,S,M,B,W F1-F7	no no no	14 9+*n*
`R4=A f`	81AF04 81Aa04 81Af04 +	A P,WP,XS,S,M,B,W F1-F7	no no no	14 9+*n*
`R0=C f`	81AF08 81Aa08 81Af08 +	A P,WP,XS,S,M,B,W F1-F7	no no no	14 9+*n*
`R1=C f`	81AF09 81Aa09 81Af09 +	A P,WP,XS,S,M,B,W F1-F7	no no no	14 9+*n*
`R2=C f`	81AF0A 81Aa0A 81Af0A +	A P,WP,XS,S,M,B,W F1-F7	no no no	14 9+*n*
`R3=C f`	81AF0B 81Aa0B 81Af0B +	A P,WP,XS,S,M,B,W F1-F7	no no no	14 9+*n*
`R4=C f`	81AF0C 81Aa0C 81Af0C +	A P,WP,XS,S,M,B,W F1-F7	no no no	14 9+*n*

Example: I want to use D1, but I need to save its value:

```
CD1EX
R0=C A
```

You see?

34.3 Copying a save register to a working register

You cannot work with values stored in the save registers, so here are the instructions to recover information stored on them:

Mnemonic	Opcode	Fields	Carry	Cycles
`A=R0`	110	All	no	20.5
`A=R1`	111	All	no	20.5
`A=R2`	112	All	no	20.5
`A=R3`	113	All	no	20.5
`A=R4`	114	All	no	20.5
`C=R0`	118	All	no	20.5
`C=R1`	119	All	no	20.5
`C=R2`	11A	All	no	20.5
`C=R3`	11B	All	no	20.5
`C=R4`	11C	All	no	20.5

34.4 Copying a save register to a working register within a field

Mnemonic	Opcode	Fields	Carry	Cycles
`A=R0 f`	81AF10	A	no	14
	81Aa10	P,WP,XS,S,M,B,W	no	9+*n*
	81Af10 +	F1-F7	no	
`A=R1 f`	81AF11	A	no	14
	81Aa11	P,WP,XS,S,M,B,W	no	9+*n*
	81Af11 +	F1-F7	no	
`A=R2 f`	81AF12	A	no	14
	81Aa12	P,WP,XS,S,M,B,W	no	9+*n*
	81Af12 +	F1-F7	no	
`A=R3 f`	81AF13	A	no	14
	81Aa13	P,WP,XS,S,M,B,W	no	9+*n*
	81Af13 +	F1-F7	no	
`A=R4 f`	81AF14	A	no	14
	81Aa14	P,WP,XS,S,M,B,W	no	9+*n*
	81Af14 +	F1-F7	no	
`C=R0 f`	81AF18	A	no	14
	81Aa18	P,WP,XS,S,M,B,W	no	9+*n*
	81Af18 +	F1-F7	no	
`C=R1 f`	81AF19	A	no	14
	81Aa19	P,WP,XS,S,M,B,W	no	9+*n*
	81Af19 +	F1-F7	no	
`C=R2 f`	81AF1A	A	no	14
	81Aa1A	P,WP,XS,S,M,B,W	no	9+*n*
	81Af1A +	F1-F7	no	
`C=R3 f`	81AF1B	A	no	14
	81Aa1B	P,WP,XS,S,M,B,W	no	9+*n*
	81Af1B +	F1-F7	no	
`C=R4 f`	81AF1C	A	no	14
	81Aa1C	P,WP,XS,S,M,B,W	no	9+*n*
	81Af1C +	F1-F7	no	

34.5 Exchanging between save and working register

Those are really cool. First we write the letter of the working register, then the name of the save register, and finally EX for “EXchange.”

Mnemonic	Opcode	Fields	Carry	Cycles
`AR0EX`	120	All	no	20.5
`AR1EX`	121	All	no	20.5
`AR2EX`	122	All	no	20.5
`AR3EX`	123	All	no	20.5
`AR4EX`	124	All	no	20.5
`CR0EX`	128	All	no	20.5
`CR1EX`	129	All	no	20.5
`CR2EX`	12A	All	no	20.5
`CR3EX`	12B	All	no	20.5
`CR4EX`	12C	All	no	20.5

34.6 Exchange between save/working register with field

Mnemonic	Opcode	Fields	Carry	Cycles
`AR0EX f`	81AF20 81Aa20 81Af20 +	A P,WP,XS,S,M,B,W F1-F7	no no no	14 9+*n*
`AR1EX f`	81AF21 81Aa21 81Af21 +	A P,WP,XS,S,M,B,W F1-F7	no no no	14 9+*n*
`AR2EX f`	81AF22 81Aa22 81Af22 +	A P,WP,XS,S,M,B,W F1-F7	no no no	14 9+*n*
`AR3EX f`	81AF23 81Aa23 81Af23 +	A P,WP,XS,S,M,B,W F1-F7	no no no	14 9+*n*
`AR4EX f`	81AF24 81Aa24 81Af24 +	A P,WP,XS,S,M,B,W F1-F7	no no no	14 9+*n*
`CR0EX f`	81AF28 81Aa28 81Af28 +	A P,WP,XS,S,M,B,W F1-F7	no no no	14 9+*n*
`CR1EX f`	81AF29 81Aa29 81Af29 +	A P,WP,XS,S,M,B,W F1-F7	no no no	14 9+*n*
`CR4EX f`	81AF2A 81Aa2A 81Af2A +	A P,WP,XS,S,M,B,W F1-F7	no no no	14 9+*n*
`CR3EX f`	81AF2B 81Aa2B 81Af2B +	A P,WP,XS,S,M,B,W F1-F7	no no no	14 9+*n*
`CR4EX f`	81AF2C 81Aa2C 81Af2C +	A P,WP,XS,S,M,B,W F1-F7	no no no	14 9+*n*

35 Pointers

D0 and D1 can only contain five nibbles, and this is because all addresses are encoded using five nibbles. They are thus called pointers because according to their value, they “point” a specific nibble in memory.

A and C are also special: they are the only registers that can be used with D0 and D1.

35.1 Giving D0 or D1 a value

We can load 2, 4 or 5 nibbles in D0 or D1. If we only load two, the other three are left untouched; if we load only four, the last one is left untouched.

We will simply write:

```
D0= ..
D1= ..
```

Perhaps it may look strange to only load only two or four nibbles. In fact, sometimes you’ll load a five-nibble value, and then only change what needs to be changed.

All you have to write is `D0= ..` (with `..` being 2, 4 or 5 nibbles). You can do the same with D1.

The space after the = sign is important with HP-ASM! Otherwise, HP-ASM won't recognize it. MASD, however, doesn't require the space, but we will always write the space in this book to maintain HP-ASM compatibility.

The hex forms depend on the number of nibbles we load:

Mnemonic	**Opcode**	**Fields**	**Carry**	**Cycles**
`D0= yz`	19zy	First 2 nibbles	no	6
`D0= wxyz`	1Azyxw	First 4 nibbles	no	9
`D0= vwxyz`	1Bzyxwv	First 5 nibbles	no	10.5
`D1= yz`	1Dzy	First 2 nibbles	no	6
`D1= wxyz`	1Ezyxw	First 4 nibbles	no	9
`D1= vwxyz`	1Fzyxwv	First 5 nibbles	no	10.5

As you can see, when you type `D1= 00120`, the Code object will be encoded as "1F02100," so the nibbles to be loaded into D1 are reversed.

I told you that you would love the Saturn…but love or hate it, I'm sure you feel *something*. ;)

35.2 Adding to or subtracting from D0 or D1

We don't always need to load a value into D0 or D1, because we can just move the pointer. Remember this piece of code?

```
A=DAT0 A
D0=D0+ 5
PC=(A)
```

The second line is the instruction used to increment D0; if a minus sign "-" is used, D0 will decrement instead.

Note: don't forget the space between the sign and the value, as it's `D0=D0+ 5` rather than `D0=D0+5.` If you forget the space, HP-ASM will report an error. If you are using MASD, however, you can safely leave the space out.

You can add or remove from 1 to 16 to D0 or D1; if you want to remove more, you will have to repeat these instructions as many times as necessary.

Here they are:

Mnemonic	**Opcode**	**Fields**	**Carry**	**Cycles**
`D0=D0+ n`	16(*n*-1)	The entire register	yes	8.5
`D0=D0- n`	18(*n*-1)	The entire register	yes	8.5
`D1=D1+ n`	17(*n*-1)	The entire register	yes	8.5
`D1=D1- n`	1C(*n*-1)	The entire register	yes	8.5

n is an integer from 1 through 16. In addition, the carry is affected. For example, if you have #FFFFFh in the pointer register and add a value, the carry is set because there is an overflow. As usual, subtraction can affect the carry as well. Note that these instructions always operate in hexadecimal mode, regardless of whether the calculator is in decimal mode.

35.3 Copying A (or C) to D0 (or D1)

The A register is useful: it has 5 nibbles, just like D0 and D1! We can make D0 or D1 point to the 5 nibbles inside A or C, like this:

Mnemonic	**Opcode**	**Fields**	**Carry**	**Cycles**
`D0=A`	130	A field of D0	no	9.5
`D0=C`	134	A field of D0	no	9.5
`D1=A`	131	A field of D1	no	9.5
`D1=C`	135	A field of D1	no	9.5

It is also possible to only copy four nibbles, leaving the fifth one untouched. In other words, the MSB nibble (MSB = Most Significant Bits) is left in D0 or D1, and four nibbles from A or C overwrite the other four nibbles.

The mnemonics here are similar, with the only difference being an "S" added to the end.

Mnemonic	**Opcode**	**Fields**	**Carry**	**Cycles**
`D0=AS`	138	Four nibbles only	no	8.5
`D0=CS`	13C	Four nibbles only	no	8.5
`D1=AS`	139	Four nibbles only	no	8.5
`D1=CS`	13D	Four nibbles only	no	8.5

35.4 Exchanging field A of A/C with D0/D1

You already know the instructions to exchange between save registers and working registers, but here are some which exchange field A of A or C with D0 or D1.

I like these a lot, because if I want to check for the prologue of an object, I can use one of those exchange registers to modify D0 (or D1) and temporarily save the original value of D0 or D1. First the instructions are listed, and then examples are given.

Mnemonic	**Opcode**	**Fields**	**Carry**	**Cycles**
`AD0EX`	132	Five nibbles	no	9.5
`AD1EX`	133	Five nibbles	no	9.5
`CD0EX`	136	Five nibbles	no	9.5
`CD1EX`	137	Five nibbles	no	9.5

Each object starts with a prologue, which is a five nibble binary number. The stack doesn't contain objects, but rather the addresses of objects.

Why?

As you know, when your program starts, the HP already uses D0, D1, B, and D. D0 points to the address of the next RPL object to use, D1 points to the first level of the stack, D is the free memory in five-nibble blocks (each unit of D is 5 nibbles available) and B points to the RPL return stack.

When we start a program, we can use pointers, but when we quit, we must restore the original values. There are two ways to do that: save all the registers and restore them before we leave, or temporarily save them ourselves.

You can call subroutines found in ROM from your programs, and many are very useful. When you want to save all pointers, you can simply write:

```
   GOSBVL 0679B
```

This entry point is called =SAVPTR and is the same on both the 48 and 49. If you are using MASD and have an entry point table loaded, you can call it by the entry point name instead, like the following:

```
   GOSBVL =SAVPTR
```

As you can see, all you have to do to use an entry point name is to replace the address with the entry point name. This does not work with HP-ASM, or if you do not have an entry point table installed, however.

We will learn more about jumps like these later. Once you have called this routine, you can do what you want with D0 or D1. This works because there is a routine at #0679Bh that will save D0, D1, B and D. Later, when we study where those values are saved, I will show you why the HP uses some part of the RAM to store some important values.

Before leaving, we can restore the pointers with this code:

```
GOSBVL 067D2
```

This entry point is called =GETPTR and is also the same on both the 48 and 49.

However, if we just want to do something very simple, we may not need to save pointers.

Let's do an example.

I want a code program that drops the first level of the stack, but only if it contains a string. First, you have to learn how to do a drop. Next, you must learn how to determine the type of an object on the stack.

We know that D1 points to the first level of the stack. The stack does not contain objects, but instead it contains their addresses. This is better: when you do a SWAP, the addresses on level 1 and 2 are exchanged! The objects don't move, so we can swap the two levels by only moving 10 nibbles. This means it's just as fast when swapping large objects!

This also explains why some strange occurrences. Sometimes when there are two objects on the stack, and one is altered, the second one changes as well!

This happens because when an object is DUPed on the stack, it's the address which is duplicated, not the object. So some commands will modify the first level of stack, and because the two addresses are the same, the object on level 2 is also modified.

This is also why the NEWOB command exists: it duplicates the object and puts its address on the stack.

So, if we have three objects on the stack, the stack will actually contain:

Memory address	Content
`D1+15`	00000
`D1+10`	address of third object
`D1+5`	address of second object
`D1`	address of first object

When our program starts, D1 points to the address of the first object. If we want to have D1 point to the next object's address, we do a:

```
D1=D1+ 5
```

If we use `C=DAT1 f` or `A=DAT1 f` we will read the specified nibbles from the object on level 2.

If we want to do a drop, we can simply move D1 to the object in level 2 and free 5 nibbles of memory (since they are no longer used).

A DROP is then:

```
D1=D1+ 5
D=D+1 A
```

This moves D1 to the next object of the stack and increments D by 1, as it contains the free RAM in five-nibble blocks. We add 1 to it because that frees five nibbles of RAM, since each address on the stack uses five nibbles of memory.

But, we cannot do a drop without checking what's on the stack, as we could get a "Try To Recover Memory?" message. Assembly language is cool, but make a mistake and you could corrupt your memory.

So, here is a DROP that does not check if the stack is empty:

```
D1=D1+ 5
D=D+1 A
A=DAT0 A
D0=D0+ 5
PC=(A)
@
```

Don't forget the @, because it tells the assembler where the end of the source is.

Notice that our code moved D1 5 nibbles further (thus, the object on level 2 becomes the object on level 1) and we free 5 nibbles of memory. The three instructions that follow are the usual ones to return to RPL: we read the address of the next RPL object (`A=DAT0 A`), we update D0 (`D0=D0+ 5`), and we continue to the next RPL object (`PC=(A)`).

Compile that code, and store it inside a variable. Then, put 3 or 4 objects on stack, and run it. Each time you run it, the object on level one is dropped. Just make sure not to run it with an empty stack!

Let's do a better example: first we'll check if the stack is empty, and if it's not, we'll do a DROP:

First, we read the address on level 1 of stack. If it's equal to *zero*, the stack is empty. Since the stack contains addresses, we have to check whether 5 nibbles are null there.

We'll read the 5 nibbles, which form the address of the first object of the stack:

```
C=DAT1 A
```

Now C contains the address of the first object on the stack, or #00000h if the stack is empty. So? We need a test, so we'll test whether C is equal to #00000h, using an A field:

```
?C=0 A
```

After this test, we have to tell the assembler what should be done whether the test is true (C=0) or not (C different than 0).

The mnemonic used is "GOYES" followed by a label.

So, write:

```
?C=0 A
GOYES QUIT
...
*QUIT
...
```

If field A of C is equal to zero, we'll jump to the label QUIT. Remember a label is a line that starts with a *.

If C is not equal to zero, the execution continues at the line just after the `GOYES`.

Here it is:

```
C=DAT1 A      % read what is on level 1
?C=0 A        % if it's equal to zero, we jump to QUIT
GOYES QUIT    % otherwise, we have an object on level 1:
D1=D1+ 5      % we move D1 5 nibbles further
D=D+1 A       % and we free 5 nibbles of memory
*QUIT         % here, we QUIT:
A=DAT0 A      % we read the next RPL object's address
D0=D0+ 5      % update D0
```

```
PC=(A)        % and we jump to the object
@             % end of source :-)
```

As you can see, we use the % symbol to add comments.

Programmers use comments to remember what each piece of their code does. It's also very useful when you give code you wrote to someone else. The assembler will ignore everything that follows the % until the line ends.

We could have commented and spaced it like this instead:

```
% check if level 1 is empty

C=DAT1 A
?C=0 A
GOYES QUIT

% DROP

D1=D1+ 5
D=D+1 A

% we QUIT here

*QUIT
A=DAT0 A
D0=D0+ 5
PC=(A)

@
```

You should put comments in your sources. That will help you know where you are, and when you use old source again, like from several weeks earlier, you don't have to research what each piece of code does.

Now we can write the code we originally wanted: drop the first level of the stack if it's a string.

As before, we first must read the address of the object on the first level of stack. If it's null, we'll quit:

```
C=DAT1 A
?C=0 A
GOYES QUIT
```

Now, we have to ensure that the object on level 1 is a string. Every object on the HP is identified by a 5-nibble prologue at its start, so that we can simply check the prologue to determine the object type.

Here we'll need to use D0 or D1 to point to the object, but we can lose neither D0 nor D1's value, so we can use an exchange instruction:

```
CD1EX
```

Now C contains the D1 value, and D1 contains C. Because C contained the address of the object on level 1 of the stack, D1 now points to the object.

The prologue is the first five nibbles of the object, so we'll use A and read five nibbles:

```
A=DAT1 A
D1=C
```

We have now read the object's prologue and have returned D1 to its previous value, which was saved in C with `CD1EX`.

Here we used `CD1EX` to have D1 point to something else while not losing the original value of D1, as it was stored in C. We used A so C keeps the D1 value.

Now we have to compare A to the string's prologue value. The string prologue is #02A2Ch, and strings always start with these 5 nibbles. (In the next part we will discuss all objects, explaining their prologues and how they're encoded.)

All we have to do here is load #02A2Ch into C and compare A to C using an A field. If A=C, we can drop the object, but otherwise we quit.

Here is the final code to perform our drop while checking for a string on level 1 of the stack:

```
C=DAT1 A        % read level 1's address
?C=0 A          % equal to zero? empty stack --> we quit
GOYES QUIT
CD1EX           % D1 points to the object, C contains old D1
A=DAT1 A        % read the first 5 nibbles, i.e. prologue
D1=C            % restore D1
LC 02A2C        % load #O2A2Ch in C, string's prologue
?A#C A          % # means: different; check if A not equal to C
GOYES QUIT      % if different, it's not a string --> we quit
D1=D1+ 5        % here we do the DROP, D1+5
D=D+1 A         % and we free 5 nibbles of RAM
*QUIT           % here we quit:
A=DAT0 A
D0=D0+ 5        % usual RPL jump...
PC=(A)
@
```

Try it: it'll work!

If stack is empty, it does nothing. If the stack contains a string, it's dropped, but if it's not a string, nothing is done.

Note: if HP-ASM tells you it doesn't understand "?A#C A" it's because 'HP' is set inside OPT. When HP is set, the not-equal-to symbol is = with a line through it. When PC is set, the not equal to sign is #.

The HP not equal to sign is [ALPHA][Right-Shift][1]. The # is [Right-Shift][Divide key].

Now let's continue with instructions:

35.5 Exchanging 4 LSB nibbles A of A/C with D0/D1

The instructions below exchange the 4 LSB nibbles (the first four nibbles) of working registers A or C with D0 or D1. The MSB nibble is left untouched:

Mnemonic	Opcode	Fields	Carry	Cycles
AD0XS	13A	Four nibbles only	no	8.5
AD1XS	13B	Four nibbles only	no	8.5
CD0XS	13E	Four nibbles only	no	8.5
CD1XS	13F	Four nibbles only	no	8.5

Those are similar to the previous ones, but here only four nibbles are exchanged.

36 Reading memory within a field

We know how to modify D0 or D1 so it can point to where we want in memory. We also know how to load values into D0 or D1 (using D0/D1= ..). Here, only registers A and C will be used, and we're going to describe the DAT0 and DAT1 instructions we have used.

The following line of code will copy 16 nibbles from the data that D1 points to into C:

```
C=DAT1 W
```

This code copies 12 nibbles, or the mantissa field (M) from working register A to the data pointed to by D0:

```
DAT0=A M
```

Here are the mnemonics:

Mnemonic	Opcode	Fields	Carry	Cycles
`A=DAT0 A`	142	A	no	23.5,3.5
`A=DAT0 B`	14A	B	no	19.5
`A=DAT0 f`	152a 152f +	P,WP,X,XS,S,W,M,B F1-F7	no no	20+*n*,1+*n*/2
`A=DAT1 A`	143	A	no	23.5,3.5
`A=DAT1 B`	14B	B	no	19.5
`A=DAT1 f`	153a 153f +	P,WP,X,XS,S,W,M,B F1-F7	no no	20+*n*,1+*n*/2
`C=DAT0 A`	146	A	no	23.5,3.5
`C=DAT0 B`	14E	B	no	19.5
`C=DAT0 f`	156a 156f +	P,WP,X,XS,S,W,M,B F1-F7	no no	20+*n*,1+*n*/2
`C=DAT1 A`	147	A	no	23.5,3.5
`C=DAT1 B`	14F	B	no	19.5
`C=DAT1 f`	157a 157f +	P,WP,X,XS,S,W,M,B F1-F7	no no	20+*n*,1+*n*/2

The register that receives the nibbles is located to the left of the equal sign (=).

It's not the *value* of the pointer that is moved, but rather the nibbles *where* the pointer D0 or D1 points.

Example: If I want to read 5 nibbles at #00100h, I do:

```
D0= 00100
C=DAT0 A
```

D0 contains #00100h, but C will contain the five nibbles that are at #00100h in memory.

As you see in the table above, fields A and B have special forms in their hexadecimal form, which is the form that is encoded inside the Code object. Reading two nibbles, field B, from D0 can be encoded using one of two different ways in the code object. This means the mnemonic `C=DAT0 B` can be encoded as either 14E or 1566.

Using 14E inside the code is both smaller and faster: 1566 will need 22 cycles but 14E will only need 19.5 cycles. The Saturn processor, when reading or writing data, is somewhat specialized for using A and B fields. In a way, it "prefers" these two fields.

37 Reading from memory with a value

We can also use a number in place of a field, so instead of:

```
   C=DAT0 A
```

one can use:

```
   C=DAT0 5
```

What is the difference? In the code object it's not encoded the same way. In addition, `C=DAT0 A` needs 23.5 cycles, and `C=DAT0 5` needs 24, so as you see, using A here is better! But, using `C=DAT0 16` is better than using `C=DAT0 W`, as `C=DAT0 16` needs 35 cycles, and `C=DAT0 W` needs 36 cycles. That's only a difference of one cycle, but when used in a loop that is run, for example, 100 times, there are 100 fewer cycles used!

It's a little bit more complicated to give the cycles with the instructions, but when you want the fastest possible routine, you'll use those cycles to choose which instructions to use.

Here they are:

Mnemonic	Opcode	Fields	Carry	Cycles
`A=DAT0` *n*	15A(*n*-1)	n nibbles	no	19+*n*
`A=DAT1` *n*	15B(*n*-1)	n nibbles	no	19+*n*
`C=DAT0` *n*	15E(*n*-1)	n nibbles	no	19+*n*
`C=DAT1` *n*	15F(*n*-1)	n nibbles	no	19+*n*

38 Writing to memory within a field

Here we copy nibbles from a working register (A or C) to the memory:

Mnemonic	Opcode	Fields	Carry	Cycles
`DAT0=A A`	140	A	no	19.5
`DAT0=A B`	148	B	no	16.5
`DAT0=A f`	150a 150f +	P,WP,X,XS,S,W,M,B F1-F7	no no	19+*n*
`DAT1=A A`	141	A	no	19.5
`DAT1=A B`	149	B	no	16.5
`DAT1=A f`	151a 151f +	P,WP,X,XS,S,W,M,B F1-F7	no no	19+*n*
`DAT0=C A`	144	A	no	19.5
`DAT0=C B`	14C	B	no	16.5
`DAT0=C f`	154a 154f +	P,WP,X,XS,S,W,M,B F1-F7	no no	19+*n*
`DAT1=C A`	145	A	no	19.5
`DAT1=C B`	14D	B	no	16.5
`DAT1=C f`	155a 155f +	P,WP,X,XS,S,W,M,B F1-F7	no no	19+*n*

As you can see here, fields A and B have specific and optimized instructions. As you know, A is useful for addresses. B, with two nibbles, is commonly used for bytes. As we'll learn later, characters are also encoded using two nibbles.

39 Writing to memory with a value

Like before, instead of using a field, we give the value we want to read:

Mnemonic	**Opcode**	**Fields**	**Carry**	**Cycles**
`DAT0=A n`	158(*n*-1)	*n* nibbles	no	18+*n*
`DAT1=A n`	159(*n*-1)	*n* nibbles	no	18+*n*
`DAT0=C n`	15C(*n*-1)	*n* nibbles	no	18+*n*
`DAT1=A n`	15D(*n*-1)	*n* nibbles	no	18+*n*

Suppose you want to write one nibble:

```
DAT0=A P
```

If P is not equal to zero (because you use it for something else), you can write:

```
DAT0=A 1
```

This will write one nibble, and it will be encoded as “1590” inside the code object (159 and 1-1 = 1590)

That’s not very difficult now, is it?

40 Jumping and tests

As I’ve told you (but not really explained) every instruction is located at some address in memory, which means we can do jumps inside our programs. Usually, a program’s instructions are executed one after another. But sometimes, after a test or just because we want the code to continue somewhere else, we will want to jump to some other part.

A jump is also called “branching” because we “branch” to some other place in memory.

Register PC is used for that here. You know it contains the next instruction to be executed, so if we modify PC, we can change the flow of the instruction’s execution.

We will discuss three kinds of jumps:

a) The *relative* jump.
This jumps +n or -n nibbles from our present position. We don’t really know precisely where in memory we are going to jump, but we know how many nibbles before or after it will be.
This introduces a new word to you: the *offset*. An offset is the numerical value that represents the displacement.
It’s a *signed* value, which means it can be either *positive* or *negative*. Thus, we can jump either backward or forward.
For example, if we want to jump 100 nibbles further, the offset is +100.

b) The *absolute* jump
This jumps to a specific address. As an address is a 5-nibble number, like #ABCDE, an absolute jump means jumping to address ABCDE and continuing execution there.

c) The *indirect* jump
More complicated (more fun). It’s not jumping to an address, but reading one address, to which we will jump, at another address.
For example, if I do an *absolute* jump to ABCDE, I will go directly to ABCDE in memory and continue execution there.
If I do an *indirect* jump, I will read 5 nibbles at #ABCDE and then jump to the address found in those 5 nibbles.

When you put jumps in your code using labels, the assembler will automatically generate the jumps for you. If you jump forward, it directly codes the offset, and if you jump backwards, it codes the complement of 2 of the address to indicate a negative offset, thus jumping backwards in memory.

For example, suppose I have the following jump in my code:

```
    ?C=0 A
    GOYES QUIT
    ...
    *QUIT
```

The assembler will generate the ... code and then calculate the offset so that the `GOYES` jumps to the code that is encoded after the *QUIT line:

Memory address	Encoded in code object	Instructions
#n	8AAzy	?C=0 A
#n + ...	...	…
#n + yz	remaining code	*QUIT label

The assembler will calculate the offset and encode it using two nibbles (yz). The `?C=0 A` will be encoded as 811zy inside the code object.

If the jump is going backwards, the assembler will use the two’s complement to code the jump value.

Why is this important?

Because the jumps after a test are encoded using *two signed nibbles* inside the Code object, it means sometimes it won’t be able to do the jump because the offset is too big. You’ll have to write something different for your code to work. This will be explained later.

41 Relative, absolute, and indirect jumps

41.1 Conditional jumps: GOC and GONC

A conditional jump is not an automatic jump: the jump will occur according to the value of something else: here, the carry.

Two instructions can be used to jump according to the carry state: `GOC` and `GONC`

`GOC` means Go On Carry
`GONC` means Go On No Carry (or Not Carry)

Example: Suppose I remove 16 from A(A) (field A of working register A). If the carry is set, it means that A was less than 16 (0 to 15). The following code will check if an underflow happens:

```
    C=C-16 A
    GOC ERROR
    ...
    *ERROR
    % underflow occurred!
```

`GOC` will only jump if the carry bit is set. If it’s not, the execution continues just after the `GOC` ... line.

`GONC` will only jump if the carry is not set. If it’s set, it will not jump.

NOTE: these checks do not change the carry value. They check whether the carry is set but don’t change its state. After the jump, the carry is still set until an instruction that modifies it clears it.

Mnemonic	Opcode	When?	Cycles
`GONC` label	5yz	If carry is clear	4.5 or 12.5
`GOC` label	4yz	If carry is set	4.5 or 12.5

There are two cycle values: 3 cycles are needed if the jump is not made, and 10 cycles are needed if the jump is made. It is therefore beneficial to choose the test that is less likely to jump, if known.

As you can see here, two nibbles, yz, are used to code the relative jump. This means we have eight bits to code the offset. The value of these two nibbles is *signed* using complement of 2, meaning that `GONC` and `GOC` can jump up to 128 nibbles *backwards*, or 127 *forward*.

If the label more than 127 nibbles ahead or 128 nibbles before, the assembler will tell you the jump is too long.

Thus, this:

```
GOC ifCarry
% if no carry here..
```

will become:

```
GONC noCarry
GOTO ifCarry
*noCarry
% we continue here if no carry
```

The `GOTO` uses one more nibble to code jumps for a total of 3 nibbles. If the assembler still says the jump is too long, we will use `GOVLNG`, which can jump to *any* point in memory since it uses 5 nibbles to code the jump address. This works because, as you should remember, all addresses use a maximum of 5 nibbles.

Remember that `GOC` and `GONC` are only able to jump 128 nibbles before and 127 nibbles after the present position.

If you need a bigger jump, you must change the test, using `GOTO`, `GOLONG` or `GOVLNG` (GO Very LoNG) instructions, which are described in the next section.

41.2 Unconditional jumps: GOTO, GOLONG, GOVLNG

Unconditional means the jump will be done without any checking, so one could call it a “blind jump.”

A difference between `GOTO`, `GOLONG` and `GOVLNG` is the number of nibbles needed to code jumps: 3, 4 or 5.

41.2.1 GOTO

Here it is:

Mnemonic	Opcode	When?	Cycles
`GOTO` label	6xyz	Always (blind jump)	14

`GOTO` will make the program continue the execution from the LABEL point inside of the source code. It needs 14 cycles to complete. Because 3 nibbles are used to code jumps, it can jump 2048 nibbles before the `GOTO` position or 2047 nibbles after.

41.2.2 GOLONG

Mnemonic	Opcode	When?	Cycles
`GOLONG` label	8Cwxyz	Always (blind jump)	17

This is a bigger jump, using four nibbles to code jumps, so we can jump 32768 nibbles before the `GOLONG`'s position or 32767 nibbles further.

When you code, first try `GOTO`. If the assembler says the jump is too long, try `GOLONG`. If it still says the jump is too long, you'll have to use `GOVLNG` (as this one uses 5 nibbles, it won't complain). This is to use the instructions that take both the fewest cycles and fewest nibbles.

41.2.3 GOVLNG

Mnemonic	Opcode	When?	Cycles
`GOVLNG` label	8Dvwxyz	Always (blind jump)	18.5

Here, as all addresses are coded using 5 nibbles, we do a direct jump to any point of memory. In this case it's not an offset encoded, but rather an address.

41.2.4 Unconditional jump to the address in A or C

Two instructions are used here: `PC=A` and `PC=C`. We directly modify the PC register and load A or C inside of it. The next instruction will be fetched from the address in A or C.

Mnemonic	Opcode	When?	Cycles
`PC=A`	81B2	Field A of register A	26,3.5
`PC=C`	81B3	Field A of register C	26,3.5

Five nibbles are moved from A or C directly to PC.

41.2.5 Unconditional jump with exchange

Here, instead of copying field A of A or C to PC, we exchange the values. PC makes the flow of instructions continue to the address in field A of A or C, and the previous value of PC is stored inside A or C, because it's an exchange.

Mnemonic	Opcode	When?	Cycles
`APCEX`	81B6	Field A of register A	19
`CPCEX`	81B7	Field A of register C	19

41.2.6 Unconditional and indirect jump

Now for the indirect jump we talked about earlier. We first read five nibbles at the address in A or C and then load them into PC, thus "jumping" there.

Mnemonic	Opcode	When?	Cycles
`PC=(A)`	808C	Field A of register A	26,3.5
`PC=(C)`	808E	Field A of register C	26,3.5

Example: if at address #ABCDE inside A or C there are five nibbles #FGHIJ, then execution continues at #FGHIJ

41.2.7 Saving PC contents

Those two instructions are not used to do jumps, but are useful to save the PC value before a jump. Here they are:

Mnemonic	Opcode	When?	Cycles
`A=PC`	81B4	Field A of register A	11
`C=PC`	84B5	Field A of register C	11

42 Calling a subroutine

When you create a subroutine, it's usually to save space inside of your code, when some piece of code is called a lot. If you need to call a routine (a piece of code) multiple times it's better (if you want to save space) to put it inside a subroutine. It's just a label followed by the code that will be called. When you want, you can return back to where you called the subroutine, and continue running your program.

When jumping to a subroutine, the processor loads the address of the instruction that follows the jump instruction into RSTK (the Return STacK area). Then, the execution continues to where you jumped, until a return instruction is found.

When the return instruction is found, it's loaded into PC, and the program "returns" from the subroutine and execution continues.

The difference here between a usual jump is that an address must be recorded so we can go back.

RSTK is eight levels high, with each level being five nibbles wide, but only five levels can be used. This means you can have five levels of subroutines, or eight if you don't allow *any* interruptions (more on that later).

There are several return instructions available. Some will just return, others will return and modify the carry bit (set it or clear it) and so on.

A subroutine is a *relative* jump, so it's an *offset* which is coded (unless you use `GOSBVL`, GOSuB Very Long, which encodes a specific five-nibble address).

42.1 GOSUB

Here three nibbles are used to encode the offset of the jump, so we can jump to a subroutine 2048 nibbles before or 2047 nibbles after the `GOSUB`'s position).

Mnemonic	Opcode	How?	Cycles
`GOSUB` label	7xyz	1 RSTK level used	15

42.2 GOSUBL

This stands for GOSUB Long. Here we use four nibbles to encode the offset, so we can jump to a subroutine 32768 nibbles before or 32767 nibbles after.

Mnemonic	Opcode	How?	Cycles
`GOSUBL` label	8Ewxyz	1 RSTK level used	18

42.3 GOSBVL

Here we directly encode the five nibbles of the address:

Mnemonic	**Opcode**	**How?**	**Cycles**
`GOSBVL` label	8Fvwxyz	1 RSTK level used	19.5

42.4 Returning from a subroutine

There are several instructions to do this:

Mnemonic	**Opcode**	**How?**	**Cycles**
`RTN`	01	Usual return	11
`RTNSC`	02	Sets the carry bit as well	11
`RTNCC`	03	Clears the carry bit instead	11
`RTI`	0F	Allows interrupts	11
`RTNSXM`	00	Sets XM bit	11

`RTN` simply removes the return address from `RSTK` and jumps back just after the instruction that called the subroutine.

`RTNSC` and `RTNCC` will return and set or clear the carry bit, so it can be used to call a subroutine that will use the carry bit to give you information according to the state of the carry bit.

`RTI` will allow interrupts again. We'll understand this better when we study how interruptions work, and how they're disallowed or re-allowed by the routines in the HP's ROM (we'll disassemble all that, yes!!)

`RTNSXM` will return and set the XM (eXternal Module missing) bit.

42.5 Returning according to the value of the carry bit

You can choose whether to return depending on the value of the carry bit. The instructions are:

Mnemonic	**Opcode**	**When?**	**Cycles**
`RTNC`	400	If carry is set	4.5 or 12.5
`RTNNC`	500	If carry is cleared	4.5 or 12.5

`RTNC` will return only if the carry is set. If it's clear, the testing uses 4.5 cycles; if it's set, the return needs 12.5 cycles.

`RTNNC` will return only if the carry is cleared. 4.5 cycles are needed if the return is not done and 12.5 cycles are used if it returns.

43 The tests

When we do a test, we compare a register to zero or to another register.

Here you will see two values for the cycles needed. The first one is the cycles used for the test to complete if no jump occurs, and the second is the number of cycles used if the test is completed and the jump is made.

Each test will start with a question mark (?). Next comes a working register (A or C), and then an operator (=, >, <, etc.).

On the line that follows a test, there must be an *action* command. It can be a jump, using `GOYES`, but it can also be a return from a subroutine if we use `RTNYES`.

43.1 Comparing registers to zero

43.1.1 Equal to zero?

Here we check whether a working register is equal to zero. The symbol to check equality is =.

Mnemonic	Opcode	Fields	Carry	Cycles
`?A=0 A`	8A8 yz	A	yes	13.5/21.5
`?A=0 f`	9a8 yz	P,WP,X,XS,S,W,M,B	yes	8.5+*n*/16.5+*n*
`?B=0 A`	8A9 yz	A	yes	13.5/21.5
`?B=0 f`	9a9 yz	P,WP,X,XS,S,W,M,B	yes	8.5+*n*/16.5+*n*
`?C=0 A`	8AA yz	A	yes	13.5/21.5
`?C=0 f`	9aA yz	P,WP,X,XS,S,W,M,B	yes	8.5+*n*/16.5+*n*
`?D=0 A`	8AB yz	A	yes	13.5/21.5
`?D=0 f`	9aB yz	P,WP,X,XS,S,W,M,B	yes	8.5+*n*/16.5+*n*

In the resulting code object, the "yz" is an offset (signed) that is used for the jump if the test is true.

If the test is true then the carry bit is set.

These instructions need 8.5+*n* cycles (*n* = number of nibbles of the field used for comparison) if the test result is false and 16.5+*n* if the test result is true and the jump is made.

You may use "`GOYES` label" or "`RTNYES`" after those instructions. `GOYES` will make the flow of execution jump to the label of your choice inside your code, and `RTNYES` will return from a subroutine.

Two nibbles are used to code the offset, and it's a signed value, so how far can it jump? Well! Good! 128 nibbles before or 127 nibbles after the jump.

43.1.2 Different from zero?

The symbol for "different" depends on the mode you set in the HP-ASM options: with HP it will be a = with a line through it (not equal to sign) and with "PC" it will be a pound sign (#). With MASD, you can use either one. I strongly encourage you to use #, so you'll be able to write your code on your PC/Mac/Linux system and send the source to your HP, rather than have to find/replace symbols for each other.

Mnemonic	Opcode	Fields	Carry	Cycles
`?A#0 A`	8AC yz	A	yes	13.5/21.5
`?A#0 f`	9aC yz	P,WP,X,XS,S,W,M,B	yes	8.5+*n*/16.5+*n*
`?B#0 A`	8AD yz	A	yes	13.5/21.5
`?B#0 f`	9aD yz	P,WP,X,XS,S,W,M,B	yes	8.5+*n*/16.5+*n*
`?C#0 A`	8AE yz	A	yes	13.5/21.5
`?C#0 f`	9aE yz	P,WP,X,XS,S,W,M,B	yes	8.5+*n*/16.5+*n*
`?D#0 A`	8AF yz	A	yes	13.5/21.5
`?D#0 f`	9aF yz	P,WP,X,XS,S,W,M,B	yes	8.5+*n*/16.5+*n*

The carry will be set if it tests to be true, otherwise it's clear.

43.2 Equality of two registers

Here the test will be true if equal or false if not equal.

Mnemonic	Opcode	Fields	Carry	Cycles
`?A=B A`	8A0 yz	A	yes	13.5/21.5
`?A=B f`	9a0 yz	P,WP,X,XS,S,W,M,B	yes	8.5+*n*/16.5+*n*
`?B=C A`	8A1 yz	A	yes	13.5/21.5
`?B=C f`	9a1 yz	P,WP,X,XS,S,W,M,B	yes	8.5+*n*/16.5+*n*
`?A=C A`	8A2 yz	A	yes	13.5/21.5
`?A=C f`	9a2 yz	P,WP,X,XS,S,W,M,B	yes	8.5+*n*/16.5+*n*
`?C=D A`	8A3 yz	A	yes	13.5/21.5
`?C=D f`	9a3 yz	P,WP,X,XS,S,W,M,B	yes	8.5+*n*/16.5+*n*

43.3 Inequality of two registers

Mnemonic	Opcode	Fields	Carry	Cycles
`?A#B A`	8A4 yz	A	yes	13.5/21.5
`?A#B f`	9a4 yz	P,WP,X,XS,S,W,M,B	yes	8.5+*n*/16.5+*n*
`?B#C A`	8A5 yz	A	yes	13.5/21.5
`?B#C f`	9a5 yz	P,WP,X,XS,S,W,M,B	yes	8.5+*n*/16.5+*n*
`?A#C A`	8A6 yz	A	yes	13.5/21.5
`?A#C f`	9a6 yz	P,WP,X,XS,S,W,M,B	yes	8.5+*n*/16.5+*n*
`?C#D A`	8A7 yz	A	yes	13.5/21.5
`?C#D f`	9a7 yz	P,WP,X,XS,S,W,M,B	yes	8.5+*n*/16.5+*n*

43.4 Less than and greater than

Mnemonic	Opcode	Fields	Carry	Cycles
`?A>B A`	8B0 yz	A	yes	13.5/21.5
`?A>B f`	9b0 yz	P,WP,X,XS,S,W,M,B	yes	8.5+*n*/16.5+*n*
`?B>C A`	8B1 yz	A	yes	13.5/21.5
`?B>C f`	9b1 yz	P,WP,X,XS,S,W,M,B	yes	8.5+*n*/16.5+*n*
`?A<C A`	8B2 yz	A	yes	13.5/21.5
`?A<C f`	9b2 yz	P,WP,X,XS,S,W,M,B	yes	8.5+*n*/16.5+*n*
`?C<D A`	8B3 yz	A	yes	13.5/21.5
`?C<D f`	9b3 yz	P,WP,X,XS,S,W,M,B	yes	8.5+*n*/16.5+*n*
`?A<B A`	8B4 yz	A	yes	13.5/21.5
`?A<B f`	9b4 yz	P,WP,X,XS,S,W,M,B	yes	8.5+*n*/16.5+*n*
`?B<C A`	8B5 yz	A	yes	13.5/21.5
`?B<C f`	9b5 yz	P,WP,X,XS,S,W,M,B	yes	8.5+*n*/16.5+*n*
`?A>C A`	8B6 yz	A	yes	13.5/21.5
`?A>C f`	9b6 yz	P,WP,X,XS,S,W,M,B	yes	8.5+*n*/16.5+*n*
`?C>D A`	8B7 yz	A	yes	13.5/21.5
`?C>D f`	9b7 yz	P,WP,X,XS,S,W,M,B	yes	8.5+*n*/16.5+*n*

43.5 Less than or equal to or greater than or equal to

Mnemonic	Opcode	Fields	Carry	Cycles
`?A>=B A`	8B8 yz	A	yes	13.5/21.5
`?A>=B f`	9b8 yz	P,WP,X,XS,S,W,M,B	yes	8.5+*n*/16.5+*n*

Mnemonic	Opcode	Fields	Carry	Cycles
`?B>=C A`	8B9 yz	A	yes	13.5/21.5
`?B>=C f`	9b9 yz	P,WP,X,XS,S,W,M,B	yes	8.5+*n*/16.5+*n*
`?A<=C A`	8BA yz	A	yes	13.5/21.5
`?A<=C f`	9bA yz	P,WP,X,XS,S,W,M,B	yes	8.5+*n*/16.5+*n*
`?C<=D A`	8BB yz	A	yes	13.5/21.5
`?C<=D f`	9bB yz	P,WP,X,XS,S,W,M,B	yes	8.5+*n*/16.5+*n*
`?A<=B A`	8BC yz	A	yes	13.5/21.5
`?A<=B f`	9bC yz	P,WP,X,XS,S,W,M,B	yes	8.5+*n*/16.5+*n*
`?B<=C A`	8BD yz	A	yes	13.5/21.5
`?B<=C f`	9bD yz	P,WP,X,XS,S,W,M,B	yes	8.5+*n*/16.5+*n*
`?A>=C A`	8BE yz	A	yes	13.5/21.5
`?A>=C f`	9bE yz	P,WP,X,XS,S,W,M,B	yes	8.5+*n*/16.5+*n*
`?C>=D A`	8BF yz	A	yes	13.5/21.5
`?C>=D f`	9bF yz	P,WP,X,XS,S,W,M,B	yes	8.5+*n*/16.5+*n*

As usual, after each of these instructions you can put a GOYES or put a RTNYES. The jump is done is the test is true.

43.6 Testing a bit

Here we have the instructions that can test the value of a bit in C or A. Only the first 16 bits, or four nibbles, of the working registers can be tested.

Twelve cycles are needed to finish the test and continue if it's false, and 21 cycles are needed if it's true and a jump is performed.

Two nibbles are used inside the code (#yz) to encode the relative jump offset, so we can jump 127 nibbles before or 128 nibbles after the test's position.

Mnemonic	Opcode	Fields	Carry	Cycles
`?ABIT=0` *n*	8086n yz	Bit number n	yes	12.5/20.5
`?ABIT=1` *n*	8087n yz	Bit number n	yes	12.5/20.5
`?CBIT=0` *n*	808An yz	Bit number n	yes	12.5/20.5
`?CBIT=1` *n*	808Bn yz	Bit number n	yes	12.5/20.5

44 Using register P

The P register can only contain one nibble. When your program starts, its value will be zero. P not only defines how the loading instructions work (`LC` and `LA`), but it also defines the width of the WP register (from nibble 0 to nibble P).

Often, the P register is used as a loop counter, as long there is no `LC`, `LA`, or WP-related instruction inside of the loop. Not only are there instructions to give P a value, but there are also instructions to increment or decrement its value.

You can also test the P register, exchange it with the C register, and other things.

The mnemonics are below, followed by an explanation of each:

Mnemonic	Opcode	Fields	Carry	Cycles
`P=` *n*	2n	The nibble of P	no	3
`P=P+1`	0C	The nibble of P	yes	4
`P=P-1`	0D	The nibble of P	yes	4
`?P#` *n*	88n yz	The nibble of P	yes	7.5 or 15.5
`?P=` *n*	89n yz	The nibble of P	yes	7.5 or 15.5

Mnemonic	Opcode	Fields	Carry	Cycles
`C=P n`	80Cn	The nibble of P	no	8
`P=C n`	80Dn	The nibble of P	no	8
`C+P+1`	809	The nibble of P	yes	9.5
`CPEX n`	80Fn	The nibble of P	no	8

44.1 P= *n*

Don't forget the space after the "=" sign!! It gives the value n to P.

44.2 P=P+1

This increments the value of P by 1. NOTE: whichever mode the Saturn is working under, hexadecimal or decimal, the incrementing is done under *hexadecimal* mode. The carry is set if an overflow occurs.

44.3 P=P-1

This decrements the value of P by 1. NOTE: whichever mode the Saturn is working under, hexadecimal or decimal, the incrementing is done under *hexadecimal* mode. The carry is set if an overflow occurs.

44.4 ?P# *n*

This test checks whether P is different than n. Two nibbles inside the code object (signed) will be used to encode the offset of the relative jump. A `RTNYES` will be coded using #00h; any other value is a signed offset, just like the previous tests. The carry will be set if the test is true.

44.5 ?P= *n*

This test will check if P is equal to n. The carry will be set if the test is true.

44.6 C=P *n*

Gives the value of the P register to nibble number n of working register C. *n* is between 0 and 15, of course.

44.7 P=C *n*

Gives the value of nibble number *n* of working register C to P.

44.8 C+P+1

Adds the value of C(A), P and 1 to field A of C. Strange instruction? You will see it can be very useful. :)

This instruction also always operates in hexadecimal mode, regardless of the mode of the Saturn.

44.9 CPEX *n*

This instruction exchanges nibble number *n* of working register C with the P register.

45 The RSTK stack

RSTK is the return stack where return addresses are stored while subroutines are running. You can also use RSTK to save the working register C for some time, but don't forget to pop the saved value before you quit. It's faster than using a save register, so it can be cool to use.

Mnemonic	Opcode	Fields	Carry	Cycles
`RSTK=C`	06	Field A of C	no	9
`C=RSTK`	07	Field A of C	no	9

The instruction C=RSTK is also very useful for embedding data in your code, like this:

```
GOSUB AfterData
INCLUDE MyDataGoesHere
*AfterData
C=RSTK                          %C now points to your data
```

46 Registers IN and OUT

Inputs and outputs done by the processor go through registers IN and OUT. Here the X field will be quite useful because the OUT register is 3 nibbles wide. The IN register is 4 nibbles wide.

We have here four instructions:

Mnemonic	Opcode	Fields	Carry	Cycles
`OUT=CS`	800	Nibble #0h of C	no	5.5
`OUT=C`	801	Field X of C	no	7.5
`A=IN`	802	Four nibbles of A	no	8.5
`C=IN`	803	Four nibbles of A	no	8.5

ANOTHER BUG

You cannot use `A=IN` or `C=IN` instructions unless they are located on an even address, or they won't work.

So, instead of writing `A=IN` or `C=IN` in source code, we will always use subroutines in ROM that will do `A=IN` or `C=IN` and then return to our program.

We'll use:

```
GOSBVL =CINRTN instead of C=IN (=CINRTN is 01160 on the HP 48 and 00212 on the HP 49)
GOSBVL =AINRTN instead of A=IN (=AINRTN is 0115A on the HP 48 and 0020A on the HP 49)
```

Those routines contain:

```
C=IN          A=IN
RTN           RTN
```

This bug has been fixed in the Saturn+ emulated processor in the ARM-based calculators, but it is best to avoid this bug anyway to maintain backwards compatibility.

47 Status Bits (ST)

We have many flags we can use in our programs. There are four nibbles inside of ST, that's 4 * 4 = 16 flags; but some are already used by the Saturn. Until you learn how to use them well, we'll avoid playing with bits 12, 13 and 14 (we'll use bit 15 to enable or disable interrupts, but be patient as we'll come to that soon).

Those status bits are like the flags you use in RPL.

Here are the instructions, and explanations follow:

Mnemonic	**Opcode**	**Fields**	**Carry**	**Cycles**
`CLRST`	08	First 3 nibble of ST	no	7
`C=ST`	09	X field of C	no	7
`ST=C`	0A	X field of C	no	7
`CSTEX`	0B	X field of C	no	7
`ST=0` *n*	84n	Bit n of ST	no	5.5
`ST=1` *n*	85n	Bit n of ST	no	5.5
`?ST=0` *n*	86n yz	Bit n of ST	yes	8.5 or 16.5
`?ST=1` *n*	87n yz	Bit n of ST	yes	8.5 or 16.5

47.1 CLRST

Clears the first 3 nibbles of ST. The last nibble is not cleared because it contains special values for the processor.

47.2 C=ST

Copies 3 nibbles of ST to C.

47.3 ST=C

Copies 3 nibbles from C to ST.

47.4 CSTEX

Exchanges 3 nibbles from C and ST.

47.5 ST=1 *n*

Sets bit n of ST to 1.

47.6 ST=0 *n*

Clears bit n of ST, therefore setting it to 0.

47.7 ?ST=0 *n* and ?ST=1 *n*

Those two instructions are used to check the value of a single bit of the ST register. We can test if it's set or cleared. You can put a `GOYES` or a `RTNYES` with a label after those instructions if you want.

48 Hardware Status Bits (HST)

There are eleven instructions that can be used to affect the HST bits. The useful bits of HST are:

- SR (Service Request)
- SB (Sticky Bit)
- MP (Module Pulled)
- XM (eXternal Module missing)

Mnemonic	**Opcode**	**Fields**	**Carry**	**Cycles**
`CLRHST`	82F	Four bits of HST	no	4.5
`XM=0`	821	XM bit	no	4.5
`SB=0`	822	SB bit	no	4.5
`SR=0`	824	SR bit	no	4.5
`MP=0`	828	MP bit	no	4.5
`HST=0` *n*	82n	1 or more bits	no	4.5
`?XM=0`	831	XM bit	yes	7.5 or 15.5
`?SB=0`	832	SB bit	yes	7.5 or 15.5
`?SR=0`	834	SR bit	yes	7.5 or 15.5
`?MP=0`	838	MP bit	yes	7.5 or 15.5
`?HST=0` *n*	83n	1 or more bits	yes	7.5 or 15.5

That's a lot to remember, isn't it?

`CLRHST` will clear the four bits of HST. You can also use other instructions, like `XM=0` or `SB=0` to clear a single bit of HST. There are four bits, thus four instructions to clear the bits.

There is also another way to set some bits of HST to zero using a formula, like this:

value = XM + (2 * SB) + (4 * SR) + (8 * MP)

Where 1 means to **clear** a bit and 0 means to set a bit, if you want to have:

XM = 0
SB = 0
SR = 1
MP = 1

You will calculate:

value = 0 + (2 * 0) + (4 * 1) + (8 * 1)
= 0 + 0 + 4 + 8
= 12

So by doing `HST=0 12` one can clear only the SR and MP bits. Note that an `HST=1 n` instruction does not exist.

There are instructions to test each of the four bits available, and the instruction `?HST=0 n` can be used to test whether some (or all) bits of HST are equal to zero, using the same formula as above. In this case, 1 in the formula also means that the test will check whether the bit is **clear**.

After those test instructions, you can put a `GOYES` or `RTNYES` if the test is true.

These instructions need six cycles to perform the test and continue if it's false or 13 cycles is the test is true and the jump is done. It may be a jump to a label or a return using `RTNYES` if we are inside a subroutine.

For example, let's write some code to return from a subroutine if bit XM is set to 1:

```
value   = 1 + (0 * 2) + (0 * 4) + (0 * 8)
        = 1
```

This means the code can be written like this:

```
   ?HST=0 1
   RTNYES
```

If we want to jump to the label "REACT" if XM=1 and SR=1, we first must calculate the value:

```
value   = 1 + (0 * 2) + (1 * 4) + (0 * 8)
        = 1 + 4
        = 5
```

So our code looks like this:

```
   ?HST=0 5
   GOYES REACT
```

To test the SB (Sticky Bit) after a rotation of bits or nibble on a working register we have:

```
   ?HST=0 2
   GOYES RESULT_LOSS
```

etc.

49 Saturn DEC/HEX mode

The Saturn processor can work under two modes: hexadecimal mode (base 16) or decimal mode (base 10). As we studied before, the latter uses compacted BCD.

Remember that with BCD, 4 bits are used for each decimal number to encode. Look back a few dozen pages for more about this.

Mnemonic	**Opcode**	**Fields**	**Carry**	**Cycles**
`SETHEX`	04	Hexadecimal mode	no	4
`SETDEC`	05	Compact BCD mode	no	4

50 Interruptions

Thanks to Christoph Giesselink for contributing this section.

Mnemonic	Opcode	Fields	Carry	Cycles
`INTOFF`	808F	Disable keyboard interrupts	no	7
`INTON`	8080	Enable keyboard interrupts	no	7
`RSI`	80810	Reset interrupts	no	8.5

What is an interrupt? Normally the CPU executes one instruction after another. But sometimes we want to have a reaction to external events, like keyboard input for example. We can solve this by continuously reading the keyboard input port. We call this "polling," where the external event should stop the CPU execution, call a specified subroutine that handles the event, and return to the application. This is called "Interrupt" handling. The code called by an interrupt is called the "interrupt handler."

The HP 48 and 49 have one interrupt handler which continues program execution at address #0000F. The interrupt is called by different sources: keyboard, timer, serial port, etc. Some of them are maskable; some are not maskable. A maskable interrupt source like the keyboard, timer or serial port can be disabled, but non-maskable interrupt sources like the [ON]-key always occur. But the interrupt isn't retriggerable: that means when we are in the interrupt service routine no other interrupt can occur until we leave the interrupt service routine with a RTI command. This is done with an internal CPU setting, a so-called "interrupt in service" flag. This is set when an interrupt occurs and the PC is set to #0000F and is released by the RTI command.

50.1 Disable all interrupts

Sometimes it's useful to disable all interrupts, even those which are non-maskable. But the only safe place where an interrupt cannot occur is inside an interrupt service routine. So we have to execute an interrupt and continue execution when the interrupt is in service.

This can be done with:

```
ST=0 15        % disable all interrupts
```

Normally, when an interrupt occurs the HP operating system checks bit 15 of the ST register, and if it's cleared, the OS sets bit 14 of the ST register (meaning that an interrupt is pending) and immediately returns with a `RTN`. But now that the "interrupt in service" flag is set, no interrupts can occur any more. If we want to avoid this first call, we have to create an interrupt before we enter the critical section where no interrupt is allowed.

To simplify this you can call:

```
GOSBVL =DisableIntr      % =DisableIntr is 01115 on the 48 and 26791 on the 49
```

This code clears bit 15 of ST, disables the keyboard interrupts with `INTOFF`, and executes an interrupt.

50.2 Enable all interrupts

To enable all interrupts it is best to call the `RTI` instruction, which clears the "interrupt in service" flag. If an interrupt is pending (bit 14 of the ST register is set) we should reset the interrupt system as well.

To enable the interrupt system we are calling the following subroutine with a `GOSUB REINT`:

```
*REINT
ST=1 15        % flag enable all interrupts
?ST=0 14       % interrupt pending?
GOYES NOINT    % no, continue
```

```
RSI                % reset interrupt system
*NOINT
RTI                % clear "interrupt in service" flag
```

To simplify this we can call:

```
GOSBVL =AllowIntr       % =AllowIntr is 010E5 on the 48 and 26767 on the 49
```

This code sets bit 15 of ST, enables the keyboard interrupts with `INTON`, resets the interrupt system if an interrupt is pending, and clears the "interrupt in service" flag with an `RTI` command.

51 Bus-related instructions

We will discuss the bus in more depth later, but you can use this section as a reference when learning all about the bus. The bus is linked to the processor, and there is much to learn about modules, memory configurations, and more. That could be a painful part...

The Saturn only has one bus. Usually one might find separate buses, for example, for data and instructions, but because the Saturn only has one, it is "multiplexed." The instructions here will be useful for configuring or unconfiguring a specific module of memory, so some parts are visible and others not.

There are three instructions that are not used on the HP 48: `BUSCB`, `BUSCC` and `BUSCD`.

The mnemonic means: BUS = Bus, C = Command, followed by letter of the command.

Some of the Saturn+ instructions in the HP 49G+ and 48GII, however, are based on the `BUSCC` instruction. That is why a number of the new Saturn+ instructions start with 80B.

Mnemonic	Opcode	Fields	Carry	Cycles
`RESET`	80A		no	7.5
`SREQ?`	80E		no	9.5
`CONFIG`	805	Field A of C	no	13.5
`UNCNFG`	804	Field A of C	no	14.5
`C=ID`	806	Field A of C	no	13.5
`SHUTDN`	807		no	6.5
`BUSCB`	8083		no	10
`BUSCC`	80B		no	8.5
`BUSCD`	808D		no	10

51.1 RESET

All modules that can be configured are set to the unconfigured state once this command is used. The ROM will not be unconfigured because it has no manager.

51.2 SREQ?

This doesn't seem to be used. It was supposed to issue the POLL bus command and latch the system bus into C(0) on the old HP-71, which was the first implementation of the Saturn architecture.

51.3 CONFIG

Used to configure a memory module. This is explained in section 66 later in this book.

51.4 UNCNFG

Unconfigures a memory module whose base address is in field A of C.

51.5 C=ID

Copies the ID of the current module of memory to field A of C. Its content will vary depending on the status of the module. This is explained in part discussing "Covered memory."

51.6 SHUTDN

Sets the calculator to low power usage mode. Only some essential components remain powered, and the HP sleeps until it is awakened by one of the following conditions:

- The timers (1 or 2) when both the wake bit (WKE) and the MSB in the corresponding timer control register are set
- A non-zero value in the IN register (caused when a pressed key is recognized)
- An interrupt from the serial receive buffer

51.7 BUSC*x*

These instructions are not used on the HP 48 or the traditional Saturn-based HP 49. The BUSCC instruction's opcode is used as a prefix for many of the new Saturn+ instructions in the 48GII and 49G+.

52 Filler instructions

There are three instructions that can be used as "filler" because they don't do anything. The only real purpose they serve is to keep entry points at the same place when writing code with absolute addressing by filling in gaps when code is removed. Absolute addressing means that the jumps to entry points within the code are all absolute jumps to specific memory addresses rather than relative jumps, which gives better performance but can make code maintenance a nightmare and places severe restrictions on how the final program can be installed on the calculator.

These "NOP" (no operation) instructions are available in 3, 4, and 5 nibble lengths.

Neither HP-ASM nor MASD understand these instructions, but you can always enter their opcodes manually by prefixing them with a dollar sign ($) if you absolutely feel you have to use them.

Mnemonic	Opcode	Fields	Carry	Cycles
`NOP3`	420		no	3 or 10
`NOP4`	6300		no	11
`NOP5`	64000		no	11

53 ARM system instructions

The calculators based on the ARM architecture, including the 49G+ and 48GII, have additional Saturn+ instructions, mostly for the purpose of interfacing with the ARM-based underlying architecture. HP has not released detailed information about these instructions, but the following table lists the new instructions and what little information has been gleaned from the comp.sys.hp48 newsgroup about them.

Mnemonic	**Opcode**		**Description**
`RPL2`	80B00	+	Returns to RPL (`A=DAT0.A D0+5 PC=(A)`).
`OFF`	80B01	+	Turn the calc off. Pressing [ON] will resume execution.
`CONFIGD`	80B03	+	Configures a 4KB block containing the 16-line header with the address in C(A) (must be multiple of 4K).
`REMON`	80B05	+	Enables remote control ([ON]-[R]) of the calculator. Transmits the display and accepts keyboard input through the USB port.
`ACCESSSD`	80B06	+	SD card access. Puts the card’s free space in the lower 8 nibbles of A, or a negative number corresponding to an error message if it fails, such as when a card is not present. Affected by P.
`GETTIME`	80B11	+	Allows you to use the timer, since it is no longer present in I/O RAM. Returns in C(W) a free running timer, in D(W) a countdown timer, and in B(W) a timer that starts counting from zero and goes up. Disable all interrupts before calling or it will crash the calculator.
`UNCNFGD`	80B13	+	Unconfigures the 16-line header and refreshes it.
`REMOFF`	80B15	+	Disables remote control ([ON]-[S]) of the calculator.
`PORTTAG?`	80B16	+	Returns in A(A) the port number (0-3) corresponding to the tag name specified in DAT1, stored with the length and then the characters.
`SETTIME`	80B21	+	Sets the real-time clock using the number of ticks in C(W).
`REFRESHD`	80B23	+	Refreshes the 16-line header.
`RESETOS`	80B31	+	Performs an On-A-F reset.
`SETLNED`	80B33	+	Sets the header to C(A) lines and refreshes the display. Use C=0 to disable the new header.
`AUTOTEST`	80B41	+	Diagnostics tests from the [ON]-[D] menu.
`SETOFFD`	80B43	+	Sets the header display offset to C(X) & 0x7FF bytes.
`BEEP2`	80B50	+	Uses the beeper. C(A) = time (ms). D(A) = frequency (Hz). P=0.
`NATIVE?`	80B51	+	Sets the carry if the native function represented by the next two nibbles is undefined, and clears the carry if it is defined.
`MOVEDN`	80B60	+	High-speed, native equivalent of MOVEDOWN. Takes the number of nibbles in C(A), the start of the source in D0, and the start of the destination in D1.
`OUTBYT`	80B65	+	Transmits byte in A(B) by infrared.
`MOVEUP`	80B70	+	High-speed, native equivalent of MOVEUP. Takes the number of nibbles in C(A), the end of the source in D0, and the end of the destination in D1.
`SERIAL`	80B71	+	Returns the serial number of the 49G+ into DAT1 using 10 bytes with ASCII encoding.
`CRTMP`	80B80	+	High-speed, native equivalent of CREATETEMP. Takes the size in C(A).
`HST=1.`*`n`*	80B82h	+	Sets individual bits of HST, with XM=1, SB=2, SR=4, and MP=8. Add up the values to set multiple bits at once. (The decimal value *n* in the mnemonic is encoded as its hexadecimal equivalent.)
`XM=1`	80B821	+	Sets XM. Alternate mnemonic for `HST=1.1`.
`SB=1`	80B822	+	Sets SB. Alternate mnemonic for `HST=1.2`.
`SR=1`	80B824	+	Sets SR. Alternate mnemonic for `HST=1.4`.
`MP=1`	80B828	+	Sets MP. Alternate mnemonic for `HST=1.8`.
`?HST=1.`*`n`*	80B83h	+	Checks whether individual bits of HST are set, with XM=1, SB=2, SR=4, and MP=8. Add up the values to check multiple bits at once. (The decimal value *n* in the mnemonic is encoded as its hexadecimal equivalent.)
`?XM=1`	80B831	+	Checks whether XM is set. Alternate mnemonic for `?HST=1.1`.

Mnemonic	Opcode		Description
`?SB=1`	80B832	+	Checks whether SB is set. Alternate mnemonic for `?HST=1.2`.
`?SR=1`	80B834	+	Checks whether SR is set. Alternate mnemonic for `?HST=1.4`.
`?MP=1`	80B838	+	Checks whether MP is set. Alternate mnemonic for `?HST=1.8`.
`HSCREEN`	80B92	+	Returns the height of the screen into A(A).
`KEYDN`	80BA0	+	Reads the keyboard. Uses C(A).
`WSCREEN`	80BA2	+	Returns the width of the screen into A(A).
`GOSLOW`	80BB0	+	Waits for (C(A) / 183) milliseconds.
`MIDAPP?`	80BB2	+	Clears carry on an HP 49G+, sets carry on the 48GII.
`BIGAPP?`	80BC2	+	Sets carry in HP 49G+, clears carry on the 48GII.
`SETFLD`*n*	80BF7h	+	Defines new fields F1 through F7 (where n = 1-7) for instructions. The field mask goes in C(W). (The values for *n* in the mnemonic, 1 through 7, become 8 through F to replace *h* in the opcode.)
`ARMSYS`	80BFE	+	Sets the ARM PC to the 32-bit value in C(B), after first clearing bits 0 and 1 to make sure the address lies on a 4-byte boundary.
`ARMSAT`	80BFF	+	Calls ARM code at the Saturn address specified by C(A).

There are some other 80Bxx instructions supported by the Saturn+, but they do not (yet) have mnemonics and are completely undocumented, aside from a few appearances in the comp.sys.hp48 newsgroup.

54 ARM enhancement instructions

A few more instructions were added to the ARM-based calculators to provide native ARM replacements for common operations. Little is known about them at the time of this writing, but the new instructions are below. They do not appear to have mnemonics, so you will need to manually enter their opcodes (proceeded with a $ symbol) to use them. This list is mainly provided for those interested in the low-level operation of the ARM-based calculators.

Mnemonic	Opcode		Description
not available	81B1	+	Found at the beginning of a return-to-RPL loop (`A=DAT0.A D0+5 PC=(A)`) and used to return to RPL.
not available	81B8, 81B9, 81BA 81BB, 81BC	+ +	These all apparently have something to do with a faster SKIPOB equivalent.
not available	81BD	+	Used at the very beginning of a SEMI to run faster replacement code.
not available	81BE	+	Used at the very beginning of a DOCOL to run faster replacement code.

Part V: Objects

55 Objects overview

Objects are an essential part of the HP 48 and 49. Everything put on the stack is an object, and they're all encoded the same way: they start with a 5-nibble prologue that identifies the object, and then the object follows in what is called the "body" of the object. The object is, in a way, a shell, and data is put inside of it.

What is the prologue? It's not only something to identify the object, but in fact it also contains the address of a code routine that is used to execute the object. So the prologue not only identifies something but also executes it.

Some objects, when evaluated, just put themselves on the stack, like strings. If you [EVAL] a string, it remains on the stack. But, if you put a global name onto the stack and [EVAL] it, the contents of the variable that corresponds to the global name is put on the stack. The global name's prologue was evaluated, and it pushed the variable's contents onto the stack.

We are now going to study each object type of the HP 48 and 49, so this is a rather long part. :-)

Perhaps you know there is a command called TYPE that returns a value, according to the object type that is on the stack. There are 28 TYPE values, but we'll soon see that's not enough. ;)

Of those 28 object types, only some can be used from RPL.

Some objects I will describe are *compound*, like the HOME directory object or the library object (the most interesting of all), and are made up of other objects, but others, like strings, are *simple*.

The following table gives each object type's name, its value given by the User RPL TYPE command, its System RPL code, its prologue, and whether it is simple or compound:

Object type name	TYPE	Code	Prologue	Kind
Real number	0	1	02933	Simple
Complex number	1	2	02977	Compound
String	2	3	02A2C	Simple
Array	3/4	4	029E8	Compound
List	5	5	02A74	Compound
Global name	6	6	02E48	Simple
Local name	7	7	02E6D	Simple
RPL program	8	8	02D9D	Compound
Algebraic expression	9	9	02AB8	Compound
Symbolic	N/A	A	N/A	Either
Binary integer	10	B	02A4E	Simple
Graphic object	11	C	02B1E	Simple
Tagged object	12	D	02AFC	Compound
Unit object	13	E	02ADA	Compound
XLIB name	14	0F	02E92	Simple
Directory object	15	2F	02A96	Compound
Library	16	8F	02B40	Compound
Backup	17	9F	02B62	Compound
(not used)	18	N/A	N/A	N/A
(not used)	19	N/A	N/A	N/A
System binary	20	1F	02911	Simple
Long real	21	3F	02955	Simple
Long complex	22	4F	0299D	Compound

Object type name	**TYPE**	**Code**	**Prologue**	**Kind**
Linked array	23	5F	02A0A	Compound
Character	24	6F	029BF	Simple
Code object	25	7F	02DCC	Simple
Library data	26	AF	02B88	Simple
Extended pointer (49 only) or EXT1	27	BF	02BAA	Simple
Minifont (49 only)	27	DF	026FE	Simple
EXT3	27*	N/A	02BEE	N/A
EXT4	27*	EF	02C10	N/A
Infinite precision real (49 only)	27**	N/A	0263A	Compound
Infinite precision complex (49 only)	27**	N/A	02660	Compound
Flash pointer (49 only)	27	N/A	026AC	Simple
Aplet (49 only)	27***	N/A	026D5	Compound
Infinite precision integer (49 only)	28	FF	02614	Simple
Symbolic matrix (49 only)	29	N/A	02686	Compound
Font (49 only) or EXT2	30	CF	02BCC	Simple

*These are reserved by HP for future use, if needed.
**Not used by the 49's built-in software but supported by some third-party software.
***Never fully implemented in the 49 and not used by any known software.

56 Simple Objects

56.1 Real number

Prologue:	02933	Epilogue:	none
Size:	21 nibbles	Type:	0

In memory:

Prologue (02933)	5 nibbles
Exponent	3 nibbles
Mantissa	12 nibbles
Sign	1 nibble

Compare this to the S, M and X fields of the registers – notice that they are the same?

This is a typical object type and will be used commonly. The real number is divided into three fields: the mantissa, the sign, and the exponent. The mantissa is the number, the exponent is *n* in 10^n; and the sign is coded using one nibble, where positive is #0h and negative is #9h

The mantissa uses 12 digits, and the number is encoded in memory using compacted BCD (Binary Coded Decimal). The exponent is also encoded using BCD, but in a special way according to its sign.

If the exponent is positive, the 3 nibbles are encoded using BCD.
If the exponent is negative, the exponent is encoded as follows:

```
   1000 - ABS(exponent)
```

ABS means the absolute value of the exponent, whose value goes from 0 to 499

For example, let's encode 2.79233×10^{-9}. First, we have the prologue, which is #33920h (remember, it must be reversed). Then, we have the exponent, which here is 1000 - ABS(-9). That's 1000 - 9 = 991, so we have #199 when reversed. The mantissa is 279233, and because it's encoded reversed (like everything else in memory), it's #000000332972h. Finally, the sign is placed: #0h.

In memory we have #33920 199 000000332972 0h (The spaces are for clarity and are not in the real object.)

CAUTION! The Saturn reverses the order of nibbles in memory. This is why each field here is reversed in memory!!

56.2 String

Prologue:	02A2C	Epilogue:	none
Size:	12 nibbles and up	Type:	2

In memory:

Prologue (02A2C)	5 nibbles
Size	5 nibbles
Character 1	2 nibbles
…	
Character *n*	2 nibbles

The size of a string is five nibbles (the five nibbles needed to encode the size) plus two nibbles per character. As with all other objects, the prologue is not considered in the size.

Each character is encoded using 2 nibbles, using ASCII (described below). The size is everything, except the prologue, so it's five nibbles plus 2**n* nibbles (5+2*n*) long.

ASCII is a table used to code characters and stands for American Standard Code for Information Interchange. It uses eight bits (that's one byte or two nibbles) to encode values. Using eight bits lets us have up to 2^8, or 256, different values.

In fact, the original ASCII table only used 7-bits, so only 128 different symbols existed. Today, we use extended ASCII, which uses all eight bits.

The first 32 characters have special meanings, for communicating or printing; these characters, which are called "non-printable characters," include page jumps, tabs, and other codes used between devices to communicate with each other. The first 128 characters are standard, and the second 128 characters vary from country to country and from computer to computer: DOS, Windows, Macintosh, Cyrillic, European, Western, etc.

You can take a look at the HP 48 ASCII table by pressing [Right-Shift][PRG]. At first you will see characters 128 to 191, and you can see 64 characters before using "-64" or 64 characters after using "+64." On the HP 49G, the ASCII table is reached with [Right-Shift][CAT], and on the other 49 series calculators, you must use [Right-Shift][EVAL]. On all HP 49 calculators, any character can be viewed by navigating with the arrows.

Therefore, characters from 0 to 127 are standard, as they were the first 7-bit ASCII standard. Characters from 128 to 255 vary from one platform to another (PC, Mac, etc.) and from one alphabet to another (Roman, Cyrillic, Kanji, etc.).

For example, the string "HP" will be encoded as #02A2Ch for the string prologue plus the body of the objects. Remember we have two characters, occupying four nibbles. Five nibbles are used to encode the size (#9h nibbles), and finally the four nibbles of the characters (#4850). This means that the string object is encoded as follows:

```
#C2A20 90000 8405h
```

Note that the two nibbles for each character are reversed, just like anything else.

#48h is #72d, which is the "H" character.
#50h is #80d, which is the "P" character.

We can manipulate strings using assembly language. Let's write a program that changes all letters inside a string to UPPERCASE.

First, take a look at the CHARS table using [Right-Shift][PRG], and notice that there is a fixed distance between the 'A' and 'a' characters: 32. If you have a lowercase character (in other words, a character whose value is between 97 and 122 inclusive), we can get its corresponding UPPERCASE value by removing 32. That's simple, isn't it? :-)

This example will be interesting: it will include a loop that uses the carry, where you'll see how to calculate the loop value, and several tests. It's a good but not too complex example. :)

(The program to turn your HP into a Star Trek phaser using the incredible boiling power of the IR diodes will come later, but be patient, as it looks like HP doesn't want that secret code to be revealed!)

The code is commented:

```
% -- "Too Few Arguments" upon empty stack ---

C=DAT1 A            % read first level object's address
?C#0 A              % if it's not empty (different than zero)...
GOYES STRING?       % check if it's a string
LA 00201            % load #201h into A
*ERROR              % this label is needed if there isn't
                    % a string on the first level of stack
GOVLNG =Errjmp      % =Errjmp is 05023, which is a routine that
                    % does error #201h: "Too Few Arguments"

% --- check if we have a string ---

*STRING?            % label
CD1EX               % point D1 to the object
A=DAT1 A            % read its prologue
D1=C                % and restore D1
LC 02A2C            % load string's prologue onto C
?A=C A              % if the object is a string
GOYES MAIN          % jump to MAIN
LA 00202            % otherwise, we load #202h error code
GOTO ERROR          % which is "Bad Argument Type" and
                    % jump to ERROR, where the GOVLNG to
                    % the DOERROR routine is in ROM

% --- prepare to loop ---

*MAIN               % kind of main routine
GOSBVL =SAVPTR      % =SAVPTR is 0679B, which saves registers D1, D0, B, and D
C=DAT1 A            % read string's address
D1=C                % point to it
D1=D1+ 5            % pass the prologue and point to the
                    % size nibbles
C=DAT1 A            % read size of string
C=C-5 A             % remove its length

D=C A               % and store it into D
```

```
% --- calculate loop value ---

% D contains a number of nibbles.  Since there are 2 nibbles
% per char, I'm going to divide D(A) by two, which will be
% done using DSRB A (shift right to divide by two).

DSRB A                   % divide by two
D=D-1 A                  % and remove one because I'm going to use
                         % a carry loop
GOC END                  % if carry is set, the string is empty, and then
                         % jump to the end and leave
D1=D1+ 3                 % move 3 nibbles forward, not 5,
                         % so when LOOP is entered, the D1=D1+ 2
                         % there points to the first char
                         % upon entering the loop, or points to the
                         % next char if it loops :-)

% --- loop ---

*LOOP
D1=D1+ 2                 % point to char
C=DAT1 B                 % read it into C
LA 61                    % #61h is #97d
?C<A B                   % if below 97, skip it
GOYES CHECK
LA 7A                    % #7Ah is #122d
?C>A B                   % if above 122, skip it
GOYES CHECK

% here we have a lowercase char, and remove 32 so we get
% an uppercase one:

C=C-16 B
C=C-16 B                 % (char value) - 32

DAT1=C B                 % write it to the string

*CHECK
D=D-1 A                  % is there another char?
GONC LOOP                % if so, jump to LOOP

% --- quit here ---

*END

GOSBVL =GETPTR           % =GETPTR is 067D2, which restores registers D0, D1, B, and D
GOVLNG =LOOP             % =LOOP is 2D564 on the 48 and 05149 on the 49, which returns to RPL
@
```

Now put a string like: “aBcDeFGhij” and run the code. You’ll get “ABCDEFGHIJ”. The Code object is only 84 nibbles long. It’s small, isn’t it? :)

(If you don’t get the same size, something may be wrong)

As practice, write a program to do the opposite: write code that turns to all characters in a string to lowercase.

56.3 Global name

Prologue:	02E48	Epilogue:	none
Size:	7 nibbles and up	Type:	6

In memory:

Prologue (02E48)	5 nibbles
Size	2 nibbles
Character 1	2 nibbles
…	
Character *n*	2 nibbles

Each character is encoded using ASCII, with two nibbles per character. On the stack, global names are stored inside single quotes (' ').

For example, 'HP48' is encoded as #02E48h for the prologue, #04h for the number of characters (up to 255), and finally two nibbles per character; that is:

```
#84E20 40 84054383
```

The HP does not let us create *any* name by default, so how about creating “invalid” global names? :)

Our program will take a string as parameter and put the global name on the stack. The main difference between the string and global name objects is the global name can only have up to 255 characters, as its length is encoded using two nibbles, whereas the string encodes the length using five nibbles.

NOTE: The following program **does not** check the length of the string given in input. Don’t give it a string longer than 255 characters.

If you give it an empty string, you’ll get an empty global name; otherwise it places a global name with the string’s contents on the stack.

```
C=DAT1 A          % read level 1's content
R0=C A            % save into R0 for future use (if needed)
?C#0 A            % if level 1 contains something
GOYES TEST        % check if it's a string
LA 00201          % if not, do error "Too Few Arguments"
*ERROR            % (used to jump here if a string isn't given)
GOVLNG =Errjmp    % =Errjmp is 05023, which makes the error,
                  % with the error code inside A(A)
*TEST
CD1EX             % point to the object
A=DAT1 A          % read its prologue
D1=C              % and restore D1 to first level of stack
LC 02A2C          % 02A2C = string's prologue
?A=C A            % if a string, jump to
GOYES ALLOCATE    % allocate memory
LA 00202          % Otherwise, error #202h, which is
GOTO ERROR        % "Bad Argument Type"
*ALLOCATE
GOSBVL =SAVPTR    % =SAVPTR is 0679B, which saves registers D0, D1, B, and D
C=R0 A            % C contains the address of object in level one
D1=C              % point D1 to object
```

```
D1=D1+ 5            % point to string size
C=DAT1 A            % A(A) contains string size
C=C-5 A             % remove its length field size
CSRB A              % and divide it per two, now C(A) contains
                    % number of characters inside of the string
R1=C A              % save the number of chars into R1(A)
C=C+C A             % multiply it by two as we need its size in nibbles
                    % to reserve memory
C=C+7 A             % add 7, with 5 nibbles for global name
                    % prologue and 2 for the length field
GOSBVL =GETTEMP     % =GETTEMP is 039BE, which allocates
                    % Ca nibbles and produces an "Insufficient Memory"
                    % error if there is not enough RAM. It also checks if
                    % 5 nibbles are free, so we will be able to push
                    % the reserved object onto stack.
CD0EX               % D0 = newly reserved object, now C contains its
                    % address
R2=C A              % save it into R2(A) (at the end we will push R2(A)
                    % on stack; that's why we save it)
D0=C                % D0 points to the new reserved memory
LC 02E48            % 02E48 = global name prologue
DAT0=C A            % write it
D0=D0+ 5            % and move 5 nibbles further
C=R1 B              % recover the size that was calculated
DAT0=C B            % and write it to the global name length field
C=C-1 B             % remove one, if CARRY is set
GOC PUSH            % then string is empty, so push the empty global
                    % name on stack
A=R0 A              % recover string's address
D1=A                % make D1 point to it
D1=D1+ 8            % and move 8 nibbles further. Why not 10?
                    % because inside the loop we have a D1=D1+ 2
                    % so we finally get D1=D1+ 10 ! :-)

*LOOP               % in this loop, we copy the chars one at a time
D1=D1+ 2            % move 2 nibbles forward, if we enter the loop,
                    % we get into the first char; otherwise, we move
                    % to the next two chars.
D0=D0+ 2            % move 2 nibbles forward in the new reserved obj
A=DAT1 B            % read one char from the string
DAT0=A B            % and write them in the global name
C=C-1 B             % remove 1 from the counter
GONC LOOP           % and loop until carry is set

*PUSH               % here, push the global name on stack
GOSBVL =GETPTR      % =GETPTR is 067D2, which restores registers D0, D1, B, and D
D=D-1 A             % remove 5 nibbles from RAM
D1=D1- 5            % and push one object into the stack
C=R2 A              % recover the global name address
DAT1=C A            % push it on stack
A=DAT0 A            % read next RPL object to execute
D0=D0+ 5            % update the D0 pointer to the next object
PC=(A)              % and jump to continue program flow...

@                   % *  END OF SOURCE  *
```

Try this with an empty string! If you [EVAL] the resulting empty global name when in HOME, you will get inside the hidden directory. The hidden directory is described next:

56.3.1 The empty global name and the hidden directory

You can create an empty global name, which has a special function on the HP: if you [EVAL] it, you will get inside the hidden directory of HOME. This directory contains three objects by default: Alarms, UserKeys, and UserKeys.CRC. The first one contains a list of alarms, the second one a list with every keyboard assignment, and the last one a four-nibble binary number, which is the CRC for the UserKeys. One rule: *never*, *ever* change the order of the variables in this directory.

When you are inside a folder inside of HOME, you can use this empty global name to hide any variable following it in the variable list. For example, if you have four variables, and store a null global name after the second one, the two final variables become "invisible," but you can still recall their value or use them!

You probably want to see this in action.

First, make sure you are in HOME, by simply pressing [Right-shift][']. Then put #15777h (on the 48) or #272FEh (on the 49) on the stack (make sure there's an h) and then type SYSEVAL [ENTER]. Now you have the null global name on stack. Just use [EVAL] to jump into it!

When you are not in HOME, you can use the null name to hide variables. Everything stored *after* the null global name disappears.

Before you forget: if you reorder the invisible directory's contents, you'll get a memory loss. You have been warned twice now.

To exit the invisible directory, do [Left-shift]['] (UPDIR) or [Right-shift]['] (HOME) on the 48 or [Left-shift-held]['](UPDIR) on the 49.

Do you want to hide the contents of a directory? Go into the desired directory, type #15777h SYSEVAL [EVAL] (on the 48) or #272FEh SYSEVAL [EVAL] (on the 49) to put the null global name on stack, put something like 1 on the stack, and press [SWAP][STO] to store it. We now have a null global name with the value 1 on it. That's not important, because what is important is your variables are no longer visible. To see them again, put the null global name on the stack again and PURGE it.

56.4 Local name

Prologue:	02E6D	Epilogue:	none
Size:	7 nibbles and up	Type:	7

In memory:

Prologue (02E6D)	5 nibbles
Size	5 nibbles
Character 1	2 nibbles
…	
Character *n*	2 nibbles

As you can see here, the prologue is the sole difference between a local name and a global name. But, the local name is not available all the time, like the global name.

56.5 User binary integer

There are two kinds of binary integers: a 15-nibble size, and an unlimited (within memory, of course) size. The HP only lets us create the smaller binary integer, but we can create the larger kind manually.

Prologue:	02A4E	Epilogue:	none
Size:	11 nibbles and up	Type:	10

In memory:

Prologue (02A4E)	5 nibbles
Size	5 nibbles
Digit 1	1 nibbles
…	
Digit *n*	1 nibbles

When you enter a number starting with #, the HP will convert it to the current base being used. If you are using the HEX base, if you type # without the last char (h for hex) then the HP will consider it's a hexadecimal number. If you use a letter at the end (h for hex, d for decimal, b for binary, or o for octal) then the HP will convert it automatically to the current base being used.

Each digit here will use one nibble.

As an example, let's encode #BAh:

The prologue is #02A4Eh, as the preceding table shows. The size is #00015h, as the HP won't allow anything else (see below to get around this). The value is #AB00000000000000h, so we have:

```
#E4A20 51000 AB00000000000000
```

The code below will take a binary number as input, and after running, it will place that binary number with the number of specified digits on the stack, assuming there is enough RAM. You can even create a 200-digit binary number if you want.

This program creates an 85-digit binary number:

```
GOSBVL =SAVPTR     % =SAVPTR is 0679B, which saves registers D0, D1, B, and D
LC 0005A           % reserve 90 nibbles
GOSBVL =GETTEMP    % =GETTEMP is 039BE, which allocates memory
CD0EX              % we save object's address
R0=C A             % into R0(A)
D0=C               % restore D0
LC 02A4E           % binary number prologue
DAT0=C A           % write it
D0=D0+ 5           % move 5 nibbles further
LC 00055           % size is 90-5=85, #55h
DAT0=C A           % write it to the object's size zone
GOSBVL =GETPTR     % =GETPTR is 067D2, which restores registers D0, D1, B, and D
D=D-1 A            % remove 5 nibbles of free ram
D1=D1- 5           % push object on stack
C=R0 A             % its address is inside R0(A)
DAT1=C A           % we push it
A=DAT0 A           % and here we quit
D0=D0+ 5           % as usual
PC=(A)             %
```

@

56.6 Graphic object

Prologue:	02B1E	Epilogue:	none
Size:	22 nibbles and up	Type:	11

In memory:

Prologue (02B1E)	5 nibbles
Size	5 nibbles
Number of rows	5 nibbles
Number of columns	5 nibbles
…	Pixels

This is a very commonly used object. A picture, or a GROB (GRaphic OBject), contains pixels, or more precisely, data that represents pixels. The graphic object first contains a prologue, #02B1Eh, followed by the size of the whole object, but the five nibbles of the prologue are not counted. The next two sets of five nibbles encode the number of lines and rows.

The pixels follow, coded one after another, from the upper-left corner to the lower-right corner. Bytes are used to keep them together. Because the data in a GROB is encoded using bytes rather than nibbles, each byte encodes eight bits, which form eight pixels of the GROB. A bit set to 1 is a pixel turned on, and a bit set to 0 is a pixel turned off. The word "pixel" is a contraction for "picture element."

IMPORTANT: In each byte, pixels are encoded using a reversed nibble. Don't forget that pictures must be encoded in a multiple of bytes, or two nibbles. This means that every GROB must have an even number of hexadecimal numbers to encode the pixels. Because each nibble encodes four pixels, each GROB must be a multiple of eight pixels.

Don't worry if your GROB does not have a size that is a multiple of eight pixels. We can simply "pad" the data with zeroes, meaning as many zeroes needed to create a multiple of eight must be added to the end. We could even pad the GROB with a small amount of non-zero values, if we want to hide a small amount of data inside the object.

For example, suppose I want a pixel off, a pixel on, a pixel off, a pixel on, a pixel off, and then three pixels on, all in one line. The screen would look like this, with 1's being powered pixels and 0's being blank, and the space is added for clarity:

```
0101 0111
```

This has eight pixels, and thus two nibbles, so we don't have to worry about adding trailing zeroes. When we encode this, we must reverse the contents of each nibble, so the pixels must be encoded as `1010 1110`. That's #AEh. After adding the prologue (#02B1Eh), size (#19d, or #13h nibbles), height (#1h), and width (#4h) to the beginning, our graphic is encoded as follows:

```
#E1B20 31000 10000 40000 AE00
```

Note that two trailing zeroes were added, and the addition was reflected in the size of the GROB. This was just to make the length #13h rather than #11h so you would remember that the nibbles must be inverted. In addition, this shows that you can have excess data within an image without hurting anything. Of course, if these zeroes were removed the GROB would take one byte less memory.

A note to Meta Kernel and HP 49 users: the Meta Kernel and the 49 add a zero-valued byte to the end of a graphic to indicate it is a grayscale GROB, so this GROB will have a "g" at the end of its size when shown on the stack. Simply drop these two zeroes and correct the length and the "g" will disappear.

As a refresher, the #13h (#19d) size was calculated by adding five nibbles for the size field, five nibbles for the width field, five nibbles for the height field, and, in this case, four nibbles for the pixel data.

56.7 XLIB name

Prologue:	02E92	Epilogue:	none
Size:	11 nibbles	Type:	14

In memory:

Prologue (02E92)	5 nibbles
Library ID	3 nibbles
Command number	3 nibbles

This is how a command inside a library is referenced on the HP 48 and 49. Even though you may see the library command name, the calculator uses this object internally. When you have, for example, DTAG on the stack it's an XLIB object, which the calculator translates into the command's name when displaying it onscreen. It contains the library number and the command number. When the command has a "user" form (a name you can type) it's displayed, otherwise "XLIB …" appears on the stack.

There is a command you can use to indirectly use XLIB's: the command LIBEVAL. LIBEVAL takes a six-digit binary value (#...) as an argument; the first three digits are the library number and the last three digits are the command number, both in hexadecimal.

56.8 System binary integer

Prologue:	02911	Epilogue:	none
Size:	10 nibbles	Type:	17

In memory:

Prologue (02911)	5 nibbles
Number	5 nibbles

This is a very simple object. The HP's RPL operating system uses it extensively, and it contains a binary integer encoded using five nibbles (that's 20 bits). The number will be shown according to the active base, **but** it's always encoded in hexadecimal.

56.9 Long real

The long real is also called the "extended real."

Prologue:	02955	Epilogue:	none
Size:	26 nibbles	Type:	21

In memory:

Prologue (02955)	5 nibbles
Exponent	5 nibbles

Mantissa	15 nibbles
Sign	1 nibble

The long real is more precise than the standard real. The user does not have access to this number, but the HP uses it internally for greater precision during calculations. It's encoded in a way similar to the standard real, but here some fields have greater sizes. BCD is used here, just like with the real object. Refer to the standard real information for details.

56.10 Character

Prologue:	029BF	Epilogue:	none
Size:	7 nibbles	Type:	24

In memory:

Prologue (029BF)	5 nibbles
ASCII code of character	2 nibbles

The character is one of the simplest objects available on the HP! In the case of this object, the prologue is actually bigger than the data being encoded. Each of these objects contains a character the HP is able to display. Characters 0-128 are the standard ASCII ones, but the HP has some special characters above. Here we have two nibbles per character, so we can code all 256 chars of the extended (8-bit) ASCII table. Because the original ASCII table was only 7-bit, only the first 128 characters are "universal."

56.11 Code object

Prologue:	02DCC	Epilogue:	none
Size:	10+ nibbles	Type:	25

In memory:

Prologue (02DCC)	5 nibbles
Size	5 nibbles
Saturn machine code	varies

This is the kind of object that HP-ASM, ASM Flash, MASD, Jazz and other code compilers produce.

Because it has the same structure as strings, some people use Code objects to hide strings, or some other object, inside. That's as simple as changing the prologue, which is so easy you know how to do it now, don't you? :o)

56.12 Library data

Prologue:	02B88	Epilogue:	none
Size:	10 nibbles and up	Type:	26

In memory:

Prologue (02B88)	5 nibbles
Size	5 nibbles

Data	varies

HP has made this object available to programmers so libraries that need to save values can save information inside a separate object. The coder will choose what data and how will be put inside this object, so it's a convenient way to store information, a kind of HP Cookie. ;-)

One common use is to store pieces of Code that are called by other programs; as these pieces of Code don't check the stack, this is a way to prevent the user from "accidentally" running these Code objects. You should be able to easily code a "launcher" for such objects. The goal is to protect your data.

56.13 Extended pointer

Prologue:	02BAA	Epilogue:	none
Size:	15 nibbles	Type:	27

In memory:

Prologue (02BAA)	5 nibbles
Object's address	5 nibbles
Program's address	5 nibbles

As you will learn in the next part, some areas of memory are covered by others, but sometimes we need to point something that is covered. This is the object the HP uses internally for that purpose. There are routines in ROM, which we'll discuss later, that are used to "uncover" the part we want to use. In this case, the object is what we want access to, and the program is the ROM routine that will be associated with the object's address, and will unconfigure and reconfigure memory modules as needed.

56.14 Infinite precision integer (49 only)

Prologue:	02614	Epilogue:	none
Size:	11 nibbles and up	Type:	28

In memory:

Prologue (02614)	5 nibbles
Size	5 nibbles
Digit 1	1 nibble
…	
Digit *n*	1 nibble
Sign (except for 0)	1 nibble

The size of an infinite precision integer, also called a ZINT, is five nibbles (the five nibbles needed to encode the size) plus one nibble per digit, plus one nibble for the sign. The number 0 has no nibble for the sign. As with all other objects, the prologue is not considered in the size.

Each digit is encoded using 1 nibble. The size is everything, except the prologue, so it's five nibbles plus *n* nibbles plus one sign nibble (6+*n*) long. The number 0 therefore has a length of 6 nibbles.

The sign is 0 if positive and 9 if negative.

56.15 Flash pointer (49 only)

Prologue:	026AC	Epilogue:	none
Size:	12 nibbles	Type:	27

In memory:

Prologue (026AC)	5 nibbles
Pointer to table A	3 nibbles
Pointer to table B	4 nibbles

This object functions as a memory mapping utility, which makes it easier for HP's developers to generate new ROM versions while keeping the same entry points for routines. The first pointer, which is 3 nibbles, points to a table (which we'll call table A) in system RAM at address #86037h. Because table A contains values that are 5 nibbles long, this first pointer is multiplied by 5 to get the address within table A. The value at this address is used by the system to configure the flash memory for running the routine, and we can consider this to be the bank in flash memory.

The second pointer, which is 4 nibbles, points to a table (which we'll call table B) at address #40222h. Table B also contains values that are 5 nibbles long, but these values are addresses to the specific routines that are referenced by the flash pointers. We can consider this to be the command number.

56.16 Minifont (49 only)

Prologue:	026FE	Epilogue:	none
Size:	1548 nibbles	Type:	27

In memory:

Prologue (026FE)	5 nibbles
Size	5 nibbles
Minifont ID number	2 nibbles
Minifont graphic	1536 nibbles

The minifont is relatively simple because of built-in constraints. After the prologue comes the size, which is always #607h nibbles (#1543d), because nothing in the minifont object is of variable length. The next number is the two-nibble ID number of the minifont, which must be between #0h and #FFh (#255d).

Last comes the font graphic itself. Each character is stored separately, in graphics that are 6 pixels tall and 4 pixels wide. The pixels in each one of these graphics are stored just like the pixels in a GROB, described earlier, taking 6 nibbles. Because there are 256 characters, the entire graphic section of the minifont takes 1536 nibbles.

56.17 Font (49 only)

Prologue:	02BCC	Epilogue:	none
Size:	4130 nibbles and up	Type:	30

In memory:

Prologue (02BCC)	5 nibbles
Size	5 nibbles

Font height (pixels)	2 nibbles
Font ID number	2 nibbles
Size of font name	2 nibbles
Font name	Size varies
Size of font name	2 nibbles
Font graphic	4096 nibbles

The font object is a little more complicated than the minifont. After the prologue and the size (which, as always, doesn't include the length of the prologue) is the height of the font, which is stored in two nibbles. This should be between 6 and 8 pixels. The next number is the ID number of the font. Each font in the system must have a unique ID to be accessed, and the ID number must be between #0h and #FFh (#255d).

Next is the size of the name of the font. This is also two nibbles and should be at least 8. This is because the 49 requires that the name be at least 8 characters long in order to display the font properly. After the size of the name is the name itself, which, like everything else, must have the characters in reverse order, and each character takes two nibbles. After the name, strangely, is the size of the name again, taking two nibbles once more.

Last comes the font graphic itself. Each character is stored separately, in graphics that are 8 pixels tall and 8 pixels wide, regardless of the font size. The pixels in each one of these graphics are stored just like the pixels in a GROB, described earlier, taking 16 nibbles. Because there are 256 characters, the entire graphic section of the font takes 4096 nibbles.

57 Compound Objects

57.1 Complex number

Prologue:	02977	Epilogue:	none
Size:	37 nibbles	Type:	1

In memory:

Prologue (02977)	5 nibbles
Real exponent	3 nibbles
Real mantissa	12 nibbles
Real sign	1 nibble
Imaginary exponent	3 nibbles
Imaginary mantissa	12 nibbles
Imaginary sign	1 nibble

This is the complex number object. In fact, representing a complex number is just keeping two real numbers together. The first one is the real part of the complex, and the second one is the imaginary part, with each number being encoded using 16 nibbles.

For example, let's encode (100,1000).

First comes the prologue, #77920h (reversed, as usual), then the real exponent, #200h, then the mantissa, #000000000001 (for the real part), and then the sign, #0h. Next comes the imaginary part, which is encoded as #300h for the exponent, #000000000001h for the mantissa, and finally the sign, #0h. This means the complex number is stored in memory like this:

#77920 200 000000000001 0 300 000000000001 0

57.2 Array

Prologue:	029E8	Epilogue:	none
Size:	27 nibbles and up	Type:	3/4

In memory:

Prologue (029E8)	5 nibbles
Size	5 nibbles
Kind of objects	5 nibbles
Number of dimensions	5 nibbles
Dimension 1	5 nibbles
…	
Dimension *n*	5 nibbles
Content of object 1	Size varies
…	
Content of last object	Size varies

The array object can contain any number of objects that are of the same type. When it only has one dimension, it's called a "vector." If there are two dimensions, we have a "matrix." We can also give it more than two dimensions, as five nibbles are available to encode the array's dimensions.

Usually, arrays are used to keep sets of real or complex numbers together. You can store any kind of object you would like using as many dimensions as you want, but the HP will only manage one- or two-dimensional arrays.

The prologue is #029E8h. Within the object, we must first define which objects are going to be kept inside the array object. Then, we have to tell the size of the whole array, but it depends on the kind of objects we're going to collect, and of course, how many of them are to be encoded.

You will see that encoding the length of an object is useful: the HP can use it to "jump" over the object. Most objects will have their size inside of them, but some won't, so the "epilogue" will be very useful to find the end of these objects.

What is interesting here is that even if we are collecting ten reals, there are not ten real prologues stored into the array: their prologue is only stored once. This is why the kind of objects is encoded using five nibbles and also why only **one** kind of object can be stored inside an array. Once we have set the kind of objects inside the array, we encode the number of dimensions using five nibbles: #00001h for vectors or #00002h for matrices. After the dimensions, we must tell how many objects are in *each* dimension. If we have a matrix, we will encode the number of lines and columns.

Once we have done all that, we can start encoding the objects inside the array. The way values are stored will vary according to the kind of objects stored. The prologues are not needed because we have already recorded it.

An important thing is that no matter what the size of the array is or what the objects are, if some parts are "empty," they still use memory.

For example, let's create an array of strings, with some names inside of it.

```
[ HP48S HP48G ]
```

First, we have the array prologue: #029E8h. As usual, it will be encoded reversed in memory. The total size is #00032h (you will learn why later; this takes five nibbles to encode). Next we must tell what kind of objects are included–strings–so we use #02A2Ch (five nibbles). There is only one dimension so we must write #00001h (another five nibbles), and there are two elements so we must write #00002h (five more nibbles).

Finally we encode the two strings. Since there are two nibbles per character, each string takes ten nibbles plus five more nibbles in front of it to tell its length. Because each string takes 15 nibbles, including the length field, the length is #0000Fh, reversed to show #F0000h. This means a total of 50 nibbles (hence the #32h length, as #50d = #32h), plus the prologue are needed to encode the object, like this:

```
    #8E920 23000 C2A20 10000 20000 F00008405438335 F00008405438374
```

The HP's stack doesn't show arrays of strings, so it will simply say "Array of String." If you are running the Meta Kernel, however, it will show the true array.

57.3 List

Prologue:	02A74	Epilogue:	0312B
Size:	10 nibbles and up	Type:	5

In memory:

Prologue (02A74)	5 nibbles
Object 1	Size varies
…	
Last object	Size varies
Epilogue (0312B)	5 nibbles

The list object is interesting: it can contain any kind of object, and as many objects as RAM allows. This is done by using a prologue to mark the beginning and an epilogue to mark the end, with each object (including its prologue) between them. This is found frequently in the ROM.

An empty list will be { } and 10 nibbles long: there is a prologue and an epilogue but nothing between.

For example, let's encode the list `{ 10.2 "dX" }`.

I have chosen two kinds of objects we have already seen: a real number and a string. It's quite easy: first encode each object, and then put the two between the list's prologue and epilogue:

10.2 is `#33920 100 000000000201 0`
"dX" is `#C2A20 90000 4685`

This is how the list would be encoded:

```
#47A20 339201000000000002010 C2A20900004685 B2130
```

57.4 RPL program

Prologue:	02D9D	Epilogue:	none
Size:	20 nibbles and up	Type:	7

In memory:

Prologue (02D9D)	5 nibbles
<< User RPL start (48: 2361E or 49: 389B9)	5 nibbles
…	
>> User RPL end (48: 23639 or 49: 389D4)	5 nibbles
Epilogue (0312B)	5 nibbles

When an RPL program is created, this is the object produced. If you look closely, you'll see that it's similar to a list, containing a prologue and an epilogue. Inside the RPL program, objects are found, one after another, but each object may be an address rather than the object itself. For example, suppose you create this User RPL program:

```
<<  OFF  >>
```

Instead of finding the code that turns the HP off inside this RPL object, there is only an address that points to the code. The OFF code is found at that address. When the HP displays the RPL program to you, it decodes each address into the corresponding name, if it has one.

The User RPL program `<< OFF >>` is coded this way in memory:

```
HP 48              HP 49                 System RPL
D9D20              D9D20                 ::
  E1632              9B983                 x<<
    E13A1              C0593                 xOFF
  93632              4D983                 x>>
B2130              B2130                 ;
```

Here, the nibbles are reversed, as the Saturn always does when dealing with memory. The first and last five nibbles are the RPL object: #02D9Dh is the prologue and #0312Bh is the epilogue.

Inside the object itself there is the `<<` (being #2361Eh on the 48 and #389B9h on the 49) and the `>>` (#23639h on the 48 and #389D4h on the 49), and between those delimiters is an address, #1A31Eh (on the 48) or #3950Ch (on the 49), which is the OFF command.

You can verify it: type #1A31Eh SYSEVAL [ENTER] (on the 48) or #3950Ch SYSEVAL [ENTER] (on the 49), and your HP will turn off. :-)

Every time you put a command inside a RPL program, the HP puts the address of the command.

Here the `<<` and `>>` create a block inside the RPL object. We can even put these "blocks" inside of each other.

These `<<` and `>>` are not really required in RPL programs you write, but the prologue and epilogue are required to have a valid, and thus executable, RPL program. Even if you do `<< << OFF >> >>` the HP codes something different: the `<<` and `>>` are there, but to make sure the `<< OFF >>` inside is not executed, something is inserted just before and just after.

This book is not aimed to be a tutorial on System RPL so this will not be discussed. If you would like to learn more about the secrets of RPL, there are many excellent tutorials. James Donnelly's "An Introduction to HP 48 System RPL and Assembly Language Programming" (ISBN: 1-879828-04-9, published by the Armstrong Publishing Company), Eduardo Kalinowski's "Programming in System RPL", and Hewlett Packard's RPLMAN.DOC are recommended. The first can be purchased by searching for its ISBN, and PDF's of the last two are available for download at the following locations:

http://www.hpcalc.org/details.php?id=5142
→ http://www.hpcalc.org/hp49/docs/programming/progsysrpl_pdf.zip

http://www.hpcalc.org/details.php?id=1745
→ http://www.hpcalc.org/hp48/docs/programming/rpl-pdf.zip

57.5 Algebraic expression

This object type is also called a "symbolic."

Prologue:	02AB8	Epilogue:	0312B
Size:	10 nibbles and up	Type:	9

In memory:

Prologue (02AB8)	5 nibbles
Object 1	Size varies
…	
Object *n*	Size varies
Epilogue (0312B)	5 nibbles

This is the usual algebraic expression used in RPL. Both a prologue and an epilogue are included. The content of the object is encoded using RPL-notation; although on the stack it does not look like RPL, **it is**. As you know, because it is encoded like RPL objects, there is no "+" sign (or whatever the desired command is) inside this object, but each mathematical symbol is coded using its address in ROM using five-nibble addresses.

For example, let's encode `'A+2'`:

The prologue is 021B8, as shown in the table above. Next, we must encode the global name A, which is encoded as 02E48 as prologue, 01 for the size (two nibbles used), and 41 for the ASCII code of the character 'A'. If this is confusing, look at the section on encoding global objects. Next comes the "+" command, encoded using its address in ROM (#2A2DEh on the 48 and #2F961h on the 49), and the number 2, encoded using its ROM address (#1AB67h on the 48 and #39B58h on the 49). Finally, we find the epilogue in the table above, #0312Bh. This means that the complete object, as reversed in ROM, is:

On the 48:
```
#8BA20 84E201014 ED2A2 76BA1 B2130
```

On the 49:
```
#8BA20 84E201014 169F2 85B93 B2130
```

57.6 Tagged object

Prologue:	02AFC	Epilogue:	none
Size:	varies	Type:	12

In memory:

Prologue (02AFC)	5 nibbles
Tag size	2 nibbles
First tag character	2 nibbles
…	
*n*th tag character	2 nibbles

Object	varies

A tagged object is simply an object with a tag to label it. A label of up to 256 characters is created and the object is put just after. The whole thing is kept inside the "tagged" object type.

All we do to create a tagged object is start with the tag prologue, next put the size of the string which tags the object, then put the characters of the tag, and finally encode the object to be tagged as you normally would. The size of the string used to tag is encoded using two nibbles, and thus it is limited to 256 characters.

As an example, let's tag the real number 10 using "Ten". It will be encoded like this:

```
#CFA20 30 4556E6 33920 100 000000000001 0
```

Notice that this begins with the tag prologue, #02AFCh, the number of characters used by the tag (#03h, but reversed of course) follows, and then the ASCII values of the characters forming the tag are encoded (Ten becomes #6E6554h, and is then reversed) of the chars. Finally, the real number 10 is encoded as you learned before.

57.7 Unit object

Prologue:	02ADA	Epilogue:	0312B
Size:	varies	Type:	13

In memory:

Prologue (02ADA)	5 nibbles
Object	varies
Unit	varies
Epilogue (0312B)	5 nibbles

This object is defined with a prologue and epilogue, and between them we find two values. The first one is the object that will receive the unit; it can be a real value, such as the 2 in `'2 m^2'`. The second one is the object used to encode the unit. Symbol like + or * are encoded using their address in ROM, and characters encode the various letter symbols.

For example, let's encode 300 000 000 meters per second. That's slightly over the light of speed, which is about 299 792 458 meters per second according to the HP's constant library.

First, put the prologue, which is #02ADAh. Next, put the object, which is the real number 300 000 000 and is encoded as `#33920 800 000000000003 0`.

Finally the unit is encoded. This is important: we must use RPN notation!! This shouldn't be a surprise to you, though, as you know that the HP 48 and 49 are RPL-based calculators.

Using RPN, we see that the unit "meters per second" is then: `m s /`

We have two characters, m and s. That's two strings, so by applying our knowledge of encoding strings they are:

```
#C2A20 70000 D6
#C2A20 70000 37
```

The division sign (/) comes next, to divide m by s, and then a multiplication sign (*) to multiply the number by the unit.

The epilogue is inserted at the end, so we finally get:

`#ADA20`	(prologue)
`#33920080000000000030`	(the number 300000000)
`#C2A2C70000D6`	(m)
`#C2A2C7000037`	(s)
`#86B01` (48) or `#957D2` (49)	(/)
`#68B01` (48) or `#777D2` (49)	(*)
`#B2130`	(epilogue)

57.8 Directory object

We have two kinds of directories: the HOME directory, and every other directory. HOME is often called the "root directory."

Libraries are "attached" to directories. HOME can have many libraries attached, but each subdirectory can only have *one* library attached to it. Two descriptions follow: one for the HOME directory and the other for subdirectories. This is a very complex object, so pay attention!

57.8.1 HOME directory

Prologue:	02A96	Epilogue:	none
Size:	57 nibbles and up	Type:	15

In memory:

Prologue (02A96)	5 nibbles
Number of attached libraries	3 nibbles
Number of first attached library	3 nibbles
Address of first library hash table	5 nibbles
Address of first library message table	5 nibbles
…	
Number of last attached library	3 nibbles
Address of last library hash table	5 nibbles
Address of last library mess table	5 nibbles
Offset of last object	5 nibbles
00000	5 null nibbles
Number of characters of object name 1	2 nibbles
Character number 1 of object 1	2 nibbles
…	
Character number *n* of object 1	2 nibbles

Number of characters of object name 1	2 nibbles
Object 1	varies
Size of object 1 zone	5 nibbles
…	
Number of chars of last object name	2 nibbles
Character number 1 of last object name	2 nibbles
…	
Character number *n* of last object name	2 nibbles
Number of chars of last object name	2 nibbles
Last object	varies

So, first we have the HOME prologue. It's followed by a three-nibble value, which tells how many libraries are currently attached to HOME (in other words, the libraries you have inside your HP). Even if you have not installed any libraries, the HP 48 and 49 have libraries built into ROM that are attached to HOME! Three very well known libraries are number 2, which contains ASR, RL, RLB, etc., number 171, which contains XVOL, YVOL, VERSION, etc., and number 1792, which contains IF, THEN, ELSE, etc. Although your HP may appear to be empty, these three libraries are always attached.

Next we will find 13 nibbles for each library that is attached. The first three nibbles contain the number of the library, the next five contain the address of the hash table of the library (this will be described later, but it is quite complex), and the last five contain the address of the library message table.

The hash table will be used to quickly access commands and the message table, if it exists, to look for custom error messages inside the lib. It's not the best optimization available, but an interesting one nevertheless.

After that, we have a five-nibble address, which is an offset to the *last* object inside the directory. This is very useful, and very, very clever.

Then, the objects inside the directory are encoded, one after another. For each object, the length of its name is listed first, encoded using 2 nibbles, as the length of a global name is two nibbles, and then each character of the name is given. If you are used to programming in Pascal, this is similar, as it uses the same variable naming convention. The Pascal language first codes the size and then the data; on the other hand, the C language encodes the data with a null value following it and no size field. Here, it's very useful to have the length before.

After the last character of the object's name, we have the object, and after the object, five nibbles give us the *whole* space used by object 1, from the two nibbles that encode its name length to the end of the object.

For each object, it's the same: first shown is the length of the name, then the characters that make up the name, then the object, and finally the space the whole thing uses.

NOTE 1:	If the hash table or the message table of one of the libraries is located in covered memory that cannot be accessed directly, then an extended pointer is found there. The extended pointer object, as well as covered memory, will be explained later. If a library doesn't have a message table, you will find #00000h.

NOTE 2: These numbers are available for library ID numbers:

Number range	Hewlett-Packard's recommended usage:
#000h to #100h	HP libraries in ROM (RESERVED)
#101h to #200h	HP libraries in RAM (RESERVED)
#201h to #300h	Non-HP libraries (RESERVED)
#301h to #6FFh	User libraries (FREE)
#700h to #7FFh	HP's own use (RESERVED)

As you see, of the three libraries attached to the HP, { #700h #002h #0ABh }, the first one is in the area reserved for HP's own use (presumably in application cards) and the two others are in the area reserved for libraries resident in ROM.

57.8.2 Subdirectories

In memory:

Prologue (02A96)	5 nibbles
Number library attached to directory	3 nibbles
Offset of last object	5 nibbles
00000	5 null nibbles
Number of characters of object name 1	2 nibbles
Character number 1 of object 1	2 nibbles
…	
Character number *n* of object 1	2 nibbles
Number of characters of object name 1	2 nibbles
Object 1	varies
Size of object 1 zone	5 nibbles
…	
Number of chars of last object name	2 nibbles
Character number 1 of last object name	2 nibbles
…	
Character number *n* of last object name	2 nibbles
Number of chars of last object name	2 nibbles
Last object	varies
Size of last object zone	5 nibbles

As you can see, a subdirectory can only have *one* library attached to it. If no libraries are attached, then #7FFh will be used as the number of the attached library.

NOTE:	An offset is contained inside the directory object so you can quickly and easily find the last object inside one directory object. The first object has *five* null nibbles before it, and you also will find a *five* nibble value after each object which gives you the whole length used by the object, though the last object does not have that information. We also can find the number of characters of each object's encoded name.

57.9 Library

Caution! This is a very complex object. :)

Prologue:	02B40	Epilogue:	none
Size:	varies	Type:	16

In memory:

Prologue (02B40)	5 nibbles
Size (CRC not included)	5 nibbles
Library name's size	2 nibbles
Library name's first character	5 nibbles
…	
Library name's last character	2 nibbles
Library name's size	2 nibbles
Library number	3 nibbles
Hash table offset	5 nibbles
Message table offset	5 nibbles
Link table offset	5 nibbles
Config object offset	5 nibbles
Message table array	varies
Link table	varies
XLIB 1 (kind)	1 or 3 nibbles
Library number	3 nibbles
Command number	3 nibbles
XLIB 1 objects	varies
XLIB 2 (kind)	1 or 3 nibbles
Library number	3 nibbles
Command number	3 nibbles
XLIB 2 objects	varies
…	
XLIB *n* (kind)	1 or 3 nibbles
Library number	3 nibbles

Command number	3 nibbles
XLIB *n* objects	varies
Masked object 1	varies
…	
Last masked object	varies
Config object	varies
CRC	4 nibbles

Not only is the library structure somewhat complex, but it varies too. Four objects inside the library object have a very specific structure: the hash table, the message table, the link table and the config object.

As you have seen, there are four offsets inside the library with this very purpose.

If the library has no name, its first length field will be zero, and there will **not** be a second length field. If there is a name, we find its length, then the characters, then again the length. A library without a name will not appear when you will do [Right-shift][2].

The hash table is used to increase the access speed of library commands. The message table contains all custom error messages of the library and the link table references all objects of the library. Offsets are used inside the library, and if an offset is #00000h then the object we are looking for does not exist.

We will discuss the four object types below. But first there is something else about the library that's important: there are two types of objects, those that are visible, called XLIB's, and those which are masked. For each *visible* object of the library, you will find:

- its kind (encoded using 1 to 3 nibbles)
- the library number (3 nibbles)
- the command number (3 nibbles)

The CRC is calculated using a hardware CRC circuit inside the HP. The CRC of an object is calculated from the size field (the prologue is not included) to the end of the object. The CRC is *not* a part of the object that is CRC'd. Then, the CRC is added to the library. The calculated size of the library contains neither the prologue nibbles nor the CRC nibbles.

57.9.1 Hash table

The purpose of this table is to speed up access to library commands. It's very simple, so the gain is not monstrous, but it's a good idea. Commands are distributed according to their name's length, from 1 to 16. A command can, however, have more than 16 characters, but then it will be grouped with the 16-character long commands.

Here is the memory scheme of the hash table:

Prologue:	02A4E	Epilogue:	none
Size:	varies	Type:	none

Category 1: commands whose names are 1 character long
…
Category *n*: commands whose name are *n* characters long

In memory:

Prologue (02A4E)	5 nibbles
Size of hash table	5 nibbles

First category offset	5 nibbles
…	
Last category offset	5 nibbles
Size of the names list	2 nibbles
Size of one-character command names	2 nibbles
First one-character command	2 nibbles
Number of command	2 nibbles
Second one-character command	2 nibbles
Number of command	2 nibbles
…	
*n*th one-character command	2 nibbles
Size of 2 character command names	2 nibbles
First two-character command	4 nibbles
Number of command	2 nibbles
…	
Offset to command 1	5 nibbles
…	
Offset to last command	5 nibbles

Here, a command that has no visible name is not listed in the hash table. Each library command has a name and number inside the library. This information is ordered in the hash table. The command number follows the names: just after the first one-character command we find its number, and so on. First, we order by length of name, and after each name we put the command number as inside the library, but only for visible commands. Next, we find the hash table offsets, of which there are two kinds: one to find a command according to its length and another to find the command using the offsets found at the end of the hash table.

57.9.2 Message table

The message table can have two forms: it can either be a single-dimensional array or an *indexed* single-dimensional array.

If we find an array (prologue 029E8) we will find the number of messages inside the library, followed by all messages, each one starting with its length (Pascal string encoding), and then each ASCII byte of each character.

If we find an indexed array (prologue 02A0A) we will find the number of messages and then a list of offsets to each message. Its length (still Pascal string encoding) precedes each message and is followed with one ASCII byte for each character of the message.

57.9.3 Link table

The link table is a simple binary integer. It encodes the addresses of the library's objects inside. You can divide this binary integer into chunks of five nibbles. The first one is the prologue (02A4E) and is followed by the length, which, as usual, does not include the prologue's length. After that, every five nibbles is an address to an object. Here we just have a table of addresses, one for each object inside the library.

57.10 Backup

Prologue:	02B62	Epilogue:	none
Size:	varies	Type:	17

In memory:

Prologue (02B62)	5 nibbles
Size	5 nibbles
Number of characters in object's name	2 nibbles
First character of name	2 nibbles
…	
Last character of name	2 nibbles
Object	varies
System binary prologue (02911)	5 nibbles
0	1 nibble
CRC	4 nibbles

Several objects can be put inside of a backup object. The HP will only put one object and will follow it with a hidden system binary whose first nibble is 0 along with four other nibbles containing the CRC of the object. The CRC is used to ensure that the object has not been corrupted. The first of the five nibbles of the system binary used will always be zero because the CRC is encoded using only four nibbles.

57.11 Long complex

Prologue:	0299D	Epilogue:	none
Size:	47 nibbles	Type:	22

In memory:

Prologue (0299D)	5 nibbles
Real exponent	5 nibbles
Real mantissa	15 nibbles
Real sign	1 nibble
Imaginary exponent	5 nibbles
Imaginary mantissa	15 nibbles
Imaginary sign	1 nibble

The long complex is like two extended reals together, with the first one being the real part and the second one being the imaginary part. This object is used internally by the HP for greater precision.

57.12 Linked array

Prologue:	02A0A	Epilogue:	none
Size:	32+ nibbles	Type:	23

In memory:

Prologue (02A0A)	5 nibbles
Size	5 nibbles
Size of objects	5 nibbles
Number of dimensions	5 nibbles
Dimension 1	5 nibbles
…	
Dimension *n*	5 nibbles
Pointer to first object	5 nibbles
…	
Pointer to last object	5 nibbles
Contents of first object	Size varies
…	
Contents of last object	Size varies

The linked array is a special kind of array. There are pointers inside a linked array, with each one being a shortcut to each object being encoded. If you have a big array that contains a lot of identical values, this kind of array will save a lot of memory space. For example, a pointer is 5 nibbles long, while a real is 16, so if you have a 100x100 array with five different real number values you will save a lot of space by using a linked array.

57.13 Infinite precision real (49 only)

Prologue:	0263A	Epilogue:	none
Size:	17 nibbles and up	Type:	27

In memory:

Prologue (0263A)	5 nibbles
Size of mantissa	5 nibbles
Mantissa digit 1	1 nibble
…	
Mantissa digit *n*	1 nibble
Sign (except for 0)	1 nibble
Size of exponent	5 nibbles
Exponent digit 1	1 nibble
…	
Exponent digit *n*	1 nibble

Sign (except for 0)	1 nibble

An infinite precision real is simply the prologue followed by two infinite precision integers which provide a representation of the real in scientific notation. The first integer contains all the digits in the real number, forming the mantissa. The second integer is the exponent, which determines where the decimal point is placed. The digits in these numbers are reversed, so the least significant digit is first and the most significant digit is last.

This object is not used by any software in the 49's ROM, and it cannot be produced by typing in the command line, but it is displayed by the stack. In order for the infinite precision real to be useful, you must install third-party software, such as the LongFloat library, written by Gjermund Skailand and Thomas Rast.

Because this format allows for any number to be represented in an infinite number of ways (for example, 1E-2 is the same thing as 10E-3), when generating long reals, the LongFloat library determines the number of digits used by the mantissa with the DIGITS user variable, but this does not necessarily have to be true. As an interesting side note, the LongFloat library uses an exponent of -DIGITS (that's the negative of the value in DIGITS) and a mantissa of DIGITS zeroes (with a positive sign!) when the integer 0 is represented.

More details on LongFloat are available in the LongFloat documentation. LongFloat can be downloaded below:

http://www.hpcalc.org/details.php?id=5363
→ http://www.hpcalc.org/hp49/math/numeric/longfl36.zip

Let's create a simple example. We want to represent the infinite precision real number 6432.2. Start with the prologue, 0263A, and reverse its nibbles. We now have:

```
#A3620
```

Now calculate the size of the mantissa. The length of the size is 5 nibbles, the length of the number is 5 nibbles (the decimal point is not counted), and the sign is 1 nibble. This adds up to #11d, or #0000Bh. Now reverse its nibbles and put it after the prologue:

```
#A3620 B0000
```

To represent the mantissa of 6432.2, we must turn it into a 5-digit integer, so it becomes 64322. Don't worry that this is one order of magnitude bigger; we will fix that later. Reverse its digits, and we have 22346. Add the sign nibble, 0, to the end. So far, this gives us:

```
#A3620 B0000 22346 0
```

In order to correct our multiplication by 10^1 when creating the mantissa, we must create an exponent of 10^-1. This means we must store -1 in the exponent. To create -1, once again we first need to calculate its size. The length of the size, as always, is 5 nibbles. The number 1 is one nibble long, and the sign is one nibble long. This adds up to #7d. Convert to hex (which happens to be the same), pad with zeroes and reverse as usual, put it at the end, and we now have:

```
#A3620 B0000 22346 0 70000
```

Now reverse the digits in the number 1, which doesn't change anything, and put the sign (9 for negative) after it. This gives us the following for the entire long real:

```
#A3620 B0000 22346 0 70000 1 9
```

57.14 Infinite precision complex (49 only)

Prologue:	02660	Epilogue:	none
Size:	29 nibbles and up	Type:	27

In memory:

Prologue (02660)	5 nibbles
Size of real mantissa	5 nibbles
Real mantissa digit 1	1 nibble
…	
Real mantissa digit *n*	1 nibble
Real sign (except for 0)	1 nibble
Size of real exponent	5 nibbles
Real exponent digit 1	1 nibble
…	
Real exponent digit *n*	1 nibble
Real sign (except for 0)	1 nibble
Size of imaginary mantissa	5 nibbles
Imaginary mantissa digit 1	1 nibble
…	
Imaginary mantissa digit *n*	1 nibble
Imaginary sign (except for 0)	1 nibble
Size of imaginary exponent	5 nibbles
Imaginary exponent digit 1	1 nibble
…	
Imaginary exponent digit *n*	1 nibble
Imaginary sign (except for 0)	1 nibble

It may look complicated at first, but it's actually very simple. An infinite precision complex number is simply the prologue (02660) followed by two infinite precision real numbers that do not have prologues. The first infinite precision real number is the real component of the complex number, and the second is the imaginary component. The documentation for the infinite precision real number above fully explains this format, but we will do a simple example below.

We want to represent the infinite precision complex number (6432.2, -14.57). Start with the prologue, 02660, and reverse its nibbles. We now have:

```
#06620
```

Now create the infinite precision real 6432.2 to represent the real component, ignoring its prologue. We did this in the last example, so you should have the following:

```
#B00002234607000019
```

Now create the infinite precision real -14.57 to represent the imaginary component, ignoring its prologue. You should have come up with the following:

```
#A0000754197000029
```

Put these three components together, and you will get this:

```
#06620 B00002234607000019 A0000754197000029
```

It's that easy!

57.15 Symbolic matrix (49 only)

Prologue:	02686	Epilogue:	0312B
Size:	15 nibbles and up	Type:	29

In memory:

Prologue (02686)	5 nibbles
Object 1	Size varies
…	
Last object	Size varies
Epilogue (0312B)	5 nibbles

The symbolic matrix object is interesting: it is very similar to a list, but it must have at least one object in it, and it is limited in the types of objects it is able to contain. Other than that (and the prologue), it is identical to a list object.

The types of objects allowed in a symbolic matrix are another symbolic matrix (to make it two dimensions), a real, a complex, an integer, or an algebraic object. You can put other types of objects in a symbolic matrix, but the 49's built-in routines aren't designed to handle them, so it's not recommended.

58 Primitive Code Object

Those don't really exist as objects, as you cannot have them on the stack except through a pointer. The prologue of those PCO's points to its address+5, for example:

```
+----------------------------+
| Prologue (abcde)           |  ------+
+----------------------------+        |
| At abcde: Object's address |  <-----+
+----------------------------+
```

As you can see, a PCO is defined as above: it's an address where five nibbles are found, located at the PCO address plus 5.

There you will find machine language Code, without any prologue or indication of size. This is used a lot by the HP; for example, all stack commands like SWAP, DUP, DROP, etc. are encoded using PCO's.

Someone who codes using System RPL may make calls to these PCO objects in ROM, called "primitives," though that doesn't mean the mathematical primitive function!

You can call a PCO using the SYSEVAL command. Put a binary integer number on the stack and type SYSEVAL [ENTER]. (Warning: don't do this without verifying that the binary integer points to a safe entry point, or the calculator's memory may be cleared!)

Part VI: Writing Programs

59 Doing loops

There are several ways to do loops, serving various purposes. A loop can be used to "wait" or slow down a program, because sometimes it will go too fast, or one can just repeat a set of instructions several times.

When you want to do a loop, something must be used to keep a value. It can be a register like A, B, C, D, or even P.

Here, we're going to use the register C as loop counter:

```
LC FF
*LOOP
C=C-1 B
GONC LOOP
A=DAT1 A
D0=D0+ 5
PC=(A)
@
```

This is a simple program. The number FF is loaded and the B field is used to loop. Each time the program loops, 1 is removed from the counter register, using the B field. The loop continues until the carry is set.

An important thing to remember when using the carry is that the loop is done 1 more time than the counter's value. Why?

FF was loaded into field B. 1 will be removed each loop, but when zero is read, the carry is still not set, so it loops again. The next time, #00h is in field B and removing 1 sets the carry, so the loop stops. Therefore, the loop above is executed #FFh + 1 times, so #100h times.

When the carry is used to do loops in your program, don't forget to remove one unit from the counter value before starting. Because the loop will be done one more time, it will loop the proper amount.

Here is an example of using P as counter. We are going to create a program that does a little "click" on the HP's buzzer.

To turn on the buzzer, we use the OUT register of the Saturn processor. We send #800h to OUT, and the buzzer turns on. If we want to turn it off, we send #000h to the OUT register.

Here is the example program that clicks:

```
LC 800
OUT=C
*LOOP
P=P+1
GONC LOOP
C=0 X
OUT=C
A=DAT0 A
D0=D0+ 5
PC=(A)
@
```

Compile it, store it inside a variable, put the back of your HP close to your ear, and press the key several times to run the program. You will hear a small click. :-)

By varying the time and turning off and on the speaker, you can produce sounds. There is a routine in ROM that produces sound of the frequency (in hertz) and duration (in milliseconds) that you give it.

Note that the above code only works on calculators with a real Saturn processor. The emulated Saturn+ processor in the 48GII and 49G+ cannot emulate the buzzer in this fashion at the time of this writing.

60 Reading from the keyboard

When we want to read a key, we have to consider that keys are internally located along lines. We're going to have current sent over some lines, and we will look at the output to check if a key was pressed. Pressing a key "closes" a circuit, which we can detect.

This introduces the concept of "couples." We are going to first send a specific value to OUT, then read IN, and finally check whether a key was pressed by comparing the IN with what should be found according to the OUT sent.

Below is the keyboard layout of the HP 48. Below each key name are two numbers: one on the left, and one on the right. The one on the left will be put into OUT, and if the key is pressed we will find the corresponding value on the right in the IN register.

A 002/0010	B 100/0010	C 100/0008	D 100/0004	E 100/0002	F 100/0001
MTH 004/0010	PRG 080/0010	CST 080/0008	VAR 080/0004	(UP) 080/0002	NXT 080/0001
' 001/0010	STO 040/0010	EVAL 040/0008	(LEFT) 040/0004	(DOWN) 040/0002	(RIGHT) 040/0001
SIN 008/0010	COS 020/0010	TAN 020/0008	(SQRT) 020/0004	(POWER) 020/0002	(INV) 020/0001
ENTER 010/0010		+/- 010/0008	EEX 010/0004	DEL 010/0002	(BACKSPACE) 010/0001

(ALPHA) 008/0020	7 008/0008	8 008/0004	9 008/0002	(DIVIDE) 008/0001
(LEFT-SHIFT) 004/0020	4 004/0008	5 004/0004	6 004/0002	(MULTIPLY) 004/0001
(RIGHT-SHIFT) 002/0020	1 002/0008	2 002/0004	3 002/0002	(SUBTRACT) 002/0001
ON /8000	0 001/0008	. 001/0004	SPC 001/0002	(ADD) 001/0001

Below is the keyboard layout of the HP 49, labeled in the same form as the 48 layout above:

A 020/0001	B 020/0002	C 020/0004	D 020/0008	E 020/0010	F 020/0020
APPS 020/0080	MODE 010/0080	TOOL 008/0080	(LEFT) 040/0004	(UP) 040/0008	(RIGHT) 040/0001
VAR 004/0080	STO 002/0080	NXT 001/0080		(DOWN) 040/0002	

HIST 010/0040	EVAL or CAT 008/0040	' or EQW 004/0040	SYMB 002/0040	(BACKSPACE) 001/0040
(POWER) 010/0020	(SQRT) 008/0020	SIN 004/0020	COS 002/0020	TAN 001/0020
EEX 010/0010	+/- 008/0010	X 004/0010	(INV) 002/0010	(DIVIDE) 001/0010
(ALPHA) 080/0008	7 008/0008	8 004/0008	9 002/0008	(MULTIPLY) 001/0008

(LEFT-SHIFT) 080/0004	4 008/0004	5 004/0004	6 002/0004	(SUBTRACT) 001/0004
(RIGHT-SHIFT) 080/0002	1 008/0002	2 004/0002	3 002/0002	(ADD) 001/0002
ON /8000	0 008/0001	. 004/0001	SPC 002/0001	ENTER 001/0001

60.1 Checking for a single key-press

Make a program loop until the [SPC] key is pressed:

```
INTOFF         % turn off keyboard interrupts
*LOOP          % LOOP starts here
LC 001         % we load #001h into C(A) on the 48 (change this to LC 002 for the 49)
OUT=C          % send it so SPC line receives current
GOSBVL =CINRTN % =CINRTN is 01160 on the HP 48 and 00212 on the HP 49
?CBIT=0 1      % does C contains #0002h ? (change this to ?CBIT=0 0 for the 49)
GOYES LOOP     % no? jump to LOOP
INTON          % allow keyboard interrupts
A=DAT0 A       % return to RPL
D0=D0+ 5
PC=(A)
@
```

A few notes here. First, INTOFF and INTON have been used. This is to tell the HP we're going to handle the keyboard in our code. It is important to use INTON at the end of the source. Then, if you look at the couple of values for the [SPC] key you will find that it's 001/0002 on the 48 and 002/0001 on the 49.

One could load 0002 (on the 48) or 0001 (on the 49) into A, and then compare register C with register A. But, if one only considers the first nibble, #0002h in hex is the same as #0010b in binary, or #0001h in hex is the same as #0001b in binary. This means all one has to do is check whether bit 1 is equal to zero on the 48 or whether bit 0 is equal to zero on the 49. If so, the key has not been pressed, and it will loop again.

Remember that the first bit is number zero and the second bit is number one.

There are others ways to check for a key. Next we'll use one that uses a logical operator, and the concept of a "mask."

60.2 Checking for key combinations

Masks are a useful concept when it comes to detecting multiple key-presses. Let's write a program that will quit if [1] and [9] are pressed at the same time.

The output for 1 and 9 keys is #002h (#0010b) and #008h (#1000b) on the 48, and, in an interesting coincidence, vice versa on the 49. Performing an OR of the two values gives #Ah (#1010b).

You can make this process simpler by using your calculator! Simply put #2h and #8h on the stack and press [MTH][F][NXT][A][B] (or simply type OR). This performs an OR on the first two levels of stack

NOTE:	You must be in HEX mode to get the value you will put inside your program. HEX mode is [MTH][F][A].

The IN values will be found using OR. The couples are #8h and #2h, so #8h OR #2h = #Ah

Source is:

```
INTOFF
*LOOP
LC 00A
OUT=C
GOSBVL =CINRTN % =CINRTN is 01160 on the HP 48 and 00212 on the HP 49
LA 000A
?C#A A
GOYES LOOP
INTON
A=DAT0 A
D0=D0+ 5
PC=(A)
@
```

60.3 Reading the entire keyboard

And now, what if we want to check if any key has been pressed? We can simply do an OR of all output masks, and then check whether the value read from the keyboard is different than zero.

But, the [ON] key is special: it's directly wired to a pin on the Saturn processor. To test whether [ON] has been pressed, we must do a `C=IN` and then check whether bit 15 is null. If it is null, [ON] was pressed.

The OR mask of all output masks is #1h OR #2h OR #4h OR #8h OR #10h OR #20h OR #40h OR #80h OR #100h. That returns #1FFh.

The source to test whether any key has been pressed follows:

```
INTOFF
*LOOP
LC 1FF
OUT=C
GOSBVL =CINRTN % =CINRTN is 01160 on the HP 48 and 00212 on the HP 49
?C=0 A
GOYES LOOP
INTON
A=DAT0 A
D0=D0+ 5
PC=(A)
@
```

There is one problem with this: as soon as we launch it, it quits. We have to flush the keyboard buffer before we can start the loop. This piece of code will flush the keyboard buffer:

```
D1=(5) "adrKEYBUFFER"+2   % should be written D1= 047DF if no entries table is available
A=DAT1 A
D1=A
C=0 A
DAT1=C A
RTNCC
```

If we use D1, we must save its value before, of course. :)

We get:

```
INTOFF                        % we turn off keyboard interrupts
*LOOP
```

```
LC 1FF                 % mask for all keys
OUT=C
GOSBVL =CINRTN         % =CINRTN is 01160 on the HP 48 and 00212 on the HP 49
?C=0 A                 % no key pressed?
GOYES LOOP             % we loop
INTON
LC(6) "adrKEYBUFFER"+2 % points to adr to keybuffer pointer, use LC 0047DF if no entries table is available
CD1EX                  % old D1 is saved in C
A=DAT1 A               % pointer to keybuffer in D1
D1=A
A=0 A
DAT1=A A               % we empty it
D1=C                   % restore D1
A=DAT0 A               % and we quit
D0=D0+ 5
PC=(A)
@
```

It goes so fast that you will need something like the following RPL program, where WAITK is the variable the code will be inside:

```
<< 0 WAIT DROP WAITK >>
```

Put this program on the stack, press [EVAL] quickly, the busy indicator should turn off, and the RPL program will remain on stack. Press any key and it will quit.

Code executes very quickly, as you can see.

60.4 Reading the [ON] key

The [ON] key is special: because it is directly wired to a pin of the processor; if we want to test it, `INTOFF` is not enough. We have to disable all interrupts, so need to set flag 15 of ST to zero. As we'll see later, in the disassembled code of the interrupt handler, if this bit is set to zero, no interrupt is done. (In fact it's a bit more complicated, but let's keep that for later. Interrupts are complex, so we'll leave that for the last part.)

Code to check for [ON]:

```
ST=0 15          % disable all interrupts
*LOOP
GOSBVL =CINRTN   % =CINRTN is 01160 on the HP 48 and 00212 on the HP 49
?CBIT=0 15       % [ON] pressed?
GOYES LOOP       % if not, we loop
ST=1 15          % we allow all interrupts
GOSUB REINT      % enable interrupts
A=DAT0 A         % and quit
D0=D0+ 5
PC=(A)
*REINT
RTI              % exit interrupt service routine
@
```

CAUTION! Don't make mistakes with `ST=0 15`, because once it's performed, you can not use [ON]-[C] or [ON]-[A]-[F]. If your program traps itself inside an infinite loop, you will have to do a hard reset using the hole below your HP, and you may lose parts of memory.

We have to use `ST=0 15` and `ST=1 15` because otherwise we cannot check for [ON] being pressed.

There is another way to check for multiple key presses: check each key, one after another. Machine language goes so fast that it will detect multiple-key presses: the program below loops until [A] and [F] are pressed together.

First, we wait for [A], and as soon it's detected, we check for [F]. This program goes so fast that it will detect both keys being pressed:

```
INTOFF                  % turn off interrupts
*LOOP
LC 002                  % out for [A]
OUT=C
GOSBVL =CINRTN          % =CINRTN is 01160 on the HP 48 and 00212 on the HP 49
LA 0010                 % in for [A]
?C#A A                  % if [A] not pressed
GOYES LOOP              % we loop
LC 100                  % out for [F]
OUT=C
GOSBVL =CINRTN          % =CINRTN is 01160 on the HP 48 and 00212 on the HP 49
?CBIT=0 0               % bit 0 = 0 ?
GOYES LOOP              % then wait for [A], [F] being not pressed
INTON                   % restore interrupts
LC(6) "adrKEYBUFFER"+2  % points to adr to keybuffer pointer, use LC 0047DF if no entries table is available
CD1EX                   % old D1 is saved in C
A=DAT1 A                % pointer to keybuffer in D1
D1=A
A=0 A
DAT1=A A                % we empty it
D1=C                    % restore D1
A=DAT0 A                % and we quit
D0=D0+ 5
PC=(A)
@
```

61 Manipulating the stack

Previously you learned that the stack ends with four null nibbles, #00000h. Though this is partially true, the *full* truth is a bit more complicated. :-)

Internally, the HP 48 calculates the stack's depth with the following code:

```
LC 806FD        % 806FD on 48G series and 49, and 7057E on S/SX
CD1EX
A=DAT1 A
C=A-C A
GOSUBL =DIV5    % =DIV5 is 06A8E, a routine that divides C(A) by 5
C=C-1 A         % we remove 1 and we have stack's depth
```

Here, D1 points to the first level of stack. The `CD1EX` moves the address in D1 to C and makes D1 point to #806FDh. At that address, using five nibbles, the address of the command line's start is stored. Next, that address is loaded inside A, and D1 is restored. Then, we calculate:

stack size * 5 = (D1 address) - (Command line address)

So we need to divide the result, C(A), by five. There is a routine in ROM to do this, located at address #06A8Eh, and it uses registers A, C, and D.

Once divided by five, remove one, and we have the stack's depth. :-)

Why do we do all that? Is it not enough just to check for the #00000h on stack? Even though it's true that the end of stack contains #00000h, there can also be the address #00000h on stack, so the stack could look like this:

```
4:
3:              4
2:     <External>   ← this is #00000h !!
1:              3
```

So we can have two (or more) #00000h on stack. This is why, if we really want to know what is the stack's depth, we have to calculate it, and not just search for the first #00000h on stack.

If you'd like, you can make your HP show this stack display. Put three numbers on the stack, and then compile this code:

```
D1=D1+ 5          % we move to level 2
C=0 A
DAT1=C A          % and set it to #00000h
D1=D1-5           % we come back to level 1
GOVLNG =LOOP      % =LOOP is 2D564 on the 48 and 05149 on the 49, which returns to RPL
```

Store this Code inside a variable, and run it over the 3 level stack you have. What do you see?

You have on level two an <External>, which is in fact #00000h.

Press [DEL] to clear the whole stack. Don't [EVAL] the <External>, as that would make your HP jump to #00000h, causing a warmstart. Your HP will only restart, which is nothing dangerous, but don't play with fire. :)

Here is the code that calculate the real depth of the stack:

```
CD1EX             % we push D1 into C
D1= 806FD         % and make D1 point to #806FDh
A=DAT1 A          % we read command's line start
C=A-C A           % and calculate D1 address minus command line address
GOSUBL =DIV5      % =DIV5 is 06A8E, a routine that divides C(A) by 5
C=C-1 A           % and remove one
```

C contains the stack's real depth, and this routine uses registers A, B, C, and D1.

61.1 Drop

Coding a DROP is very easy. Here, we don't need to know the stack's real depth. We're just going to read the first's level value. If it's zero, then the stack "should" be empty. If it's not equal to zero, we drop one level.

You know that D1 points to the stack's address. Each time we increment D1 we reduce the stack's size, and each time we decrement it we make the stack bigger. The stack only contains addresses, each one being 5 nibbles wide, so if we read from the stack, we'll use field A.

If we increment D1 by 5, we do a drop. The code below does it:

```
C=DAT1 A
?C=0 A
GOYES QUIT
D1=D1+ 5
```

```
D=D+1 A
*QUIT
GOVLNG =LOOP     % =LOOP is 2D564 on the 48 and 05149 on the 49, which returns to RPL
@
```

Here, we start by reading the contents of the first level. If it's null, we quit. At 2D564 (on the 48) or 05149 (on the 49) is the following code:

```
A=DAT0 A
D0=D0+ 5
PC=(A)
```

The `GOSBVL =LOOP` uses 7 nibbles in memory, and the usual instructions would need 11 nibbles. Here we save 4 nibbles by using this jump rather than using three instructions.

61.2 Swap

Since the stack only contains addresses, doing a swap is simply exchanging the addresses of level one and level two. First we must ensure that the stack is two levels deep, and if it is, we will exchange the two addresses.

We could do:

```
C=DAT1 A         % read first level's address
?C=0 A           % if it's null, we quit
GOYES QUIT
D1=D1+ 5         % we point to level 2
A=DAT1 A         % and read it into A
D1=D1- 5         % we come back to level 1
?A=0 A           % if A empty, no level 2
GOYES QUIT       % so we quit
DAT1=A A         % otherwise, we write A to DAT1 which moves level 2's address to level 1
D1=D1+ 5         % we point to level 2
DAT1=C A         % and write C there, which moves level 1's address to level 2
D1=D1- 5         % we make D1 come back to level 1
*QUIT
GOVLNG =LOOP     % =LOOP is 2D564 on the 48 and 05149 on the 49, which returns to RPL
@
```

Try it. If there is an empty stack or only one object it does nothing, but if there are two objects they are swapped! :-)

How does the HP perform a SWAP?

When we call SWAP, an RPL program is called. It contains two calls: the first one checks whether two object's addresses are on stack, and if not, an error is done. Otherwise, the SWAP code is called, and it does a SWAP without checking, because the other piece of code already checked.

It's better because this way, the HP is able to handle a #00000h on the stack, swapping it with another object. Our code will recognize any #00000h as the end of stack, so it will not do anything if the stack contains a #00000h.

Of course, the "usual" HP user should **never** have an <External> on the stack. If they do have one, chances are it's not the #00000h one, whose evaluation does a warmstart. But, Externals are extensively used by System RPL, the basis of User RPL, and it must be able to manage SWAP or any other stack function with Externals.

62 Example program: Digit Sum

Let's take what you've learned so far and put it all together to write a simple program.

We're going to write a program that takes a real number from level 1 of the stack, calculates the sum of all its digits, and returns the result as a real to level 1. We will also do simple argument checking to make sure there is an real number on level 1 by checking the prologue of the object with the standard real prologue, and we'll give an error if it doesn't match.

The program listing, written by Arnaud Amiel (thanks, Arnaud!) is below:

```
GOSBVL =SAVPTR        % =SAVPTR is 0679B, which saves registers D0, D1, B, and D
A=DAT1 A              % A holds pointer to level 1
AD1EX                 % D1 points to level 1
A=DAT1 A              % Load the prologue in A(A)
LC(5) DOREAL          % DOREAL is 02933, the real number prologue, loaded into C(A)
?C#A A
GOYES error           % If they are different, error
D1=D1+ 5              % Point to number
A=DAT1 W              % Load it in A
SETDEC                % Real numbers are stored in BCD format
P=4                   % Field M holds 12 digits: Counter P=16-12
C=0 W                 % Reset C to store our result

*NextDigit
ASR W                 % Move last digit to field X
ASR X                 % Move digit to LSB in field X
ASR X
C=C+A X               % Add it up to our counter
P=P+1                 % We counted one more digit...
GONC NextDigit        % If Carry we did all 12 digits and P=0

A=0 X                 % Reset A to store exponent
?C=0.X
GOYES finished        % If all digits are out of X, we're done
CSRC                  % Move digit out of X
?C=0 X
GOYES finished        % If all digits are out of X, we're done
CSRC                  % Move digit out of X
A=A+1 X               % Increment exponent (2 digit result: X=1)
?C=0 X
GOYES finished        % If all digits are out of X, we're done
CSRC                  % Move digit out of X
A=A+1 X               % Increment exponent (3 digit result: X=2)

*finished
CSRC                  % Move last digit from S to M
C=C+A X
DAT1=C W              % Overwrite real at level 1
SETHEX                % Go back to hexadecimal mode
GOVLNG =GETPTRLOOP    % =GETPTRLOOP is 05143, which loads the saved registers and exits

*error
LC(5) argtypeerr      % argtypeerr is 00202, the "Bad Argument Type" error message number
GOVLNG =GPErrjmpC     % =GPErrjmpC is 266DB on the 49 and 10F40 on the 48, which exits with error
@
```

This program replaces the number in stack level 1 with the sum of its digits by directly replacing the nibbles in memory at the address pointed to by stack level 1. This has some interesting side effects. The HP stores the real integers from -9 to 9 in ROM, so you cannot write to them with the `DAT1=C W` instruction. When you run this program with a negative real integer from -1 to -9, you will notice that it will return the same negative number back, having no effect on the memory. When you give it any other negative real number, it will return the sum of its digits as a positive number. In these cases, the number is stored in RAM rather than ROM so the `DAT1=C W` instruction works as expected.

There are ways to fix this. You can push the answer onto level 1 of the stack, pushing up the input so it will be on level 2, and exit the program by using `GOSBVL =PUSH%LOOP`, where PUSH%LOOP is 2A23D on the 48 and 2F899 on the 49, after putting the real number to push into A(W). To remove the input from the stack when you read it you could call `GOSBVL =POP1%`, where POP1% is 29FDA on the 48 and 2F636 on the 49, to pop the real number off the stack and put it into A(W).

The best solution would be to allocate memory for the new real number, and then point stack level 1 there instead. All of these solutions, however, make the code more complex, so they are not being shown in this simple example.

63 Example program: Perfect Squares

This program, also written by Arnaud Amiel, calculates all 10-digit perfect squares that use no more than three different numbers. There are 48 of them (47 use three different numbers and one uses only two different numbers), and this program will push all 48 of them onto the stack in about 5 seconds on a 49G+.

The algorithm is simple to understand once you know that if the list of squares is U(n), then:

U(n+1)-U(n)=2+(U(n)-U(n-1))

This has been written with full MASD syntax, complete with macros like `SAVE` and the higher-language-like constructs with the curly braces (`{` and `}`), so it will need some minor changes to assemble it with HP-ASM.

```
SAVE SETDEC                       % Save the registers, and we are using decimal numbers
LA 1111088889000                  % Load the largest invalid 10-digit square in A(M) so append 000
C=0.M LC 66665000 B=C.M           % Load U(n)-U(n-1) in B, and same thing with 000
LC 33333 RSTK=C                   % We have to test SQRT(9999999999)-SQRT(1111111111)=66666 squares
                                  % We will need all our registers so we keep the counter on RSTK
                                  % This is safe as we don't use any GOSUB
*NextNum
C=RSTK C+1.A                      % We are going to test the next square so we get the counter and increment it
SKIPNC { P=0 SETHEX LOADRPL }     % If we have counted all the squares we exit.
RSTK=C                            % We keep our counter on RSTK
B+1.M B+1.M A+B.M C=A.M           % This is where the square is calculated
                                  % We use B+1 twice because B+2 does not take into account DEC mode
                                  % We keep the number tested in A we test C
P=5 A=0.XS B=0.XS D=0.XS          % We test 10 digits so P=15-10
                                  % We test the digits in XS.  A(XS), B(XS) and D(XS) are our
                                  % three 1 digit stores so we clear them
*TestNum
P+1 GOC GoodNum                   % We went through the all 10 digits without error, so it is a good number
CSR.W                             % We shift the last digit of M to XS to be tested
?C=0.XS GOYES NextNum             % If the digit is 0 it is not a solution, we go to calculate the next square
?A#0.XS { A=C.XS GOTO TestNum }   % Do we have something in store 1?  If not
                                  % we store this digit and go to test the next digit
?A=C.XS GOYES TestNum             % If this digit is already in store 1 we go to test the next digit
?B#0.XS { B=C.XS GOTO TestNum }   % Same as before for store 2
?B=C.XS GOYES TestNum
?D#0.XS { D=C.XS GOTO TestNum }   % Same as before for store 3
```

```
?D=C.XS GOYES TestNum
GOTO NextNum                    % If we got here we have more than 3 different digits

*GoodNum
R1=A.M                          % Using PUSH% will change all registers so we backup
C=B.M R2=C.M                    % Backup U(n)-U(n-1)
ASL.M ASL.M                     % Our 10 digits were aligned to the right but they must be aligned to the left
LA 009                          % This is the exponent for a 10-digit number
A=0.S                           % Make sure the sign is right
GOSBVL =PUSH%                   % Push A as a real
SAVE                            % Save the pointers
SETDEC                          % Go back to Decimal mode as PUSH% went to HEX
A=R1.M                          % Restore our backups
C=R2.M B=C.M                    % Restore
GOTO NextNum                    % Go to check the next number
@
```

64 Creating a program for creating macros

You know that you can put parts of your programs into several files, and use the ' symbol to create links. But, as you learned in the HP-ASM documentation, you can also create macros. Remember that a macro is something precompiled that can be included inside any Code object.

For example, suppose we want to write a program that displays “YES” and “NO” choices on the menu bar, and returns <1h> if YES is pressed or <0h> if NO is pressed. We could create a GROB, with “YES” and “NO” words inside, display it on the menu area from our program, and wait for a key to be pressed!

So, if we want to put a GROB inside of our code, we don’t really need to put the whole object inside. We can just put the data we need. ASM Flash and HP-ASM are able to include a precompiled object anywhere in one of our programs. So we are going to write a →MACRO program that does two things:

- If a Code object is given to it, it removes the prologue and size fields and puts a string with the code inside on the stack
- If a GROB object is given to it, remove the prologue, the size field and the 10 nibbles that follow, which are used to code the number of lines and columns.

For example, instead of writing this at the end of a program all the time:

```
A=DAT0 A
D0=D0+ 5
PC=(A)
```

we could compile it and use →MACRO on it. Then, we call it 'RPL' and store it in the Macro directory of HOME. Each time we want to use it in a program, type the following and ASM Flash/HP ASM will put the contents of RPL there:

```
...
/RPL
...
```

If you don’t have an entry table, this is VERY cool to avoid remembering the addresses of routines in ROM: instead of remembering #2D564h, which contains the RPL exit on the 48 (#05149h on the 49), we simply code this and use →MACRO on it:

```
GOVLNG 2D564
@
```

Store this as 'RPL' and /RPL can be used in source code. This can be done with any piece of code. :)

The commented source is in the next section. It has been tested, so it should work.

Some assemblers, however, have names built-in for some addresses and functions. For example, the Meta Kernel has a command built-in for exiting to RPL.

64.1 →MACRO.S

```
!0-15
!PC

C=DAT1 A            % read first level's content
?C#0 A              % if it's not null, we start
GOYES BEGIN
LA 00201            % otherwise do an error; A(A) contains error code
GOVLNG =Errjmp      % =Errjmp is 05023, which does the error

*BEGIN              % here we begin
R1=C A              % save first level's address into R1(A)
GOSBVL =SAVPTR      % =SAVPTR is 0679B, which saves registers D0, D1, B, and D
C=R1 A              % put level one address into C
B=C A               % and save it into B (see why below)
D0=C                % D0 points to object's first nibble
GOSBVL =SKIPOB      % =SKIPOB is 03019, which calls SKIP so D0
                    % points to the first nibble after the object
CD0EX               % C contains that address
C=C-B A             % calculate (end address - start address)
                    % so C = length of object
R2=C A              % R2a contains object's length (in nibbles)
C=R1 A              % get back object's address saved in R1(A)
D0=C                % and make D0 point to it
A=DAT0 A            % read its prologue
LC 02DCC            % 02DCC = Code prologue
?A#C A              % if there isn't a code object,
GOYES GROB?         % check if there's a GROB
% Code
C=R1 A              % C = address of object
C=C+10 A            % skip the code prologue and size (10 nibbles)
R1=C A              % and save it back to R1(A)
C=R2 A              % here C = length to allocate for string
C=C-10 A            % remove 10 (since we have 10 fewer nibbles to copy)
R2=C A              % and save it back to R2(A)

*ALLOCATE
GOSBVL =MAKE$N      % =MAKE$N is 05B7D, which reserves a string Ca long
CD0EX               % C = address of first nibble of the newly reserved
                    % string
D1=C                % D1 point to it (because the COPY routine I'm going
                    % to use copies from D0 to D1..)
C=R1 A              % C is the "corrected" object's address
                    % corrected = prologue and non-used fields skipped
D0=C                % make D0 point to it
C=R2 A              % C = number of nibbles to copy (R2 = the length
                    % calculated using SKIPOB)
```

```
GOSBVL =MOVEDOWN     % =MOVEDOWN is 0670C, which is the
                     % COPY DOWN routine (explained below, with others)
GOSBVL =GETPTR       % =GETPTR is 067D2, which restores registers D0, D1, B, and D
C=R0 A               % R0 contains reserved string's address
D=D-1 A              % remove 5 nibbles of the free RAM
GOC EXIT             % if CARRY = not enough memory to push object
                     % on the stack
D1=D1- 5             % add one object to the stack: D1-5
DAT1=C A             % write object's address (that is: pushes object
                     % on stack)
*EXIT                % here we find the usual return to
GOVLNG =LOOP         % =LOOP is 2D564 on the 48 and 05149 on the 49, which returns to RPL

*GROB?               % check if there's a GROB as the object
LC 02B1E             % 02B1E = GROB prologue
?A=C A               % if a GROB, update R1(A) and R2(A)
GOYES OK
GOVLNG =GETPTRLOOP   % =GETPTRLOOP is 05143, which recovers the saved
                     % registers and quits (using the usual RPL return)
*OK                  % here there's a GROB, so:
C=R1 A               % get the start of GROB's object
C=C+16 A             % we add 20 to it, skipping the prologue, the size,
C=C+4 A              % and the ten nibbles of number of lines and columns
R1=C A               % wrote back to R1a
C=R2 A               % C = calculated length
C=C-16 A             % remove 20 from it
C=C-4 A
R2=C A               % and wrote back to R2a
GOTO ALLOCATE        % we reserve, copy and quit

@
```

In this program, R1(A) and R2(A) are used. R0 isn't used because the routine used to reserve a string sends the address of the reserved string back into R0.

First, the first level of the stack is read. If it's empty, #00201h is loaded into A(A). The routine at #05023h does an error, using the error code inside of A(A) (all routines used are described just below the explanations about that code).

If it's not empty, the object's length is calculated. I have used R1(A) to keep the first object's address, because I'll need it later when I do the copy from the object to my reserved string. I also use the SKIPOB routine located at #03019h, which will be described below.

Once done, D0 points to the first nibble just after the object. An exchange instruction is then used, and then the length of the object is calculated using this formula: length = (end address - start address). R2(A) is used to keep the object length, which is needed to copy the object later.

Then, I get the object's pointer, and check if its prologue is a Code prologue. If so, I update R1(A) and R2(A), by adding the number of nibbles skipped to R1(A) (10 for Code, 10 for GROB) and then removing that number from R2(A), the number of nibbles to copy. When you want to modify →MACRO so it can handle the object kind you want, just add tests and update R1(A) and R2(A) the same way.

Once it's done, we recover the string's length and reserve it using a ROM subprogram found at #05B7Dh, which allocates C(A) nibbles in a string object. When it comes back, R0(A) contains the address of the string, which will be put on the stack later, and D0 points inside the string.

Then, all I do is use the COPY DOWN copy routine in ROM (see below for all routines used) but it copies from D0 to D1. When I return from reserving the string, D0 points to the string. I make D0 point to the object and D1 to the reserved string, and then put the number of nibbles I want to copy into C(A).

Once the copy is done, the object is put on the stack. A GOC is used so if there are not five nibbles available, the object isn't pushed and it quits. The next garbage collection would then remove the string that was reserved.

64.2 ROM routines used in this program

64.2.1 #5023h, also called Errjmp or ERROR_A

Put the usual error code into A(A), and call this routine using a GOVLNG. You **must not** have registers saved when you call it, so if registers need to be recovered, do it first and then call this routine.

64.2.2 #679Bh, also called SAVPTR or SAVE_REG

Let's disassemble it and see how it works. Here is the commented source of SAVE_REG:

```
CD0EX         % move D0 address to C
DO= 8072F     % and make D0 point to #8072Fh (see below)
DAT0=C A      % write D0 to it (we save D0)
D0= 806F8     % make D0 point to #806F8
CD1EX         % move D1 address to C
D1=C          % and write it there
DAT0=C A      % so we save D1
D0= 806F3     % make D0 point to #806F3h
C=B A         % move B(A) to C(A)
DAT0=C A      % and save B(A) to #806F3h
D0= 807ED     % make D0 point to #807EDh
C=D A         % move D(A) to C(A)
DAT0=C A      % and save D(A) to #807EDh
RTNCC         % return AND clear the CARRY flag
```

Notice that the register D0 is lost when we return, but D1, B, and D still have their values. You can see that the HP saves D0, D1, B and D to some addresses in memory: those are reserved just for that. Later in this document, I will describe an area of memory called the "System area." There, we have space reserved so we can save the registers. When we recover the registers (see below) the HP reads where we saved. This is why we can only save registers one time; old values are lost for new ones.

64.2.3 #3019h, also called SKIPOB (Skip OBject)

This one is a little more complicated.

Before calling this routine, make D0 point to its first nibble. When it returns, D0 points to the first nibble after the object (skipping the object in D0), the carry is cleared, ST1 is cleared, and P is set to 0. This routine also uses D0, A, and C, so if one of those registers contains something you really need, save it somewhere! :)

64.2.4 #05B7Dh, also called MAKE$N or RES_STR (REServe STRing)

This ROM routine uses D0, D1, A, B, C, D, R0.

Before calling it, put the number of chars you want reserved into C(A); if there is not memory, an "Insufficient Memory" error is done (registers should be saved before calling this routine, as any error recovers them).

If there is enough space in the temporary memory area, the subroutine returns, and D0 points *inside* the string on its first nibble, and its address is stored in R0a. When you quit the program, you will put that address onto the stack to put the string on it.

64.2.5 #0670Ch, also known as MOVEDOWN or COPY DOWN

I have used a call to this subroutine because memory-handling routines are very well written in the ROM. There are two routines to copy data in memory: COPY DOWN and COPY UP.

What's the difference? When you use COPY DOWN, C(A) nibbles are copied from D0 to D1, moving D0 and D1 with "+*n*" (*n* a value the routine sets), and COPY UP copies from D0 to D1, but from the *end* of the zone being copied (D0 and D1 are decremented during the copy).

I use COPY DOWN since I copy from an object to another, from the beginning of each to the end. If I copied backwards in memory, I would have used COPY UP.

When you call COPY DOWN or COPY UP, D0 must point to where data will be read, and D1 to where it will be written. C(A) is used to say how many nibbles we want to have copied.

Those two routines (COPY DOWN at #0670Ch and COPY UP at #066B9h) use D0, D1, A and C.

64.2.6 #067D2h, also called GETPTR or GET_REG

This routine restores the previous saved registers; in ROM, we find this:

```
D0= 807ED    % where D is saved
C=DAT0 A
D=C A
DO= 806F3    % where B is saved
C=DAT0 A
B=C A
DO= 806F8    % where D1 is saved
C=DAT0 A
D1=C
D0= 8072F    % where D0 is saved
C=DAT0 A
D0=C
RTNCC
```

Of course, we can call an address *after* 067D2. What does this mean? Easy :-)

If we want to recover D0 or B and D0, we will call the routine, but not at its beginning! This also works for all routines: you can choose from *where* you start it, as long as it finishes with a `RTN`.

Here is the same code as before, but with addresses:

```
#067D2h      D0= 807ED    % where D is saved (80E9B on the 49)
#067D9h      C=DAT0 A
#067DCh      D=C A
#067DEh      DO= 806F3    % where B is saved
#067E5h      C=DAT0 A
#067E8       B=C A
```

```
#067EAh     DO= 806F8    % where D1 is saved
#067F1h     C=DAT0 A
#067F4h     D1=C
#067F7h     D0= 8072F    % where D0 is saved (8076B on the 49)
#067FEh     C=DAT0 A
#06801h     D0=C
#06804H     RTNCC
```

If you just want to recover D0, simply call `GOSBVL 067F7`.

65 More built-in ROM routines

There are many more routines available to the programmer. These are, however, sparsely documented. Before calling them it is recommended that you save the working registers, as most of the routines use them, though they do not touch the scratch registers. The best sources of information have been compiled by Mika Heiskanen and Carsten Dominik. Mika's list can be downloaded at the following location:

http://www.hpcalc.org/details.php?id=1783
→ http://www.hpcalc.org/hp48/programming/entries/ent_srta.zip

Carsten's lists are available at the following addresses:

http://staff.science.uva.nl/~dominik/hpcalc/entries

http://www.hpcalc.org/details.php?id=5146
→ http://www.hpcalc.org/hp49/programming/entries/edb.zip

http://www.hpcalc.org/details.php?id=5475
→ http://www.hpcalc.org/hp48/programming/entries/entriesdb48pdf.zip

http://www.hpcalc.org/details.php?id=5476
→ http://www.hpcalc.org/hp49/programming/entries/entriesdb49pdf.zip

Joe Horn produced a cross reference comparing the entries in the 48 with those in the 49, and this can be very useful for those writing code for both calculators. This is available for download at the following location:

http://www.hpcalc.org/details.php?id=3248
→ http://www.hpcalc.org/hp49/programming/entries/xref4849.zip

Note that most of the entries in these documents are dedicated to System RPL, but quite a few assembly routines are also given.

As you will see by looking at the above list, there is a the large number of entry points, even when counting only those available within assembly language code. Because each entry point name maps to its own memory address, it is highly recommended that you have an entry point table installed on your calculator if you are going to be doing much development away from a PC. Of course, an alternative is to print a list of the entry points, and memorize the most common ones.

66 HP 48 memory management

A big thanks goes to Christoph Giesselink for writing this section!

The Saturn chip has a 4-bit data bus and a 20-bit address bus. This means that the addressable area is 2^{20} = 1 048 576 nibbles, numbered from #00000h to #FFFFFh using the hexadecimal base. As we learned earlier, there are two nibbles in a byte, so we divide 1 048 576 nibbles by 2 and get 524 288 bytes, or 512KB.

66.1 Daisy-chain basics

The third and fourth generation Saturn chips—Clarke in the S series and Yorke in the G series—have six built-in memory controllers. They allocate:

Device	Function on S/SX	Function on G/G+/GX
NCE1 (ROM)	ROM (256KB)	ROM (512KB)
HDW	I/O registers (32B)	I/O registers (32B)
NCE2 (RAM)	RAM (32KB)	RAM (32KB,128KB)
CE1	Port1 (32KB,128KB)	Bank switcher
CE2	Port2 (32KB,128KB)	Port1 (32KB,128KB)
NCE3	unused	Port2 (up to 4MB)

The memory controllers are linked together by a technique called a *daisy-chain configuration*. When the Saturn sends a CONFIG command to the bus, the first controller in the chain receives this information. When a controller is fully configured it passes the command through to the next controller in the chain. By this way all controllers can be configured.

The daisy chain is:

```
    SATURN  →  HDW  →  NCE2  →  CE1  →  CE2  →  NCE3  →  NCE1
```

The memory controllers can configure the size and address location of most of the allocated modules. If two devices are configured with overlapping address ranges, the device with the higher access priority is selected and the device with the lower priority is covered. That's the reason for the term "covered". The priority-controlled access to the devices is different from the daisy chain order. This is done by a special arbitration. If there's no device configured at the requested address the NCE1 (ROM) controller will answer.

The priority of the devices is:

1. HDW (highest priority)
2. NCE2
3. CE2
4. CE1
5. NCE3
6. NCE1 (lowest priority)

If we want to skip a device that covers a device with lower priority, we have to "unconfig" (unconfigure) the higher-priority device with the `UNCNFG` command. Then we have access to the wanted device. After we have finished using the desired device we should reconfigure the device again.

66.2 Configuration of devices

After a CPU reset or after the `RESET` command, all devices, except the ROM, are unconfigured.

The order of the device configuration is:

1. HDW address
2. NCE2 size + address
3. CE1 size + address
4. CE2 size + address
5. NCE3 size + address

The size and address value of the NCE2, CE1, CE2 and NCE3 must be a multiple of #01000h (#04096d nibbles, or 2KB), so the smallest address range we can use is 2KB. The HDW size is always 64 nibbles so the base address must then be a multiple of #00040h (#00064d). All devices should be configured, even if they are unused (meaning they would be configured with the minimum size of 2KB).

IMPORTANT: If two modules are configured on the same address, they are unconfigured by priority and NOT in the reverse order of configuration!

66.3 RESET, CONFIG, UNCNFG, and C=ID

The device configuration is controlled by four commands:

66.3.1 RESET

Unconfigures all devices.

66.3.2 CONFIG

Configure a device using the data in C(A). For all devices, except HDW, `CONFIG` must called twice. This is because it first must specify the size viewed in the Saturn address area and then specify the base address where the device is viewed.

66.3.2.1 Size

The parameter for the size is calculated by the formula:

```
#100000h - #<module_size> = #<size_we_give_it>
```

So, if we want to give the module a size of 32KB (#10000h), we have to find:

```
#100000h - #<module_size> = #10000h
```

This means the module size is #F0000h, which must be given in the C(A) register for the first `CONFIG` command. This means we code it like this:

```
LC F0000            % 32KB
CONFIG              % set device size
```

Why such a strange number? Let's make another example. We want to configure a 128KB card using 384KB as its size at address #00000h.

```
#100000h - #<size_we_give_it> = #<module_size> = #100000h - #C0000h = #40000h
```

The code looks like this:

```
LC 40000            % 384KB
CONFIG              % set device size
LC 00000            % address
CONFIG              % set device address
```

The 128KB card will be configured at #00000h through 3FFFFh and mirrored at #80000h through #BFFFFh. Why is this mirrored?

The reason for this behavior is in the size argument. The first `CONFIG` argument is not really a size but rather a "don't care" mask for the address line. A one bit means "Compare this address line" and a zero bit means "Ignore the address line".

The memory manager uses the address lines A19 through A12. The address lines A11 through A0, which are not connected, are automatically "don't care", as their contents are ignored. Let's write the 384KB (#40000h) size information in binary form and use an 'X' for "don't care" instead of a zero.

A19	A18	A17	A16	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0	
X	1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	= #40000h

As you can see, only A18 is relevant! In the base address we used, #00000h, A18 is cleared, so all addresses where A18 is cleared select the device that has the address ranges #00000h through #3FFFFh and #80000h through #BFFFFh. In all other addresses, A18 is set. The 128KB card is mirrored because the real size is smaller then the configured one.

66.3.2.2 Address

The address parameter tells the device controller where the device should be viewed. But there are some implications. Because the address lines A11 through A0 are not connected and because some address bits may be "don't care", these bits will be ignored because of the size information, so the size and address must be a multiple of #01000h.

Example: We want to locate a 128KB card at #A0100h. Which address area will be used?

```
LC C0000          % 128KB
CONFIG            % set device size
LC A0100          % address
CONFIG            % set device address
```

We simply write the address in binary form and "AND" it with the "don't care" mask of the size information. The result is the first address of the area.

```
  1010 0000 0001 0000 0000b = #A0100h
& 1100 0000 0000 0000 0000b = #C0000h
-----------------------------------
= 1000 0000 0000 0000 0000b = #80000h
```

In our example the 128KB device is configured at #80000h through #BFFFFh.

The next example will show what will happen when the configured size is smaller than the real size of the device and the address is not on a boundary of the memory size.

Example: We want to locate 64KB of a 128KB card at #A0000h. Which part we will see?

```
LC E0000          % 64KB
CONFIG            % set device size
LC A0000          % address
CONFIG            % set device address
```

The 128KB card is configured at #A0000h through #BFFFFh, but we will see the 64KB nibble of the card at the address #A0000h. This is because the address lines A16 through A0 of the 128KB chip are directly connected with the address lines of the Saturn chip (after a nibble to byte conversation inside the Saturn chip). This means the memory controller isn't able to move the position of the 128KB card. So, just remember that all memory modules can only begin on a boundary of their own size, please.

If we want to configure the HDW controller, no size information is needed. There's only one `CONFIG` for the start address.

66.3.3 UNCNFG

This command unconfigures a device, specified by the address in C(A). Normally we use the base address of the uncovered device (the one with the highest priority on this address), but don't forget what we learned about the size information in the CONFIG part above.

Example: We want to unconfigure a 128KB card in a 64KB frame at #A0000h.

```
LC A0000                % address of configured device
% It is possible to use LC A0100 above but it's not recommended
UNCNFG                  % unconfigure device
```

66.3.4 C=ID

Gets information on which device must be configured next and places it in the C register:

1. If there are none, you'll get #00000h in C(A)
2. If there is one, you'll have to check the contents of C(B):

Contents of C(B)	Contents of C(A)
#01h	size of NCE3
#03h	size of NCE2
#05h	size of CE1
#07h	size of CE2
#19h	address of HDW
#F2h	address of NCE3
#F4h	address of NCE2
#F6h	address of CE1
#F8h	address of CE2

You can get the last size or address used for the current memory device, except HDW, by removing the last two nibbles of C (with `C=0 B`). When the current memory address is HDW you only have to clear the last six bits.

Example: Configure the next unconfigured device in its old address space.

```
*CONFIGURE
C=ID                    % get next device to configure
?C=0 A                  % all devices configured
GOYES QUIT
A=C B                   % save device information
C=0 B                   % clear device information to get size/address
CONFIG
?ABIT=0 7               % was it a size information
GOYES CONFIGURE         % yes, then configure address
*QUIT
```

66.4 CONFIG after RESET

What is the default configuring of an HP 48 after `RESET`? Let's make an example. We want to configure a 48GX with:

1.	I/O registers	at #00100h
2.	128KB System RAM	at #80000h, size 128KB
3.	128KB RAM card in slot1	at #C0000h, size 128KB
4.	512KB RAM card in slot2	at #C0000h, size 128KB
5.	Bank switcher	at #7F000h, size 2KB

This is the default mapping of a HP 48GX with a card plugged in port1 and port2.

```
RESET                   % unconfigure all devices
```

```
    LC 00100            % address I/O registers
    CONFIG              % address of HDW
    LC C0000            % 128KB System RAM
    CONFIG              % size of NCE2
    LC 80000            % address System RAM
    CONFIG              % address of NCE2
    LC FF000            % size 2KB Bank switcher
    CONFIG              % size of CE1
    LC 7F000            % address Bank switcher
    CONFIG              % address of CE1
    LC C0000            % 128KB size + address of RAM cards
    CONFIG              % size of CE2
    CONFIG              % address of CE2
    CONFIG              % size of NCE3
    CONFIG              % address of NCE3
```

Now all devices are configured. A `C=ID` command will return #00000h in the C(A) register.

66.5 UNCNFG example

The ROM in a 48GX is covered at several addresses. Let’s use the configuration of the example before. In the next example we want to read five nibbles from the ROM at #FF000h. At this address the ROM is covered by the RAM cards in slot 1 and 2, so we have to remove the memory controllers CE2 and NCE3 from the daisy chain. After reading the data we have to insert the removed memory controllers again. We will assume that the address line A19 is selected. (This will be further explained in the bank switcher section.)

```
    LC C0000            % address of 128KB RAM cards
    UNCNFG              % unconfig card in slot 1 (CE2)
    UNCNFG              % unconfig card in slot 2 (NCE3)
    D0= FF000           % read address
    A=DAT0 A            % read the ROM content
    CONFIG              % config size of CE2
    CONFIG              % config address of CE2
    CONFIG              % config size of NCE3
    CONFIG              % config address of NCE3
```

This was easy, but we assumed that two RAM cards were plugged. What can we do when we don’t know if RAM cards are plugged? We have to rewrite the example before.

```
    LC C0000            %address of 128KB RAM cards

    % unconfig card in slot1
    D0= 8052F           % D0= (=CONFTAB)+#4
    C=DAT0 S
    ?C=0 S              % card in slot 1
    GOYES QUIT1         % no
    UNCNFG              % unconfig card in slot 1 (CE2)
    *QUIT1

    % unconfig card in slot2
    D0= 80531           % D0= (=CONFTAB)+#6
    C=DAT0   S
    ?C=0     S          % card in slot 2
    GOYES QUIT2         % no
    UNCNFG              % unconfig card in slot 2 (NCE3)
```

```
*QUIT2

D0= FF000              % read address
A=DAT0 A               % read the ROM content

% reconfig all unconfigured cards
*CONFIGURE
C=ID                   % get next device to configure
?C=0 A                 % all devices configured
GOYES QUIT
C=0 B                  % clear device information to get size/address
CONFIG
GOTO CONFIGURE
*QUIT
```

66.6 Default configuration settings

66.6.1 HP 48S/SX memory configuration

from #00000h to #000FFh	256 nibbles of ROM
from #00100h to #0013Fh	64 nibbles of I/O RAM which cover 64 nibbles of ROM
from #00140h to #6FFFFh	about 223KB of ROM
from #70000h to #7FFFFh	32KB RAM covers 32KB of ROM
from #80000h to #BFFFFh	128KB card in slot 1
from #C0000h to #FFFFFh	128KB card in slot 2 covers 2KB of unused NCE3 (from #D0000h to #D0FFFh)

66.6.2 HP 48G memory configuration

from #00000h to #000FFh	256 nibbles of ROM
from #00100h to #0013Fh	64 nibbles of I/O RAM which cover 64 nibbles of ROM
from #00140h to #7DFFFh	about 251KB of ROM
from #7E000h to #7EFFFh	empty slot 1, configured to a 2KB size; empty slot 2, covered by 2KB of the empty slot 1; 2KB of ROM are covered by the 2KB given to the empty slot 1
from #7F000h to #7FFFFh	bank switcher, given 2KB which covers 2KB of ROM
from #80000h to #8FFFFh	32KB of RAM which covers 32KB of ROM
from #90000h to #FFFFFh	The rest of the ROM (about 224KB)

66.6.3 HP 48G+ and GX (without RAM card) memory configuration

from #00000h to #000FFh	256 nibbles of ROM
from #00100h to #0013Fh	64 nibbles of I/O RAM which cover 64 nibbles of ROM
from #00140h to #7DFFFh	about 251KB of ROM
from #7E000h to #7EFFFh	empty slot 1, configured to a 2KB size; empty slot 2, covered by 2KB of the empty slot 1; 2KB of ROM are covered by the 2KB given to the empty slot 1
from #7F000h to #7FFFFh	bank switcher, given 2KB which covers 2KB of ROM
from #80000h to #BFFFFh	128KB of RAM which covers 128KB of ROM
from #C0000h to #FFFFFh	128KB of ROM (uncovered)

66.6.4 HP 48GX memory configuration with 32KB in port 1

from #00000h to #000FFh	256 nibbles of ROM
from #00100h to #0013Fh	64 nibbles of I/O RAM which cover 64 nibbles of ROM
from #00140h to #7DFFFh	about 251KB of ROM
from #7E000h to #7EFFFh	empty slot 2, configured to a 2KB size; 2KB of ROM are covered there by the 2KB given to the empty slot 2
from #7F000h to #7FFFFh	bank switcher, given 2KB which covers 2KB of ROM
from #80000h to #BFFFFh	128KB of RAM which covers 128KB of ROM
from #C0000h to #CFFFFh	32KB card in slot 1 which covers 32KB of ROM
from #D0000h to #FFFFFh	96KB of ROM (uncovered)

66.6.5 HP 48GX memory configuration with 128KB in port 1

from #00000h to #000FFh	256 nibbles of ROM
from #00100h to #0013Fh	64 nibbles of I/O RAM which cover 64 nibbles of ROM
from #00140h to #7DFFFh	about 251KB of ROM
from #7E000h to #7EFFFh	empty slot 2, configured to a 2KB size; 2KB of ROM are covered there by the 2KB given to the empty slot 2
from #7F000h to #7FFFFh	bank switcher, given 2KB which covers 2KB of ROM
from #80000h to #BFFFFh	128KB of RAM which covers 128KB of ROM

from #C0000h to #CFFFFh	128KB card in slot 1 which covers 128KB of ROM

66.6.6 HP 48GX memory configuration with 128KB in port 1 and 2

from #00000h to #000FFh	256 nibbles of ROM
from #00100h to #0013Fh	64 nibbles of I/O RAM which cover 64 nibbles of ROM
from #00140h to #7EFFFh	about 253KB of ROM
from #7F000h to #7FFFFh	bank switcher, given 2KB which covers 2KB of ROM
from #80000h to #BFFFFh	128KB of RAM which covers 128KB of ROM
from #C0000h to #CFFFFh	128KB card in slot 1 which covers 128KB of slot 2 and 128KB of ROM covered by both

66.7 GX bank switcher and other ROM stuff

The biggest changes from the HP 48S series to the 48G series are the doubled ROM size and the ability to use a memory card up to 4MB in slot 2. They decided to handle cards larger than 128KB by splitting them into ports of 128KB each with a technique called “bank switching”. What is a bank? A bank in this meaning is an address area shared by more than one device. In contrast to the “covered technology” explained before, all shared devices use the same address area. We use something called a bank switcher to switch the banks (as if you couldn’t figure out what it did by its name). Memory cards in slot 2 are divided into banks of 128KB.

HP implemented this with a new address line (A19) for the bigger ROM and a new memory controller for the bank switcher. To save a chip pin, the address line A19 and the NCE3 control line are multiplexed (meaning they both use the same chip pin). The address line A19 is needed to select the upper 256KB of the ROM, so the NCE3 line has become “chip select” for the memory card in slot 2.

The bank switcher itself has to generate the five upper address lines (A21-A17) for the memory chip in slot 2 as well as an enable signal (BEN) to deselect the memory when the A19/NCE3 pin is used as address line A19. It’s implemented with an external “latch.” By reading a value from the bank switcher memory area the addresses A6-A1 are latched.

The content of the address lines is saved in the following method:

A1 → A17
A2 → A18
A3 → A19
A4 → A20
A5 → A21
A6 → BEN

If BEN=0 then port 2 is always disabled; if BEN=1 then port 2 is controlled by the NCE3 signal.

66.7.1 Switch between A19 and NCE3

Here we will assume that the I/O registers are configured at #00100h, the bank switcher at #7F000h, and the system RAM at #80000h. The control bit DA19 to switch between A19 and NCE3 is placed at bit 3 of the address #00129h. The problem is that this register has different meanings on writing and reading. So, it’s not allowed to read data from the address, manipulate the DA19 bit and write it back. But the system saves the last written value at a global variable in system RAM. Be careful about the order of the DA19 and BEN access.

Example: Switch to upper ROM access.

```
D0= 7F000              % BEN=0, disable port 2
C=DAT0 B
D0= 8069A              % D0=(5)LINECOUNTg
C=DAT0 B               % get content of LINECOUNT
CBIT=1 7               % DA19=1, enable ROM
D0= 00128              % D0=(5)LINECOUNT
DAT0=C B
```

Example: Switch to port 2 access.

```
D0= 8069A              % D0=(5)LINECOUNTg
C=DAT0 B               % get content of LINECOUNT
CBIT=0 7               % DA19=0, disable ROM
D0= 00128              % D0=(5)LINECOUNT
DAT0=C B
D0= 7F040              % BEN=1, enable port 2
C=DAT0 B
```

66.7.2 Select a bank

We now know how the hardware works and that we're controlling the upper address lines with a read from the memory controller. The formula to select a bank is:

```
#<memory_address> = #7F000 + #40 + 2 * n
```

(where #7F000 is the base address of the bank switcher, #40 represents a set BEN signal and *n* is the selected bank (0-31) in port 2)

Example: I want to select bank 2 of a 512KB card, so I do:

```
D1= 7F044
A=DAT1 B
```

The read byte must be thrown away.

66.7.3 Invalid card data with 4MB cards in slot 2

The message "Warning: Invalid Card Data" always appears in a GX with a plugged 4MB card in slot 2. This is a bug in the HP 48GX firmware.

The following description of this bug is based on Christoph Giesselink's personal experiments and contradicts the official HP reasoning.

The problem is, what is the last address on the address bus when one does a read with the following code?

```
D1= 7F044
A=DAT1 B
```

This content is latched. A byte read reads two nibbles so the above code reads the nibbles at #7F044 and #7F045. Because A0 of the chip's internal address bus isn't wired out, we can ignore the LSB of the reading address. So address #7F044 is latched.

But my experiences show that when I do a byte read at that address, the addresses #7F044, #7F045 and #7F046 are read. So address #7F046 is latched and that's the address of the next bank.

But what has this to do with 4MB card? Remember, when we want to select bank 31, the formula is *#7F000 + #40 + 2 * 31 = #7F07E*. With a byte read access the latched address is #7F080. But with this address BEN=0! The RAM in slot 2 is no longer selected!

But why does a 256KB card, for example, work fine? A 256KB card has two 128KB banks and use only A17 from the latched address bus.

We try to access bank 0:

```
   D1= 7F040
   A=DAT1 B
```

#7F042 is latched: BEN=1, A17=1. The second bank is selected.

We try to access bank 1:

```
   D1= 7F042
   A=DAT1 B
```

#7F044 is latched: BEN=1, A17=0. The first bank is selected.

Everything is working fine because you read and write data to the memory card with the same system.

Can we use this information to write data to bank 31 of a 4MB card? We know that bank 0 is always saved in the second 128KB area and the last bank is saved in the first 128KB area. A read on #7F07E isn't possible because of the non-existent BEN signal. But with the following code:

```
   D1= 7F03E
   A=DAT1 B
```

#7F040 is latched: BEN=1, A17-21=0. The first bank is selected!

But what about writing instead of reading to activate a bank? A difficult question; this depends on the state of the write protected switch on the memory card in slot 2.

For understanding this we have to go back into the history of the S-series calculators. On an SX calculator the two slots are connected with CE1 and CE2 of the Clarke MMU (memory management unit). Each output pin is controlled by a special input pin connected to the write-protected switch of the memory card. If the MMU detects that the card is "write protected," no chip select is generated when you try to write to the memory card, but reading just work fine.

For some reason, Hewlett Packard moved the memory cards to CE2 and NCE3 in the G-series. But the NCE3 chip select output pin is not controlled by such a special input pin to prevent writing. Therefore, GX memory cards must have special control logic on the memory card for "write protecting." But the operating system should know if the card in slot 2 "is" write protected or "not". The write protected switch of the card in slot 2 is connected to the free control input of CE1. At the GX the bank switcher flip flop is connected to CE1, so the write protect switch of the memory card in slot 2 controls writing on the bank switcher flip flop!

We discussed the reading condition before, so let's have a look what happens when we are writing to the bank switcher.

If the card is not "write protected" the expected value is latched.

Example:

```
D1= 7F040
DAT1=A B
```

The byte write command writes two nibbles at #7F040 and #7F041, so #7E041 is latched.

If the card is "write protected," we have no write access to the latch. As you remember, the Saturn CPU has a four-bit data bus, the external modules have an eight-bit data bus. So the Clarke/Yorke chips have a special unit inside for these nibble/byte conversations. If you want to write two nibbles to an external memory device and these two nibbles are stored on the same byte address, we need one write cycle to save the data. But if we want to modify only one nibble in such a byte-addressed memory, the controller must do a read/modify/write cycle (reading the byte, modifying one nibble, writing the modified byte back). The write cycle is disabled because the bank switcher is "write protected," but the read cycle modifies the latch content. The latched address is the highest address with a nibble read.

Example:

```
D1= 7F040
DAT1=A B
```

This code doesn't work; there is no change in the latched data because there is no read cycle.

Example:

```
D1= 7F041
DAT1=A B
```

This works; #7E042 is latched because of modifying the nibble located in the #7E042, #7E043 byte.

As we can see, writing makes more trouble than reading, so please use the published formula for bank switching with a byte read.

67 HP 49 memory management

67.1 Overview

The HP 49G has an Intel TE28F160-S5100 2MB flash memory chip. The HP 49G+ initially had a Silicon Storage Technology SST39VF160-70-4C-EK 2MB flash memory chip, but this chip was EOL'd on December 8, 2003, and it is not known if a different chip was later used. Both calculators use a 512KB SRAM chip, with the BSI BS616LV4010EC-70 seen on the 49G+ and both the Samsung KM684000BLT-5 and Hyundai HY628400 seen on the 49G.

From a programmer's point of view, the exact chips used don't really matter; all we care about is that the flash ROM is 2MB and the RAM is 512KB. The flash ROM is internally split into banks of 64KB, but this is mostly abstracted into 128KB banks seen by the programmer.

Like the 48, the 49 is still limited by 20-bit addressing, so the Saturn can only see 512KB at any time. This means that memory management on the 49 has a lot in common with that of the 48. The 49 requires a more advanced bank-switching system than the 48 in order to handle the 2.5MB of total memory, but the memory is still split into banks of 128KB.

Because banks are 128KB, the flash ROM is split into 16 banks. The first bank is split in half. The first 64KB half is reserved for the "boot sector". This is read-only, and the 49G enforces this in hardware. The second 64KB half is available for user storage on ROM versions through 1.18, and it is used by the system for part of the CAS on higher ROM versions. The remaining 15 of the 128KB banks are either used by the system or available to the user, and the exact numbers depend on the ROM version.

The RAM is divided into two ports. The first port, port 0, is shared with the HOME directory and is 256KB. The second port, port 1, is not shared. On the 49G, it is also 256KB. On the 49G+, port 1 is only 128KB, because the underlying operating system (Kinpo OS) and the emulation layer (Saturn emulator) use the remaining 128KB. Some of this reserved memory is available to ARM programmers, but that is outside the scope of this book.

The 49G+ operating system also takes two additional 128KB banks of flash, resulting in fewer flash memory banks available to the user.

Below is a table showing the ROM and RAM bank usage on "traditional" ROMs. This means that ROM versions under 1.20 are for the 49G only, and ROM versions 1.20 and higher are for the 49G+ only.

ROM version	ROM Bank 0	System ROM banks	User ROM banks	RAM Port 0	RAM Port 1
1.05 – 1.18	half boot / half user	next 7 banks	last 8 banks	2 banks	2 banks
1.19 Beta 6	half boot / half system	next 7 banks	last 8 banks	2 banks	2 banks
1.20 +	half boot / half system	next 8 banks	last 7 banks	2 banks	1 bank

As you can see, there is a total of 20 banks of memory (ROM+RAM) in the HP 49, adding up to the 2.5MB mentioned earlier. Only 4 banks can be visible at any time, so the system must selectively map whichever banks are needed at a given time to one of these four addressable banks.

Inside the ".flash" flash update file for the 49G, each of these banks has its own name. The built-in ROM update tool uses these names to determine where in flash to store each transferred bank. The first half of ROM bank 0 is called Part0, and the second half is called FS. The first full 128KB bank is called System, and the remaining banks are called Part1, Part2, Part3, Part4, Part6, Part7, and so on. Note that there is no bank named Part5.

The best way to show how the memory can be addressed is with a table. Each line in the table represents a 128KB bank of addressable space, with the four banks adding up to the Saturn's 512KB maximum.

from #00000h to #3FFFFh	mapped to one of the first 4 banks of flash ROM
from #40000h to #7FFFFh	mapped to any one bank of flash ROM or either bank of RAM port 1
from #80000h to #BFFFFh	normally mapped to first bank of RAM port 0
from #C0000h to #FFFFFh	normally mapped to second bank of RAM port 0

As you can see, one of the first four banks of ROM is always mapped at any given time, and both banks of port 0 of RAM are normally mapped. The last 12 banks of ROM, or one of the 3 of the first 4 banks of ROM that are not already mapped, all must share the second bank of address space.

67.2 Hardware details

The HP 49G uses the same fourth generation Saturn chip with the code name Yorke as the 48G series. For better understanding of the next sections, the basics from the section "HP 48 memory management" should be read first. We have the same memory controllers with the names NCE1, HD, NCE2, CE1, CE2 and NCE3 (with the exception of the first in the order of daisy chain configuration).

The 49G+ is similar to the 49G from a software perspective, but because the underlying hardware is different, though abstracted by the emulation layer, not everything in this section will be valid. Christoph Giesselink graciously wrote the remainder of this section.

67.3 A19 and NCE3 sharing

As we learned from the HP 48, the A19 and NCE3 hardware signals use the same connector on the Yorke chip. The behavior of this pin is controlled by the DA19 bit in the LINECOUNT register. To avoid any trouble in the 49G this shared pin is always used as memory controller NCE3, and never as address line A19. This means also that the maximum addressable size of one module is 256KB ($2^{19}/2$).

67.4 Memory controllers

Device	Function on 49G
NCE1	Flash chip (read only) (2MB)
HDW	I/O registers (32B)
NCE2	RAM (256KB)
CE1	Bank switcher
CE2	RAM (128KB)
NCE3	RAM / Flash chip (read/write) (128KB)

67.4.1 NCE1 (Flash chip)

This is the memory controller for the 2MB flash chip. Because this memory controller was designed for ROM access, writing to this controller isn't possible. To get the complete 2MB of the chip visible in the 512KB address area of the Saturn chip, the 2MB are divided into 16 128KB banks controlled by the bank switcher.

from #00000h to #3FFFFh	contain one of the first four 128KB banks
from #40000h to #7FFFFh	contain one of the sixteen addressable 128KB banks
from #80000h to #BFFFFh	mirror of the #00000h to #3FFFFh area
from #C0000h to #FFFFFh	mirror of the #40000h to #7FFFFh area

Remember, this is the controller with the lowest priority and can be covered by any other memory controller.

67.4.2 HDW

The I/O register controller.

67.4.3 NCE2 (RAM)

This controller manages a 256KB bock of the 512KB RAM. The RAM part containing the HOME directory and port 0 data is normally configured from #80000h to #FFFFFh in the Saturn address area.

67.4.4 CE1 (Bank switcher)

The bank switcher is implemented as a six-bit latch, like the 48GX. We also have the same misbehavior of setting wrong data with a read command like with the 48GX bank switcher. In contrast to the 48GX, it’s allowed to set the bank switcher latch with a write command, which works as expected. This is possible because on the 49G the corresponding card detection line is set to read/write capabilities.

The lower two bits of the six-bit latch select one of the four (2^2) banks of the flash for the #00000h - #3FFFFh address area. The upper four bits control the sixteen (2^4) possible banks of the flash in the #40000h - #7FFFFh address area. The six-bit latch is connected to the address lines A1-A6.

A1 $\rightarrow$ bit 0 bank select area #00000h - #3FFFFh

A2 → bit 1 bank select area #00000h - #3FFFFh
A3 → bit 0 bank select area #40000h - #7FFFFh
A4 → bit 1 bank select area #40000h - #7FFFFh
A5 → bit 2 bank select area #40000h - #7FFFFh
A6 → bit 3 bank select area #40000h - #7FFFFh

In contrast to the HP 48 series, this controller isn't configured by default.

Example: Set #00000h to Flash bank 2 and #40000h to Flash bank 12 (zero based numbering)

```
LC FF000            % 2KB
CONFIG              % set size CE1 (bank switcher)
LC 3F000            % address
CONFIG              % set device address CE1 (bank switcher)
LC 64               % 1100100b
                    % 1100 10 0
                    % =12  =2 =address line A0
D0=C
DAT0=A B            % set the latch with bank 12 and bank 2 information
UNCNFG              % unconfig CE1 (bank switcher)
```

67.4.5 CE2 (RAM)

This controller manages a 128KB block of the 512KB RAM and contains a half part of port 1 data. It's unconfigured by default.

67.4.6 NCE3 (RAM / flash chip)

This controller manages the last 128KB block of the 512KB RAM and contains the other half part of port 1 data. It's also unconfigured by default.

The other use of NCE3 is to allow write access to the flash chip connected to NCE1. Therefore the LED bit in the LCR register must be set.

Example: Prepare NCE3 for flash write access at #40000h

```
LC FF000            % 2KB
CONFIG              % set size CE1 (bank switcher)
LC 3F000            % address
CONFIG              % set device address CE1 (bank switcher)
LC FF000            % 2KB
CONFIG              % set size CE2 (RAM 1)
LC 3E000            % address
CONFIG              % set device address CE2 (RAM 1)
LC C0000            % 128KB
CONFIG              % set size NCE3 (ROM)
LC 40000            % address
CONFIG              % set device address NCE3 (ROM)
D1= 0011C           % load LCR - LED Control Register
LC 8                % turn on LED
DAT1=C 1
```

After this code sequence we have full read/write access to the 128KB flash chip bank, selected by the upper four bits of the bank switcher, in the memory area from #40000 to #7FFFF. Remember, a flash chip cannot be used as simple RAM. For writing to the chip we have to use a special manufacturer-specific command set which isn't part of this tutorial.

67.5 Restoring bank switcher content

Example: Restore bank switcher content to operating system value

```
LC FF000                % 2KB
CONFIG                  % set size CE1 (bank switcher)
LC 3F000                % address
CONFIG                  % set device address CE1 (bank switcher)
D0= 860B8               % =CurROMBank2 get last written BS address [A]
C=DAT0 B                % CE1 (bank switcher), address (3F0xxH)
D0=C                    % reload BS address
DAT0=C B                % dummy write to reload BS
UNCNFG                  % unconfig CE1 (bank switcher)
```

68 Graphics

Christoph Giesselink deserves a **huge** thank-you for writing this section! Please note that some of the numbers in this section assume a 64-pixel tall screen.

68.1 General

The calculators based on the Clarke or Yorke chip (38G, 39G/40G, 48S/SX/G/G+/GX, and 49G) use a liquid crystal display (LCD) with a resolution of 131 by 64 pixels. The LCD is controlled by the Clarke/Yorke chip and two commercial column driver chips.

A 4096 Hz interrupt redraws the next display row. Because you have 64 rows refreshed by a 4096 Hz interrupt the whole display has a refresh rate of 4096 Hz / 64 rows = 64 Hz. Refreshing one display line takes about 22-23 µs every 244 µs. A UMA (Unified Memory Architecture) layout is used, meaning that the RAM for the CPU and display controller is not separated. The advantage of this is that the CPU has full control over the display, and there's no need to copy data from the CPU to the display memory. The disadvantage is that when the display is refreshed, the CPU must be stopped to guarantee the RAM access for the display controller. This will slow down the CPU speed.

The display is controlled by several I/O registers. This section assumes that the I/O registers are mapped at #00100 (this is the normal setting). The display only works with Graphic Objects (GROBs), meaning that each bit in the display memory represents a pixel on the display.

Example: Read number of last updated row to A(B).

```
LC 3F                   % mask of 6 row bits
D0=(5) "LINECOUNT"      % =LINECOUNT is 00128
A=DAT0 B                % get last updated row
A=A&C B                 % clear DA19 and M32 bits
```

This row counter is a down counter. The top row has number #3Fh and the bottom row number #0h. This code is primarily used for synchronizing output.

Example: Wait until the last row is drawn.

```
LC 3F                   % mask of 6 row bits
D0=(5) "LINECOUNT"      % =LINECOUNT is 00128
```

```
*LOOP
A=DAT0 B              % get last updated row
A=A&C B               % clear DA19 and M32 bits
?A#0 B                % not zero
GOYES LOOP            % then wait
```

68.2 Main / Menu display

The display area is divided into two parts by hardware: the "Main display" area and the "Menu display" area. What's the difference between these areas? On top of the display you have the main display area. In this area the GROB data can be greater or smaller than the viewed one. We're able to shift the display contents with reprogramming the display controller without changing the GROB data. On the bottom, the menu area has a fixed length of 131 pixels. This design is very useful for the HP calculators, because you're able to scroll or modify the main display content without modifying the menu area.

With the `LINECOUNT` register you can configure how many lines the main area and the menu area use. For internal reasons it isn't possible to set the main area to zero lines. Using #0 or #1 as the `=LINECOUNT` argument is the same as using #3F, allowing the main area to use the full 64 lines.

Example: Split "Main display" to 56 lines and "Menu display" to 8 lines height.

```
D0=(5) "LINECOUNT"    % =LINECOUNT is 00128
LC 37                 % 56 lines main display
DAT0=C B
```

As we see, we have to program the last line of the "Main display" area. So it's not possible to have no "Main display" area. But programming 63 into the `LINECOUNT` register will purge the "Menu display" area.

All addresses of the display controller are BYTE aligned! That mean, that every display line begins on an even address. Each display area has its own base address pointing to a GROB data field, so it's important that the GROB data field begins on an even address. The base address of the "Main display" area is located at #00120h-#00124h (`=DISP1CTL`) and the base address of the "Menu display" area is located at #00130h-00134h (`=DISP2CTL`). These registers are W/O (write only) and have to point to the beginning of the display GROB in RAM. The display GROB is just a normal data field of a GROB object, the representation of the pixel state.

Because these and two other display registers are "write only," the operating system uses so-called "ghost" registers in RAM, saving the last written value for restoring the original content. Assuming a default RAM layout, the addresses of the ghost registers are:

	Register	**Ghost**	**38G**	**48SX**	**39G/40G/48G*x*/49G*x***
=DISP1CTL	#00120	=DISP1CTLg	#F062C	#7050E	#8068D
=LINENIBS	#00125	=LINENIBSg	#F0631	#70513	#80692
=LINECOUNT	#00128	=LINECOUNTg	#F0639	#7051B	#8069A
=DISP2CTL	#00130	=DISP2CTLg	#F0634	#70516	#80695

These entry points are not in all entry point tables, so it is possible that you will have to manually type the addresses even if you have an entry point table installed.

Example: Restore the "Main display" area pointer.

```
D0=(5) "DISP1CTLg"    % =DISP1CTLg is 8068D on the 48G series and 49
C=DAT0 A
D0=(5) "DISP1CTL"     % =DISP1CTL is 00120
DAT0=C   A
```

68.3 Main display area setup

The dimension of the GROB in the “Main display” area may be different from the display size, so we have to adjust the display control registers for a correct view.

We have three main registers to control the main display area:

#00120h - #00124h	=DISP1CTL	Display Start Address (W/O)
#00100h	=BITOFFSET	Left Margin of GROB (R/W)
#00125h - #00127h	=LINENIBS	Right Margin of GROB (W/O)

`DISP1CTL` defines the base address of the bit pattern of the GROB. Remember, the start address must be even! With the lower 3 bits of the `BITOFFSET` register [OFF2 OFF1 OFF0] we can scroll the GROB up to 7 (2^3-1) pixels to the left.

Example: Scroll “Main display” area content 2 pixels to the left.

```
D0=(5) "BITOFFSET"    % =BITOFFSET is 00100
LC A                  % DON bit + 2 pixel shift
DAT0=C 1
```

But how can we scroll the display content with more than 7 pixels? That’s easy. For 8 pixels we simply add 2 (2 * 1 nibble = 2 * 4 bit = 8 bit = 8 pixel) to the screen base address.

Example: Scroll “Main display” area content 13 (8+5) pixels to the left.

```
D0=(5) "DISP1CTLg"    % =DISP1CTLg is 8068D on the 48G series and 49
C=DAT0 A              % remember, we have to use the ghost copy
C=C+1 A               % 8 pixel
C=C+1 A
D0=(5) "DISP1CTL"     % =DISP1CTL is 00120
DAT0=C A
D0=(5) "BITOFFSET"    % =BITOFFSET is 00100
LC D                  % DON bit + 5 pixel shift
DAT0=C 1
```

So, now we have to adjust the `LINENIBS` register for setting the right margin of the GROB. When we want to show a line of 131 pixels we have to calculate the number of nibbles needed for each line.

Number of nibbles = (CEIL(131 / 8) * 8) / 4 = 34

We have to reserve 34 nibbles for each display line. But why 34 nibbles and not 33 (CEIL(131 / 4))? As I told you, the display addresses are byte aligned, so you have to divide the number of pixels (131) by the number of bits (pixels) in a byte (8).

131 / 8 = 16.375

Of course you can’t reserve 0.375 bytes, so you have to reserve 17 bytes over all.

CEIL(131 / 8) = 17 bytes

With 17 bytes we’re able to control 136 pixels (17 * 8), so we have to reserve 34 nibbles (136 / 4) for the line. But here we have a new problem. When we scroll the display content more than 5 pixels (136 - 131) to the left, we need some more “extra” nibbles to view the line correctly, because the line is now larger than the 136 reserved pixels (131 pixel viewed + 6 pixel shifted). The following formula takes this into account.

Address of next line = Address of current line + 34 + Right Margin + FLOOR(Left Margin / 4) where

Address of next line - Address of current line = number of nibbles of the display line

For our example line of 131 pixels and no left margin shift, the right margin register must be loaded with:

```
LINENIBS  =  Right Margin = Address of next line - Address of current line  - 34 - FLOOR(Left Margin / 4)
          =  34                                                             - 34 - FLOOR(0          / 4)
          =  0
```

Example: Configure GROB 159 * 64 with 3 pixels scrolled to the left.

No. of nibbles = (CEIL(159 / 8) * 8) / 4 = 40

LINENIBS = 40 - 34 - FLOOR(3/4) = 6

```
D0=(5) "BITOFFSET"    % =BITOFFSET is 00100
LC B                  % DON bit + 3 pixel shift
DAT0=C 1
D0=(5) "LINENIBS"     % =LINENIBS is 00125
LC(3) 6               % right margin
DAT0=C 3
```

Example: Configure GROB 159 * 64 but now with 4 pixels scrolled to the left.

No. of nibbles = (CEIL(159 / 8) * 8) / 4 = 40

LINENIBS = 40 - 34 - FLOOR(4/4) = 5

```
D0=(5) "BITOFFSET"    % =BITOFFSET is 00100
LC(1) B               % DON bit + 3 pixel shift
DAT0=C 1
D0=(5) "LINENIBS"     % =LINENIBS is 00125
LC(3) 5               % right margin, #4 will also work here
DAT0=C 3
```

Because of the BYTE aligned display controller interface the next line address is internally calculated as

Address of next line = Address of current line + (34 + Right Margin + FLOOR(Left Margin / 4) * 2) & #FFE

Never use this formula for calculating the right margin; it's only for demonstration purposes! So you can also use #4 as Right Margin in the prior example. The HP function `=WINDOWRIGHT`, for example, uses this behavior. Instead of recalculating the right margin value when the BITOFFSET value changes from 3 to 4, the right margin value is always subtracted with 2 at this point.

Here's also a tricky example from Jean-Yves Avenard reversing the "Main display" area in the screen:

```
LINENIBS  =  Right Margin = Address of next line - Address of current line  - 34 - FLOOR(Left Margin / 4)
          =  -34                                                            - 34 - FLOOR(0          / 4)
          =  -68
          =  #FBC (3 nibble width)
```

Program:

```
GOSBVL =SAVPTR        % =SAVPTR is 0679B
D0=(5) "DISP1CTLg"    % =DISP1CTLg is 8068D on the 48G series and 49
A=DAT0 A
LC FBC0074E           % #0074E = 1870 = 55 * 34 = offset to end of main display area
```

```
    C=C+A A
    D0=(5) "DISP1CTL"      % =DISP1CTL is 00120
    DAT0=C 8
    GOVLNG =GETPTRLOOP     % =GETPTRLOOP is 05143
```

68.4 Menu display area setup

The GROB in the “Menu display” area must always have a width of 131 pixels (34 nibbles), because you can’t adjust the left and right margins. The y-dimension is controlled by writing the `LINECOUNT` register.

We have one main register to control the menu display area:

#00130h - #00134h =DISP2CTL   Menu Start Address (W/O)

DISP2CTL defines the base address of the bit pattern of the GROB. Remember, the start address must be even!

68.5 Display I/O Register Overview

This section discusses the bits of the display I/O registers in numeric order.

(R/W) = Read/Write
(R/O) = Read/Only
(W/O) = Write/Only

#00100h=BITOFFSET (R/W)

Bit 3 [DON]
This is the display enable flag; this bit must be set to activate the refresh of the display. If it’s cleared, the display is off, and the CPU isn’t interrupted every 4096 Hz. If you want to increase your calculator’s speed, just turn off the display.

Bit 2-0 [OFF2 OFF1 OFF0]
This controls the left margin of the main display area. The number is the vertical offset of the display. If you want to scroll more then seven pixels you have to increment the base address of the main display area by two. When Bit 2 is set, using a vertical offset greater than three, you have to adjust the right margin value as well.

#00101 =CONTRAST (R/W)

Bit 3-0 [CON3 CON2 CON1 CON0]
The four lower bits of the five-bit contrast value. Higher values mean darker contrast.

#00102 =DTEST (R/W)

Bit 3-1 [VDIG LID TRIM]
Internal control bits, which should be 0!

Bit 0 [CON4]
This is the upper bit of the five-bit contrast value.

#00103 =DSPCTL (W/O)

Bit 3-0 [LRT LRTD LRTC BIN]
Internal control bits; should be 0! Playing around with these control bits may damage your display! You have been warned!

#0010B =ANNCTRL (R/W)

Bit 3-1 [LA4 LA3 LA2 LA1]
The four lower bits of the six-bit annunciator value. Each bit is representing an annunciator symbol. 0 means OFF, and 1 means ON.

LA4 = Alert
LA3 = Alpha
LA2 = Right Shift
LA1 = Left Shift

#0010C =ANNCTRL+1 (R/W)

Bit 3 [AON]
This is the annunciator enable flag. Set this bit to enable the annunciators. The annunciators are independent from the display DON bit, so you can switch off the display and still use the annunciators.

Bit 2 [XTRA]
This bit is unused.

Bit 1-0 [LA6 LA5]
The two upper bits of the six-bit annunciator value.

LA6 = Transmitting
LA5 = Busy

Example: Switching ON the Alpha annunciator, all others OFF.

```
   D0=(5) "ANNCTRL"
   LC 84                    %AON or LA3
   DAT0=C B
```

#00120 =DISP1CTL (W/O)
This is the five-nibble Main display start address (LSB first). Please refer to sections 68.2 and 68.3 for more information.

#00123 =LINENIBS (W/O)
This is the three-nibble Main Display Right Margin configuration register (signed value, LSB first). Please refer to section 68.3 for more information.

#00128 =LINECOUNT (W/O)

Bit 3-0 [LC3 LC2 LC1 LC0]
The four lower bits of the six-bit screen height value. Please refer to section 68.2 for more information.

#00128 =LINECOUNT (R/O)

Bit 3-0 [LC3 LC2 LC1 LC0]
The four lower bits of the six-bit line draw value. Please refer to section 68.1 for more information.

#00129 =LINECOUNT+1 (W/O)

Bit 3 [DA19]
Address line 19 control. Leave this bit unchanged please!

Bit 2 [M32]
Unused. Leave this bit unchanged please!

Bit 1-0 [LC5 LC4]
The two upper bits of the six bit screen height value. Please refer to section 68.2 for more information.

#00129 =LINECOUNT+1 (R/O)

Bit 3 [DA19]
Value of the address line 19 control.

Bit 2 [M32]
Unused.

Bit 1-0 [LC5 LC4]
The two upper bits of the six-bit line counter value. Please refer to section 68.1 for more information.

#00130 =DISP2CTL (W/O)
This is the five-nibble Menu display start address (LSB first). Please refer to sections 68.2 and 68.4 for more information.

That's all for now. We hope you enjoyed this, and watch for Edition 4 in the future!

69 Additional reading

Although this book is intended to give a comprehensive introduction to assembly language programming, it still cannot be your only source of programming information. Most programs written in assembly language still have some System RPL or User RPL code to perform some operations, so it is important to know those languages as well. In addition, other assembly language tutorials have been written over the years that can provide different points of view if you are confused by any of the topics in this book. We have compiled a list of recommended additional reading below.

69.1 Internal development tools

We mentioned this earlier, but it is important enough that we feel it should be mentioned again. The best source for information about using MASD on the 49 is the documentation written by Cyrille de Brébisson, the programmer of MASD. Although it can be a bit tough to follow at times, it comprehensively covers all of MASD's unique features.

http://www.hpcalc.org/details.php?id=2986
→ http://www.hpcalc.org/hp49/docs/programming/masddocs.zip

The 49 also includes a built-in library creator, briefly mentioned in the section on the development library, but more comprehensively documented in the below document:

http://www.hpcalc.org/details.php?id=2980
→ http://www.hpcalc.org/hp49/docs/programming/crlib.zip

The library creator also provides the ability to extend the built-in menus, described in the following document:

http://www.hpcalc.org/details.php?id=2981
→ http://www.hpcalc.org/hp49/docs/programming/extprg.zip

When you write your own program and package it as a library, you may want to include documentation accessible through the help catalog. This is supported by the library creator and is documented in the file below:

http://www.hpcalc.org/details.php?id=2979
→ http://www.hpcalc.org/hp49/docs/programming/addhelp.zip

69.2 Assembly language

HP's original SASM document for programming the Saturn using the HP syntax is available in text format below:

http://www.hpcalc.org/details.php?id=1746
→ http://www.hpcalc.org/hp48/docs/programming/sasm.zip

Matthew Mastracci wrote a comprehensive document on the Saturn architecture and the hardware of the 48, available as a plain text document at the following address:

http://www.hpcalc.org/details.php?id=1753
→ http://www.hpcalc.org/hp48/docs/programming/saturnd.zip

Peter Geelhoed has provided a large collection of simple example programs written in assembly language, which anybody learning Saturn assembly should find useful. It is available in PDF format below:

http://www.hpcalc.org/details.php?id=5007
→ http://www.hpcalc.org/hp49/docs/programming/mltut.zip

69.3 System RPL

HP's original RPLMAN document for programming the HP 48 in System RPL is available in text format below:

http://www.hpcalc.org/details.php?id=1743
→ http://www.hpcalc.org/hp48/docs/programming/rplman.zip

Also mentioned earlier is Eduardo Kalinowski and Carsten Dominik's excellent book, Programming in System RPL, which provides a comprehensive tutorial and reference for System RPL programming in a PDF document:

http://www.hpcalc.org/details.php?id=5142
→ http://www.hpcalc.org/hp49/docs/programming/progsysrpl_pdf.zip

69.4 User RPL

The instruction manuals provided with the 48 and 49 series provide a good introduction to User RPL, but far more additional information is available separately.

HP's 48G Series Advanced User's Reference manual, no longer in print, is available for download. It is a huge (nearly 40MB) PDF file with over 700 pages of documentation covering all aspects of User RPL programming and a reference of all User RPL commands.

http://www.hpcalc.org/details.php?id=6036
→ http://www.hpcalc.org/hp48/docs/misc/hp48gaur.zip

HP's 49G Advanced Users Guide, though not as easy-to-read as the 48G's AUR, provides a reference for the 49's new User RPL commands and can be downloaded in PDF format below:

http://www.hpcalc.org/details.php?id=2998
→ http://www.hpcalc.org/hp49/docs/misc/49g_aug.zip

One-Minute Marvels, a collection of 100 User RPL programs that each take only a minute to type in, was written by Richard Nelson and Wlodek Mier-Jedrzejowicz and provides an excellent set of examples for somebody wanting to learn User RPL tricks. The PDF document is available below:

http://www.hpcalc.org/details.php?id=1691
→ http://www.hpcalc.org/hp48/docs/programming/1minmarv.zip

70 Index