

Writing Externals for Hermes

Copyright © 1993 by Sam Shafritz of Intermind Software



Issue 1: Writing Your First External, August 1993

WE4H Explained:

Writing Externals for Hermes, or WE4H, are "tech notes" designed to bring the ability of programming external into the hands of the average Hermes SysOp. You do not need to be familiar with any programming language or techniques to use and benefit from WE4H. In fact, if you are familiar with Pascal, most of the tips revealed here will seem obvious or trivial. This newsletter is for intelligent people with no programming background.

Each issue of WE4H will concentrate on a specific area of Hermes external development. The exception is this first issue: Writing Your First external. This issue will take you through the entire process from beginning to end of writing and compiling your very first Hermes external.

Why WE4H Exists:

When I first started writing my external I was incredibly frustrated at how long it took to get my questions answered. I would want to continue writing, but I needed an answer before I could continue. My only source of information at that time was Frank Price on the Hermes Support BBS. Here is one of his actual responses to a question of mine:

(1/1): Sane.p
Name: AOC Software, Inc. #1 Last on: 1/12/92
Date: Mon, Jan 12, 1992 12:39:47 PM

It is in the Sane.p interface. Please don't ask programming questions again.

AOC Software, Inc.

You can see why it is so easy to get discouraged. Not only did he not answer my question sufficiently, but he told me not to ask questions again! I found his response both frustrating and confusing. This was the man who wrote, "I would be glad to answer specific questions about external on the support BBS."

During my first few months of learning, I would dream about some mythical source of information that could answer all my questions quickly and completely. Well, through WE for H I hope to create such a source of information. Most Hermes SysOps learn how to use the application of Hermes incredibly efficiently. They learn all the tricks and consider themselves experts on Hermes. But why do they stop there? That's only the first step. I urge you to go beyond that. Until you know how to write your own external for Hermes you will never be able to fully customize your BBS. I'm not necessarily advocating writing external and releasing them for the public's use -- although that is certainly an option you will have. But wouldn't it be nice to be able to whip off a quick external that'll fix your current problem?

For example, I needed a list of every user on my board who was 34 years old. Not greater than 34 years old, not less than 34 years old, but exactly 34 years old. (You might be wondering why I needed such a list, although this is irrelevant I'll explain anyway. I had forgotten a user's name but knew he was 34 years old. I realized there couldn't be that many 34 years olds on my board and once I saw his name I would recognize it.) So I whipped off an external that listed every 34 year old and voilà, I had a list of

four users. I immediately remembered his name after seeing it and the problem was solved. Had I not been able to write such an external, I would have been forced to look at every single user's name until I found the right one.

Now, admittedly, this is a weird example. But I'm sure you can think of similar situations where being able to manipulate externals would be useful. I think the most common reason average Hermes SysOp's don't learn how to write externals is because they believe this myth that programming is incredibly complicated and it takes years to learn. Or they believe they must take some sort of course or read dozens of books to understand how to write externals. Nothing could be further from the truth.

If you are a reasonably intelligent person, (I.Q. > 100) read this entire issue of WE4H, and want to learn, you will be able to write your very own external in less than one week. However, reading this and other issues may not answer all your questions. You may have very specific questions that I don't cover. If this is the case I want to hear from you. Ask me your specific questions. If you like me and hate to wait for U.S.P.S. Mail, or even E-Mail, then call me! If you're willing to pay for the call I'll be happy to answer any question I can. This phone number will be valid from September 29, 1993 until December ??, 1993. **(412) 648-4618**, this is a VOICE number -- I do not have a BBS currently running. Please call between 9 AM and 9 PM EST. If you don't wanna pay for the call, or you're just too embarrassed to call voice, my U.S.P.S. address for the same time period is:

Sam Shafritz
Tower C Room 513
3990 Fifth Ave.
Pittsburgh, PA 15213-3524

My Internet address is: SJSST32@VMS.CIS.PITT.EDU

My account number on the Hermes Support BBS is: 665

Please don't hesitate to call or write. I remember exactly how frustrating it is, and I can probably answer your question in two seconds and get you rolling again.

First things first:

The absolute first step in writing externals for Hermes is to get yourself a copy of THINK Pascal v4.0.1 or higher. Other Pascal compilers will work and even other languages will work, but there are several advantages to THINK Pascal v4.0.1. First, all my examples here will be in Pascal and specifically THINK Pascal v4.0.1. Second, Hermes itself was written in THINK Pascal and it is somewhat excepted as a standard in the Hermes development community. If you have a choice -- go with THINK Pascal.

Where can you get THINK Pascal? Lots of places. I'm not going to tell you where to buy it from and I'm certainly not going to advocate pirating. My advise is to get it from wherever you normally buy software. Usually mail-order places like Mac Connection or Mac's Place have the best deals. Software stores like EggHead usually sell it at very inflated prices.

A few words here before we get started on the necessity of understanding. I have found throughout not only my Hermes external experience, but with all my computing experience, that people have a terrible problem accepting that they don't understand something. The problem with this is that they have trouble just accepting that something works -- even though they don't know why. They will want to question and investigate why something is the way it is. Although educationally speaking this is a fine and even admirable character trait, it impedes the achievement of the main goal. Your goal here is not to understand the workings of your computer, your goal is to write externals. My point is, if something confuses you or your not quite 100% sure of something. . . don't worry about it! Just accept that there are going to be things you're not going to understand.

Once you've got THINK Pascal v4.0.1 or higher the next thing you need is a file from the Hermes Support

BBS called "HEI##." The ## refers to the current version of Hermes. As I'm writing this the current version is 2.2 so the file you need would be called "HEI22." In case you are wondering, that stands for **Hermes External Interface**.

Special Note about HEI22

I looked at the copy of HEI22 I got from the Hermes Support BBS. There is a problem. It appears in this version of HEI the documentation was left out. Therefore if this is your first copy of HEI you are getting, go with HEI211 instead of HEI22.

The file will most likely be compacted with Stuffit. In the archive there should be 1 folder. Inside that folder should be 3 items; 2 folders and 1 text file.

TEXT FILE:	HermHeaders.p
FOLDER:	Sysop Externals
FOLDER:	User Externals

You may ignore everything in the "Sysop Externals" folder for now. However, unstuff everything else in the archive. There are two files you need to print out: "HermHeaders.p" and "User Externals Docs." Don't try and save your printer or paper and just refer to these files on the screen; **print them out!** The HermHeaders.p file is a TEXT file but it is also a Pascal file. You can open it in a normal text editor to print it but the formatting won't be perfect. THINK Pascal has a print command, so you may be better off printing it from there. The "User Externals Docs" file is located in the "User Externals" folder. It is a Microsoft® Word file.

I suggest printing either the "HermHeaders.p" or the "User Externals Docs" on colored paper. There is no need to print both of them on color paper, normal white paper is fine for one of them. The way I've always done it is to print the "HermHeaders.p" on light blue paper and the "User Externals Docs" on plain white paper. The reason for this is simple. When you are programming you are going to be constantly referring to either of these documents. You will need to know at an instant which is which. The color-coding helps tremendously. Please notice that I did **not** say you should **read** either of these files, just print them out for future reference. Reading them at this stage will no doubt just confuse you.

Now you are ready to compile your first external. Did that shock you? Did you think you needed to know a lot more first? Well please note that I said **compile** your first external, not **write** your first external. The first external you will compile will be the infamous "Update Phone." You may have noticed a lot of stuff about Update Phone in the "User Externals" folder when you were unstuffing it. You probably saw a lot of weird looking files with .p's or .1's after their names. The file with a .1 after it's name is called a project. This is just something for Pascal and you really don't need to know anymore about it right now. If you have your copy of THINK Pascal currently running quit it before you begin.

Double-click on the file with a .1 after it's name, it should be titled "Update Phone#.1." Now several things might happen when you open this project for the first time. It will probably ask you a questions like, "Update Phone#.1:HardDrive:Some Folder:Blah Blah Blah: Etc. Needs to be built in a certain order, change it to that order?" Or you might get a questions like, "The current project needs to be uncompressed or overwritten, should I do it?" In either event, say YES.

Now, either the "UpdatePhone.p" file will open or it will say, "Cannot find 'UpdatePhone.p' would you like to look for it?" Yes, you want to look for it. I don't know where you unstuffed it to so I can't tell you where it is -- but you should be able to find it. Once you find it, it should open and you'll see a window with a text file that should look like greek to you. Don't worry! In fact don't even look at the text in the window, just look to the corners of the screen. That's right! You don't need to understand one word of Pascal in order to compile this external. The next step is to hit ⌘ B.

It will almost definitely tell you it can't find "DRVRRRuntime.lib." This file might be a little more difficult to find. It should have come with your copy of THINK Pascal. Look inside the folder called "Pascal

Folder." There should be a folder there called "Libraries." Inside that folder is where the file you want lies. If you still can't find it, hit cancel, and go back to the finder. Search for the file with System 7's speedy file search. (If you are not using System 7 you should be.)

After you've found that file guess what... you're gonna have to find **another** one! This time the file is "Interface.p." It should be located in your "Pascal Folder."

It may and it may not ask you to find the "HermHeaders.p" file, but if it does, no sweat -- you unstuffed it so you should be able to find it.

HermHeaders.p Problems?

Depending on what version of HEI## you have you may and may not encounter several problems with your copy of HermHeaders.p. It is unfortunate that Frank Price released several copies of HEI## with these "mistakes" in the HermHeaders.p file. Hopefully with Robert taking over, future versions of HEI## will be "mistake" free.

The reason I use quotes around the word mistakes is because only one of these problems is truly a mistake on Mr. Price's fault. But for your purposes you can just assume they are all mistakes. You must make several changes in your HermHeaders.p file for this external to compile. Use the find command in Pascal and search for "privates: Handle;". If you don't find this then you are in good shape! However if you do find this you must change the line to read "privates: myPrivsHand;". If you made a change hit ⌘ S.

The other "mistakes" you will correct later. When you are compiling the HermHeaders.p file if it says something like "DSPPBPTr is not declared," then simply change the offending undeclared thingy to a "longint." So the line "nodeDSPPBPTr: DSPPBPTr;" would then read, "nodeDSPPBPTr: longint;". If you make changes hit ⌘ S.

At this point if you successfully found all those files the external is compiled! Congratulations! Now it is time to take your eyes out of the corner and actually look at the window with that greek text. Here is what I want you to do. Just scroll down the window at a fairly speedy pace and become familiar enough with the format of Pascal so the mere sight doesn't scare you. As you are scrolling down be on the lookout for the words, "Your current phone number is:". When you find it the whole line should look like this:

```
Outline('Your current phone number is:', true, 0, procs[1]);
```

You might have already deduced that whatever is in the "s (single quote marks) gets printed to the screen. If you didn't deduce that don't worry... there will be other chances for you to show off your deducing skills. Now the question on your mind should be, "What is this 'Outline' thingy and how can it help me?" Well I'm glad you asked. This "Outline thingy" as you so elegantly put it, is called a **procedure**. What's a procedure you ask? Exactly what the name implies it is -- a procedure, a plan, a proceeding. So what's Outline's procedure? Well if you think about it, displaying words to the screen is only one fifth of the necessary things to do in a BBS environment. You also have to send it through the modem, do a line feed, set the color, and put it in the buffer. Luckily for you the procedure Outline does all of that for you! So guess what, those "other" four things Outline does -- you don't have to worry about them! Infact, if you didn't understand what I meant by "buffer" or "line feed" it's not a problem; you don't **need** to know.

So now you are probably wondering, "If I don't need to know these 'other' things, then what is all that other junk at the end of the Outline procedure?" Well that's another good question. Lets take it one part at a time. Here's the stuff you called 'junk'...

```
, true, 0, procs[1]);
```

First of all, the comma is just a field separator. The comma before the "true" just separates the words being displayed (by the way, words or a collection of characters are called "strings.") from the true or false field. (by the way, true or false fields are called "booleans.")

Notice the parentheses at the beginning and end of the OutLine procedure as well as any other procedure. Make sure all your procedures have a set.

Now, what does that true or false, or boolean, field control? Well if it is true, before the string is displayed a RETURN is hit. That is the string is displayed on the next line down. Here is an example of OutLines with the boolean fields set as true and false:

Example 1:

```
OutLine('Line Number One', TRUE, 0, procs[1]);
OutLine('Line Number Two', TRUE, 0, procs[1]);
```

Would display like this:

```
Line Number One
Line Number Two
```

Example 2:

```
OutLine('Line Number One', FALSE, 0, procs[1]);
OutLine('Line Number Two', FALSE, 0, procs[1]);
```

Would display like this:

```
Line Number OneLine Number Two
```

It takes awhile to get used to which boolean value to use and when, but you'll get the hang of it. Now, on to the other "junk"... What's that 0 mean after the boolean field? Well as usual we have the comma to separate fields. This time the comma is separating the boolean field from the integer field. (fields of numbers are either integers, bytes, reals, or longints) The zero in the integer field in this case controls the color in which the string is displayed. The numbers for the colors are the same as in Hermes. When you are posting a message, and you hit control-p#, what color is control-p0? Get it? Don't worry, if the user doesn't have ANSI it doesn't matter what number you put there. (Just make it 0 through 7 though okay?)

Now for the last piece of junk... What is that "procs[1]" thing? Well for starters "procs" is just short for "procedure." All this field wants to know is which Hermes procedure you are using. There are 17 Hermes procedures, OutLine is just one of them. How do you know which procedure is which? Ah! It is time to refer to what you printed out before... get out your "User External Docs."

Do **not** read any of it -- it'll just confuse you. Instead flip to page 5 (if you used normal paper) or until you see "Hermes Procedures." The first procedure mentioned is bCR. bCR just moves the cursor to the next line down, a line feed if you will. This procedure is built in to OutLine, but sometimes you need a line feed and don't need to OutLine any string. But I'm getting off the subject... The important thing is to notice where it says "{selector = 0}." That means when calling bCR you use "procs[0]." Can you guess what selector OutLine is? Turn to the next page. Well I hope you figured out it is selector 1. Every procedure has a selector from 1 to 17. That is how you know what to put in the procs[#] field at the end of every procedure.

Wait! There is one last piece of junk, the semi-colon. The semi-colon is used alot in Pascal -- it's like a period at the end of a sentence. You place a semi-colon at the end of almost every -line in Pascal, but

(and here's the kicker) not **every** line. It is very difficult to explain which lines get a semi-colon and which don't. If you've ever taken Spanish as a foreign language it's like knowing when to use Ser vs. Estar, or Por vs. Para. I couldn't possibly teach you at this stage when to use the semi-colon. However I can give you some helpful advice. Your Pascal program knows when to use a semi-colon even if you don't. So when in doubt, don't put one. Later when you try and compile your project your Pascal compiler will tell you where it wants a semi-colon.

Okay, so now you understand the entire OutLine procedure, as well as the general concept of any Hermes procedures. So here is what I want you to do: change that line.

```
OutLine('Your current phone number is:', true, 0, procs[1]);
```

That's boring! Spice it up a little! Change the color and change the string. (leave the boolean and procs fields as they are though) Here is what a creative line might look like:

```
OutLine('Yo! In case you are wondering, your phone # is:', true, 5, procs[1]);
```

There... that's a nice green, creative line isn't it? Now when you change your line don't copy my creative suggestion word for word -- that defeats the purpose. When you are done hit ⌘ S. It's a good idea to get into the habit of saving after every change. Hit ⌘ B again and see if it compiles. If it doesn't it means you have done something wrong. If it compiles successfully you are ready for the next big step; to actually create an external. You are actually going to create a file with the icon of an external, with the file type of an external, and it will even smell like an external. You will replace your Update Phone external with your creation and actually run the program.

First step is to open ResEdit. ResEdit v2.1 should have come with your copy of THINK Pascal, but if it didn't get yourself a copy of ResEdit. Open the external that was included in the HEI### archive in ResEdit. (It's called "Update Phone.") You should see two resource types. One is called "HRMS" and the other "XHRM." Ignore the XHRM. Select the HRMS and hit ⌘ C. Create a new ResFile by selecting new from the ResEdit file menu and paste the HRMS you copied there. (⌘ V) You should name the file "HRMS.rsrc" and save it in the same folder as where the file "Update Phone#." exists.

You can quit ResEdit now and go back to Pascal. In THINK Pascal select "Run Options..." from the "Run" menu. Near the top click on the "Use Resource File:" check box and select the "HRMS.rsrc." file you just created. Click on the "OK" button.

Now it's time to actually create your external. Make sure Hermes is not running and select "Build Code Resource" from the "Project" menu in THINK Pascal. Name it whatever you want but place it in your "Externals" folder located in your "Hermes Files" folder. ("Smart Link" should be checked.)

Launch Hermes, logon, and check out your very own external creation. Run it several times. Look at it. Marvel at what you created. Call over a friend and say "Look at that! I made that!" Run it several more times and enjoy the high.

Declaring Variables:

The first version of this issue I did not include this section. One of my editors told me he was confused whenever I mentioned something about declaring a variable. After re-reading the issue I saw that I did indeed refer to declaring variables but never really explained how. Like most things it sounds difficult, but is very simple.

There are two places where you can define variables in a Hermes external. (Actually there are more, but we won't worry about them now okay?) The two different places represent the two different types of variable declarations: Local Declarations and Private Declarations. The main differences between the two are where they are declared and how many nodes use them.

Local variables are declared at the beginning of your external, where it says "**Procedure** MAIN." There are two local variable declarations in the Update Phone external; "T1" and "i". Local variables are only used by 1 node at a time and therefore you only need 1 copy of each variable. However, with that advantage comes a severe limitation. You cannot use a local variable from one stage to another. For "stage-hopping" variables you need Private declaration.

Private variables are declared towards the end of the "HermHeaders.p" file. Where it says "myPrivs = **record**." There are two private variable declarations in the Update Phone external; "stage" and "ActiveOn". Private variables are used by every node at the same time and therefore you need 10 copies of each. The way you go about making 10 copies is with an "array." Now how do you actually declare these things? I mean what do you type? Follow these instructions and you can't go wrong.

```
VariableName: TypeOfVariable;
```

The stuff to the left of the colon is just the variable name. This can be anything you want, with the exception of key pascal words. (like "if" or "then") Beginning programmer (and some expert programmers as well) like to use long variable names. This helps them remember what variable does what and it makes the whole coding processes seem more like speaking english. I hate long named variables. For me they are just a pain to type out each time I need to use them. My variables are usually 1 or 2 letters long. Occasionally, when I'm feeling crazy, I'll make 'em longer. Anyway the choice of long or short names is totally up to you. Name 'em after old girl/boy friends, name 'em after your kids, name 'em whatever you want.

The stuff to the right of the colon does the actual declaring. The possible types of variables you can declare is limitless. However for your purposes there are only four main things you will declare variables as; string, integer, longint, or boolean.

```
VariableName: array[1..10] of TypeOfVariable;
```

When declaring an array, for private declarations, use the above format. When you are declaring multiple variables of the same kind it is not necessary to place each variable on a separate line. Instead use a comma like this:

```
VariableName1, VariableName2: TypeOfVariable;
```

That about covers variable declarations. If you are confused later on, refer back to this section.

Understanding Concepts:

It is now time to actually begin writing your own external. What will your external do? It will let the user input two numbers, add them together, and display the result. Sounds simple enough doesn't it? Unfortunately, in order for you to do this, you must actually understand some real concepts of Pascal. Don't get scared! Think of this as learning how to use a program, not learning how to program. The program you are learning how to use is THINK Pascal, and typing in certain phrases is how this program works. We are going to take this one step at a time and I'll be with you the whole way.

First step is to make a copy of your previous creation so we don't ruin your only copy. All that is necessary is to duplicate the "UpdatePhone.p" file. Do the same thing you did last time and double-click on the project. ("Update Phone#.1") This time all of the files should be found, but if they are not you know what to do.

Now I'm going to let you in on a little secret. Almost every Hermes external begins the exact same way. Over half of the adding external you are going to write has already been written for you. All that junk you scrolled by before when you were looking for the words, "Your current phone number is:", is standard for every external no matter what the external is going to do. This is why I always stress something when teaching new external writers how to write for Hermes. What I stress is that it is **unnecessary** to start

from scratch when beginning a new external. The name of the game is modification. You never write an entire external from beginning to end, but instead you just modify an existing external's code. This is what you are going to do now.

First step is to get rid of all the stuff in the code you are modifying that is specific to the Update Phone external. Go back to where you saw the words, "Your current phone number is:". (It might be different now because you changed the line, remember?) Take a look at all the stuff around that. You should see something like this:

```

case privates^^.stage[curNode^] of
  1:
    begin
      bCR(procs[0]);
      OutLine('Your current phone number is:', true, 0, procs[1]);
      OutLine(n[curNode^]^thisUser.phone, true, 0, procs[1]);
      bCR(procs[0]);
      bCR(procs[0]);
      YesNoQuestion('Do you wish to change it? ', false, procs[6]);
      privates^^.stage[curNode^] := 2;
    end;

```

This is where it all begins. All the stuff above that "1:" you'll never need to touch. (Well not never, but almost never; for your purposes just assume it's never)

You will also need the stuff at the very end. I'm talking about all those "ends" and "otherwise" things in bold.

Now, what do we want this external to do? Add! So, first we need to explain what the external will do to the user. How you go about this is up to you; you know how the OutLine procedure works. However, for those very uncreative types I'll give you a suggestion:

```

OutLine('Adding v1.0 By: Your Name', true, 0, procs[1]);
bCR(procs[0]);
OutLine('You enter two numbers and this', true, 0, procs[1]);
OutLine('program will add them together.', true, 0, procs[1]);

```

Now how do you position your OutLine's in the external? Well hold on! You're not ready for that yet. First you need to learn about something called a prompt. You use prompts all the time in Hermes but maybe you've never really noticed them. The type of prompt we'll be using for this external will be a "NumbersPrompt." It you read the description of a NumbersPrompt in the "User External Docs" (okay, you can read it now if you really want) you might be confused. Here is the format of a NumbersPrompt compliments of the "User External Docs."

procedure NumbersPrompt (prompt, accepted: STR255; high, low: integer; the Rout:

ProcPtr); {sel

Nothing confused me more than this. I didn't understand how to transform that general form into a real working NumbersPrompt. Here is the key. In the general form the fields separated by a comma are of the same kind. That is they are either all strings, boolean, integers, etc. The fields separated by a semi-colon are of different kinds. So look at the first two fields, they are separated by a comma therefore they must be of the same kind, and they are; strings. (STR255 just means a string with a maximum length of 255 characters.) So what do the first two fields do? Well the string you pass for "prompt" is the string that will be displayed. The string you pass for "accepted" are all the character you want a user to be able to enter -- **except** for numbers. Numbers come next with the "high" and "low" fields. As you might have guessed you place in those fields the highest and lowest numbers you want the user to be able to enter. If you don't want the user to be able to enter **any** numbers at all, make the high number lower than the low number. (Like -1 for high and 0 for low) "The Rout" is the same as with OutLine but #5 instead of #1. (procs[5]) So, now that you are thoroughly confused here is what your real working NumbersPrompt should look for this part of the external:


```
NumbersPrompt('Enter the first number: ', ", 1000, 0, procs[5]);
```

I just picked 1,000 as the high number arbitrarily. The maximum value you can pass for an integer is about 32,000. Okay, so how do you blend this NumbersPrompt and the OutLines you wrote before into the first stage of this external? Stage? Stage? Were you confused by that term? "What's a stage," you ask. Remember that "1:" I talked about awhile back? That marked the first stage. There are two times when you **must** end your current stage and start a new one when writing Hermes externals; after you prompt and after you display text. You don't need to worry about the latter because we won't be displaying text for awhile, but you will be prompting very shortly. So, after you prompt -- how do you end the stage and begin a new one? First step is to set a variable. In Pascal the way you set a variable is to let it := a certain value. (In case you are wondering := is pronounced "gets.") The variable you need to set looks ferocious, but it's really just a pussy cat once you get to know it.

```
Privates^^.stage[curNode^] := ?
```

Don't worry about what those ^^s mean. I've been at this for over a year and I only **think** I know what the deal with the up arrows is. I believe it's called "dereferencing a handle." The point is you don't need to understand them, just know when to use them. The best way to learn and understand is by example, so I'll shut up and just give you an example of how to put together your first stage.

```
1:
  begin
    bCR(procs[0]);
    OutLine('Adding v1.0 By: Your Name', true, 0, procs[1]);
    bCR(procs[0]);
    OutLine('You enter two numbers and this', true, 0, procs[1]);
    OutLine('program will add them together.', true, 0, procs[1]);
    bCR(procs[0]);
    bCR(procs[0]);
    NumbersPrompt('Enter the first number: ', ", 1000, 0, procs[5]);
    privates^^.stage[curNode^] := 2;
  end;
2:
  begin
  end;
```

Now, the next thing we need to work on is the next stage, stage number two. The first thing we'll do is to look at what the user entered. The string that holds whatever the user typed at the prompt is:

```
n[curNode^]^curPrompt
```

This variable, I find, is too long to work with. So we are simply going to let a shorter-named variable get that long-named one. Like this:

```
t1 := n[curNode^]^curPrompt;
```

Now the nice, short, cute little "t1" is exactly the same as the mean, long, ugly "n[curNode^]^curPrompt." Now that we can work with t1 let's take a look at t1 and see what the user entered. We can first make an assumption though. The assumption is simple; it is impossible for the user to have entered anything other than what the prompt allowed. If you refer back to your prompt you will notice that you allowed only numbers from 1 to 1000 and nothing else. So, the logical conclusion might be that t1 can only possibly contain numbers from 1 to 1000 right? Wrong. There is one thing a user can **always** enter at **any** Hermes prompt. No matter what you do to your prompt you cannot prevent the user from being able to enter this. What is this magical entry? Just a simple [Return]. So we must prepare for t1 being **either** a number from 1 to 1000 **or** a return.

Here's what you'll do, I won't talk you through this. Look read this example and see if you can follow

what's going on.

```
2:
  begin
    t1 := n[curNode^].curPrompt;
    if t1 = " then
      Privates^^.stage[curNode^] := 1
    else
      begin
        {We'll put more stuff here later}
      end;
    end;
  end;
```

In case you are confused " means they hit return. (That's two single quote marks, **not** one normal quote mark) So what did I just say up there? I said if the user hit a return instead of entering a number, then go back to stage one -- start all over again. However, if they entered **anything** other than a return then do something else. Get use to this checking for returns, you'll be doing this after every prompt for the rest of your life.

Now, we can't add the numbers yet because we don't know what the second number is. So before we prompt for the next number we must somehow make our program remember what the first number is. We will do this with what is called a "privates variable." The first step is to create this privates variable. Open the "HermHeaders.p" file in Pascal and scroll down to the very end. Start scrolling up until you see something like this:

```
myPrivsHand = ^myPrivsPtr;
myPrivsPtr = ^myPrivs;
myPrivs = record
  activeOn: array[1..10] of boolean;
  stage: array[1..10] of integer;
{put your variables here, one copy for each node}
end;
```

We need to add another integer so we can use it to store the first number. You may name your variable anything you want. My advice is to keep the name short, because although long-named variables are easier to remember, they are a pain to type out. I am going to call the variable "first" but remember you may call it anything you wish. To add this new variable simply place a comma after "stage" and type in the name. It should look something like this:

```
myPrivsHand = ^myPrivsPtr;
myPrivsPtr = ^myPrivs;
myPrivs = record
  activeOn: array[1..10] of boolean;
  stage, first: array[1..10] of integer;
{put your variables here, one copy for each node}
end;
```

After you've added it hit ⌘ S, and re-open your main program. Scroll back to stage number two and now we are ready to store the first number. The t1 string which holds whatever the user entered for the first number is just that -- a string. It is not a number (integer, longint, real, etc.) and therefore cannot be added, subtracted, multiplied, etc. In order to store it in our private variable we must first convert it to an integer. To do this we will use a little procedure called "StringToNum." The procedure "StringToNum" needs a longint to dump the numeric value of the string to. I like to call my longints l1, l2, l3, etc. but you can call yours anything you want. You defined them in the same place that t1 was defined. So, the entire second stage should look something like this:

```
2:
  begin
    t1 := n[curNode^].curPrompt;
```

```

if t1 = " then
    Privates^^.stage[curNode^] := 1
else
    begin
        StringToNum(t1, l1);
        Privates^^.first[curNode^] := l1;
        bCR(Procs[0]);
        NumbersPrompt('Enter the second number: ', ", 1000, 0, procs[5]);
        privates^^.stage[curNode^] := 3;
    end;
end;

```

The external is practically done! Stage three will look strikingly similar to stage two, in that it'll start off the same way. First step, as always, after a prompt is to check for a return. Now this time what shall we do with the losers who hit return? We have several options. We could send him back to the very beginning -- easy enough. We could exit the external and print, "Sorry, you didn't enter a number!" -- also easy. Or we could do what I would recommend. Send him back to the second number prompt and make him enter a number. Now this is not as simple as just setting Privates^^.stage[curNode^] to stage 2, because look what would happen. The n[curNode^].curPrompt is a return ("), so it'll just go from stage 3 to 2 back to 1! Therefore we must change the n[curNode^].curPrompt back to the first number, so when it goes back to stage 2 nothing will get messed up. There is no law that says n[curNode^].curPrompt must be set by a prompt, you can set it to whatever you want! So how about something like this:

```

3:
begin
    t1 := n[curNode^].curPrompt;
    if t1 = " then
        begin
            Privates^^.stage[curNode^] := 2;
            NumToString(Privates^^.first[curNode^], n[curNode^].curPrompt);
        end
    else
        begin
            {More to come...}
        end;
    end;
end;

```

Okay, that takes care of the losers who hit return, now what about those cool people who enter numbers? Well we know both of the numbers, so lets add! First we will need to convert t1 to a number again. After that we'll add the numbers, convert the answer to a string, and then OutLine the answer to the screen. Sounds simple enough eh?

```

3:
begin
    t1 := n[curNode^].curPrompt;
    if t1 = " then
        begin
            Privates^^.stage[curNode^] := 2;
            NumToString(Privates^^.first[curNode^], n[curNode^].curPrompt);
        end
    else
        begin
            StringToNum(t1, l1);
            NumToString(Privates^^.first[curNode^] + l1, t2);
            OutLine('The answer is: ', true, 3, procs[1]);
            OutLine(t2, false, 0, procs[1]);
            privates^^.activeOn[curNode^] := false;
            privates^^.stage[curNode^] := 0;
            n[curNode^].activeuserExternal := -1;
        end
    end;
end;

```

end;
end;

A couple things to note here. . . you must declare t2 for this to work. Just go up to where t1, and i are declared, put a comma after t1 and voilà. All that junk after the OutLines is just what you need to do to exit your external and return the user to the external menu.

Well in case you missed it. . . **you're finished!** All that is left to do is compile the external, build a copy, and launch Hermes. Now that you understand some basic concepts you can experiment with other kinds of externals. I suggest for your second external you make a program than can add or subtract two numbers. Make a menu with all the different options.

I hope you found this issue of WE4H useful. Remember if you have any questions I want to hear from you. Good luck!

Sam Shafritz