How To Use the INIT Shell with THINK Pascal

The INIT Shell consists of two THINK Pascal projects and associated source files. This note will describe the overall organization of the INIT Shell, describe the purpose of each file in the distribution, and illustrate how to assemble a fairly complex INIT using the shell.

The “loader” project is used to create the basic INIT file. As distributed, the code in this project generates a “shell” INIT which simply displays its icon. You will add code to the “DoInstall” procedure to execute various boot-time actions. Among these actions, you may wish to apply patch traps or install VBL tasks; the “resident” project (which will be described below) has been provided for this purpose. Any code which does not have to remain resident after INIT execution may reside entirely within the “DoInstall” procedure.

The “resident” project is used to create code which will stay in the system heap after completion of the INIT execution. Code which remains resident may be used for trap patches and VBL tasks. All resident code loaded by an INIT created using the shell may share a common set of global variables contained in a handle in the system heap. A comprehensive set of inline procedures and functions, together with a special code header for the IRES resource, is provided to allow you to patch traps using Pascal (even register traps). These special routines depend upon assumptions about the state of the runtime stack. As such, they must be used at the main program level of the patch code, and should never be called from within a subroutine of the patch code.

INITShellLoader.π You’ll use this to generate the base INIT file.
INITShellLoader.p The loader base code - you shouldn’t change this.
INITDoInstall.p The loader user code - you will change this.
INITShell.rsrc Additional resources for the the INIT file.

INITShellResident.π You’ll use a copy of this to generate each IRES resource.
INITShellResident.p The resident code - you will change this.
INITShellGlobals.p The globals for the resident code - you will change this.

The Loader Project

The loader project consists of the files INITShellLoader.π, INITShellLoader.p, INITDoInstall.p, and INITShell.rsrc. The loader code contained in INITShellLoader.p is entirely self contained; you will not need to modify it.

The loader code's main routine constructs a Quickdraw environment on the stack, since an INIT can't use A5 globals. This Quickdraw environment is used by the INIT to draw its icon, and may be used by any code you write to execute at load time.

The mouse button is tested before running your code. If the button is down, your code will not be executed, however, a distinctive icon will be displayed to confirm that loading was skipped. The method used to plot icons is compatible with the standard ShowInit routine written by Paul Snively, Darin Adler, and Paul Mercer, but is written in Pascal rather than assembly language, and does not plot color icons like the later versions of ShowInit. With the framework established here, color icon support should be simple to add; I omitted it because I didn't want to code something I couldn't test.

Your code will be entirely contained in the DoInstall procedure found in INITDoInstall.p. DoInstall is passed a single function parameter; this is a callback routine which provides access to various support routines found in INITShellLoader.p. Although this method adds a bit of complexity to the code, it provides two important benefits. First, it avoids problems with the circular dependencies which would otherwise arise between the INITShellLoader and INITDoInstall units; this will be a big help to anyone still using THINK Pascal 2.0, which doesn't have the {$Z±} directive. Second, it will allow you to compile the INITShellLoader unit as a library, since it does not depend upon anything you will ever change; this is a good example of the kind of issue which faces folks who write runtime libraries. Of course, if you changed this to a library, you'd have to change the name of the entry point and have your code resource, with its own "main", call the loader's entry point.

DoInstall must alway call the callback routine with the 'initCallback' function code before doing anything else. This gives the installer an opportunity to reserve space for its internal tables and for any permanently resident variables your INIT may need.

Upon return from your DoInstall procedure, the loader mainline will check a couple of flags which may have been altered by callback routines, and either display an icon which indicates that your INIT loaded successfully, or display one of two failure icons (load failed due to a runtime error, or load was not completed due to configuration) and undo any trap patches or VBL task installations that your DoInstall procedure may have performed prior to discovering an error.

The loader mainline is able to undo trap patch and VBL task installations by providing callback routines to not only perform the installations, but also to record enough information to be able to restore a trap to its pre-patch state or remove a VBL task. Two of the parameters provided to the callback routine with the 'initCallback' function code indicate how many patches and VBL tasks your INIT will install. Failure to declare as many patches or VBLs as you actually install will only result in an inability to automatically undo them in the event of an error; you may want to take advantage of this if you're doing something tricky where a simple restoration of the trap address or removal of the VBL task won't suffice.

The callback routine accepts a function code and three integer parameters. The function code specifies what action will be taken by the loader. The following table indicates how the parameters are used for each function code.

Function	Param1	Param2	Param3
initCallback	maxPatches	maxVBLs	globalsSize
setPatch	resID	trapWord	–
setVBL	resID	count	phase
failInstall	–	–	–
doNotInstall	–	–	–
suppressIcon	–	–	–

The setPatch and setVBL functions both expect the resource ID of an 'INIR' resource which contains the code to be installed as a patch or a VBL task, respectively. In the case of the setPatch function, the second argument is the actual trap word which would be used to invoke the trap (*not* the trap number!); the loader uses bit 11 of the trap word to distinguis between an OS trap and a toolbox trap. The setVBL function expects the count and phase of the VBL task as the second and third parameters.

If an error occurs while attempting to install a patch or a VBL task, the loader sets an internal failure flag and ignores all subsequent requests to install patches and VBLs. Later, when your DoInstall code returns, the loader notes the fact that an error had occurred and undoes the trap patches and VBL installations that completed successfully. This saves you from writing a lot of error checking and recovery code, and provides a uniform method for restoring much of the environment to its original state in the event that your INIT must fail to load. This scheme could be readily extended to support undoing of modifications to low-memory globals as well.

In the event that your DoInstall code may detect errors not directly related to patching traps or installing VBL tasks, the failInstall callback code may be used to invoke the loader's failure and recovery mechanism. This has the same effect on subsequent attempts to install patches and VBLs as a failure detected by the loader, and invokes the same recovery actions when your DoInstall procedure returns.

When installation fails, the default action of the loader is to display ICN# 130 just before finishing. In the event that you have a legitimate reason (such as an unsupported hardware configuration) for not running your INIT, the doNotInstall callback function may be used to signal this situation to the loader, which will display ICN# 131. If for some reason you don't want to display an icon under certain conditions, the suppressIcon callback function is provided.

While on the subject of icons, there are two other ICN#'s which may be displayed: 128 for a successful load and 129 for a skipped load. Loading is skipped by the standard mechanism of holding down the mouse button. The successful load icon is subject to the conditional suppression provided by the suppressIcon callback. The DisplayIcon procedure is written to simply bail out if the requested ICN# resource is not present, so you may simply omit any icon you never want to have displayed.

[resident code, globals, header & header patching at load time]