B The Template Layer

`B/rtpt`: *RTP.i6t* To issue run-time problem messages, and to perform some run-time type checking which may issue such messages.

`B/utilt`: *Utilities.i6t* Miscellaneous utility routines for some fundamental I6 needs.

`B/numt`: *Number.i6t* Support for parsing integers.

`B/timet`: *Time.i6t* Support for parsing and printing times of day.

`B/tabt`: *Tables.i6t* To read, write, search and allocate rows in the Table data structure.

`B/sortt`: *Sort.i6t* To sort arrays.

`B/relt`: *Relations.i6t* To manage run-time storage for relations between objects, and to find routes through relations and the map.

`B/figst`: *Figures.i6t* To display figures and play sound effects.

`B/inxt`: *IndexedText.i6t* Code to support the indexed text kind of value.

`B/regxt`: *RegExp.i6t* Code to match and replace on regular expressions against indexed text strings.

`B/chart`: *Char.i6t* To decide whether letters are upper or lower case, and convert between the two.

`B/unict`: *UnicodeData.i6t* To tabulate casings in the character set.

`B/stact`: *StoredAction.i6t* Code to support the stored action kind of value.

`B/listt`: *Lists.i6t* Code to support the list of... kind of value constructor.

`B/combt`: *Combinations.i6t* Code to support the combination kind of value constructor.

`B/relkt`: *RelationKind.i6t* Code to support the relation kind.

`B/blkvt`: *BlockValues.i6t* Routines for copying, comparing, creating and destroying block values, and for reading and writing them as if they were arrays.

`B/flext`: *Flex.i6t* To allocate flexible-sized blocks of memory as needed to hold arbitrary-length strings of text, stored actions or other block values.

`B/zmt`: *ZMachine.i6t* To provide routines handling low-level Z-machine facilities.

`B/glut`: *Glulx.i6t* To start up the Glk interface for the Glulx virtual machine, and provide Glulx-specific printing functions.

`B/iot`: *FileIO.i6t* Reading and writing external files, in the Glulx virtual machine only.

Introduction Template B/introt

Purpose

A short introduction to the template and its organisation.

§**1. The Software Stack.** The term "software stack" is sometimes used to refer to the total body of code running in a computer. As the word "stack" implies, there tends to be a fairly clear division between components, and an architecture in which one component is supported by another. The ground on which the stack rests is the hardware. Then come device drivers, for communicating with hardware, and a kernel of very low-level code to regulate who talks to the hardware and when. On top of that are usually several layers of operating-system facilities and utility programs, and on top of all of that are Firefox, iTunes and other consumer programs which the user has made a conscious decision to run. These top-level programs are visible, while the lower layers are concealed from the user and are generally very reliable, so it is easy to forget that they are there at all.

Similarly, when writing and testing a work with I7, it's easy to think that the only code running is the code directly generated by the source text. In fact, though, it runs inside a simulated computer called the virtual machine (or VM), and the VM has a software stack just as other computers do. Here the distinction between "program" and "operating system" is more blurred, because it can afford to be – the VM only ever has a single program running, and is not connected to any valuable hardware or data, so there is no need to protect the system's integrity from a malicious program, or to protect one program from the failings of another running at the same time. But there is still a recognisable software stack. From top to bottom, the VM at run-time contains:

(a) The rules, text and tables written by the author of this specific work of IF;
(b) Material contributed by Extensions which the author chose to borrow from;
(c) Rules, phrases and other constructions made by the Standard Rules extension, whose use is compulsory;
(d1) Infrastructure for rulebooks and code for maintaining a world model common to all works of IF;
(d2) The code needed to manage complex data structures such as relations, tables, indexed text, and so on;
(e) The interface to the VM, which is abstracted so that higher levels do not need to know whether Glulx or the Z-machine is being used.

Here (d1) and (d2) are independent blocks, so the picture is roughly so:

the player typing at a keyboard

(a) MATERIAL WRITTEN BY THE AUTHOR

(b) EXTENSIONS CHOSEN BY THE AUTHOR

(c) THE STANDARD RULES

(d1) MECHANICS OF PLAY (d2) DATA STRUCTURE SUPPORT

(e) VIRTUAL MACHINE INTERFACE

virtual machine "hardware"

NI is not like a compiler for a conventional program, because it has to conjure up the entire software stack whenever it compiles anything. (Until 2008, NI compiled code which ran on top of the traditional I6 library: nowadays it compiles stand-alone code which never uses the `#Include` directive. While it's true that the I6 compiler does add a very thin extra layer of code itself – the "veneer" – in all other respects NI specifies every single line of I6 code which makes up the eventual story file.)

NI compiles layers (a), (b) and (c) from the I7 source text found in the project file, in the Extensions installed by the user, and in the Standard Rules extension which comes pre-installed. But parts (d1), (d2) and (e) are almost exactly the same I6 code whatever may be sitting above them: they are simply copied, with minor variations, from a large body of standing code installed in the I7 application which is called the "template".

§**2. Segments and Paragraphs.** The template is divided into about 40 individual "segments", each stored in its own file and with the `.i6t` file extensions. This stands for "I6 template", because the code generated by NI is Inform 6 (I6) code. What makes the file a template rather than being raw code is that it is divided into named paragraphs which contain both commentary and also I6 code: when NI turns a template into the output code, the commentary (and each paragraph heading) is stripped out.

See the Inform documentation for how to modify the template in use for any particular project: I6 code can be added before, instead of or after any named segment or paragraph, and in addition, it's possible to replace entire segment files with your own versions stored in the Materials folder for a project.

§**3. Architecture.** To recap, then, the template contributes the following parts of the software stack:

(d1) MECHANICS OF PLAY (d2) DATA STRUCTURE SUPPORT
(e) VIRTUAL MACHINE INTERFACE

A more detailed version of this diagram organises its segments follows:

NI Control. `Main.i6t`, `Load-Core.i6t` (and several others like it), `Output.i6t`, `Definitions.i6t`.

Mechanics of Play I: Rules. `OrderOfPlay.i6t`, `Actions.i6t`, `Activities.i6t`, `Rulebooks.i6t`.

II: Infrastructure. `Parser.i6t`, `ListWriter.i6t`, `OutOfWorld.i6t`, `WorldModel.i6t`, `Light.i6t`, `Tests.i6t`.

III: Basic Services. `Language.i6t`, `MStack.i6t`, `Chronology.i6t`, `Printing.i6t`, `RTP.i6t`, `Utilities.i6t`.

Data Structure Support I: Basics. `Number.i6t`, `Time.i6t`, `Tables.i6t`, `Sort.i6t`, `Relations.i6t`, `Figures.i6t`.

II: Block Values. `IndexedText.i6t`, `RegExp.i6t`, `Char.i6t`, `UnicodeData.i6t`, `StoredAction.i6t`, `Lists.i6t`, `RelationKind.i6t`, `BlockValues.i6t`, `Flex.i6t`.

Virtual Machine Interface. `ZMachine.i6t` *or* `Glulx.i6t` and `FileIO.i6t`.

To elaborate:

NI Control. As well as containing commentary and I6 code, template files can also contain commands to tell NI to do something more interesting than simply copying over material verbatim into its output. Four of the segments use this ability a great deal: the remaining segments hardly use it at all. These four segments therefore don't fit anywhere in the diagram of the software stack above. "Main.i6t" controls the top-level logic of NI and gives the sequence of operations; "Load-.i6t" specifies the basic kinds of value known to NI – numbers, times, texts, rulebooks and so on; "Output.i6t" is essentially the arrangement used to join all the many fragments of I6 from the template and the I7 source text into a single end-to-end I6 program which will become the story file; "Definitions.i6t" defines many named constants which are used across the template.

Mechanics of Play I: Rules. "OrderOfPlay.i6t" is the highest-level description of what happens when a story file runs: it contains the I6 `Main` routine, and definitions of the primitive rules in the most important rulebooks.

"Actions.i6t" and "Activities.i6t" contain code which runs actions and activities, respectively: these sit on top of "Rulebooks.i6t", which performs general rulebook-running.

Mechanics of Play II: Infrastructure. "Parser.i6t" breaks down a command typed by the player into a slate of variables which can be formed into an action. "ListWriter.i6t" is a general-purpose service for printing lists of objects satisfying various descriptions, and formatted in different ways. "OutOfWorld.i6t" provides code to handle out-of-world actions not needing direct access to VM internals: PRONOUNS and SUPERBRIEF, for instance. "Tests.i6t" provides code for the testing commands – TEST, ACTIONS, SHOWME and so forth.

"WorldModel.i6t" contains code for performing object movements and the like in such a way that the world model rules are preserved: it handles component parts, decides touchability and so on. "Light.i6t" determines the level of light and decides on visibility.

Mechanics of Play III: Basic Services. "Language.i6t" provides English-language messages issued by template routines and the Standard Rules during play: replacing this segment is the way to translate I7 story files into other languages of play. (It's the equivalent of the old I6 "language definition files", and retains most of the same structure.) "MStack.i6t" provides a small general-purpose memory stack. "Chronology.i6t" performs the continuous monitoring required to ensure that past-tense conditions work: for instance we can only determine whether or not "the Black Door has been open" if we have spent the whole time so far checking on whether it is open or not, and this is where the checking is done. "Printing.i6t" handles paragraph-breaking, the printing of object names with suitable articles attached, and miscellaneous other printing needs. "RTP.i6t" issues run-time problem messages as needed: these should appear only if the story file does something clearly illegal, and point to bugs not yet removed from the author's code. "Utilities.i6t" provides miscellaneous utility functions at a very low level, such as for unsigned comparison of two numbers.

Data Structure Support I: Basics. Many kinds of value need no maintenance, and little or no support. "Number.i6t" and "Time.i6t" contain code to parse numbers and times of day from text typed by the player, together with a few basic operations: for instance, the rounding off of times of day. "Tables.i6t" provides access to table entries, specified in a variety of direct and indirect ways; this makes use of "Sort.i6t", which abstracts a choice of sorting algorithms for use on tables and other I6 data. "Relations.i6t" provides code for testing and asserting relations: the information about what is related to what else is stored in a variety of different ways, depending on the relation, to make best use of memory. "Figures.i6t" displays figures and plays back sound effects, or rather, passes instructions to do so down to the VM.

Data Structure Support II: Block Values. Kinds of value can be divided into the ordinary ones – number, object, time and so on – together with "block values" such as indexed text, stored action and list. Block values have to be stored in dynamically allocated memory from a heap. It would be wasteful to include all of this code, and to spare memory for the heap, if no block values were actually needed, so the following segments are only compiled if the source text makes specific reference to at least one block value. "IndexedText.i6t" manages character-indexed strings of text, which can shrink or stretch to arbitrary lengths: it makes use of "RegExp.i6t" for regular expression matching and search-and-replace; and also of "Char.i6t" for code to deal with lower and upper casing of letters, and "UnicodeData.i6t" to provide character set details, mechanically converted from the Unicode 4.0 standard. "StoredAction.i6t" manages stored actions: it can convert the current action into a stored one, and also try a long-stored action so that it now takes place. "Lists.i6t" manages the flexibly sized lists produced by values whose kind is list of numbers, list of texts, list of lists of lists of stored actions, and so forth: it can merge, insert, delete, resize, rotate, and reverse lists, and makes use once again of "Sort.i6t" (q.v.) to sort them.

"BlockValues.i6t" provides the basic support for kinds of value which are stored as blocks of memory on the heap.

At the lowest level, "Flex.i6t" manages flexible memory allocation as required by "BlockValues.i6t": it organises the heap of unclaimed memory, for instance, and maks allocations and deallocations when needed.

Virtual Machine Interface. Depending on the Settings used for the I7 project being compiled, we either use "ZMachine.i6t" or "Glulx.i6t": if Glulx, we also add "FileIO.i6t", which provides support for the limited file-handling abilities offered by Glulx.

Properly speaking, there is one further template file: "Introduction.i6t". It provides the commentary you are now reading, but has no other function, contains no code, and is finished now anyway.