



LDE(X) Core Manual

Bessy Release 2.2
Implemented for LDE(X) 6.4

Copyright 2006
LDE(X) Development Team

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License."

Table of Contents

GNU GENERAL PUBLIC LICENSE.....	4
GNU FREE DOCUMENTATION LICENSE.....	7
Foreword.....	11
Requirements.....	12
Hardware.....	12
Software.....	12
Resources.....	13
LDE(X) Related.....	13
LiteStep Related.....	13
Shell Replacement Related.....	13
Welcome.....	14
A Little Bit Of History.....	14
Why LiteStep engines and also WScript engines?.....	14
The Core.....	15
Cache.....	16
fish.....	19
Why would I use it?.....	19
How does it work?.....	19
A History Lesson.....	20
How to Swim & not Sink — The fish Way.....	22
fish Operation — The LiteStep-native engine.....	22
fish Operation — The WSH Engine.....	25
sCore/fish Developer Manual.....	25
Calling the Script.....	26
Target List Directives and Comments.....	27
File Extension Includes and Excludes.....	27
Wildcards and Regular Expressions.....	28
Shell Parameters.....	29
Additional Usage Scenarios.....	30
sCore/fish and LDE(X).....	30
gHost Technical Documentation.....	32
Autohide.....	32
CatsCradle.....	32
String Length.....	33
String Flipping.....	33
Character Stripping.....	34
Power Calculation.....	34
Character to Number Conversion.....	35
Character and String Case Changes.....	36
BaseN to Decimal Conversion.....	37
Converse.....	39
Converse Operation — The LiteStep-native Engine, ckDialog Mode.....	40
Converse Operation — The LiteStep-native Engine, LSBox Mode.....	40
Converse Operation — The WSH Engine.....	41

Debug.....	43
Fog.....	44
Loop.....	45
Navigator.....	47
Navigator Operation — The LiteStep-native Engine.....	47
Navigator Operation — The WSH Engine.....	49
Profile.....	51
Basic principle.....	51
Interface-level implementation.....	52
Setting Name Array Definition.....	52
Setting Value Array Definition.....	52
The Service Code.....	53
Progress.....	57
Progress Operation — The LiteStep-native Engine.....	57
Progress Operation — The WSH Engine.....	59
Spring.....	59
Therapy.....	60
Therapy Operation – The LiteStep-native Engine.....	60
Therapy Operation — The WSH Engine.....	62
Type.....	63
Uncle.....	64
List.....	70
Metta.....	74
Stack.....	76
Utility Functions.....	77
Applaunch.....	77
Argscount.....	77
Locationcheck.....	78
LiteStep Modifications for LDE(X).....	80
Includefolder.....	80
lsapi\settingsfileparser.cpp.....	80
litestep\buildoptions.h.....	81
About Box.....	82
Lsapi\aboutbox.cpp.....	82
Litestep\litestep.rc.....	82
Evar to Announce Includefolder Availability.....	82
Lsapi/settingsmanager.cpp.....	82
Escape Codes.....	83
Lsapi/settingsmanager.cpp.....	83
FAQs.....	84
Core.....	85
Known Issues.....	87
Core.....	87

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991
Copyright (C) 1989, 1991 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software — to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it. For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

- a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of

any change.

- b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.
- c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

- 3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:
 - a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
 - b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,
 - c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

- 4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License.

However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

- 5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.
- 6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.
- 7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as

to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

GNU FREE DOCUMENTATION LICENSE

Version 1.2, November 2002
Copyright (C) 2000,2001,2002 Free Software Foundation, Inc.
51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- a) Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- b) List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- c) State on the Title page the name of the publisher of the Modified Version, as the publisher.
- d) Preserve all the copyright notices of the Document.
- e) Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- f) Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- g) Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- h) Include an unaltered copy of this License.

- i) Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- j) Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- k) For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- l) Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- m) Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- n) Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- o) Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

6. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

7. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

8. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the

Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

9. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

10. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

11. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

Foreword

This document contains all available technical, in-depth, documentation for the Core systems, both native LifeStep versions and the powerful WSH-based ports. Core is an evolving system and the developers will attempt to keep this document in step with changes, but we would appreciate any comments from readers.

In general, any Core systems need to have full documentation before they can be included in a release. In some cases, though, we accept that full documentation may be provided shortly after a formal release. SDK documentation is expected, though.

Requirements

NOTE : Additional requirements may be imposed by plugins, icons, schemes or 3rd party interfaces.

Hardware

300 MHz Pentium II processor (faster is recommended for transparency effects)

32 MB of free RAM (16 MB of free RAM minimum, more needed for additional plugins)

Software

Windows 2000 Service Pack 1 or later (Service Pack 4 recommended) or Windows XP (Service Pack 2 recommended)

Internet Explorer 5.5 or later (5.0 will work, but with limitations)

Windows Scripting Host support (WScript 5.6 or better is recommended, although WScript 5.1 will work with some limitations)

Resources

LDE(X) Related

Discussion forum : ldex.terica.net/board.htm
Sourceforge.net : www.sourceforge.net/projects/ldexforwindows/
Main Site : ldex.terica.net

LiteStep Related

LiteStep.net : www.litestep.net
LSThemes : www.ls-themes.org
LiteStep.com : www.litestep.com
LiteStep Universe: www.ls-universe.info
LS Mailing List : wuzzle.org/list/litestep.php
LSI : beyondconvention.net/ohussain/l/installer/
Dist.ru: www.dist.lite-step.ru
Open Theme Standard: o-t-s.sourceforge.net

Shell Replacement Related

Loose-Screws : www.loose-screws.com
Blizzle.com : www.blizzle.com
Lost In the Box : www.lostinthebox.com
Shell-Shocked: www.shell-shocked.org

Welcome

Core is the underpinning of the LDE(X) system: it delivers an abstracted set of code interfaces that are both used within Core and available for use by interfaces and plugins, as documented in the SDK. The use of abstracted interfaces and the SDK approach means that Core development is not constrained by deep hooks from interfaces — as long as the SDK remains compatible, the abstracted interfaces can be extended and evolved.

The use of a service request and dependency tracking system allows developers to optimize their work by loading only the minimum of Core services necessary.

Please note that this document is not really intended for interface-level developers; it is a technical reference for those interested in developing additional Core systems or revising/porting the existing work.

A Little Bit Of History

LDE(X)'s Core was formally established with LDE(X)6.0. Before that release the scripting systems were embedded in the LDE(X) interface system. A design decision for the 6.0 release was to fully separate the Core and interface code so that the system was more extensible, and the later releases have worked to maintain this principle.

Why LiteStep engines and also WScript engines?

The WSH engines were originally designed to allow Core to be used with non-LiteStep shells, but they have proven useful under the LiteStep shell as well. Both systems coexist and are user-selectable; whenever possible, the two systems have been kept syntax compatible to allow a hybrid setup to be utilized.

Although the support for other shells is still a work in progress, the WSH engines do provide some features not available through the native LiteStep engines, such the setting of registry values.

The Core

Cache

The cache service provides a runtime batch-processing facility under LiteStep. It differs from the profile service, examined later, in that the profile service is designed to carry out predefined series of actions; it has no built-in support for adding to, removing from or modifying the profile in question. Cache implements more flexible batch-processing features but this comes at the cost of decreased readability and less efficient handling of quoted arguments and paths.

The service exposes a basic set of functions: the minimum required to create, maintain and execute batch actions.

Cache creation is done dynamically through the use of the `!addtocache` script:

```
*Script bang !addtocache
*Script gotoif ["%{args:2}" <> ""] namedcache
*Script exec !VarSet cachename "defaultcache"
*Script goto updatecache
*Script label namedcache
*Script exec !VarSet cachename %{args:1}
*Script exec !VarSet namedcache "1"
*Script label updatecache
*Script exec !if ["%{namedcache}" = "1"] [!VarSet value %{args:2}]
    [!VarSet value %{args}]
*Script exec !listadd "%{cachename}" "%{value}"
*Script exec !VarRemove [cachename][namedcache][value]
*Script ~bang
```

As can be seen from the source, this script is essentially a wrapper around the list service's `!listadd` function, but it also provides for the definition of a default cache variable, `%{defaultcache}`. It may seem redundant to use the list script for this purpose — after all, the cache could also be defined using `!varset` — but the employment of this script provides a safety check not available with direct definition; it will only add entries that do not already exist.

Specific caches can be created or targeted by passing the name of the containing `mzvar` as the second argument to the script.

Entries can be removed from a cache using the related script, `!removefromcache`:

```
*Script bang !removefromcache
*Script gotoif ["%{args:2}" <> ""] namedcache
*Script exec !VarSet cachename "defaultcache"
*Script goto updatecache
*Script label namedcache
*Script exec !VarSet cachename %{args:1}
*Script exec !VarSet namedcache "1"
*Script label updatecache
*Script exec !if ["%{namedcache}" = "1"] [!VarSet value %{args:2}]
    [!VarSet value %{args}]
*Script exec !listrem "%{cachename}" "%{value}" "0"
*Script exec !VarRemove [cachename][namedcache][value][cacheindex]
*Script ~bang
```

The one thing to note about this script is that it passes zero as the third argument to the list service's `!listrem` function. This means that all matching entries will be removed — a useful feature for making sure that matching entries

are removed from directly-defined arrays.

The !optimisecache script has a similar value when dealing with directly-defined arrays; it will remove all duplicates from a named cache, if given the name as an argument, or the default cache in the case of being called alone. As with the other functions, it's basically a wrapper around a list service function (!listremdup in this case):

```
*Script bang !optimisecache
*Script gotoif ["${args:1}" <> ""] namedcache
*Script exec !VarSet cachename "defaultcache"
*Script goto updatecache
*Script label namedcache
*Script exec !VarSet cachename ${args:1}
*Script exec !VarSet namedcache "1"
*Script label updatecache
*Script exec !listremdup "${cachename}"
*Script exec !VarRemove [cachename][namedcache]
*Script ~bang
```

Finally, the entries in the cache are executed using the !runcache script, which loops through the list and executes the value of each element.

```
*Script bang !runcache
*Script exec !if [${args:1} = ""] [!VarSet cachename "defaultcache"]
[!VarSet cachename {args:1}]
*Script exec !VarSet runcache_loopcounter "0"
*Script exec !VarSet runcache_totalloopcount "%${cachename}:count"
*Script exec !VarSet runcache_suppresscheck
"%{runcache_totalloopcount}"
*Script exec !VarAdd runcache_suppresscheck -1
*Script label runcache_loop
*Script exec !VarAdd runcache_loopcounter 1
*Script exec !if ["${runcache_loopcounter}" =
"%{runcache_suppresscheck}"] [!VarRemove SCORE_SUPPRESS_ALL]
[!VarSet SCORE_SUPPRESS_ALL "suppress"]
*Script exec "%${cachename}:${runcache_loopcounter}"
*Script gotoif ["${runcache_loopcounter}" =
"%{runcache_totalloopcount}"] exit
*Script goto runcache_loop
*Script label wshpause
*Script gotoif ["${SCORE_FISH_CALLS}" = "0"] wshsafe
*Script gotoif ["${SCORE_THERAPY_CALLS}" = "0"] wshsafe
*Script exec !pause 200
*Script goto wshpause
*Script label wshsafe
*Script label exit
*Script exec !flushcache
*Script ~bang
```

Sharp eyes will notice three things about this code. The first is that it borrows a feature from the profile service; it sets the %SCORE_SUPPRESS_ALL mzvar to "suppress" until the end of execution and removes it for the last entry. The details of this variable are covered later in the sections on fish and gHost, but the essential explanation is that the presence of this variable prevents the WSH ports of the various LiteStep-native services from showing a recycle prompt after they finish their work. This is the default behavior for sCore/fish, for example.

The second thing is the wshpause/wshsafe loop. Due to the overhead involved

with the WSH scripts, sCore/fish and therapy use call counters to track how many copies of the scripts are active at any given time and adjust their execution accordingly in order to prevent overloads. Although this is generally effective, services such as cache and profile can place higher demands on these services — so an additional check of these variables is done for safety, especially when used on slower systems.

The final element of note is that !runcache automatically clears the cache after the execution is finished. This is done through the !flushcache script, shown here:

```
*Script bang !flushcache
*Script exec !if [%{args:1} = "" ] [!VarRemove "defaultcache"]
                [!VarRemove {args:1}]
*Script ~bang
```

fish

fish is arguably the most powerful and useful backend ever developed under LiteStep and inherits many facilities from the old LSI engine. The gHost collection, which encompasses the majority of other Core services, runs second only to fish in terms of complexity.

fish 0.2 was a major update that brought in Windows Scripting Host code to sit alongside the reliable, LS native fish engine. The mode in which the fish engine should operate is controlled by a variable (`$CORE_FISH_MODE$`) check within Core — if this variable is not found, the native LS version of the engine is used (the values are noted in the LDE(X) SDK for developer reference). The implementations on the interface side are compatible so as to avoid any issues arising from differing syntaxes; old interfaces can use the WSH or LS-native versions of fish simply by altering the `CORE_FISH_MODE` value as detailed later in this document.

Why would I use it?

- fish is portable & nearly totally independent of whatever code you want it to change.
- fish is small and easy to deploy.
- fish has an built-in debug system that can be enabled to dump any required information to disk. Simply use the markers as discussed later, run the debug enable function (`!debugon`) and away you go.

How does it work?

As of the time of writing, the LiteStep-native version of fish has moved to 1.0.1 from the original 0.0.1 system. This follows a significant increase in the power of the fish system (0.1.1d) and new to 0.1.2 threading support.

fish uses markers to process strings. To illustrate this, we will use the aforementioned “`!debugon`” function to show what can be done. This discussion will use the new 0.1.x protocols (Note that these are subject to change in later revisions).

The marker is user defined; in this case we have chosen to use

```
POPICONS
```

In the world of LiteStep, we can only insert inert text as comments, so the marker is present as

```
;POPICONS
```

wherever it is needed, e.g.

```
*Popup "Enable Popup Icons" !iconson ;POPICONS
;POPICONS *Popup "Disable Popup Icons" !iconsoff
```

Using this code above, it is fairly obvious that the first line will result in a popup menu entry. The second line has no effect due to the presence of a comment flag: LiteStep doesn't care what lives behind any ";" characters — it simply stops processing the line at that character.

fish can use this to its advantage: we can use the text behind the ";" to target specific lines of text for manipulation. This is nothing unique to fish, however, as the LSI engine in LDE-X | LSI R3.0 was the first production system to use this approach. Accordingly, before we go on to discuss the new fish engine, it is perhaps beneficial to present a little history to show how this new system is superior to the old LSI in almost every way. Of course, you can skip ahead if you so wish.

A History Lesson

The initial Series X release of LDE, LDE-X R1.0, was a complete ground-up rewrite of the existing system. One of the major aims in LDE-X was to make configuration and use far simpler than had been the case in the nine preceding versions. Limitations of the various scripting and file editing solutions (then change.dll and textedit.dll 1.41) forced the backend engine to match entire strings of code and replace them with full lines of code. change.dll could not handle E-Vars and TextEdit, prior to version 2.0, was unable to perform parametric (regexp) searches. As such, the reliability of the backend engine was severely limited - this was also true for LDE-X R2.0.

With the development of LDE-X Release 3, it was decided that a completely new approach, one leading to code independence in the backend engine, was required. TextEdit 2.0 provided the key facility to enable this — parametric searches. To enable the backend, now fully independent, to work and configure the system code correctly, the use of inert comment-based marker flags was implemented. The Lovingly Scripted Interface (LSI) was at that time comprised of two independent elements: the frontend code (.box files, .rc files, etc.) and then the backend engine that would configure both itself and the frontend code.

The LSI was built around two fundamental components: mzScript and TextEdit 2.0+.

The LSI used a simple construct in its early days that looked something like:

```
*Script bang !favouritesoff
*Script exec !textreplace "step.rc" "(.*)" ;FAVON$" ";FAVON \1"
*Script exec !textreplace "step.rc" "^;FAVOFF (.*)" "\1 ;FAVOFF"
*Script ~bang
```

(The use of '\$' by textedit was dropped for 2.4.12. For the use of "\$" and "^", please check the TextEdit 2.x documentation. Please note that this is the version that LDE(X) currently uses.)

When the new rc parser was added to LiteStep 0.24.6 (post 17th January 2001) this system encountered a number of problems. The parser now would throw away the “;” and anything else at load time. As such, the !favouritesoff example above became essentially:

```
*Script bang !favouritesoff
*Script exec !textreplace “step.rc” “(.*)” “”
*Script exec !textreplace “step.rc” “”
*Script ~bang
```

The release of the source code for TextEdit 2.0 allowed this to be addressed with the addition of an escape code (see the module documentation for more information), resulting in the release of TextEdit 2.1I. The LSI function then looked like:

```
*Script bang !favouritesoff
*Script exec !textreplace “step.rc” “(.*) \~FAVON$” “\~FAVON \1”
*Script exec !textreplace “step.rc” “^\~FAVOFF (.*)” “\1 \~FAVOFF”
*Script ~bang
```

This essentially formed the last version of the LSI and was provided in Release 3.71.

The limitations of the LSI became evident as the system grew from the initial Release 3.0 to the Release 3.5 code base. Code duplication was massive and the engine was heavy, slow and crude compared to other potential implementations. It had, however, served reliably and capably for the lifetime of LDE-X | LSI Release 3.x series (some 1/3 of the total life of LDE).

With the advent of 2001, and after 18 months of development, the necessity of a re-write became evident. Much of the code had become unnecessary and could be stripped out; there was plenty of room for improvement, and Release 3.71 was solid enough to last the period until the new “alfred” system could be completed.

The configuration and LDE parser (not LiteStep's parser!) engine was the first of the hurdles to be tackled. The LSI was too cluttered to be of a great deal of help, so a ground-up rewrite of the parser routines began.

What had been absent from the LSI was a generic processing routine: all functions, such as !favouritesoff, made the changes themselves. This meant that a considerable period of time was wasted in ensuring that, following feature additions requiring LSI processing, the relevant functions handled the new code.

For example, if a new file was to be processed in relation to the favourites AEB being switched off, it would be necessary to ensure that !favouritesoff processed it through:

```
*Script bang !favouritesoff
*Script exec !textreplace “step.rc” “(.*) \~FAVON$” “\~FAVON \1”
*Script exec !textreplace “step.rc” “^\~FAVOFF (.*)” “\1 \~FAVOFF”
*Script exec !textreplace “new.rc” “(.*) \~FAVON$” “\~FAVON \1”
*Script exec !textreplace “new.rc” “^\~FAVOFF (.*)” “\1 \~FAVOFF”
*Script ~bang
```

This would also require that a matching change be made to the !favouriteson function to reverse the changes. This leads to almost complete duplication — the only thing different being the search/replace strings (the two quoted strings at the end of each line above).

Additionally, vulnerability to syntax errors creeps in when a single, generic parser is not employed.

These limitations were addressed through the development of fish.

How to Swim & not Sink — The fish Way

fish is built around a generic parser structure. The parser is called (since 0.1.4) in one of two ways: the recommended (new) method is illustrated below and the older, still supported method afterward.

```
*Script bang !favouritesoff
*Script exec !mzparser_off "FAV"
*Script ~bang
```

The older method is :

```
*Script bang !favouritesoff
*Script exec !VarSet fishvar1 "FAV"
*Script exec !mzparser_off
*Script ~bang
```

In the first case, by passing "FAV" as an argument, mzparser_off now automatically defines fishvar1. In the latter case, since fishvar1 has already been declared previous to calling the parser function, there is no need to provide an argument and the parser does not attempt to redefine the fishvar1 variable. The advantage of the new method is that it is simpler to use and more flexible in operation.

There are two fish engines available for use. Only one is available at any time for the current user and this is set universally for all interfaces of that user under LDE(X) — that is, all interfaces available to a specific user can be set to use the LS native engine or all can use the WSH engine — but other users can have their own, different, preference of engine, and the each user can change this setting at any time.

fish Operation — The LiteStep-native engine

The best way to understand the way that fish operates is to follow through a process flow associated with one action. The action itself is unimportant - the sequence of events that result are the significant part of this discussion.

fish 0.1.5 moved to textedit 2.4.13 — the use of the \$ as an end of line (EOL) marker in the search string was a potential issue for future versions of LiteStep. The module was changed to use _ instead. The code samples below are written

with this in mind.

fish 0.1.6 added support for post-fish command execution using a variable (`%{postfishcommand}`).

To start, let us consider the function associated with turning popup icons off:

```
*Script bang !IconsOff
*Script exec !mzparser_off "POPICON"
*Script ~bang
```

A couple of things are worth noting before we get into the discussion:

- All the parsers utilize the gHost suite's progress bar implementation to provide feedback as the changes are made (assuming that `$gHostHome$` is defined and working).
- A script called `!updatehost` is supported in fish to perform actions directly after the on and off fish actions have been completed. Potential uses are to prompt for a restart or some other action. If this script is not defined, there is no effect on the operation of fish or the host environment.

To simplify the code, both on and off actions utilize a generic match script (`!mzparser_match`), shown here:

```
*Script bang !mzparser_match
*Script exec !varset parameter %{args:1}
*Script exec !varset type %{args:2}
*Script gotoif ["%{type}" = "end"] end
*Script gotoif ["%{type}" <> "start"] exit
*Script exec !VarSet fishvar2 "\~%{parameter} (.*)"
*Script exec !VarSet fishvar3 "\1 \~%{parameter}"
*Script goto construct
*Script label end
*Script exec !VarSet fishvar2 "(.*) \~%{parameter}"
*Script exec !VarSet fishvar3 "\~%{parameter} \1"
*Script label construct
*Script exec !VarSet fishvar4 "@%{fishvar2}_@ @%{fishvar3}@"
*Script label exit
*Script exec !VarRemove parameter
*Script exec !VarRemove type
*Script ~bang
```

In the `!mzparser_match` code, fish generates the (mutated) string (e.g. `@(.*) ;POPICON#@ @;POPICONON \1@`) and then calls `!mzparser_do` which is where the action takes place, as governed by the defined content of `fishvar4`.

```
*Script bang !mzparser_do
*Script exec !fisherrorcheck
*Script exec !If ["%{fishOK}" = "1"] [!do_scripts %{fishvar4}]
*Script exec !fisherrorcheck
*Script exec !If ["%{fishOK}" = "1"] [!do_boxes]
*Script exec !fisherrorcheck
*Script exec !If ["%{fishOK}" = "1"] [!do_misc]
*Script exec !fisherrorcheck
*Script exec !If ["%{fishOK}" = "1"] [!do_userfiles]
If DEBUG
*Script exec !debugfunction "]log" "$Core$\Debug\fish-debug.txt"
    "mzparser_do completed."
EndIf
```

```
*Script label exit
*Script ~bang
```

The call to !fisherrorcheck is a safety measure to avoid problems:

```
*Script bang !fisherrorcheck
*Script exec !VarRemove fishOK
*Script gotoif ["${fishvar1}" <> ""] check2
*Script exec !debugfunction alert "fish failed check - fishvar1 is
    ${fishvar1}."
*Script exec !debugfunction "log" "$Core$\Debug\fish-debug.txt"
    "fish failed check - fishvar1 is ${fishvar1}"
*Script goto error
*Script label check2
*Script gotoif ["${fishvar2}" <> "(.*)"] check3
*Script exec !debugfunction alert "fish failed check - fishvar2 is
    ${fishvar2}"
*Script exec !debugfunction "log" "$Core$\Debug\fish-debug.txt"
    "fish failed check - fishvar2 is ${fishvar2}"
*Script goto error
*Script label check3
*Script gotoif ["${fishvar3}" <> ""] allOK
*Script exec !debugfunction alert "fish failed check - fishvar3 is
    ${fishvar3}"
*Script exec !debugfunction "log" "$Core$\Debug\fish-debug.txt"
    "fish failed check - fishvar3 is ${fishvar3}"
*Script goto error
*Script label allOK
*Script exec !VarSet fishOK 1
*Script goto exit
*Script label error
*Script exec !VarRemove fishOK
*Script label exit
*Script ~bang
```

Each !do_ action is very similar, so for this discussion to remain concise we will only mention of the many scripts of this type:

```
*Script bang !do_userfiles
*Script exec !textreplace @$userprofile$applications-rc@ ${fishvar4}
*Script exec !textreplace @$userprofile$desktop-rc@ ${fishvar4}
*Script exec !textreplace @$userprofile$user-labels-rc@ ${fishvar4}
*Script exec !textreplace @$userprofile$user-popup-rc@ ${fishvar4}
*Script exec !textreplace @$userprofile$user-wallpaper-rc@
    ${fishvar4}
*Script bang
```

In each line, the syntax becomes !textreplace @file@ @;POPICONOFF (.*)@ @\1 ;POPICONOFF@ (remembering the POPICON example from earlier).

The !mzparser_cleanup routine is called to remove all the variables from memory once they are redundant.

The sharp eyes out there will notice that fish supports the dumping of variable definitions to disk via the !mzparser-debug_dumpvars function that is activated when debug mode is active:

```
If DEBUG
*Script bang !mzparser-debug_dumpvars
*Script exec !debugfunction "log" "$Core$\Debug\fish-debug.txt"
    "Listing variables in parser...."
```

```

*Script exec !debugfunction "log" "$Core$\Debug\fish-debug.txt"
    "fishvar1: %{fishvar1}"
*Script exec !debugfunction "log" "$Core$\Debug\fish-debug.txt"
    "fishvar2: %{fishvar2}"
*Script exec !debugfunction "log" "$Core$\Debug\fish-debug.txt"
    "fishvar3: %{fishvar3}"
*Script exec !debugfunction "log" "$Core$\Debug\fish-debug.txt"
    "fishvar4: %{fishvar2} %{fishvar3}"
*Script exec !debugfunction "log" "$Core$\Debug\fish-debug.txt"
    "Listing completed."
*Script ~bang
EndIf

```

fish Operation — The WSH Engine

This was contributed by Scott Kearn (Amtal), who joined the development team during the development of LDE(X) 6.1 and headed the production of the WSH scripts from that time onward. It performs exactly the same overall operations as the LS-native engine, but has improved potential for error handling but adds some additional notification features and flexibility in the files that it targets for processing. This comes at the cost of higher resource usage, however; this tradeoff should be considered when choosing the engine to employ.

The WSH engine has been designed with the capability to adjust XML and registry settings as well as the text files that contain LiteStep settings, although not all of these features are implemented at present. The interesting feature for LDE(X) in all this is that the WSH fish allows a single directory to be specified within the target file index and all files will be recursively processed by the fish system. This greatly simplifies the assignment of target files when compared to the native LiteStep engine, but there are dangers in relation to binary files and so some degree of care should be exercised — of particular note, the file extension filters mentioned shortly should be employed when possible.

sCore/fish Developer Manual

The WSH port of fish aims to be highly flexible and adaptable; essentially all aspects of its operation can be controlled through the use of mzvars, most notably SCORE_FISH_DEBUG, SCORE_FISH_PROGRESS and SCORE_FISH_SUPPRESS. The first controls the writing of debug output, while the second and third control the displays of progress bars and notification dialogs, respectively.

Two files should be considered when examining the source code: fishconfig.rc (this provides the integration between LiteStep and the WSH script) and fish.wsf (the script itself).

Calling the Script

The WSH port of fish is implemented in the fish.wsf script, which is called in the following way:

```
fish.wsf //job:language targetstring on|off targetlistfiles  
[progress] [debug] [suppress] [appname]
```

The first four parameters are required; the following three are assumed to be false if omitted, and the last argument can be used to override the automatic shell detection that the script attempts to do. "0" and "false" are recognized as false boolean values for the first three optional arguments; "noprogess," "nodebug" and "nosuppress" may also be specified for the corresponding arguments. Anything else will be interpreted as true. This variation in values is designed to both increase the readability of the configuration lines that call the fish script and provide a means of overriding global settings. Under LDE(X), these default settings are specified by E-Vars that may be specified directly or matching mzvars that are specified at runtime.

The //job:language argument specifies the language version of the fish script to use. As of the time of this writing the only available version is coded in JScript and is specified using "//:job:fish_js".

By default, fish shows an information popup after it has finished its file processing, notifying the user that a recycle or restart of the shell may be required for the changes to take effect. The suppress argument may be used to suppress this notification. The same effect can be achieved by setting the SCORE_FISH_SUPPRESS E-Var or mzvar to anything other than "0" or "false."

Progress bars and debug settings are controlled through similar means; the variable names to use are SCORE_FISH_PROGRESS and SCORE_FISH_DEBUG, respectively.

The last argument — appname, usually referenced in the LDE(X) WSH integration code through the %apptarget} variable — can be used to override the automatic shell detection feature discussed below. This feature is common to most of the WSH services.

The "targetlistfiles" argument is used to tell the script which files to process. This is not done on the command line due to the potential length of the file list, but instead is done by pointing the script to a file containing directives specifying which files to process: the target list of files. The format of this list is described in the following sections.

Note that as with the LiteStep-native version of fish, the WSH port will automatically recycle LDE(X) if !updatehost is defined and the nouupdate mzvar is absent, but this will only occur if the SCORE_FISH_SUPPRESS variable is set to a true value as well. If suppression is not enabled, or if it is being used under a different shell or in a different environment, fish's default behavior is to simply notify the user that this may need to be done — it will not do it itself. This decision was made because of the different ways that the changes may affect different shells,

and to leave the option to the developer or themer as to whether or not a recycle should be automated (e.g. by including the recycle directive in the calling !bang). It is highly recommended that any automated recycles involving fish also involve a delay, since it is otherwise possible for a recycle to occur before all of the files have been updated.

Target List Directives and Comments

In addition to the list of files and directories to process, the target list files parsed by fish can contain a mixture of embedded directives and comments. These are prefixed by the hash character ("#").

To include a file or directory of target list files, similar to the inclusion of a C source file, the target list file can use a line similar to the following:

```
#include %userprofile%\Desktop\foo.list
```

This will cause fish to read in `foo.list` and parse it for additional targets. Note that includes are recursively processed, but fish does not do any checking to prevent infinite recursion loops. Make sure to take the appropriate precautions when using this directive.

End-of-line comments are not supported, but full line comments may be included as follows:

```
# I am a comment.
```

Note the space between the hash sign and the comment text.

Processing directives themselves are specified in standard path notation and can contain environment variable references, which will be expanded automatically. For example:

```
C:\Temp\mylog.log  
%appdata%\directory  
%UserDataLocation%\user-ui.rc
```

File Extension Includes and Excludes

sCore/fish can be configured to include or exclude specific file extensions during folder processing. (Note that these are exclusive; either include parameters or exclude parameters should be used, but not both.) This is configured by placing a `parse-rules.cfg` file in the targeted directory root, and the rules will be applied to all subfolders during recursive processing. The file should have the following format:

```
exclude  
gif  
bmp  
zip
```

The first line must be either "include" or "exclude" as a bare string. Each

following line should be a file extension to be excluded or included. Note that these rules are specific to each directory. (In other words, if these directories are listed as directories containing target list files:

```
%userprofile%\Desktop\foo  
%userprofile%\Desktop\bar
```

And the foo directory contains a parse-rules.cfg file, its rules will only be applied to its files and subdirectories. They will not be applied to the bar directory.)

If no parse-rules.cfg file is present, fish will parse all files during directory processing.

Wildcards and Regular Expressions

fish does not support DOS-based wildcards, but it does support regular expressions. In addition to being influenced by concerns over portability, the decision to support one but not the other resulted from concerns over a bug in the way that different Windows operating system versions deal with wildcards and long file names. In many cases the matches are carried out based only on the first three characters of the file name extension; unfortunately, this is generally the case in Windows 2000 and XP, and may lead to unpredictable results.

In order to trigger this kind of processing, a regular expression pattern must be specified at the end of a path using the "regexp:" prefix. This requirement is necessary due to the fact that a number of regular expression metacharacters are legal file name characters as well, and some patterns could be valid as either file names or regexps. There would otherwise be no way of reliably determining which format was the intended one.

Since some developers may not be familiar with regular expression syntax, the following list will demonstrate how to translate common DOS wildcard formats into regular expression patterns. More detailed information on regexps may be found in the Windows Script 5.6 documentation, available from MSDN's scripting pages.

Multiple character matches:

DOS	foo*.nfo
Regular Expression	foo.*\nfo

Single character matches:

DOS	foo?.nfo
Regular Expression	foo.\nfo

In regular expression syntax, the period (".") matches any character except a newline. When used by itself it is equivalent to the DOS "?" wildcard. However,

since it is a metacharacter (a character with a special meaning) it needs to be escaped when it is to be matched literally. This is done by using the backslash character ("\"). The asterisk ("*") metacharacter has the same meaning in regular expressions as it does in DOS: it matches zero or more characters. To understand how this works, consider the following fish command:

```
fish.wsf foo off %userprofile%\Desktop\regexp:foo.\.rc
```

fish will recognize that the third argument is a regular expression pattern due to its "regexp:" prefix. When it interprets this, it will match any target list files on the current user's desktop with names that match the following pattern: "foo", followed by one character, followed by ".rc". This means that it will match "foo1.rc", "foo\$.rc" and "foo-.rc", but not "foobar.rc" or "foo11.rc". However, if this command were given instead:

```
fish.wsf foo off %userprofile%\Desktop\regexp:foo.*\.rc
```

All files on the user's desktop with names starting with "foo" and ending with ".rc" will be matched.

Directives of this type are used for Core file processing in LDE(X), among other jobs, as well as for specifying which target list files are to be parsed. The core_wshfishtargets.txt file is a good example of how it can be employed:

```
%LDERootDataLocation%\global-fileprefs.rc
%LDERootDataLocation%\global-coreprefs.rc
%LDERootDataLocation%\global-uiprefs.rc
%LDERootDataLocation%\user-ui.rc
%LDERootDataLocation%\user-plugins-global.rc
%LDERootDataLocation%\Plugins\regexp:.\.rc
%LDERootDataLocation%\Plugins\regexp:.\.box
%LDECoreLocation%\dev\dev.rc
```

These directives instruct fish to process specific files, along with all .rc and box files in the user's root Plugins directory (and subdirectories).

Regular expression patterns may be used anywhere that regular paths may be used, including include directives. However, there is currently no support for including or excluding subdirectories based on regular expressions.

Shell Parameters

The utility script used by fish sets variables specifying whether the current shell uses text, xml or registry configurations and what variation of these it uses. In order to cooperate with fish each shell needs to have a configuration file located in the JSLib\AppCfg directory, named with a .cfg extension (not required but recommended for consistency) and with its first line structured as follows:

```
shell executable name:configuration type|comment character code
```

A blank line may follow this, followed by whatever notes the shell developer or contributor may feel are necessary.

The shell executable name must be the name of the shell program as it appears

in the process list; the configuration type must be one of "text", "xml" or "registry" and the comment character code must be the hexadecimal code for the comment character used in the shell's configuration files, e.g. 3B for ";". (For registry configs this should be the character that should be appended to a key or value name to set it as disabled. The hyphen ("-", 2D) is suggested as a standard.) This comment character code should be omitted for xml configurations; if it is present, it will be ignored.

fish, like the other WSH services, will check to see which of the configured applications are running and, out of these, which has the lowest PID; the matching configuration file will then be used. Specifying the name of the shell executable as the script's appname argument will force the matching configuration file to be loaded and used instead.

(This is particularly useful in situations where an explorer window might be kept open while litestep.exe is killed and restarted. The WSH-related code in LDE(X) uses this approach to make sure that the scripts load LiteStep configurations.)

Additional Usage Scenarios

Although the original application of fish suggested that it was useful, primarily, for toggling single options, users and developers should note that it may be applied to multiple lines simultaneously. This allows for blocks of code to be toggled. This is primarily of interest under shells such as Blackbox where fish can be used, for example, to enable or disable certain sections of a user's menu.

In this respect fish may be considered and employed as a simple macro processor when dealing with text files. Similar macro processing functionality may be produced when dealing with registry or xml configurations by chaining several fish directives together (through batch files, WSH scripts or shell-supported macro/script commands).

sCore/fish and LDE(X)

Beginning with LDE(X) 6.1.b2.3700, this version of fish was integrated into Core through the LiteStep-specific code in fish*.rc (and later fishconfig.rc). This was done through alternate definitions of the !mzparser_on and !mzparser_off bangs.

LDE(X) developers should continue to use these bangs in their code so that it will be portable across both fish implementations. This means that direct command-line customization (such as setting the progress, debug and suppress options) is not available in the usual manner. It is available, however, through variable settings. These will have no effect on the LiteStep-specific version if present and will be replaced by defaults if not.

- Progress: as a default this setting will inherit the value of CORE_GHOST, if specified, or CORE_GHOST_PROGRESS, if it is specified and CORE_GHOST is not.

Otherwise it will have the default value of "noprogess."

The progress setting can be overridden by giving an appropriate value to the SCORE_FISH_PROGRESS variable before calling the appropriate bang. For example:

```
*Script exec !VarSet SCORE_FISH_PROGRESS "progress"  
*Script exec !mzparser_on "foo"
```

Note that the SCORE_FISH_PROGRESS variable will be removed after the bang has finished. This applies in all of the cases mentioned here.

- Debug: as a default this setting will inherit the value of the DEBUG variable, if present; its default value is "nodebug."

While developers are strongly encouraged to rely on this default behavior, there may be cases in which enabling or enabling debug support for particular actions may be useful. In these cases the override value may be set using the SCORE_FISH_DEBUG variable:

```
*Script exec !VarSet SCORE_FISH_DEBUG "nodebug"
```

Again, it is important to note that this temporary value will be cleared after bang execution.

- Suppress: developers are strongly encouraged to override the suppress setting only in cases where a scripted recycle will be involved, and in those cases the override value should be set to true (any value other than "0" or "false").

Due to the way that the !mzparser_on and !mzparser_off commands are implemented, fish.wsf is called multiple times. The default values for the suppress argument are true for every call to fish other than the last, and false for the last call. Only the value of the last call's suppress argument may be overridden. This is done using the SCORE_FISH_SUPPRESS variable:

```
*Script exec !VarSet SCORE_FISH_SUPPRESS "suppress"
```

Suppression may also be done by setting an appropriate value for the SCORE_SUPPRESS_ALL variable: this will affect all sCore-based services at once.

gHost Technical Documentation

Autohide

Everything lives in `$ghosthome$\services\autohide.rc`. The script is quite complex, but usage is as simple as providing the `!HandleUIElement` script with two (optionally, three) arguments.

The script supports several modes, as set by the first argument, which may be either “autohide” or “autoraize.” The second argument gives the name of the LSBox to be handled — it can be any box (child, parent or standalone). The general format is this:

```
*Script exec !HandleUIElement "autohide" "PanelNameToAutoHide"
```

or

```
*Script exec !HandleUIElement "autoraize" "PanelNameToAutoRaise"
```

The third argument is optional and is dependent on the mode in which the function is being used.

```
*Script exec !HandleUIElement "autohide" "PanelNameToAutoHide"  
"OpacityToFadePanelFrom"
```

or

```
*Script exec !HandleUIElement "autoraize" "PanelNameToAutoRaise"  
"ZorderOfLoweredPanel"
```

CatsCradle

The CatsCradle service was originally designed to provide a baseN-to-decimal conversion feature for the Core. In the process of developing this service it became evident that others would be required and, as a result, CatsCradle is also able to reverse string contents, calculate their length and carry out exponential operations on decimal input. Since the baseN conversion feature builds on the base provided by these smaller services, they will be covered first.

A couple of notes are in order before proceeding to an examination of the script source. The first of these is in regard to the variable names that CatsCradle uses internally; the names are lengthy and may look intimidating, but this naming scheme was necessary in order to avoid name collisions between the various scripts during their employment by the baseN converter (mzScript has no variable scope. Essentially, all mzvars are global.)

Secondly, the scripts make use of the `%{variable:sep}` property introduced with mzScript 1.0. When this is set to "" it is possible to walk a variable character by character; essentially, it becomes an array of one-character elements. The individual indexes can be accessed using a counter variable, much like using

pointer arithmetic in C.

Note that CatsCradle currently cannot handle values containing spaces. This is due to limitations in mzScript's argument handling routines. Developers will need to code around this limitation until it is addressed in a future module revision.

String Length

```
*Script bang !stringlength
*Script gotoif [%{args:2} = ""] exit
*Script exec !Varset lengthinputstring %{args:1}
*Script exec !Varset lengthreturnvariable %{args:2}
*Script exec !Varset lengthinputstring:sep ""
*Script exec !Varset lengthwalkcount 0
*Script label walkstringloop
*Script exec !VarAdd lengthwalkcount 1
*Script gotoif ["%{lengthinputstring:%{lengthwalkcount}}" <> ""]
    walkstringloop
*Script exec !VarAdd lengthwalkcount -1
*Script exec !Varset %{lengthreturnvariable} "%{lengthwalkcount}"
*Script exec !VarRemove [lengthwalkcount][lengthreturnvariable]
    [lengthinputstring]
*Script ~bang
```

This script takes two arguments: the string value, which can be either numeric or text, and the name of the variable in which the length of the string should be stored. (Note that only the value of the second argument is checked; this makes it possible to use `!stringlength` to check for empty strings as well as get the count of from defined variables.) Assuming that both arguments are provided, the script sets the separator character to an empty string and checks each array element to see if it actually holds a value. If so, the count is incremented and the script proceeds to the next element; if not, the count is decremented to strip off the out-of-bounds index and the resulting total is stored in the specified return variable.

String Flipping

As should be expected, `!flipstring` allows developers to reverse the character order in a string. This can be useful in handling numeric input and even for those awkward modules that expect BGR color inputs rather than RGB (the former is a legacy color implementation used in early LiteStep revisions).

```
*Script bang !flipstring
*Script exec !Varset flipinputstring %{args:1}
*Script gotoif ["%{flipinputstring}" = ""] exit
*Script exec !Varset flipreturnvariable %{args:2}
*Script gotoif ["%{flipreturnvariable}" = ""] exit
*Script exec !stringlength "%{flipinputstring}"
    "flipcharacterposition"
*Script exec !Varset fliploopcount "1"
*Script exec !Varset stringtoflip "%{flipinputstring}"
*Script exec !Varset stringtoflip:sep ""
```

```

*Script exec !VarSet flipinputstring:sep ""
*Script label flipstring
*Script exec !VarSet flipinputstring:%{fliploopcount}
    "%{stringtoflip:%{flipcharacterposition}}%"
*Script exec !VarAdd flipcharacterposition -1
*Script exec !VarAdd fliploopcount 1
*Script gotoif ["%{flipcharacterposition}" <> "0"] flipstring
*Script exec !VarSet %{flipreturnvariable} "%{flipinputstring}"
*Script label exit
*Script exec !VarRemove [flipcharacterposition][fliploopcount]
    [stringtoflip][flipreturnvariable][flipinputstring]
*Script ~bang

```

Note that this script calls the !stringlength function so that it knows how long the input string is. This allows the string to be walked backward into a holding variable that is then assigned to the return variable at the end of the process.

Character Stripping

The !stripcharacter script allows for the removal of a specified character from either the beginning or end of a string; this can be useful when extracting a variable name applying to x and y coordinates, for example, since those variable names usually consist of a common name with the single character “x” or “y” appended.

The script takes a string value as its first argument, the name of a return variable and the character to strip as its second and third, and an optional “start” or “end” argument as its last. By default the character is removed from the end of the string if found.

```

*Script bang !stripcharacter
*Script exec !quotedargscount "myargscout" %{args}'
*Script gotoif [%{myargscout} < "3"] exit
*Script exec !VarSet searchchar %{args:3}
*Script exec !VarSet tempstring %{args:1}
*Script exec !VarSet %{tempstring}:sep ""
*Script exec !if [%{args:4} <> "start"] [!VarSet stripmode "end"]
    [!VarSet stripmode "start"]
*Script exec !if ["%{stripmode}" = "end"]
    [!if [%{tempstring:%{%{tempstring}:count}} = %{searchchar}]
    [!VarRemove %{tempstring}:%{%{tempstring}:count}]]
    [!if [%{%{tempstring}:1} = %{searchchar}]
    [!VarRemove %{tempstring}:1]]
*Script exec !VarSet %{args:2} %{%{tempstring}}
*Script exec !VarRemove [tempstring][searchchar][stripmode]
*Script label exit
*Script exec !VarRemove myargscout
*Script ~bang

```

Power Calculation

mzScript currently does not provide any sophisticated math functions — including power operations. However, it is possible to use its existing features to

code up such a function. CatsCradle's answer to this problem is the following script, !compute_power. Consistent with the previous two functions, this one expects three arguments: the input value (note that this is not sanity checked in the current implementation; passing a character string to the function is not recommended), the power factor to apply and the output variable in which the calculated result is stored.

```
*Script bang !compute_power
*Script exec !Varset powerinputvalue %{args:1}
*Script gotoif ["%{powerinputvalue}" = ""] exit
*Script exec !Varset powervalue %{args:2}
*Script gotoif ["%{powervalue}" = ""] exit
*Script exec !Varset powerreturnvariable %{args:3}
*Script gotoif ["%{powerreturnvariable}" = ""] exit
*Script exec !Varset powercomputedvalue 1
*Script gotoif ["%{powervalue}" = "0"] return
*Script exec !VarSet powerloopcount 1
*Script label powerloop
*Script exec !VarMul powercomputedvalue %{powerinputvalue}
*script exec !VarAdd powerloopcount 1
*Script gotoif ["%{powerloopcount}" <= "%{powervalue}"] powerloop
*Script label return
*Script exec !VarSet %{powerreturnvariable} %{powercomputedvalue}
*Script label exit
*Script exec !VarRemove [powerloopcount][powerinputvalue][powervalue]
[powerreturnvariable][powercomputedvalue]
*Script ~bang
```

A simple loop is performed to carry out the multiplication to the desired power level and, at the end, the calculated result is assigned to the return variable.

Character to Number Conversion

Very simple little routine to map the position of the input character in the alphabet (US/GB) to the return variable. It also supports an offset to the return value — useful for getting the corresponding ASCII code, for example.

```
*Script bang !chartonumber
*Script exec !VarSet char %{args:1}
*Script gotoif ["%{char}" = ""] exit
*Script exec !Varset chartonumberoffset %{args:2}
*Script exec !Varset chartonumberreturnvariable %{args:3}
*Script gotoif ["%{chartonumberreturnvariable}" = ""] exit
*Script exec !changecharcase "upper" "char"
*Script exec !Varset chartonumberalpha "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
*Script exec !Varset chartonumberalpha:sep ""
*Script exec !Varset chartonumbercount 1
*Script label chartonumberloop
*Script gotoif ["%{char}" =
"%{chartonumberalpha:%{chartonumbercount}}"]
chartonumberloopdone
*Script exec !Varadd chartonumbercount 1
*Script gotoif ["%{chartonumbercount}" >
"%{chartonumberalpha:count}"] chartonumberloopdone
*Script goto chartonumberloop
*Script label chartonumberloopdone
*Script exec !If ["%{chartonumbercount}" >
```

```

    "{chartonumberalpha:count}"
    [!Varset %{chartonumberreturnvariable} "{char}"]
    [!Varset %{chartonumberreturnvariable} "{chartonumbercount}"]
*Script exec !If [{"%{chartonumberreturnvariable}" <> "{char}"]
    [!VarAdd %{chartonumberreturnvariable} %{chartonumberoffset}]
*Script label exit
*Script exec !VarRemove [chartonumberreturnvariable][char]
    [chartonumberoffset][chartonumbercount][chartonumberalpha]
*Script ~bang

```

Note that this function is case-insensitive; it calls the !changecharcase script, discussed below, before doing string comparisons. (Technically this is not necessary, as string comparisons in mzScript 1.0 are also case-insensitive — but this should allow the script to remain usable given any changes in this regard.) Also note that if the character is not in the A-Z range the original value will be assigned to the return variable.

Character and String Case Changes

Case changes in CatsCradle are done using two functions: !changecharcase and !changestringcase. The latter calls the former, so it's presented here first:

```

*Script bang !changecharcase ; call with two arguments : upper/lower
variable_name_holding_char_to_be_converted
*Script gotoif [%{args:2} = ""] exit
*Script exec !VarSet ctl_mode %{args:1}
*Script exec !VarSet ctl_char %{args:2} ; call with name of variable
holding char to be converted
*Script gotoif [{"%{ctl_char}" = ""] exit
*Script exec !if [{"ctl_mode}" = "upper"] [!VarSet changecasecase
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"]
*Script exec !if [{"ctl_mode}" = "lower"] [!VarSet changecasecase
"abcdefghijklmnopqrstuvwxyz"]
*Script gotoif [{"changecasecase}" = ""] exit
*Script exec !VarSet changecasecase:sep ""
*Script exec !VarSet changecasecounter 1
*Script exec !VarSet changecaseref "%{ctl_char}"
*Script label lower
*Script exec !if [{"%{ctl_char}" =
"%{changecasecase:%{changecasecounter}}"] [!VarSet %{ctl_char}
"%{changecasecase:%{changecasecounter}}"]
*Script gotoif [{"%{ctl_char}" <> "%{changecaseref}"] exit
*Script exec !VarAdd changecasecounter 1
*Script gotoif [{"changecasecounter}" > "%{changecasecase:count}"]
exit
*Script goto lower
*script label exit
*Script exec !VarRemove [ctl_mode][ctl_char][changecasecase]
*Script exec !VarRemove [changecasecounter][changecaseref]
*Script ~bang

```

Basically, this is a simple array-to-array mapping. This is a change from the previous revision of the script in which the characters were hardcoded; the current revision is more efficient and allows for additional characters to be added to the sets as and when needed.

Unlike most of the CatsCradle functions, !changecharcase operates directly on

the variable passed to it by name in its second argument; it does not use a return variable. The first argument is the name of the case to which the character should be changed: either “lower” or “upper.”

!changestringcase uses the same argument format and simply walks through the string, calling !changecharcase for each character:

```
*Script bang !changestringcase
*Script gotoif [%{args:2} = ""] exit
*Script exec !VarSet ctl_string_mode %{args:1}
*Script exec !VarSet return_stringname %{args:2}
*Script gotoif ["%{%{return_stringname}}" = ""] exit
*Script exec !VarSet ctl_string {%{return_stringname}}
*Script exec !VarSet ctl_string:sep ""
*Script exec !VarSet ctl_string_length "{ctl_string:count}"
*Script exec !VarSet ctl_string_position "0"
*Script label string_convert_loop
*Script exec !VarAdd ctl_string_position 1
*Script exec !changecharcase "{ctl_string_mode}"
    "{ctl_string:%{ctl_string_position}"
*Script gotoif ["%{ctl_string_position}" = "{ctl_string_length}"]
    string_convert_loop_exit
*Script goto string_convert_loop
*Script label string_convert_loop_exit
*Script exec !VarSet %{return_stringname} "{ctl_string}"
*Script exec !VarRemove [ctl_string_mode][return_stringname]
    [ctl_string][ctl_string_length][ctl_string_position]
*Script label exit
*Script ~bang
```

BaseN to Decimal Conversion

This was coded up principally to allow the comdlgLS color picker to be used and, where necessary, convert its hexadecimal return values for colors to decimal format. There are various other uses, naturally.

```
*Script bang !convertfrombasen
*Script gotoif [%{args:1} = ""] error1
*Script gotoif [%{args:2} = ""] error1
*Script gotoif [%{args:3} = ""] error1
*Script exec !VarSet basen %{args:1}
*Script exec !VarSet convfrombasen_sourcevalue %{args:2}
*Script exec !VarSet convfrombasen_destinationvariable %{args:3}
*Script goto main
*Script label error1
*Script exec !msgbox "CatsCradle encountered a problem before it got
    started!"
*Script exec !msgbox 'args unquoted were %=%{args}%= '
*Script goto exit
*Script label main
*Script exec !stringlength "{convfrombasen_sourcevalue}"
    "convfrombasen_sourcevalue_length"
*Script exec !flipstring "{convfrombasen_sourcevalue}"
    "convfrombasen_sourcevalue"
*Script exec !VarSet convfrombasen_loopcount 1
*Script exec !VarSet convfrombasen_convertedvalue 0
*Script exec !VarSet convfrombasen_convtempvalue 0
*Script label conversionloop
```

```

*Script exec !VarSet convfrombaseNpowerfactor
    "%{convfrombaseNloopcount}"
*Script exec !VarAdd convfrombaseNpowerfactor -1
*Script exec !if ["%{convfrombaseNloopcount}" > "1"]
    [!compute_power "%{baseN}" "%{convfrombaseNpowerfactor}"
    "convfrombaseNmultiplyfactor"]
    [!VarSet convfrombaseNmultiplyfactor "1"]
*Script exec !VarSet convfrombaseNsourcevalue:sep ""
*Script label basecheck
*Script exec !VarSet convfrombaseNcharactervalue
    "%{convfrombaseNsourcevalue:%{convfrombaseNloopcount}}"
*Script gotoif ["%{baseN}" <= "9"] multiplybyfactor
*Script exec !VarSet convfrombaseNtestforchar
    "%{convfrombaseNcharactervalue}"
*Script exec !VarMul convfrombaseNtestforchar 1
*Script gotoif ["%{convfrombaseNtestforchar}" =
    "%{convfrombaseNcharactervalue}"] multiplybyfactor
*Script exec !chartonumber "%{convfrombaseNcharactervalue}" "9"
    "convfrombaseNcharactervalue"
*Script label multiplybyfactor
*Script exec !VarRemove convfrombaseNtestforchar
*Script exec !VarSet convfrombaseNconvtempvalue
    "%{convfrombaseNmultiplyfactor}"
*Script exec !VarMul convfrombaseNconvtempvalue
    "%{convfrombaseNcharactervalue}"
*Script exec !VarAdd convfrombaseNconvertedvalue
    "%{convfrombaseNconvtempvalue}"
*Script exec !VarAdd convfrombaseNloopcount 1
*Script gotoif ["%{convfrombaseNloopcount}" <=
    "%{convfrombaseNsourcevalue:length}"] conversionloop
*Script exec !VarRemove [convfrombaseNloopcount]

[convfrombaseNmultiplyfactor][convfrombaseNcharactervalue]*Script
exec !VarSet %{convfrombaseNdestinationvariable}
    %{convfrombaseNconvertedvalue}
*Script label exit
*Script exec !VarRemove [baseN][convfrombaseNsourcevalue]
    [convfrombaseNsourcevalue:length][convfrombaseNconvertedvalue]
    [convfrombaseNpowerfactor]
*Script ~bang

```

Despite the length of this function, its operation is actually quite straightforward — with readability added by the calls to the other functions provided by CatsCradle.

Although the `!chartonumber` script will safely process non-letter characters, this script uses the `mzScript` default behavior when it is asked to perform a mathematic operation on a character or string – it takes the value of the string or character to be zero — to avoid unnecessary calls to this function. In order to make use of this behavior the script duplicate the main variable as a test variable and multiplies it by one; text would result in a value of 0 for the test, as would a zero numeric value, but numeric values in the test variable retain their original values after the multiplication. Comparing the test and main variable values after this step allows the nature of the value to be determined; if the values are not equal, the current value in the main variable is a character and the script proceeds to call `!chartonumber`.

Converse

There are three modes for this component, two of them utilizing LiteStep modules and one of them utilizing WSH. Of the native versions, one uses LSBox and the other uses ckDialog. The creation/destruction scripts are trivial for both, to be honest. The major complexities arise with the dialog code itself, particularly for ckDialog.

The purpose of the Converse service is to allow for user confirmation of actions that might break or slow down the system — requesting the unload of a module required by Core, for example. The base Converse system also provides scripts for handling various aspects of LSBox dialog displays, primarily in regard to system performance demands, and for notifying the user that manual configuration might be required. These scripts (dialogeffect, dialogdisplay, dialogfxoff and configurationrequired) are located in converse.rc.

The handling of user choices is done by the following scripts when using one of the native methods:

```
*Script bang !doit
If DEBUG
*Script exec !debugfunction "log" "$Core$\Debug\converse-debug.txt"
    "doit called."
EndIf
*Script exec !VarSet SCORE_SUPPRESS_ALL "1"
*Script exec !If [{"updatehost"} = ""] [!VarSet nouupdate "1"]
*Script exec !%{actionname}
*Script exec !VarRemove SCORE_SUPPRESS_ALL
*Script exec !VarRemove nouupdate
*Script exec !If [{"dialogmode"} = "ckdialog"] [!closedialog]
*Script exec !If [{"dialogmode"} = "lsbox"] [!LSBoxDestroy
    %{dialogname}]
*Script exec !VarRemove [dialogmode][dialogname][actionname]
If DEBUG
*Script exec !debugfunction "log" "$Core$\Debug\converse-debug.txt"
    "doit completed."
EndIf
*Script ~bang

*Script bang !abortit
If DEBUG
*Script exec !debugfunction "log" "$Core$\Debug\converse-debug.txt"
    "abortit called."
EndIf
*Script exec !VarRemove actionname
*Script exec !If [{"dialogmode"} = "ckdialog"] [!closedialog]
*Script exec !If [{"dialogmode"} = "lsbox"] [!LSBoxDestroy
    %{dialogname}]
*Script exec !VarRemove dialogname
If DEBUG
*Script exec !debugfunction "log" "$Core$\Debug\converse-debug.txt"
    "abortit completed."
EndIf
*Script ~bang
```

Both reference the %{actionname} mvar, which contains the command to execute if the user confirms the action; this needs to be set before showing the

dialog, as the dialog itself is responsible for calling !doit and !abortit as necessary.

Converse Operation — The LiteStep-native Engine, ckDialog Mode

The module is somewhat memory-heavy and for this reason, given that it is likely to be rarely used, Core 2.1 tracks the total number of served dialogs that are open. If there are no such dialogs open, the module is unloaded:

```
*Script start !varSet ckdialogcount "0"
*Script start !unloadmodule "$core$\modules\ckdialog\ckdialog.dll"
```

It is then reloaded on demand with the dialog creation code:

```
*Script bang !opendialog
*Script exec !if [{"ckdialogcount"} = "0"]
    [!reloadmodule "$core$\modules\ckdialog\ckdialog.dll"]
*Script exec !VarAdd ckdialogcount "1"
*Script exec !VarSet {%dialogname}_ckshown "1"
*Script exec !ckCreateDialog {%dialogname}Dialog {%dialogname}
*Script exec !VarRemove dialogname
*Script ~bang
```

The close dialog contains similar code:

```
*Script bang !closedialog
*Script exec !VarRemove {%dialogname}_ckshown
*Script exec !ckaction "{%dialogname}.show" "close"
*Script exec !VarAdd ckdialogcount "-1"
*Script exec !VarRemove dialogname
*Script exec !if [{"ckdialogcount"} = "0"]
    [!unloadmodule "$core$\modules\ckdialog\ckdialog.dll"]
*Script ~bang
```

The container script that should be called is !dodialog:

```
*Script bang !dodialog
*Script exec !VarSet dialogname {%args:1}
*Script gotoif [{"dialogname"} = ""] exit
*Script exec !if [{"%{%dialogname}_ckshown"} = "1"] [!closedialog]
    [!opendialog]
*Script ~bang
```

The reason for this is that ckDialog allows multiple instances of any of its dialogs to be displayed. This causes trouble for ensuring that all are closed down, etc. and for that reason, Converse's ckDialog mode tracks the existence of a dialog-specific mzvar built from the corresponding {%dialogname} mzvar to determine whether a dialog is supposed to be opened or closed down.

Converse Operation — The LiteStep-native Engine, LSBox Mode

The LSBox mode for Converse is best described by reference to the in-built dialog scripts are available in Core. Interface developers are free to use their own, but these are always available so long as the developer provides the required .box files. One example is the warning box that can be displayed when

a user requests the unload of a module:

```
*Script bang !warning_unload
*Script exec !varset dialogmode "\sbox"
*Script exec !varset dialogname "breakage"
*Script exec !FileExist "$DialogFolder$\%{dialogname}.box"
    '!LSBoxCreate "$DialogFolder$\%{dialogname}.box"' '!msgbox
    'Converse : $fileerrorstring$ : $DialogFolder$\
    %{dialogname}.box'"
*Script ~bang
```

Under the bunny interface this is used, in combination with the metta service, to warn users of potential problems unloading the wharf module:

```
*Script bang !wharf_unload
*Script exec !VarSet actionname "wharf_DoUnload"
*Script exec !warning_unload
*Script ~bang
```

When the user selects an entry calling the !wharf_unload script, Converse displays the breakage box, which contains the following code:

```
*Shortcut "" 66 276 "$LSImageFolder$\Dialogs\Yes.png" .none .none #0T
    !doit
*Shortcut "" 168 276 "$LSImageFolder$\Dialogs\No.png" .none .none #0T
    !abortit
```

On selecting either the yes or no shortcut the action specified in `%{actionname}` is either executed or discarded, depending on whether !doit or !abortit is called, and the box is closed.

The remaining Core-provided dialogs — !warning, !slowdown and !delay — are implemented similarly.

Note that the `$DialogFolder$` E-Var is an abstraction of a common location for these dialogs, seeing as the box files and images are not provided by Core itself. This path, and the required set of files, needs to be specified and provided by the interface.

Converse Operation — The WSH Engine

Unlike the LiteStep-native versions, the sCore implementation of Converse uses only one script to handle the dialogs:

```
*Script bang !converse_do
*Script exec !If ["%{SCORE_CONVERSE_DEBUG}" = ""]
    [!VarSet SCORE_CONVERSE_DEBUG
    "%{SCORE_CONVERSE_DEBUG_DEFAULT}"]
*Script exec !If ["%{conversetextstring}" = ""]
    [!VarSet conversetextstring wshvar:ldex:converse%{dialogname}]
*Script exec !VarSet converseexec
    "$LDEWISHLocation$\converse\converse.wsf"
    //job:%{SCORECONVERSEJOB} "%{dialogname}"
    "%{conversetextstring}" "! %{actionname}" "%{conversetextrgb}"
    "%{conversetextpos}" "%{converseimagepath}"
    "%{converseimagepad}" "%{SCORE_CONVERSE_DEBUG}"
*Script exec !VarRun converseexec
*Script exec !VarRemove SCORE_CONVERSE_DEBUG
```

```

*Script exec !VarRemove dialogname
*Script exec !VarRemove converseexec
*Script exec !VarRemove actionname
*Script exec !VarRemove conversetextstring
*Script exec !VarRemove conversetextrgb
*Script exec !VarRemove conversetextpos
*Script exec !VarRemove converseimagepath
*Script exec !VarRemove converseimagepad
*Script ~bang

```

The converse.wsf script, as evident above, is capable of taking a substantial number of arguments in order to customize the appearance of the resulting dialog; the display of the dialog is itself done using an application called Autolt, which builds it using standard Win32 API calls. The general way of calling converse.wsf is as follows:

```

converse.wsf //job:language title text action textcolor textposition
image padding debug

```

As with sCore/fish, the language argument determines the implementation of Converse to be used within the .wsf package. Currently only a JScript version is available, so the first argument will be “//job:converse_js”. The second, third and fourth arguments are required and specify the title and text of the dialog, and the action to be performed if the user confirms it, respectively.

The following four arguments allow for customizing the dialog's appearance. The first allows the developer to specify the color for the dialog text, in RGB format; the second allows for the specification of the offset of this text from the top of the dialog, and its width and height, in pixels and separated by the “|” character; the third provides the path to an emblem image, and the last allows for the specification of the padding between the dialog's emblem image and the dialog borders, in pixels once again. (If one value is given, it will be used for both horizontal and vertical padding. If two are provided, separated by the “|” character, the first will be used for the horizontal offset and the second for the vertical.) Finally, the last argument allows for debug output to be written. These values may be set through the matching mzvars in the !converse_do code above.

In the !converse_do script above, the textcolor, textposition, image and padding arguments are blank; this is allowed under LDE(X) due to the fact that defaults for these values are provided by a Core JScript library file, versiontracker.js. Similarly, the text is referenced using a “wshvar:” prefix. This allows converse to hook into the same localization system used to display fish dialog boxes, etc. in the user's own language. The string is parsed by converse.wsf and converted to a path/variable name combination that is then evaluated to get the localized string. In the case of a breakage dialog, for example, the argument that would be passed to converse.wsf is this:

```

“wshvar:ldex:conversebreakage”

```

Converse.wsf would take this, locate the converse.js file in \$Core\$JSLib\locale\EN\ldex and then use the value of the “conversebreakage” variable as the dialog text. It would then check for a localized version under a

matching sibling directory, and if found, will replace this default value with the localized text for the variable. Any string not starting with “wshvar:” will be used literally.

The title argument works in much the same way: converse.wsf will check to see if the string resolves to a variable available in the locale libraries. If it does, it will use the resulting string; if not, it will use the value of the title argument directly.

To illustrate how all of this comes together, this is the WSH version of !warning_unload:

```
*Script bang !warning_unload
*Script exec !varset dialogname "Breakage"
*Script exec !converse_do
*Script ~bang
```

!converse_do translates this to the following script call, using foo as the actionname value:

```
"$LDEwISHLocation$\converse\converse.wsf" //job:converse_js
"Breakage" "wshvar:ldex:converseBreakage" "! foo" "" "" "" ""
"nodebug"
```

Converse.wsf fills in the defaults, changing this to

```
"$LDEwISHLocation$\converse\converse.wsf" //job:converse_js
"converseBreakagetitle" "wshvar:ldex:converseBreakage" "! foo"
"255,255,255" "218|232|66" "path\to\core.jpg" "5|5" "nodebug"
```

Then it resolves the conversebreakage variable to its default English text ("This may break LDE(X)! Are you sure you want to do this? Please ensure that you have a recent backup in case you need to reverse the changes."), resolves converseBreakagetitle in the same way (to "Breakage," which is then prefixed with "LDEx://") and builds the resulting dialog.

Of course, all of this happens behind the scenes. Unless an interface developer wants to customize the dialogs his or her interface can show, calling !warning_unload is all that is required.

Debug

Everything lives in \$ghosthome\$\services\debug.rc. The script is very simple, as is the usage:

```
*Script bang !debugfunction
*Script exec !VarSet action %{args:1}
*Script exec !VarSet filename %{args:2}
*Script exec !VarSet output %{args:3}
*Script gotoif ["%{action}" = "alert"] alert
*Script gotoif ["%{action}" = "log"] log
*Script goto exit
*Script label alert
*Script gotoif ["%{filename}" = ""] exit
*Script exec !msgbox %{filename}
*Script goto exit
*Script label log
```

```

*Script gotoif [{"filename}" = ""] exit
*Script gotoif [{"output}" = ""] exit
*Script exec !textreplace @%{filename}@ @%{output}@
*Script label exit
*Script exec !varremove [action][filename][output]
*Script ~bang

```

Two modes of debug output are available in debug 1.2 — message boxes or file-based logging. These are best used with a conditional, so that you can enable or disable the debug output by setting or removing a variable definition.

The message box output is generally only suited to situations where very little feedback is desired, else you will be overrun by message boxes. To use this mode, the line below is all you need.

```

*Script exec !debugfunction "alert" "This string will be shown"

```

File logging of debug output is more suitable for handling complex and interwoven scripts where a significant amount of debug output is to be expected. Usage is shown in the line below.

```

*Script exec !debugfunction "log" "path\filename.extension"
    "This string will be written"

```

The debug support for the WSH engines is limited to file-based logging only at present, and each service writes to a dedicated file in the LDE(X) log directory.

Fog

Everything lives in \$ghosthome\$\services\fog.rc.

It's a very simple little bit of code - the trick is in the integration code that will be presented shortly.

```

*Script bang !fog
*Script exec !If [{"statusname}" = ""] [!VarSet statusname
    %fargs:1}]
*Script exec !If [{"status}" < "0"] [!VarSet status "0"]
*Script gotoif [{"statusname}" = ""] exit
*Script gotoif [{"fogpercent}" <> ""] percent
*Script exec !VarMul status 2.55
*Script exec !If [{"fogfloat}" = ""] [!VarInt status]
*Script exec !If [{"status}" > "255"] [!VarSet status "255"]
*Script goto do
*Script label percent
*Script exec !If [{"fogfloat}" = ""] [!VarInt status]
*Script exec !If [{"status}" > "100"] [!VarSet status "100"]
*Script exec !VarRemove fogpercent
*Script label do
*Script exec !therapy_tracer "%{statusname}" "%{status}"
*Script label exit
*Script ~bang

```

Note the use of the argument handling at the beginning of the !fog script. This comes from the calling function, as illustrated in the code sections here.

```

*Script bang !panel-opacityf
*Script exec !varset status %fargs:1}

```

```

*Script exec !varset targetfile "$suiprefsfile$"
*Script exec !varset mzvar "1"
*Script exec !fog "panel-alphablend_focus"
*Script exec !setpanelopacity
*Script ~bang

```

The function above also uses an argument handler — it is designed to be called in the manner shown below:

```
*Popup "40%" !panel-opacityf "40"
```

The `!panel-opacityf` function takes the '40' and sets the associated variable (`!panel-alphablend_focus` here, but it can be anything) to that value. The `!targetfile` variable is set (for the therapy tracer that will be used in the `!fog` function) at this point to give the user implementation the greatest flexibility.

The final undiscussed function of note is the call to `!setpanelopacity`: this is part of the specific use of `!fog` in this example, but similar functions may be needed in order for most interfaces to update their environments with the changes.

```

*Script bang !setpanelopacity
*Script exec !LsBoxSetTransparency PanelBG !panel-alphablend_focus
    !panel-alphablend_nonfocus
*Script ~bang

```

The reason that this is necessary is that `!fog` merely calculates, sets and stores the new value of `!panel-alphablend_focus`. It is up to the interface developer to apply it.

Loop

Everything lives in `$ghosthome$services\loop.rc`. Loop saw a major update in Core 2.2 and becomes much more useful for repeating events. The old `LSBox`-specific implementation is now provided by a wrapper function and will be shown later in this section.

It's a very simple little bit of code — the implementation code is where most of the control is available.

```

*Script bang !loop
*Script exec !VarSet looplimit [%{args:1}]
*Script exec !VarSet looponaction [%{args:2}]
*Script exec !VarSet loopoffaction [%{args:3}]
*Script exec !VarSet loopcount "1"
; checking those pause values.
*Script exec !ifexist loop_pause [!varset loop_temp_pausecheck
!loop_pause][!varset loop_temp_pausecheck !loop_pause-default]
*Script exec !VarSet loop temptemp_pausecheck !loop_temp_pausecheck
*Script exec !VarMul loop_temp_pausecheck 1
*Script exec !if [%{loop_temp_pausecheck} =
!loop temptemp_pausecheck][!ifexist loop_pause [!varset
loop_pausevalue !loop_pause][!varset loop_pausevalue !loop_pause-
default]][!varset loop_pausevalue 50]
*Script exec !VarRemove
!loop_temp_pausecheck !loop temptemp_pausecheck

```

```

; should now have sane pause value in even worst case screw-ups.
*Script label loop
*Script exec !execute [%{looponaction}]
*Script exec !pause [%{loop_pausevalue}]
*Script exec !execute [%{loopoffaction}]
*Script exec !pause [%{loop_pausevalue}]
*Script gotoif [%{loopcount} = [%{looplimit}]] exit
*Script exec !varadd loopcount 1
*Script goto loop
*Script label exit
*Script exec !VarRemove
[looplimit][loopcount][loopoffaction][looponaction]
*Script ~bang

```

This code is reasonably simple. The lengthy variable names make it look more intimidating than it is. It's simply a loop event with the first argument defining the number of loops, and each loop has an on and off event that can be used.

The LSBox wrapper is also quite simple :

```

*Script bang !loop_lsbox-shortcut
*Script exec !VarSet looplimit [%{args:1}]
*Script exec !VarSet loop_lsbox-shortcut_onaction [!LSBoxGroupShow
[%{args:2}]
*Script exec !VarSet loop_lsbox-shortcut_offaction [!LSBoxGroupHide
[%{args:2}]
*Script exec !loop [%{looplimit}] [%{loop_lsbox-shortcut_onaction}]
[%{loop_lsbox-shortcut_offaction}]
*Script exec !VarRemove [targetgroup][loop_lsbox-
shortcut_onaction][loop_lsbox-shortcut_offaction]
*Script ~bang

```

Note the use of the argument handling at the beginning of the !loop script. The arguments come from the calling function, as illustrated in this code:

```

*Script bang !StartInfoLDE
*Script exec !LSBoxCreate "$LiteStepDir$Panels\Dialogs\start.box"
*Script exec !pause 50
*Script exec !loop_lsbox-shortcut "1" "510"
*Script exec !loop_lsbox-shortcut "1" "520"
*Script exec !loop_lsbox-shortcut "1" "521"
*Script exec !loop_lsbox-shortcut "1" "522"
*Script exec !loop_lsbox-shortcut "1" "530"
*Script exec !loop_lsbox-shortcut "1" "540"
*Script exec !loop_lsbox-shortcut "1" "550"
*Script exec !loop_lsbox-shortcut "1" "560"
*Script ~bang

```

The first argument sent to the !loop_lsbox-shortcut function is the number of complete cycles (integer only!) that you wish to perform (on-off-on, nothing else!) and the second argument sent to the function is the shortcut group number. Easy, huh?

Navigator

Navigator Operation — The LiteStep-native Engine

Navigator currently supports two methods of operation: one for setting a folder location and one for file doing the same for a file. Both are called with two arguments, the first being the internal variable name in which to store the result and the second being designed as a description of the intended target (shown in the titlebar of the browse-for-file or folder dialog) — "text editor" or similar. All code resides in \$ghosthome\$\services\navigator.rc. Minor changes were made in the update for Core 2.2, mainly to tidy the code up and make it slightly more efficient.

The folder location script is called via !setloc:

```
*Script bang !setloc
*Script exec !VarSet name {%args:1}
*Script exec !VarSet description {%args:2}
*Script exec !IfExist navnowrite [!VarSet navnowrite "1"] [!VarSet
navnowrite "0"]
*Script exec !If [%{navnowrite} = "0"] [!VarSet targetfile
"%{args:3}"]
*Script gotoif [%{navnowrite} = "1"] splitnavwrite
*Script gotoif [%{args:3} = ""] nosplitarg
*Script goto setsplitmode
*Script label splitnavwrite
*Script gotoif [%{args:4} = ""] nosplitarg
*Script label setsplitmode
*Script exec !If [!"%{navnowrite}" = "1"] [!VarSet splitmode
%{args:3}][!VarSet splitmode {%args:4}]
*Script label nosplitarg
*Script gotoif [!"%{name}" = ""] exit
*Script gotoif [!"%{description}" = ""] exit
*Script gotoif [%{navnowrite} = "1"] skip

*Script gotoif [!"%{targetfile}" = ""] exit
*Script label skip
*Script exec !ComDlgSetTitle "$chooselocationofstring$ {%description}
- $presscanceltoskipstring$"
*Script gotoif [!"%{splitmode}" = ""] foldermode
*Script exec !varset navlocation:sep ":"
*Script exec !ComDlgFolder !VarSet navlocation "^@dp^"
*Script gotoif [!"%{splitmode}" = "path"] pathmode
*Script gotoif [!"%{splitmode}" = "drive"] drivelettermode
;*Script goto pathmode
*Script goto exit
*Script label pathmode
*Script exec !navigatorwrite "%{navlocation:2}"
*Script goto exit
*Script label drivelettermode
*Script exec !navigatorwrite "%{navlocation:1}"
*Script goto exit
*Script label foldermode
*Script exec !ComDlgFolder !navigatorwrite "^@dp^"
*Script label exit
*Script exec !navigatorcleanup
*Script ~bang
```

The file location script is called via !setfile:

```

*Script bang !setfile
*Script exec !VarSet name %{args:1}
*Script exec !VarSet description %{args:2}
*Script exec !If ["%{navnowrite}" = "0"] [!VarSet targetfile
%{args:3}]
*Script gotoif ["%{name}" = ""] exit
*Script gotoif ["%{description}" = ""] exit
*Script gotoif ["%{navnowrite}" = "1"] skip
*Script gotoif ["%{targetfile}" = ""] exit
*Script label skip
*Script exec !ComDlgSetTitle "$please locate string$ %{description} -
$press cancel to skip string$"
*Script exec !ComDlgFile !navigatorwrite "^@fp^"
*Script label exit
*Script exec !If ["%{error}" <> ""] [!navigatorcleanup]
*Script ~bang

```

The !navigatorwrite function is called by both scripts and provides a central, easily maintained way of storing the results of a user's file and folder choices:

```

*Script bang !navigatorwrite
*Script exec !VarSet navtemp %{args:1}
If DEBUG
*Script exec !debugfunction "alert" "target is %{navtemp}"
EndIf
*Script exec !If ["%{navtemp}" <> ""] [!VarSet navigatortarget
"%{navtemp}"]
*Script gotoif ["%{navtemp}" = ""] exit
If DEBUG
*Script exec !debugfunction "alert" "%{navigatortarget}"
*Script exec !debugfunction "alert" "%{name}"
*Script exec !If ["%{navnowrite}" = "1"] [!debugfunction alert
"%{targetfile}"]
EndIf
*Script exec !If ["%{navmzvar}" <> ""] [!VarSet %{name}
%{navigatortarget}]
*Script exec !If ["%{navevar}" <> ""] [!SetVar %{name}
%{navigatortarget}]
*Script exec !If ["%{navosvar}" <> ""] [!envset %{name}
%{navigatortarget}]
*Script exec !If [%{navnowrite} = "0"] [!therapy_tracer "%{name}"
%=%{navigatortarget}%=]
*Script label exit
*Script exec !navigatorcleanup
*Script ~bang

```

Sharp eyes will notice that !navigatorwrite supports three features, each of which can be toggled on or off using appropriate mzvars. The normal behavior is to write the resulting path to a file, similar to what is done by the therapy service; this can be overridden by setting the %{navnowrite} variable to anything other than an empty string. This isn't of much use on its own, but in combination with the other options it allows for the result to be written to a mzScript variable, Windows environment variable or both depending on the presence of the %{navmzvar} and %{navosvar} variables, respectively.

The implementation is usually via a set of dedicated scripts within the host environment, but a limited set is provided by Core. These scripts provide ways of setting common application paths and locations that are likely to be shared

across interfaces:

```
*Script bang !setpfilesloc
*Script exec !setloc "PROGRAMSPATH" "Program Files"
"$Core$\scripts\native\litestep\globalfiles\globalfolders.rc"
*Script ~bang

*Script bang !seteditor
*Script exec !setfile "EDIT" "text editor"
"$LDERootDataLocation$\global-fileprefs.rc"
*Script ~bang
```

These scripts live in \$Core\$\scripts\native\litestep\globalfiles\globalfiles.rc.

Navigator Operation — The WSH Engine

sCore/navigator replaces the !navigatorwrite script with processing done by the WScript engine itself. This is primarily to allow for compatibility with shells other than LiteStep. To illustrate this, consider the code for !setloc:

```
*Script bang !setloc
*Script exec !VarSet name {%args:1}
*Script exec !VarSet description {%args:2}
*Script exec !VarSet targetfile {%args:3}
*Script exec !If [{"%args:4"} <> ""] [!VarSet splitmode {%args:4}]
[!VarSet splitmode "folder"]
*Script exec !If [{"%name"} = ""] [!VarSet error "1"]
*Script exec !If [{"%description"} = ""] [!VarSet error "1"]
*Script exec !If [{"%navnowrite"} = "1"] [!VarSet targetfile
"nowrite"]
*Script exec !If [{"%targetfile"} = ""] [!VarSet error "1"]
*Script exec !If [{"%navmzvar"} <> ""] [!VarSet targetfile
"shell:!.exe varset >>var<< >>value<<:%{targetfile}"]
*Script exec !If [{"%navosvar"} <> ""] [!VarSet targetfile
"osvar:%{targetfile}"]
*Script exec !If [{"%navssf"} = ""] [!VarSet navssf "ssfDrives"]
If DEBUG
*Script exec !If [{"%SCORE_THERAPY_DEBUG"} = ""]
[!If [{"%SCORE_THERAPY_DEBUG_DEFAULT"} = ""]
[!VarSet SCORE_THERAPY_DEBUG "debug"]
Else
*Script exec !If [{"%SCORE_THERAPY_DEBUG"} = ""]
[!If [{"%SCORE_THERAPY_DEBUG_DEFAULT"} = ""]
[!VarSet SCORE_THERAPY_DEBUG "nodebug"]
EndIf
*Script exec !If [{"%SCORE_THERAPY_SUPPRESS"} = ""]
[!If [{"%SCORE_THERAPY_SUPPRESS_DEFAULT"} = ""]
[!VarSet SCORE_THERAPY_SUPPRESS "suppress"]]
*Script exec !If [{"%SCORE_THERAPY_DEBUG"} = ""] [!VarSet
SCORE_THERAPY_DEBUG "%{SCORE_THERAPY_DEBUG_DEFAULT}"]
*Script exec !If [{"%SCORE_NAVIGATOR_DEBUG"} = ""]
[!VarSet SCORE_NAVIGATOR_DEBUG
"%{SCORE_NAVIGATOR_DEBUG_DEFAULT}"]
*Script exec !If [{"%apptarget"} = ""]
[!VarSet apptarget "%{LDELSEngine}"]
*Script gotoif [{"%error"} = "1"] exit
*Script exec !VarSet navexec
"$LDEWishLocation$\navigator\navigator.wsf"
//job:%{SCORENAVJOB} folder {%name} "%{description}"
```

```

    "%{targetfile}" %splitmode %navssf "%apptarget}"
    "%SCORE_NAVIGATOR_DEBUG}" "%SCORE_THERAPY_DEBUG}"
    "%SCORE_THERAPY_SUPPRESS}"'
*Script exec !varRun navexec
*Script label exit
*Script exec !navigatorcleanup
*Script ~bang

```

In general, the WSH port of Navigator works in the same way as the native version when implemented under LiteStep — but there are two lines that deserve special attention. The first is the !varset line containing this string:

```
"shell:!.exe varset >>var<< >>value<<:%{targetfile}"
```

The navigator.wsf script uses a generalized method of setting shell variables; this is based on string substitution, replacing ">>var<<" with the name of the variable to be set and ">>value<<" with the return value of the browse dialog, if any. In the case of LiteStep with mzScript loaded — this being the default scenario under LDE(X) — the script translates this into !varset var value, using !.exe to communicate with the shell. Similar strings can be developed to allow it to be employed under other shells and in different configurations.

In keeping with this shell-independent attitude, the script sets environment variables not by using !envset, which is LiteStep-specific, but by a call to the system's WMI service. The code for this can be found in \$Core\$\JSLib\CoreLib.js:

```

function writeEnvVal(envName, envValue) {
    var env = wscript.GetObject("winmgmts://")
    + syso.ExpandEnvironmentStrings("%USERDOMAIN%")
    + "/root/cmiv2:win32_Environment");
    var envInst = env.SpawnInstance_();
    envInst.UserName = syso.ExpandEnvironmentStrings("%USERNAME%");
    envInst.Name = envName;
    envInst.VariableValue = envValue;
    envInst.Put_();
    envInst = env = null;
}

```

("syso" is an instance of a wscript.Shell object.)

Note that this function only supports setting user environment variables. This is done in order to err on the side of caution, but it can be easily modified to support the system context if necessary.

The second line of note in the !setloc script is this:

```
*Script exec !If ["%{navssf}" = ""] [!VarSet navssf "ssfDrives"]
```

This allows a developer to specify the root folder for the "browse for folder" dialog box that shows up when !setloc is called. The available values are listed in CoreLib.js for reference. Similar customization features are available in !setfile as well:

```

*Script exec !If ["%{navfilter}" = ""] [!VarSet navfilter
    "All Files (*.*)|*.*"]
*Script exec !If ["%{navinitdir}" = ""] [!VarSet navinitdir "null"]

```

Here the content of the `%{navfilter}` mzvar is used to set the file type filter for the browse dialog, and `%{navinitdir}` can be set to the path at which to start: `$programspath$,` for example.

Unlike the native version, sCore/navigator does not itself write the results to disk; it farms this job out to the WSH port of therapy. Because of this it can pass the value of `%{SCORE_THERAPY_SUPPRESS}` and `%{SCORE_THERAPY_DEBUG}` to that script, allowing recycle notifications to be shown or suppressed on demand. Note that default values will be calculated for these variables if the LiteStep-native version of therapy is active.

Profile

The profile service is, currently, a strictly LiteStep-native batch action approach that uses a paired-array mzScript setup: one holding the names of the values to be set (with the prefix “F_” to designate that the value is to be set using fish, “T_” to be set using therapy or “B_” to designate that the command should be executed directly) and the other array holding the values to be assigned to these settings or used as arguments. One benefit of this approach is that both name and value arrays can inherit content from other arrays; an additional benefit is that the profile arrays can be updated on-the-fly if needed. A slight disadvantage is that the available profile data is loaded in memory rather than being disk-based; a large number of profiles could present memory issues or stability concerns, so the inclusion of large number of profiles at any given time is generally discouraged. (Developers may want to consider the use of a setting that will allow profiles to be enabled and disabled on demand, as they may be useful only at certain times.)

With version 2.0, profile gained the ability to create new profiles, either for the user to keep to themselves, or to provide as part of the shared pot of profiles in the main LDE(X) folder.

Basic principle

The architecture is quite basic. For the purposes of this document, due to the size of the default arrays (from which properties can be inherited by other profiles), we will confine the discussion to example profiles which are much smaller.

Although it is technically possible to implement multi-dimensional arrays in mzScript using different separator characters for each dimension, this is unwieldy and, in practice, subject to breakage from unexpected values. Accordingly, the profile service is designed to use two similarly-named arrays: as noted earlier, these are one for the setting names and one for the setting values. The setting value array is a simple list of values but there are some special

issues to consider in regard to the array containing the corresponding values. This array will be examined first in the following discussion.

Note that each profile is best designed to reside in a single file, in order to make it easily installable and maintainable, and the implementation is left up to the interface designer. Core does not provide any profiles of its own.

Interface-level implementation

Setting Name Array Definition

The process of creating a profile settings array is simple. It's a two-step process, wherein a script that creates the values is defined

```
*Script bang !createprofile_test
*Script exec !VarSet test:1 "testvalues"
*Script exec !VarSet test:2 "F_TEST"
*Script ~bang
```

and then called by a two line code insert:

```
*Script start !createprofile_test
*Script start !scriptremove !createprofile_test
```

Although these two steps could be combined into one — using only the second and third lines from the first code snippet and using `start` instead of `exec` — this format is more easily maintained and less prone to causing the introduction of obscure bugs.

It's important to note the value given for `%{test:1}` here: this is the name of the array containing the values to be applied.

Setting Value Array Definition

The settings values array is created in the same way as the previous one:

```
*Script bang !createprofilevalues_test
*Script exec !VarSet testvalues:1 ""
    ; syncs the arrays for ease of definition
*Script exec !VarSet testvalues:2 "0"
*Script exec !VarRemove testvalues:1
    ; designed to offset the array as required
*Script ~bang

*Script start !createprofilevalues_test
*Script start !scriptremove !createprofilevalues_test
```

The current system offsets the value array to ease profile development; without this offset the value would actually match the following index in the settings array. This can cause problems when debugging large profiles with dozens of settings so an empty first element is introduced to keep the indices

synchronized. This element is removed at the end of the profile definition in order to keep it from needlessly being retained in memory.

Note that in order for the profile to be correctly applied this removal is required; the profile service code relies on the actual array offset.

The Service Code

In the interest of brevity only portions of the profile service code will be reproduced here. The full code lives in `$ghosthome/services/profile.rc`.

Profiles are applied by calling the `!applyprofile` script with the name of the profile to apply — in the case of the sample code above, “test” — as its only argument.

The first thing to note is that after setting up the structure for looping through the profile arrays, the service code contains this line:

```
*Script exec !VarSet SCORE_SUPPRESS_ALL “suppress”
```

This sets a variable that will prevent the WSH engines from presenting recycle and restart dialogs. It is automatically removed before the final call, allowing the prompts to be shown at the end of the processing cycle. Automatic recycling is also prevented for both the native and WSH ports of fish using this line:

```
*Script exec !VarSet nouupdate “1”
```

For each setting profile looks at the first two characters of the array element value and assigns a default action type if they do not match a valid prefix; currently, this means that in the absence of a prefix the value will be taken as the name of a bang to execute.

```
*Script exec !varset prefixtemp “${nameholder:1}”
*Script gotoif [“${nameholder:2}” <> “_”] noprefixfound
*Script gotoif [“${prefixtemp}” = “F”] prefixfound
*Script gotoif [“${prefixtemp}” = “T”] prefixfound
*Script gotoif [“${prefixtemp}” = “B”] prefixfound
*Script goto noprefixfound

*Script label prefixfound
*Script exec !VarSet profileactiontype “${prefixtemp}”
*Script exec !VarRemove nameholder:1 ;Remove action prefix
*Script exec !VarRemove nameholder:1 ;Remove underscore
*Script goto doaction

*Script label noprefixfound
*Script exec !VarSet profileactiontype “B”
*Script goto bangcheck

*Script label doaction
*Script exec !VarRemove prefixtemp
```

Once the action type is defined, the script branches based on this value and executes the appropriate action.

```
*Script gotoif [“${profileactiontype}” <> “F”] therapycheck
```

```

*Script label fishcheck
*Script exec !If
    [“%{profileactionstoapply}:%{applyprofile_loopcounter}”
    = “1”] [!mzparser_on “%{nameholder}”]
    [!mzparser_off “%{nameholder}”]
*Script goto actiondone

*Script label therapycheck
*Script gotoif [“%{profileactiontype}” <> “T”] bangcheck
*Script exec !If [“%{targetfile}” = “”]
    [!If [“%{profiledefaulttargetfile}” <> “”]
    [!VarSet targetfile “%{profiledefaulttargetfile}”]]
*Script exec !profile_unescape
*Script exec !therapy_tracer “%{nameholder}” “%{unescapedaction}”
*Script exec !VarRemove [unescapedaction]
*Script goto actiondone

*Script label bangcheck
*Script gotoif [“%{profileactiontype}” <> “B”] actiondone
*Script exec !profile_unescape
*Script exec !If [“%{unescapedaction}” <> “”]
    [!varset bangexec “!%{nameholder} %unescapedaction”]
    [!varset bangexec “!%{nameholder}”]
*Script exec !varrun bangexec
*Script exec !varremove [bangexec][unescapedaction]
*Script goto actiondone

*Script label actiondone
*Script exec !VarRemove [profileactiontype][nameholder]

```

Sharp eyes will notice that the therapy and bang execution sections call a script named !profile_unescape. This is a utility script that does a simple character substitution on the setting value (stored internally in %unescapedaction), allowing paths to be used without being split into array elements. In the current implementation this is limited to using “|” to represent the “.” character, which would otherwise be interpreted as an array element separator and cause the arrays to lose synchronization.

The creation code is slightly more involved, but also fairly straightforward. The choice of shared or private profiles is driven by “COMMON” or “PRIVATE” as the first argument to the !profile_create script. Without this, we exit. We use CatsCradle to simplify the comparison code by forcing any incoming argument into upper case. We then quit if the first argument doesn't match what we expect.

AutoIT is pressed into service to provide a text input dialog for the user to name their profile. If they enter no name, we quit. For the moment, we echo this back to the user for visual confirmation, but this will be removed when the creation code is known to be stable. If the profile name is the same as an existing profile, we quit. This was safer than overwriting.

The next step is to walk the default reference array and see if any settings have changed. We need to determine from the reference array whether the current 'setting' is for an actual setting, or whether it is a bang command. If it is a setting, we then need to establish whether this is for a fish command or for a therapy command. Based on these filtering actions, we can then query whether a setting

is on or off (fish), or whether the value is different (therapy or bang command). When we see a difference, we add it to our internal new profile list, otherwise we iterate the counter and compare the next one. Eventually, we hit the end of the reference profile and can write out our new profile.

To generate the on-disk file, we copy a template into place as the new file and then build the file contents to provide both the profile and also the interface inputs. There is a chunk of header and footer information that is hard-coded into place, but otherwise the write actions take place in a loop with a substitution of the required information.

```

*Script bang !profile_create
*Script gotoif [%{args} = ""] exit ; we need to have an incoming
profile type. All other input is ignored.
*Script exec !VarSet profilename %{args:1} ; added to allow private
or common profile creation. Drives file destination.
*Script exec !msgbox "Got this far"
*Script exec !changestringcase "upper" "%{profilename}"; use
catscradle function to ease comparison code :)
*Script gotoif ["%{profilename}" = "COMMON"] valid_type
*Script gotoif ["%{profilename}" = "PRIVATE"] valid_type
*Script goto exit ; invalid input so quit. Could force to be common,
if that is desired.
*Script label valid_type
*Script exec !a3exec "$Core$\scripts\autoit\textinput.au3" "varset"
"newprofilename"
*Script gotoif ["%{newprofilename}" == ""] exit
; Check if array name being passed is invalid and abort.
*Script gotoif [%{%{newprofilename}} = ""] valid_name ; currently
don't allow for existing profiles to be overwritten.
*Script exec !msgbox "$createprofileexistenceerrorstring$"
*Script goto exit
*Script label valid_name
*Script exec !VarSet counter "1" ; offset for array headers.
*Script exec !VarSet referenceprofile "default"
*Script exec !VarSet lengthofreferenceprofile
"%{%{referenceprofile}:count}"
;setup array headers.
*Script exec !VarSet %{newprofilename}:1 "%{newprofilename}actions"
*Script exec !VarSet %{newprofilename}actions:1 ""
;setup actions array separator for drive path safety
*Script exec !VarSet %{newprofilename}actions:sep "^"

;loop to walk the default profile for the current interface.
*Script label loop
*Script exec !VarAdd counter 1
*Script exec !msgbox "loop cycle : %{counter}" ; DEBUG CODE XXXX
*Script exec !VarRemove setting_type

;evaluate the reference array setting.
*Script exec !VarSet nameholder "%{%{referenceprofile}:%{counter}}%"
*Script exec !msgbox "nameholder set to : %{nameholder}"
*Script exec !VarSet nameholder:sep ""
*Script exec !msgbox "nameholder first char is %{nameholder:1}" ;
DEBUG CODE XXXX
*Script exec !if ["%{nameholder:1}" = "F"] [!VarSet setting_type "F"]
*Script exec !if ["%{nameholder:1}" = "T"] [!VarSet setting_type "T"]
*Script exec !if ["%{nameholder:1}" = "B"] [!VarSet setting_type "B"]

```

```

; remap array action name to lose prefix
*Script exec !VarRemove nameholder:1 ;Remove action prefix
*Script exec !VarRemove nameholder:1 ;Remove underscore
;find out what check to run
*Script gotoif ["%{setting_type}" = "F"] fish_setting
*Script gotoif ["%{setting_type}" = "T"] therapy_setting
*Script gotoif ["%{setting_type}" = "B"] bang
*Script exec !msgbox "Undefined type. Exited." ; DEBUG CODE XXXX
*Script goto exit ; failsafe
*Script label fish_setting
;check status of fish setting against reference profile
; if default is on, check for our being off.
*Script gotoif ["%{referenceprofile}actions:%{counter}" = "0"]
fish_setting_check2
*Script gotoif ["%#{nameholder}%" = "0"] buildnewprofile
*Script goto nochange
*Script label fish_setting_check2
; if default is off, check for our being on.
*Script gotoif ["%#{nameholder}%" = "1"] buildnewprofile
*Script goto nochange
*Script label therapy_setting
;check status of therapy setting against reference profile
*Script gotoif ["%{referenceprofile}actions:%{counter}" <
"%#{nameholder}%" ] buildnewprofile
*Script goto nochange
*Script label bang
;check value of bang setting against reference profile
*Script gotoif ["%{referenceprofile}actions:%{counter}" <
"%#{nameholder}%" ] buildnewprofile
*Script goto nochange
*Script label buildnewprofile ; different from default profile so
need to save the change.
*Script exec !VarSet %{newprofilename}:%{counter}
"%{setting_type}_%{nameholder}"
*Script exec !VarSet %{newprofilename}actions:%{counter}
"%#{nameholder}%"
*Script label nochange
*Script gotoif ["%{counter}" < "%{lengthofreferenceprofile}"] loop
*Script label createprofilefile
*Script exec !if ["%{profiletype}" = "COMMON"] [!VarSet profilefile
"$UI$\includes\profiles\%{newprofilename}.rc"]
*Script exec !if ["%{profiletype}" = "PRIVATE"] [!VarSet profilefile
"$UserDataLocation$\profiles\%{newprofilename}.rc"]
; Can assume that the common profiles folder exists, but may not be
true for the private location. Let's check and create it if needed.
*Script exec !if ["%{profiletype}" = "PRIVATE"] [!direxist
"%=$UserDataLocation$\profiles%" "" "!dircreate
"%=$UserDataLocation$\profiles%"]
*Script exec !filecopy "$Core$\support\newprofile.rc"
"%{profilefile}"
*Script exec !textappend @%{profilefile}@ @&Popup
\=%{newprofilename}\= \^applyprofile \=%{newprofilename}\=@
*Script exec !textappend @%{profilefile}@ @&Script bang
\^createprofile_%{newprofilename}@

;loop to build name array from changes.
*Script exec !VarSet newprofilelength %{newprofilename}:count
*Script exec !VarSet counter "1"
*Script exec !TextAppend @%{profilefile}@ @&Script exec \^VarSet
%{newprofilename}:1 \=%{newprofilename}actions\=@
*Script label profilename_loop

```

```

*Script exec !VarAdd counter 1
*Script exec !TextAppend @%{profilefile}@ @\&Script exec \^VarSet
%{newprofilename}:%{counter} \=%{newprofilename}:%{counter}\=@
*Script gotoif ["%{counter}" <> "%{newprofilelength}"]
profilename_loop
*Script exec !textappend @%{profilefile}@ @\&Script ~bang@

*Script exec !textappend @%{profilefile}@ @\&Script bang
\^createprofileactions_%{newprofilename}@
*Script exec !VarSet counter "1"
*Script exec !TextAppend @%{profilefile}@ @\&Script exec \^VarSet
%{newprofilename}actions:1 \= \=@
*Script exec !TextAppend @%{profilefile}@ @\&Script exec \^VarSet
%{newprofilename}actions:sep \=^ \=@
*Script label profileactions_loop
*Script exec !VarAdd counter 1
*Script exec !TextAppend @%{profilefile}@ @\&Script exec \^VarSet
%{newprofilename}actions:%{counter}
\=%{newprofilenameactions}:%{counter}\=@
*Script gotoif ["%{counter}" <> "%{newprofilelength}"]
profileactions_loop
*Script exec !textappend @%{profilefile}@ @\&Script ~bang@
*Script exec !textappend @%{profilefile}@ @\&Script start
\^createprofile_%{newprofilename}@
*Script exec !textappend @%{profilefile}@ @\&Script start
\^createprofileactions_%{newprofilename}@
*Script exec !textappend @%{profilefile}@ @\&Script start
\^scriptremove createprofile_%{newprofilename}@
*Script exec !textappend @%{profilefile}@ @\&Script start
\^scriptremove createprofileactions_%{newprofilename}@

*Script label exit
*Script exec !VarRemove
[lengthofreferenceprofile][counter][setting_type][nameholder][newprof
ilename][newprofilenameactions][referenceprofile][profiletype]
*Script ~bang

```

Progress

Progress Operation — The LiteStep-native Engine

Everything lives in \$ghosthome\$\services\progress.rc.

Progress supports two modes of operation, each with their own individual usage structures.

The original mode was that of time-based operation: defining a period of time for the overall operation in seconds by declaring the period of time for each 10% block of the total (e.g. for a 20 second operation, define 2 as the period of time for each 10% block). Note that due to limitations in mzScript, the hosting LiteStep environment cannot do anything at all until the time has elapsed (i.e. this mode doesn't permit multitasking within LiteStep, despite the threading support in 0.24.6). An example of this mode is shown below, from a legacy release of LDE.

```
*Script bang !rec
```

```

*Script exec !RecycleLabelShow
*Script exec !VarSet pausevalue 200
*Script exec !progress
*Script exec "$litestepdir$misc\recycle.bat"
*Script ~bang

```

If no time (pausevalue) is defined, a default value is used — 125ms at the time of writing. This corresponds to a total delay period of 1.25 seconds.

The other mode is to drive the progress bar directly by utilizing the ability to change the visibility status of each 10% block. This mode allows the progress system to work within the actions of the environment without preventing the hosting system doing any work at all. It requires a little extra effort, but this mode is usually more useful. An example of this mode is shown below, taken from Ermintrude2's startup code.

```

*Script exec !VarSet progresskeepopen "1"
*Script exec !pause 50
*Script exec !progress "10"
*Script exec !VarSet startuprunning "YES"
    ;Tracer for startup handling
*Script exec !progress "20"
*Script exec !userdatanotinstalled
*Script exec !progress "30"
*Script exec !startenvironment
*Script exec !progress "40"
*Script exec !VarRemove startuprunning
*Script exec !progress "70"
*Script exec !versioncheck-userfiles
*Script exec !progress "80"
*Script exec !user-startup
*Script exec !progress "90"
*Script exec !pluginsstartup
*Script exec !pause 1000
*Script exec !progress "100"
...
*Script exec !progress "kill"

```

Unlike the previous mode, this requires a little more explanation. The progress code uses the presence of the `%{progresskeepopen}` mzvar to activate this mode of operation (the default being the timed operation as before). In order to ensure that the progress function does not close down the bar, a variable called `progresskeepopen` has to be defined. After calling `!progress "100"` the script will automatically set a variable named `%{progresskill}`, which will cause the service to close the progress bar on the next call. This can also be done, explicitly, by calling `!progress` with the "kill" argument, in which case the progress bar is closed immediately.

Note that progress only supports increments of 10% and cannot be rolled backward in its current implementation.

Argument handling was rewritten in the 1.4 revision of progress. In this version, calling `!progress` with no arguments will result in a standard timed progress bar if the `%{progresskeepopen}` mzvar is not defined; if it is, calling it with no arguments will toggle the box on and off. Similarly, the presence of this variable will determine whether a numerical argument will be used as a time delay setting or

percentage value.

Progress Operation — The WSH Engine

The WSH implementation of progress is used internally by sCore/fish and a number of the installation and maintenance scripts; it's not directly accessible by the shell. At this time it also relies on the presence of the user-ui.rc file and progress theme images provided by LDE(X). These limitations will be addressed in a later revision of the service that will use Autolt to provide Windows-native progress bars as an option, and as the default when the LDE(X) files are not present.

On the implementation side, sCore/progress uses .hta (HTML Application) files as hidden stub windows. These files are opened hidden, and once opened they display a content file using the HTA showModalDialog() method. The content file itself contains a single image reference along with some script to change the image source when a keypress is received.

Scripts access the progress features by using the diaLogosLib.js library. This exposes three functions: progressopen(), progressset() and progressclose(). In most cases the first function is called without an argument, although it can be passed a path to a progress .hta file directly. (When called without an argument it will use a setting from user-ui.rc to locate the proper progress theme. This is what allows it to match the progress theme setting used by the native engine.)

progressset() takes an integer percentage value, makes sure that the modal dialog window is active and sends it the appropriate keypress; when necessary, the percentage value is rounded to the nearest decade in order to work with the progress theme image sets available. Essentially, it's the equivalent of !progress "value". Finally, progressclose() is called in order to dismiss the dialog and close the HTA.

As with sCore/converse, actual use of the library is straightforward. The internals only need to be dealt with by those developers who want to produce their own progress themes, in which case the existing .hta/.html files should be used for reference.

Spring

Everything lives in \$ghosthome\$\services\spring.rc. It's a relatively simple bit of code, but it does have some issues related to different popup modules — a subject that will be dealt with shortly.

```
*Script bang !spring
*Script gotoif [%{args} = ""] exit
*Script exec !varset params %{args}
*Script gotoif [{"%{panelname}" = ""}] exit
*Script exec !If [{"%{springx}" = ""}] [!VarSet springx "0"]
```

```

*Script exec !If [{"springy}" = ""][!VarSet springy "0"]
*Script exec !DirExist "$LDERootDataLocation$\Popups\%{panelname}"
"!VarSet global%{panelname}exists '1'"
*Script exec !DirExist "$UserDataLocation$\Popups\%{panelname}"
"!VarSet local%{panelname}exists '1'"
*Script exec !if [{"global%{panelname}exists}" = "1"]
[!VarSet popuptoshow
"$LDERootDataLocation$\Popups\%{panelname}"]
*Script exec !if [{"local%{panelname}exists}" = "1"]
[!if [{"global%{panelname}exists}" = "1"]
[!VarSet popuptoshow "$UserDataLocation$\Popups\%{panelname}|
%{popuptoshow}"][!VarSet popuptoshow
"$UserDataLocation$\Popups\%{panelname}"]]
*Script gotoif [{"_xpop}" = ""] pop2
*Script gotoif [{"_xpop_compat}" <> ""] compatmode
*Script exec !varset _xpop_exec "!xPopupDynamicFolder
'%{popuptoshow}' %springx %springy '%params'"
*Script exec !varrun _xpop_exec
*Script goto exit
*Script label compatmode
*Script exec !varset _xpop_exec "!PopupDynamicFolder '%popuptoshow}'
%springx %springy '%params'"
*Script exec !varrun _xpop_exec
*Script goto exit
*Script label pop2
*Script exec !PopupDynamicFolder "%popuptoshow}" x=%springx
y=%springy %params}
*Script label exit
*Script exec !VarRemove [springx][springy][panelname][params]
*Script exec !If [{"_xpop_exec}" <> ""][!VarRemove _xprop_exec]
*Script ~bang
EndIf

```

The main consideration when employing spring is the use of the `%{springy}` mzScript variable. This can be used to take into account any magic pink in the base image: being transparent in the popup graphic rendering, this can produce the appearance of a gap between the panel and the bottom of the popup. To keep the code as flexible as possible, spring allows for dynamic adjustment of this offset through the use of this variable.

The other consideration to keep in mind is that spring was originally developed to support the `popup2` module. LS-Universe's `xPopup` module is a viable alternative, however, and uses a different syntax. It also uses a different `!bang` to invoke the popup depending on whether or not `popup2` compatibility mode is enabled. In order for spring to adjust to these different settings it checks for the presence of the `%{_xpop}` and `%{_xpop_compat}` mzvars. These must be set by the interface developer; spring currently makes no attempt to detect the modules or settings in use.

Therapy

Therapy Operation – The LiteStep-native Engine

Therapy is a key component of gHost. It originally served to detect the runtime

resolution and, if it had changed since the last session, it would make the necessary changes to the interface code before it was loaded — ensuring that the interface was built and laid out properly. With the development of mathematical support for LiteStep e-variables this functionality was removed from therapy. However, its other features were retained and play a key role in the operation of LDE(X).

Therapy provides an advanced way to write the value of variables out to disk. It expects a construct in the target file of the form

```
variablename "value" ;variablename
```

and will store the value passed to it in that location. Note that the construct must already exist when therapy is called; it does not do on-demand creation.

Therapy supports the use of a default target file — this involves much less work for the developer in those situations where an all-in-one configuration approach is used. This default file is set individually by each interface under LDE(X). Developers can specify their own by adding a line similar to the following in any of the interface's early-loading configuration files:

```
therapydefaulttargetfile "<path\to\filename.rc>"
```

In most cases the default target file will be located under \$userdatalocation\$, the standard location for interface-specific configuration files. This E-Var points to a folder named according to the following convention:

```
%userprofile%\LDEData\<LDE(X) revision>\<interface name>
```

Therapy can also target any file you wish (be careful about this!) if you set the value of `!targetfile` to the path and filename you want to use. This variable is cleared after a successful execution of the therapy script, `!therapy_tracer`.

Therapy usage is generally quite simple and flexible, given that therapy can take any value and write it out as the value for any variable, as you might expect from the syntax below:

```
*Script exec !therapy_tracer <variablename> <value>
```

Therapy can also set the value of the variable named `variablename` to `value` in addition to writing the result to disc. This can be done for both E-Vars and mzvars depending on the presence of two variables: `!evar` and `!mzvar`, respectively. (Note that setting an E-Var value may or may not work depending on the build of LiteStep being used and the modules loaded at the time.)

The `!therapy_tracer` function is shown below :

```
*Script bang !therapy_tracer
If DEBUG
*Script exec !debugfunction "log" "$Core$\Debug\therapy-debug.txt"
    "therapy_tracer called"
*Script exec !debugfunction "log" "$Core$\Debug\therapy-debug.txt"
    "statusname : !statusname = !status"
EndIf
*Script exec !If [!args:1] <> "" [!VarSet statusname !args:1]
*Script exec !If [!args:2] <> "" [!VarSet status !args:2]
```

```

*Script exec !If [{"mzvar}" <> ""] [!VarSet {%statusname}
    "{status}"]
If XLABELLOADED
*Script exec !If [{"evar}" <> ""] [!SetEvar {%statusname}
    "{status}"]
EndIf
*Script exec !VarRemove [mzvar][evar]
*Script gotoif [{"targetfile}" <> ""] targetdefined
*Script gotoif [{"#therapydefaulttargetfile%#" <> ""] defaultdefined
*Script exec !msgbox $therapyerrorstring1$
If UIVENDOR
*Script exec !msgbox $therapyerrorstring2$
EndIf
*Script exec !msgbox $therapyerrorstring3$
*Script exec !msgbox $therapyerrorstring4$
*Script goto exit
*Script label defaultdefined
*Script exec !VarSet targetfile "#therapydefaulttargetfile%#"
*Script label targetdefined
*Script exec !textreplace @%{targetfile}@ @(.*) \~{%statusname}_@
    @%{statusname} \=%{status} \= \~{%statusname}@
*Script gotoif [{"keepstatus}" <> ""] statuskept
*Script exec !VarRemove [statusname][status]
*Script label statuskept
*Script exec !If [{"updatehost}" <> ""] [!updatehost]
*Script label exit
*Script exec !VarRemove targetfile
*Script exec !If [{"updatehost}" <> ""] [!VarRemove updatehost]
If DEBUG
*Script exec !debugfunction "log" "$Core$\Debug\therapy-debug.txt"
    "therapy_tracer completed"
EndIf
*Script ~bang

```

Note that unlike fish, therapy does not execute an existing !updatehost script by default; it will only do so if the {%updatehost} flag variable is defined.

As a side note, sharp eyes will notice the presence of the XLABELLOADED E-Var in the code above. That is used in the current Core revision to flag the loading of LS-Universe's xLabel module, which provides the ability to modify E-Vars in memory. (This feature is not included in current LiteStep builds.) This module is currently provided by Core when dependencies require it; if so provided, the value of this E-Var will be set to "1."

Therapy Operation — The WSH Engine

When used under LDE(X), sCore/therapy operates in an identical manner to the LiteStep-native version — with the exception that, as with fish, it is able to display recycle prompts after it has finished its task. This behavior is off by default but can be enabled on demand by setting the SCORE_THERAPY_SUPPRESS mzvar to a value that the script will recognize as false ("0" or "nosuppress").

Used on its own, therapy.wsf is able to use the same kind of application detection as sCore/fish (and, accordingly, the comment character support as well). It is also able to set registry values.

The general format for calling the script is this:

```
fish.wsf //job:language statusname status targetfile
        [debug] [suppress] [appname]
```

Currently only a JScript implementation is available, so the first argument will always be //job:therapy_js. The last three arguments work in the same way as they do for fish, and the *statusname* and *status* arguments work in the same way as they do for the LiteStep-native version. The *targetfile* argument does as well, except when targeting the registry. To force the registry to be targeted, developers can pass “registry” to the script as the *appname* argument.

When in registry mode the *targetfile* argument must be in this format:

```
type:hive:path\to\key
```

“Type” must be either string, expand, multi or dword, corresponding to the type of value entry, and “hive” must be one of HKCU, HKLM, HKCR, HKU (users) or HKCC (current config).

When passing multi-string values in the *status* argument the individual strings must be separated by the “&|&” character combination; when passing DWORD values they must be in decimal notation. Default values for a key can be targeted using the *statusname* value of “@” or an empty string.

Type

Type allows for the setting of typefaces and font sizes through the standard Windows font dialog. Considering its similarity in function to navigator, it uses much the same naming convention and is called via !setfont:

```
*Script bang !setfont
*Script exec !VarSet fontstring %args:1}
*Script exec !VarSet fontsizestring %args:2}
*Script exec !VarSet fontfile %args:3}
*Script gotoif [%fontstring] = "" exit
*Script gotoif [%fontsizestring] = "" exit
*Script exec !ComDlgFont
*Script exec !dofont
*Script label exit
*Script ~bang
```

Essentially, this works as a wrapper for therapy, providing one target file and two variable names to use as markers. The values to be set are provided by the call to !ComDlgFont (provided by the comdlgLS module) and are parsed by the workhorse of the service, !dofont:

```
*script bang !dofont
*script exec !VarSet font ^@tn^
*Script exec !VarSet fontsize ^#d^^@ts^/10^#d^
*Script gotoif [%font] = "" exit
*Script gotoif [%fontsize] = "" exit
*Script gotoif [%fontsize] < "0"] exit
If DEBUG
*Script exec !debugfunction alert "font is %font} at size
```

```

    %fontsize}"
EndIf
*Script exec !VarSet targetfile "%fontfile}"
*Script exec !therapy_tracer "%fontstring}" "%Font}"
*Script exec !VarSet targetfile "%fontfile}"
*Script exec !therapy_tracer "%fontsizestring}" "%FontSize}"
*Script label exit
*Script exec !VarRemove [font][fontsize][fontstring][fontsizestring]
    [fontfile]
*Script ~bang

```

There is no sanity checking at all apart from checking for negative font sizes — users can pick any size and any font. For this reason it is recommended that interface developers check the value of %updatehost when calling !setfont. Automatically recycling LDE(X) after a user mistakenly selects WingDings as her or his popup font, for example, would be a strikingly bad idea.

The WSH port of type works identically to the LiteStep-native version with the single exception of the ability to pass suppress and debug values to therapy. This is done in the same way that it is done for sCore/navigator.

Uncle

Uncle provides a way of tracking the shown and hidden states of LSBoxes and is usually called via a script, as in the following code from the recycler plugin:

```

*Script bang !recyclerfloater
*Script exec !VarSet panelname "recyclerfloater"
*Script exec !VarSet foldername "$RECYCLERPLUGINPATH$"
*Script exec !VarSet filename "recyclerfloater"
*Script exec !VarSet statusname "RECFLT"
*Script gotoif ["%setdataonly}" <> ""] dataonly
*Script exec !VarSet targetfile "$RECYCLERPLUGINPATH$\config.rc"
*Script exec !dopanel
*Script label dataonly
*Script exec !VarRemove setdataonly
*Script ~bang

```

With Uncle 3.6, a modification was made to allow single line operation of Uncle as well, although this will not always be entirely suitable. A modification of the script above in this format would be as follows:

```

*Script bang !recyclerfloater
*Script exec !VarSet targetfile "$RECYCLERPLUGINPATH$\config.rc"
*Script exec !dopanel "recyclerfloater" "$RECYCLERPLUGINPATH$"
    "recyclerfloater" "RECFLT"

```

Incidentally, this illustrates the potential weakness in this format. The original script allows for another script to call it after defining %setdataonly; this allows the calling script to set the Uncle-related variables without actually toggling the box. This can be useful for making scripts more modular. In cases where this is not necessary the single-line version may be more efficient.

!dopanel is the common element in both versions. This uses the panelname,

foldername, filename and statusname variables to determine the box to show or, in the case of the single-line version, builds these variables from the argument list.

This is the code for the script, which lives in \$ghosthome\$\services\uncle.rc. The first section performs an argument counting operation:

```
*Script bang !dopanel
If DEBUG
*Script exec !debugfunction "log" "$Core$\Debug\uncle-debug.txt"
    "dopanel called."
EndIf
*Script exec !VarSet uncle_dopanel_argscheck "0"
*Script exec !if [{"filename}" = ""]
    [!VarAdd uncle_dopanel_argscheck 1]
*Script exec !if [{"foldername}" = ""]
    [!VarAdd uncle_dopanel_argscheck 1]
*Script exec !if [{"panelname}" = ""]
    [!VarAdd uncle_dopanel_argscheck 1]
*Script exec !if [{"statusname}" = ""]
    [!VarAdd uncle_dopanel_argscheck 1]
```

In the event that all of the variables are specified, as in the first version of the recycler script, !dopanel will branch to a check function; if some are specified but not all, it will instead jump to an error reporting section later down:

```
*Script gotoif [{"uncle_dopanel_argscheck}" = "0"]
    uncle_dopanel_checker
*Script gotoif [{"uncle_dopanel_argscheck}" <> "4"]
    uncle_dopanel_argerror
```

If neither of these conditions applies the script will assume that arguments have been passed and will assign them one by one to the required variables:

```
*Script gotoif [%{args:count} <> "4"] uncle_dopanel_argerror
*Script exec !VarSet panelname %{args:1}
*Script exec !VarSet foldername %{args:2}
*Script exec !VarSet filename %{args:3}
*Script exec !VarSet statusname %{args:4}
```

Note that the script checks the length of the arguments array before assigning the variables, performing the same kind of check as the first section but targeting the alternate call format.

If all argument checks are passed, the script proceeds to check the existence of the specified box file:

```
*Script label uncle_dopanel_checker
*Script exec !VarRemove uncle_file_error
*Script exec !FileExist [{"foldername"}\{"filename"}.box"
    "!VarSet uncle_file_error '0'" "!VarSet uncle_file_error '1'"
*Script gotoif [{"uncle_file_error}" = "1"] uncle_dopanel_fileerror
*Script exec !VarRemove uncle_file_error
```

This uses a function provided by the filemaster module to set the mzScript variable %{uncle_file_error} to "1" if the specified box file does not exist. If this happens the script will jump to the file error handler; if it does not, the script will proceed to the next section.

```

*Script label uncle_dopanel_chooser
*Script exec !VarRemove uncle_dopanel_argscheck
*Script exec !If ["%{panelname}shown" <> ""]
    [!destroypanel] [!drawpanel]
*Script goto exit

```

This code checks for the existence of a status variable (set by Uncle in its box creation script) in order to determine whether or not the box is already shown. Based on this it calls either the create or destroy script as appropriate.

The remainder of the script contains the error handling functions.

```

*Script label uncle_dopanel_argerror
*Script exec !msgbox "$panelargerrorstring$"
*Script goto exit

*Script label uncle_dopanel_fileerror
*Script exec !VarRemove uncle_file_error
*Script exec !FileExist "%{foldername}\%{filename}.box" ""
    " !msgbox '$fileerrorstring$ : %{filename}'"
*Script exec !DirExist "%{foldername}" ""
    " !msgbox '$foldererrorstring$ : %{foldername}'"
*Script goto exit
*Script label exit

If DEBUG
*Script exec !debugfunction "log" "$Core$\Debug\uncle-debug.txt"
    "dopanel completed."
EndIf
*Script ~bang

```

One short note on the checking of the `%{panelname}shown` variable: this makes use of the pre-parse stage within `mzScript`, wherein `%{panelname}` is evaluated before the line is performed. In the case of the recycler script the check actually performed by `mzScript` is this:

```

*Script exec !If ["%{recyclerfloatersshown}" <> ""] [!destroypanel]
    [!drawpanel]

```

The draw and destroy panel functions work to handle the panel creation and destruction routines and, if applicable, the toggling of gadget groups as well. (An example of the use of gadget groups can be found in the code for the sidebar2 plugin.) The scripts are structurally the same, varying only in the actions taken, they're coded as separate scripts to avoid unnecessarily complex conditional structures.

This is the code for the `!drawpanel` script:

```

*Script bang !drawpanel
If DEBUG
*Script exec !debugfunction "log" "$Core$\Debug\uncle-debug.txt"
    "drawpanel called."
*Script exec !debugfunction "log" "$Core$\Debug\uncle-debug.txt"
    "drawpanel using %{panelname}."
EndIf
*Script gotoif ["%{gadgetset}" = ""] nogadget
*Script exec !LsBoxGroupHide %{gadgetset}01
*Script exec !LsBoxGroupShow %{gadgetset}
*Script exec !VarRemove gadgetset
*Script label nogadget

```

```

*Script exec !VarSet %{panelname}shown "yes"
*Script exec !if [{"%{panelname}isbroken"} = "1"]
    [!LSBoxShow %{panelname}]
    [!LSBoxCreate "%{foldername}\%{filename}.box"]
*Script exec !VarRemove [panelname][foldername][filename]
*Script gotoif [{"%{STATUSMEM}"} <> "1"] nostatus
*Script exec !VarSet mzvar "1"
*Script exec !If [{"%{status}"} = ""] [!VarSet status "1"]
*Script exec !therapy_tracer "%{statusname}STATUS" "%{status}"
*Script label nostatus
*Script exec !VarRemove [statusname][status]
If DEBUG
*Script exec !debugfunction "log" "$Core$\Debug\uncle-debug.txt"
    "drawpanel completed."
EndIf
*Script ~bang

```

Two particular parts of this code are worthy of note. The first is the code dealing with the `%{gadgetset} mzvar`; as noted earlier, this allows plugins such as `sidebar2` to specify shortcut groups to toggle when a box is created or destroyed. In order to make use of this feature the variable must be defined previous to calling `!dopanel` and the shortcut groups must be identically named, with the "closed" gadget name being postfixed with "01."

The second part involves the `%{panelname}isbroken` conditional. In some cases, usually when an `LSBox` is hosting a module, the boxes need to be hidden and shown instead of destroyed and created. This behavior can be enabled by setting the "isbroken" `mzvar` for the box in question — in the current example, this would be "recyclerfloaterisbroken."

Finally, the block of code within the `STATUSMEM` conditional serves as a link to the therapy system — enabling the environment to maintain its state across sessions. Defining this `mzvar` will enable this feature. (In the `elf`, `bunny` and `Ermintrude2` interfaces this is done using a fish marker, conditional branch and call to `!varset` in the user's interface preferences file.)

In order to protect against problems, `Uncle` provides a `!resetstatus` function allows for user status data to be reverted to a default state:

```

*Script bang !resetstatus
If DEBUG
*Script exec !debugfunction "log" "$Core$\Debug\uncle-debug.txt"
    "resetstatus called."
EndIf
*Script exec "$UI$\misc\maintain.wsf" environment
If DEBUG
*Script exec !debugfunction "log" "$Core$\Debug\uncle-debug.txt"
    "resetstatus completed."
EndIf
*Script ~bang

```

Note, however, that this is hardcoded to call the environment upgrade script currently implemented for the `bunny`, `elf` and `Ermintrude2` interfaces. In future revisions this will be more generalized.

As a additional feature, `Uncle 3.x` provides the ability to reset `LSBoxes` to defined

positions using variable movement speeds and visibility settings. Scripts that make use of this function are generally of a form similar to this, drawing once again from the recycler plugin code:

```
*Script bang !resetrecyclerfloater
*Script exec !VarSet setdataonly "1"
*Script exec !recyclerfloater
*Script exec !VarSet panelx 130
*Script exec !VarSet pany 130
*Script exec !resetpanel
*Script ~bang
```

Note that the use of the setdataonly variable prior to calling !recyclerfloater. As noted earlier regarding the code for that script, this serves as a flag variable to signal that the variables related to a !dopanel call should be set, but the call itself should not be made. This allows for some modularity in the code and centralizes the definitions of the variables.)

The final Uncle function, !resetpanel, supports animated or static resetting of boxes to their original positions based on the values of two variables: movesteps and movetime. (If defined as E-Vars these will be converted to mzvars. Existing mzvar values will be preserved if they exist. In the cases of the bunny, elf and Ermintrude2 interfaces these are stored as E-Vars in the user's interface preference file.) In order to customize the reset positions the developer needs to set the %panelx and %pany variables to the desired coordinates before calling the script. If these are not set the script will use default values of twenty pixels for each.

```
*Script bang !resetpanel
If DEBUG
*Script exec !debugfunction "log" "$Core$\Debug\uncle-debug.txt"
    "resetpanel called."
EndIf
*Script exec !If [{"panelx"} = ""] [!VarSet panelx 20]
*Script exec !If [{"pany"} = ""] [!VarSet pany 20]
*Script exec !If [{"%panelname}shown" <> ""]
    [!VarSet panelshown "yes"]
*Script gotoif [{"reset-evars"} <> ""] evars
*Script gotoif [{"panelshown"} = "yes"] movebox
*Script goto writebox
*Script label movebox
*Script gotoif [{"animatedreset"} = ""] noanimreset
*Script gotoif [{"_uncle_reset_init"} = "1"] noanimreset
*Script exec !If [{"movesteps"} = ""]
    [!VarSet movesteps "%#MOVESTEPS%#"]
*Script exec !If [{"movetime"} = ""]
    [!VarSet movetime "%#MOVETIME%#"]
*Script exec !VarSet _uncle_reset_init "1"
*Script label noanimreset
*Script exec !LSBoxMoveBox %panelname %panelx %pany
    %movesteps %movetime
*Script label writebox
*Script exec !textreplace @%foldername%\%filename}.box@
    @LSBoxX (.*)@ @LSBoxX %panelx@
*Script exec !textreplace @%foldername%\%filename}.box@
    @LSBoxY (.*)@ @LSBoxY %pany@
*Script goto exit
```

```

*Script label evars
*Script exec !VarRemove reset-evars
*Script gotoif ["##{panelx}%" <> ""] evars-ok
*Script gotoif ["##{panely}%" <> ""] evars-ok
*Script exec !resetpanel
*Script goto evars-exit
*Script label evars-ok
*Script gotoif [{"{panelshown}" = "yes"}] movebox-ev
*Script goto writebox-ev
*Script label movebox-ev
*Script exec !LSBoxMoveBox %{panelname} %##{panelx}%# %##{panely}%#
    ##movesteps%# ##movetime%#
*Script label writebox-ev
*Script exec !textreplace @%{foldername}\%{filename}.box@
    @LSBoxX (.*)@ @LSBoxX \##{panelx}\#@
*Script exec !textreplace @%{foldername}\%{filename}.box@
    @LSBoxY (.*)@ @LSBoxY \##{panely}\#@
*Script label exit
*Script exec !unclecleanup
If DEBUG
*Script exec !debugfunction "log" "$Core$\Debug\uncle-debug.txt"
    "resetpanel completed."
EndIf
*Script ~bang
*Script label evars-exit

```

Sharp eyes will notice that there are two modes of operation, toggled by the presence of the `{reset-evars}` variable. When this is set, the current values for the `panelx` and `panely` variables are ignored; instead, the script uses the values of the E-Var names given for `panelx` and `panely`. When using this mode it is required that developers provide valid names for the values of these variables before calling the script. Uncle does not attempt to check if the resulting value would be a valid one to pass to `!LSBoxMoveBox`, although it will replace the call with one using the standard `mzvar` syntax if either of the E-Vars does not exist.

List

The list service is based on third-party code, originally developed by tnl, that has been modularized and adjusted to work within the LDE(X) framework. It provides various methods for manipulating mzScript array variables. Each function can be loaded independently and will load its own dependencies as needed.

Although some of the services can be duplicated using the functions from the stack service, the equivalent ones here usually have additional features that may make them more applicable. `!listadd` is one example.

This script takes two arguments: the name of the array variable to target, which will be created if it does not exist, and the value to add. Before adding to the list, however, the script checks the existing values; as a result, duplicates will not be added. This check is done through a call to `!listfind`, as shown in the code below:

```
*script bang !listadd
*script exec !varset _listadd_retref %{args:1}
*script exec !varset _listadd_pending %{args:2}
*script exec !varset _listadd_create "false"
*script exec !if ["%{_listadd_retref}" = ""]
    [!varset _listadd_create "true"]
*script gotoif ["%{_listadd_create}" = "false"] append
*script exec !varset %{_listadd_retref} "%{_listadd_pending}"
*script goto cleanup
*script label append
*script exec !varset _listadd_retval "%{_listadd_retref}"
*script exec !listfind _listadd_found "%{_listadd_retval}"
    "%{_listadd_pending}" "0"
*script gotoif ["%{_listadd_found}" <> "0"] dontadd
*script exec !if ["%{_listadd_retval}" = ""]
    [!varset %{_listadd_retref} "%{_listadd_pending}"]
    [!varset %{_listadd_retref}
    "%{_listadd_retval}%{listsep}%{_listadd_pending}"]
*script label dontadd
*script label cleanup
*script exec !varremove _listadd_create
*script exec !varremove _listadd_retref
*script exec !varremove _listadd_retval
*script exec !varremove _listadd_pending
*script exec !varremove _listadd_found
*script ~bang
```

The "0" in the `!listfind` call checks for any and all occurrences of the given string; the index, if found, is then stored in the `_listadd_found` mzvar. The script branches based on this result, adding the value to the array only if this call returns a zero value.

Description of the full range of scripts is beyond the scope of this manual, but the interfaces for tnl's original code have been maintained. Documentation for these features is available in `$Core$\scripts\native\litestep\list\list.html`. Two special notes are in order, however.

The first is in regard to the `!listsep` script. Originally, this was designed to manipulate the global list separator character in mzScript, but this feature no

longer exists with mzScript 1.0. The script has accordingly been stripped down to its bare essential: setting the mzvar `listsep` to a particular value so that it can be passed to the other list services:

```
*script bang !listsep
*script exec !varset listsep %{args:1}
*script ~bang
```

Secondly, it's worth exploring the features of the `!listexec` script, which is both powerful and somewhat obscure in usage. This does not, as one might expect, run the contents of each list element. Instead, it uses the value of each element as a substitution in a command template. This is done through two calls to separator replacement scripts:

```
*script bang !listexec
*script exec !varset _listexec_working "%{args:1}%{listsep}empty"
*script exec !varset _listexec_exec %{args:2}
*script gotoif ["%{_listexec_working}" = "%{listsep}empty"] nomore
*script exec !stack_pop "_listexec_working" "%{listsep}"
*script exec !varset _listexec_working "%{_stack}"
*script label nexttoken
*script gotoif ["%{_listexec_working}" = ""] nomore
*script exec !stack_shift "_listexec_working" "%{listsep}"
*script exec !varset _listexec_token "%{stack_retval}"
*script exec !if ["%{_stack}" = ""] [!varremove _listexec_working]
    [!varset _listexec_working "%{_stack}"]
*script exec !varset _listexec_exec2 "%{_listexec_exec}"
*script exec !listresep _listexec_exec2 - "%{_listexec_token}"
*script exec !listresep3 _listexec_exec3 _listexec_exec2 "."
*script exec !varrun _listexec_exec3
*script goto nexttoken
*script label nomore
*script exec !varremove _listexec_exec
*script exec !varremove _listexec_exec2
*script exec !varremove _listexec_exec3
*script exec !varremove _listexec_token
*script ~bang
```

Using the stack functions (discussed in their own section), this script walks through the list of values given in its first argument, calls `!listresep` to substitute each value for each hyphen in the command template and then passes the resulting string to `!listresep3`, which replaces each period with a space.

An example may explain this best. Consider this code:

```
!listexec pitas:pizzas:zebras !varset.-.tasty
```

On the first pass through, the `!listexec` script would translate the command template to this:

```
!varset.pitas.tasty
```

This would be converted on the second pass to this, and executed:

```
!varset pitas tasty
```

In the end, the result would be the creation of three mzvars, `%{pitas}`, `%{pizzas}` and `%{zebras}`, all with the value "tasty."

Bear in mind that the current implementation does suffer from some

weaknesses. It does not support command templates containing quotes, or array definitions surrounded by them; it will also fail if either the period or hyphen characters come at the end of the command template. (This last point is actually due to limitations in the `!listresep` and `!listresep`s scripts and may be addressed in a future revision.)

Finally, two scripts were added to the list service during its port to LDE(X), originally developed for another service and merged in due to overlapping scope. These are the `!listtovars` and `!listclearvars` scripts.

```
*script bang !listtovars
*script gotoif [%{args:1} = ""] exit
*script exec !varset _listtovars_name %{args:1}
*script label loop
*script exec !varadd _listtovars_count "1"
*script exec !varset _listtovars_target
    %{_listtovars_name}%{_listtovars_count}
*script gotoif ["%{_listtovars_name}:%{_listtovars_count}" = ""]
    exit
*script exec !varset %{_listtovars_target}
    %{_listtovars_name}%{_listtovars_count}
*script exec !varset %{_listtovars_name}count %{_listtovars_count}
*script goto loop
*script label exit
*script exec !varremove _listtovars_name
*script exec !varremove _listtovars_count
*script exec !varremove _listtovars_target
*script ~bang
```

The `!listtovars` script, shown above, carries out a simple operation; it creates `mzScript` variables based on the original variable name and the individual element index, creates a count variable in the same way, and assigns the index values and count total to each, respectively. To illustrate, consider these commands:

```
!varset myarray "foo:bar:baz"
!listtovars foo
```

The end result would be the definition of the list `foo`, the variables `foo1` through `foo3`, set to "foo," "bar" and "baz," respectively, and the `foocount` variable, set to "3." As can be seen, this provides an ability to automatically create sequential variables from a list, something not possible with `!listexec`.

`!listclearvars` essentially undoes the previous function. It checks for the existence of the first index and count variables and, if found, removes the series:

```
*script bang !listclearvars
*script gotoif [%{args:1} = ""] exit
*script exec !varset _listclearvars_name %{args:1}
*script gotoif ["%{_listclearvars_name}1" = ""] exit
*script gotoif ["%{_listclearvars_name}count" = ""] exit
*script label loop
*script exec !varadd _listclearvars_count "1"
*script exec !varremove %{_listclearvars_name}%{_listclearvars_count}
*script gotoif ["%{_listclearvars_count}" =
    "%{_listclearvars_name}count"] loopexit
*script goto loop
*script label loopexit
```

```
*script exec !varremove %[_listclearvars_name]count
*script label exit
*script exec !varremove _listclearvars_name
*script exec !varremove _listclearvars_count
*script ~bang
```

Metta

Metta comes into play when a user requests the unloading or disabling of a module that either the Core or the interface recognizes as being critical to its operation. This recognition is set up through a combination of the Metta service and module-specific configuration files.

The module files follow a format similar to this, used by Ermintrude2 to add Metta support to its hotkey module:

```
If HKY

*Script bang !Keys_Unload
*Script exec !VarSet module
    "$UI$\Modules\Utils\multikeys\multikeys.dll"
*Script exec !Module_UnLoad
*Script ~bang

*Script bang !Keys_Off
*Script exec !VarSet moduletag "HKY"
*Script exec !Module_Off
*Script ~bang

Else

*Script bang !Keys_On
*Script exec !VarSet moduletag "HKY"
*Script exec !Module_On
*Script ~bang

EndIf
```

The two module- and interface-specific elements here are the values of the module and moduletag variables, which are the full path to the module to unload and the fish marker used to enable and disable loading of the module by the interface, respectively.

The !module_* scripts are defined by Metta:

```
*Script bang !Module_Unload
If DEBUG
Script exec !debugfunction log "$Core$\Debug\metta-debug.txt"
    "Module_Unload called with %{module}"
EndIf
*Script exec !VarSet actionname "UnloadModule %{module}"
*Script exec !warning_unload
*Script exec !VarRemove module
*Script ~bang

*Script bang !Module_Off
If DEBUG
*Script exec !debugfunction log "$Core$\Debug\metta-debug.txt"
    "Module_Off called with %{moduletag}"
EndIf
*Script exec !VarSet actionname "mzparser_off %{moduletag}"
*Script exec !warning
*Script exec !VarRemove %{moduletag}
*Script exec !VarRemove moduletag
*Script ~bang
```

```
*Script bang !Module_On
*Script exec !VarSet %{moduletag} "1"
*Script exec !mzparser_on "%{moduletag}"
*Script exec !VarRemove moduletag
*Script ~bang
```

These scripts are essentially ways to wrap module unload and disable calls so that they automatically fork out to Converse for confirmation. (See the discussion of the !warning and !warning_unload scripts in the section on this service.) The script to enable a module is provided for consistency.

Sharp eyes will note, however, that the !module_on and !module_off scripts are not simple wrappers for fish. They also set or remove, as appropriate, matching mzvars — something potentially useful when executing caches, profiles or other batch actions that may need to determine the action taken by the user.

The Core provides a small set of Metta scripts for its own modules and these are loaded directly by the service's metta.rc. For example:

```
*Script bang !ckHotSpots_Unload
*Script exec !VarSet module
    "$Core$\Modules\ckHotSpots\ckHotSpots.dll"
*Script exec !Module_Unload
*Script ~bang
```

Note that there are no matching on or off scripts, as these modules are loaded or bypassed according to the service dependency system.

Stack

The stack service provides a set of functions for performing standard push, pop, shift and unshift operations on mzscript arrays. These functions perform much the same way that they do in standard languages such as perl, except for the fact that the original variable is kept intact; all changes are made to one or two return variables, depending on the function. For example, these commands —

```
!varSet foo "bar"  
!stack_push "foo" "baz"
```

— would leave `%{foo}` unchanged but set `%{_stack}` to `"bar:baz"`. This is intended as a safety measure but can have unexpected consequences if it is not taken into account. To expand on this example, consider this sequence of commands:

```
!varSet foo "bar"  
!stack_unshift "foo" "foo"  
!stack_push "foo" "baz"
```

Developers might normally expect that this would result in `foo` ending up with the value `"foo:bar:baz"`. However, not only is this not the case — `%{foo}` is unchanged, as it was in the previous example — but `%{_stack}` will not have this value, either. It will end up as `"bar:baz"` again. In order to work as expected, `foo` needs to be updated with the results of the function call:

```
!varSet foo "bar"  
!stack_unshift "foo" "foo"  
!varSet foo "%{_stack}" ; foo is now "foo:bar"  
!stack_push "foo" "baz"  
!varSet foo "%{_stack}" ; foo is now "foo:bar:baz"
```

Since all of the stack scripts clear the `%{_stack}` and `%{_stack_retval}` mzvars when starting it is possible to use an `!If` directive for error checking:

```
!varSet foo "bar"  
!stack_unshift "foo" "foo"  
!If ["%{_stack}" <> ""] [!varSet foo "%{_stack}"] [!msgbox "error 1"]  
!stack_push "foo" "baz"  
!If ["%{_stack}" <> ""] [!varSet foo "%{_stack}"] [!msgbox "error 2"]
```

This can also be used to check for the use of incorrect separator characters, etc.

Stack stores the results of its functions in two variables: `%{_stack}`, which contains the array contents after the operation, and `%{_stack_retval}`, which contains the value returned by the pop and shift functions.

The first argument to each function is the name of the array to be processed. In the case of `!stack_push` and `!stack_unshift` the second argument is the value to be pushed or unshifted onto the array, and each function can optionally take a final argument: the string or character to be used as the array separator.

Utility Functions

Applaunch

LDE(X) 6.3 introduced the ability to launch applications after toggling an interface into full-screen or hidden mode. This is done by launching the application with either the `!launch_fs` or `!launch_hideall` scripts:

```
*Script bang !launch_fs
*Script gotoif [%{args:1} = ""] exit
*Script gotoif ["%{fullscreenlaunchcommand}" = ""] skipui
*Script exec %{fullscreenlaunchcommand}
*Script label skipui
*Script exec %{args}
*Script label exit
*Script ~bang

*Script bang !launch_hideall
*Script gotoif [%{args:1} = ""] exit
*Script gotoif ["%{hidealllaunchcommand}" = ""] skipui
*Script exec %{hidealllaunchcommand}
*Script label skipui
*Script exec %{args}
*Script label exit
*Script ~bang
```

These scripts simply check for the existence of the appropriate `launchcommand` `mzvar` and run its contents if found; the interface developer needs to define these variables so that they can be employed. In `Ermintrude2` this is done in the `command-aliases.rc` file:

```
*Script start !varset fullscreenlaunchcommand "!FullScreenMode"
*Script start !varset hidealllaunchcommand "!togglestuff"
```

Being simple wrapper functions, the combination of these scripts and variable definitions allows for application launch scripts to be portable across different interfaces.

Argscout

As the name suggests, the scripts provided by this service allow for run-time counting of arguments or, more appropriately, non-empty arguments. This is done through two scripts. The first counts bare-string arguments:

```
*Script bang !argscout
*Script gotoif [%{args:count} = "0"] exit
*Script exec !VarSet argstocheck '%{args}'
*Script exec !VarSet argstocheck:sep ""
*Script exec !VarSet argscounter 1
*Script exec !VarSet argscout %{argstocheck:%{argscounter}}
*Script label argscoutloop
*Script exec !VarAdd argscounter 1
*Script gotoif ["%{argstocheck:%{argscounter}}" <> ""] argscoutloop
*Script exec !VarAdd argscounter -2
```

```

*Script exec !VarSet %{argscount} "%{argscounter}"
*Script exec !VarRemove [argscount][argscounter][argstocheck]
*Script label exit
*Script ~bang

```

When passed a list of arguments, this script will count valid entries from the second element to the end of the argument array, storing the result in the mzvar named in the first argument. Note that this is a change from previous revisions, which used the value of the last argument as the return variable. The reason for this change becomes apparent when looking at the source for the quoted argument variation:

```

*Script bang !quotedargscount
*Script gotoif [%{args:count} = "0"] exit
*Script exec !VarSet argstocheck %{args}
*Script exec !VarSet argstocheck:sep '"'
*Script exec !VarSet argscounter 2
*Script exec !VarSet argscount %{argstocheck:%{argscounter}}
*Script label argscountloop
*Script exec !VarAdd argscounter 2
*Script gotoif ["%{argstocheck:%{argscounter}}" <> "" ] argscountloop
*Script exec !VarAdd argscounter -4
*Script exec !VarMul argscounter .5
*Script exec !VarSet %{argscount} "%{argscounter}"
*Script exec !VarRemove [argscount][argscounter][argstocheck]
*Script label exit
*Script ~bang

```

Although similar to the !argscount script, !quotedargscount looks at every second element, splitting the total argument string at its double quote marks. This allows it to count as single arguments those values that contain spaces. If the script is passed a variable reference that evaluates to an empty string, the count loop will terminate at this entry, signaling a problem with the argument list.

This method illustrates why the return variable is now the first argument instead of the last: it would not be defined in the case of an error. Or worse, it would be left holding a previously set value, leading to false positive results.

Locationcheck

Due to various quoting issues with mzScript and other modules, it is impossible to make a LiteStep-based environment completely safe for long path names. This becomes readily apparent when dealing with values such as this:

```

"$Core$\scripts\WSH\myscript.wsf" "someone's string"

```

In order to maintain the full path this variable would need to be enclosed within single quotes. However, this would result in the truncation of the variable value at "someone," additionally leaving an unbalanced set of double quotes.

In addition to this, scripts called by any LiteStep-native code often need to be able to reliably determine the root LDE(X) directory. Although it is possible to do this using environment variables or registry values, the sCore services take a

less invasive approach and examine their runtime paths, looking for the LDE(X) folder in order to determine the installation root.

This poses a potential problem, as it is possible to install the system to a different directory or rename it, or install it in a path containing spaces. The locationcheck script checks for these conditions and, if they exist, notifies the user of what needs to be done. It then opens an explorer window to the install directory and kills the LiteStep process:

```
*Script bang !locationcheck
*Script exec !VarSet ldexlocation "$LiteStepDir$"
*Script exec !VarSet ldexlocation:sep "."
*Script exec !VarSet ldexlocation "${ldexlocation:2}"
*Script exec !VarSet ldexlocation:sep " "
*Script exec !if ["${ldexlocation:2}" <> ""]
    [!VarSet ldexpatherror "1"]
*Script exec !VarSet ldexlocation:sep "\"
*Script exec !VarSet ldexlocationcount "${ldexlocation:count}"
*Script exec !VarAdd ldexlocationcount -1
*Script exec !VarSet ldexlocation
    "${ldexlocation:%{ldexlocationcount}%"
*Script exec !if ["${ldexlocation}" <> "LDE(X)"]
    [!VarSet ldexpatherror "1"]
*Script gotoif ["${ldexpatherror}" <> "1"] safepath
*Script exec !msgbox "$locationerrorstring$ ($litestepdir$)"
*Script exec !msgbox "$locationerrorquitstring$"
*Script exec !VarRemove ldexpatherror
*Script exec !VarSet browseexec "explorer.exe" "$LiteStepDir$"
*Script exec !VarRun browseexec
*Script exec !VarSet killexec "$LDEWISHLocation$\kill.wsf"
    $LDELSEngine$
*Script exec !VarRun killexec
*Script exec !pause 10000
*Script label safepath
*Script exec !VarRemove [ldexlocation][ldexlocationcount]
*Script ~bang
```

Note that the script uses kill.wsf to end the LiteStep process — calling normal termination bangs such as !quit do not work.

LiteStep Modifications for LDE(X)

LDE(X) is a sophisticated system, but that sophistication has required some modifications to the standard LiteStep engine. These changes are, at the time of writing, unique to the certified builds of LiteStep for LDE(X). These builds should not be used in connection with other distributions, but can be used with OpenLDE distributions or LDE(X) (naturally).

Includefolder

This is an essential modification for much of LDE(X) to work. If this feature is missing, much of LDE(X), including the Core, will fail. Accordingly, we advise that you only deploy certified builds of LiteStep under this system. The details of this modification are in the following sections.

Isapi\settingsfileparser.cpp

Replace the `_ProcessLine` function with the code below:

```
///=====
///
/// _ProcessLine
///
void FileParser::_ProcessLine(LPCTSTR ptzName, LPCTSTR ptzValue)
{
    ASSERT_ISSTRING(ptzName); ASSERT_ISSTRING(ptzValue);
    if (lstrcmpi(ptzName, _T("if")) == 0)
    {
        _ProcessIf(ptzValue);
    }
    else if (lstrcmpi(ptzName, _T("include")) == 0)
    {
        TCHAR tzPath[MAX_PATH_LENGTH] = { 0 };
        GetToken(ptzValue, tzPath, NULL, FALSE);
        FileParser fpParser(m_pSettingsMap);
        fpParser.ParseFile(tzPath);
    }
}
#ifdef LS_CUSTOM_INCLUDEFOLDER
// includefolder implementation, basically from ilmcuts :)
// tutorial over #lsdev (gulp) - he's very patient :)
// the first iteration broke because EVars weren't supported by the
// code and also findData.cFileName doesn't hold the full path
// This revision fixes these problems and is better commented because
// I'm dumb :)
    else if (lstrcmpi(ptzName, _T("includefolder")) == 0)
    {
        TCHAR tzPath[MAX_PATH_LENGTH]; // path+pattern
        TCHAR tzFilter[MAX_PATH_LENGTH]; // path only
        VarExpansionEx(tzPath, ptzValue, MAX_PATH_LENGTH);
        // expands string in ptzValue to tzPath - buffer size
        // defined by MAX_PATH_LENGTH
        PathUnquoteSpaces(tzPath);
        // strips quotation marks from string
    }
}
#endif
```

```

PathCombine(tzFilter,tzPath,_T("*.rc"));
// tzFilter now is tzPath appended with *.rc - the API
// takes care of trailing slash handling thankfully.
// Hard-coded filter for *.rc files to limit search
// operation.
WIN32_FIND_DATA findData;
// defining variable for filename
HANDLE hSearch = FindFirstFile(tzFilter, &findData);
// Looking in tzFilter for data :)
if (hSearch != INVALID_HANDLE_VALUE)
{
    BOOL FoundNextFile;
    do
    {
        if (!(findData.dwFileAttributes
        & FILE_ATTRIBUTE_DIRECTORY) &&
        (!(findData.dwFileAttributes &
        FILE_ATTRIBUTE_SYSTEM) &&
        !(findData.dwFileAttributes &
        FILE_ATTRIBUTE_HIDDEN)))
        // stripping out directories, system and
        // hidden files as we're not interested in
        // them and MS throws these kind of files
        // around from time to time....
        {
            // Processing the valid cFileName data
            // now.
            TCHAR tzFile[MAX_PATH_LENGTH];
            FileParser fpParser(m_pSettingsMap);
            PathCombine
            (tzFile,tzPath,findData.cFileName);
            // adding (like above) filename to
            // tzPath to set tzFile for opening.
            fpParser.ParseFile(tzFile);
        }
        FoundNextFile = FindNextFile(hSearch,
        &findData);
    }
    while(FoundNextFile);
    FindClose(hSearch);
}
}
#endif // includefolder
else
{
    m_pSettingsMap->insert(SettingsMap::value_type(ptzName,
    ptzValue));
}
}

```

litestep\buildoptions.h

Activates the includefolder code above:

```

// Adds includefolder functions to lsapi.
// 0.24.7 default: DISABLED
#define LS_CUSTOM_INCLUDEFOLDER

```

About Box

Announces the modifications made and rejigs the about box layout.

Lsapi\aboutbox.cpp

Replaces the 0.24.7 string with i686,includefolder. Also changes font size from 14 to 12 and extends the themeAuthor string to 21 characters.

```
case WM_INITDIALOG:
{
    // set title font
    HFONT hTitleFont = CreateSimpleFont("Verdana", 12, false);
    SendDlgItemMessage(hwnd, IDC_TITLE, WM_SETFONT,
(WPARAM)hTitleFont, FALSE);
    // set title with LS version
    SetDlgItemText(hwnd, IDC_TITLE, "i686,includefolder");
    // set Theme info
    char themeAuthor[21] = { 0 };
    char themeName[21] = { 0 };
    char themeOut[MAX_LINE_LENGTH] = { 0 };
}
```

Litestep\litestep.rc

Adds 0.24.7 to the About LiteStep caption for the window since the above modification has removed it from the previous location.

```
CAPTION "About LiteStep 0.24.7"
```

Evar to Announce Includefolder Availability

Lsapi/settingsmanager.cpp

A minor adjustment to add a couple of lines to this file, as shown below, to set an Evar when includefolder is available for use. Provides a safeguard to check against in LDE(X). The i686 Evar definition is included for completeness. The bitbucket and documents lines are added to provide a positional guide for adding these definitions.

```
#ifdef LS_CUSTOM_INCLUDEFOLDER
    SetVariable("includefolder", "1");
#endif
    SetVariable("i686", "1");
    SetVariable("bitbucket",
        "::{645FF040-5081-101B-9F08-00AA002F954E}");
    SetVariable("documents",
        "::{450D8FBA-AD25-11D0-98A8-0800361B1103}");
```

Additionally, the OS "detection" code was updated as shown below :

```
if (OsVersionInfo.dwMajorVersion == 6)
{
```

```

        SetVariable("Longhorn", "true");
        SetVariable("os", "Longhorn");
    }
    else if (OsVersionInfo.dwMajorVersion == 5)
    {
        if (OsVersionInfo.dwMinorVersion >= 2)
        {
            SetVariable("win2003", "true");
            // Windows 2003 (5.2)
            SetVariable("os", "Win2K3");
        }
        else if (OsVersionInfo.dwMinorVersion >= 1)
        {
            SetVariable("winXP", "true");
            // Windows XP (5.1)
            SetVariable("os", "winXP");
        }
        else
        {
            SetVariable("win2000", "true");
            // Windows 2000 (5.0)
            SetVariable("os", "win2K");
        }
    }
    else if (OsVersionInfo.dwMajorVersion >= 4) // windows NT 4.0
    {
        SetVariable("winNT4", "true");
        SetVariable("os", "winNT4");
    }
}
}

```

Escape Codes

This is experimental, following a suggestion from ilmcuts. It's not an entirely convincing approach, though it may prove useful to some. Since it is experimental, it might be withdrawn in a future build and is not available in the main 0.24.7 CVS.

Lsapi/settingsmanager.cpp

Similar to the evars added for i686 and includefolder checks, we can add a buildoption controlled set of single character Evars for use as escape codes.

```

#ifdef LS_CUSTOM_ESCAPECODES
    SetVariable("c", "!");
    SetVariable("b", "!");
    SetVariable("e", "$");
    SetVariable("n", "\n");
    SetVariable("q", "\\");
#endif

```

FAQs

Core

1. How do I use Core to deliver my interface?

You should first check the SDK. It contains a comprehensive list of Core features and usage examples that should ease the learning curve and speed up the development of your interface. The SDK contains information relating to delivering your new interface in a manner compatible with other Core-dependent interfaces and environments.

2. What's the WiSH modification all about?

The WiSH modification started as a contribution from Amtal during the development of LDE(X)6.1 and has since become tightly integrated with the LDE(X) system. It is aimed at moving the majority of the advanced scripting to more standard languages (javascript, vbscript, etc.) by using the Windows Script Host (WSH).

The advantage of using standard languages, compared to mzScript under LiteStep, is portability to other systems and ease of developer access. mzScript is a unique language that requires a fair amount of time to learn and work with compared to existing, universal languages like javascript or vbscript. Portability to other shells and systems may be of limited value due to the designed-in reliance on LiteStep modules, but the standardized language provides significant benefits.

A further advantage is the ability to integrate advanced error handling and flexibility over the mzScript-native code.

3. Why is WiSH a contributed part of LDE(X) rather than a designed-in part of the environment?

It's no longer a "contributed" system in the sense of an add-on. It has become a core part of the system and deservedly so. It's become designed-in, like most parts of LDE(X) — consider it a natural evolution of the code.

4. Why on earth is fish called fish, etc.?

It used to be an acronym, but the expanded form was lost to the mists of time a while back. I suppose one could be retrofitted so consider this hidden contest #1 in the documentation and feel free to submit one if you wish :)

Uncle was derived from User iNterface Control Engine.

therapy was named because it "treats" values within files and the session. It used to do more than this, but progress with LiteStep 0.24.6 meant that part of it became redundant.

Converse was named because it gets involved in dialogs (dialogues... yep — it was a stretch, but coders' collective sense of humor can be a little odd from time to time).

Many of the others are obvious.

5. Why is LDE called LDE, etc.?

This is documented in detail in various places. LDE used to be the Leaf Desktop Environment, but they had no further need of it at the end of 2002. After some negotiation, the name and project was transferred to the LDE development team. The project has no connections with its former owners and there was no need for a fork either, which was helpful. The LDE development team was subsequently created in 2003/2004 to give collective ownership of the project combined with stable and sensible management.

Known Issues

Core

- To get full feature support, the host system really should have Internet Explorer 5.5 and WScript 5.6+ installed. This will not be the default case for clean installations of Windows 2000, for example.
- Certain anti-virus and anti-spyware products may disable components such as the Scripting.FileSystemObject as part of their default settings, and in these cases the WSH scripts may fail silently.