AUTHORING SYSTEM FOR FLEXIBLE AND RAPID DEVELOPMENT OF ON SCREEN DISPLAY (OSD) APPLICATIONS

Mohan K Mohankumar and David M Ihnat
Network Systems Group, Zenith Electronics Corporation.

Abstract

The On Screen Display (OSD) used in Zenith's Cable TV set-top boxes utilize an Authoring System to allow development of control screens, data structures and state definitions. These, when combined with information provided from the RDBMS, define its display and behavior capabilities.

This paper provides an overview of Zenith's ScreenPlay(TM) Authoring System used for the development of downloadable dialogs. Also discussed are the components of the Authoring System with respect to the modular, object-oriented design, and the advantages and limitations of the implementation.

INTRODUCTION

Within the context of the cable set-top box environment, an Authoring System can be defined as a software system that helps developers create multimedia programs or presentations without requiring the painstaking skills involved in traditional programming[1]. Today's set-top boxes, in addition to enabling scrambling and authorizing of video signals, play a key role in providing for interactive TV and information services. An interactive OSD (On Screen Display) application, once downloaded to the set-top box, controls the display and behavior of the box, depending on keystrokes from the viewer and resulting events. Some of the key features provided by the application, (also referred to as a *Dialog* in the subsequent sections), are such capabilities as scanning the program listings, making one-button selections from the listings for display or later recording, etc. The OSD application may be designed with many different *look and feel* user interfaces, as required by the cable TV MSOs. An Authoring System that is user-friendly and flexible is needed to rapidly develop and maintain the dialogs.

The remainder of this paper provides, in order, an overview of the design and implementation of the Screenplay Authoring System. Key advantages and limitations of the project as implemented are described. This is followed by sample screens. Conclusions are then drawn with respect to the experiences of the developers in this effort.

DESIGN AND IMPLEMENTATION

Design Requirements

As the set-top boxes are provided with greater resources, such as a faster microprocessor, more memory, and more complex support circuitry, the capabilities of the software that drives them must grow with it. When the design process was started, it was decided that the following criteria were basic to a successful Authoring System (AS):

- *Ease of Use.* The dialog development environment must be easy to learn and use, requiring only rudimentary programming skills.
- *Standard GUI.* The AS must provide a Graphical User Interface (GUI) that complies with the Common User Access (CUA).

- *Modular Dialog Development.* Provision must exist to permit definition of both internal functions and access to external libraries.
- *Procedural Language.* The AS must provide support for procedural actions for data and screen manipulation. This is accomplished via a C-like script language.
- *Ease of Maintenance.* It must be easy to maintain the dialogs as well as the authoring tools.
- *External Dynamic Data Access.* Provision must exist to permit reference to external dynamic data when developing a dialog.
- *Field Maintainability.* It must be feasible to hand over the dialog maintenance to the operators at the headend.
- *Authoring System Modularity.* The Authoring System components must exhibit a high degree of modular design to ease future enhancement.

Implementation

The Authoring System is independent of any other component of the set-top box control system. As shown in figure 1, it has three active components, the Dialog Editor (**DE**), the Dialog Compiler (**DC**), and the Dialog Assembler (**DA**). These three components may be independently executed on different target platforms, if necessary[2]. Commonly, the DE is the only visible component; the DC and DA are transparent to the users. Dialog developers are only concerned with producing the dialogs, not with how the dialog gets translated to a form that the set-top boxes will understand.

The Authoring System produces, as its end product, a data store which describes the *static component* of the dialog; that is, the display and behavior of all data transmitted to the controller portion of the system. This data store is in a format suitable for input to the download process, which is part of the Information Gateway System. The download preparation process then combines it with the dynamic data from the SQL database(s) to produce a fully functional dialog with static and dynamic components, that is ready to be transmitted.

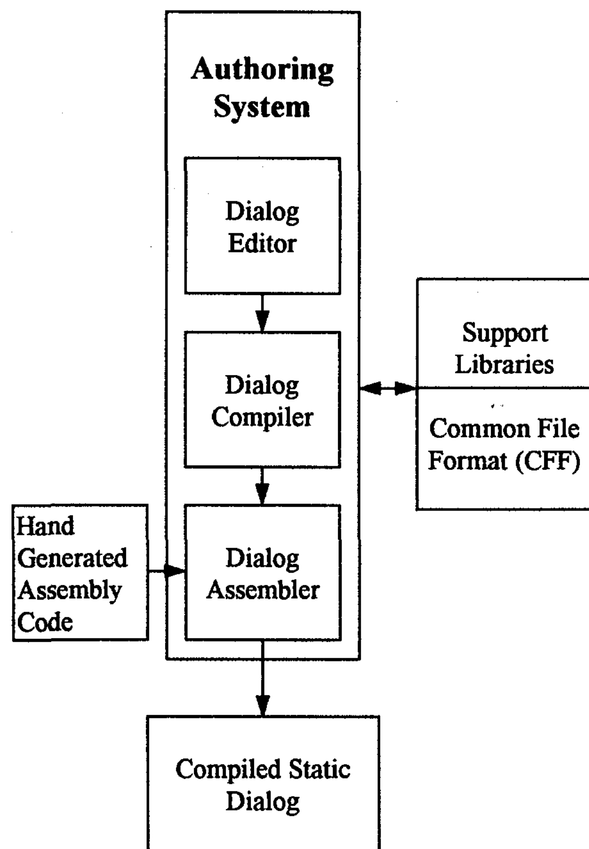

Figure 1. Authoring System

Dialog Editor

The Dialog Editor is a WYSIWYG, ('What-you-see-is-what-you-get'), screen-oriented editor that enables the user to describe an interactive session in terms of multi-screen forms, display and input fields, and actions to be taken on user input or in response to events in the decoder or resulting from downloaded dynamic data and/or commands from the headend. The actions are expressed in a script language

with a syntax reminiscent of 'C'. The output of this an editing session is a Dialog file.

At the time of the initial design, MS-Windows had established a widely installed base capable of running on minimal, standardized IBM personal computers; this drove the decision to implement the Dialog Editor as an MS-Windows application. Originally the entire Authoring System was implemented as a standard 16 bit application. Due to growing complexity and capabilities of authored dialogs, it has become necessary to convert the environment to a full 32-bit model. The eventual growth path is to convert to a full Presentation Manager (PM) application under OS/2.

Internally, each dialog consists of multiple sections, which in turn consist of multiple forms, in an hierarchical relationship. Each form is displayed to the dialog author as a graphic expression of what the viewer will see on the television screen, and set of transitions and/or actions that will be executed for each of the buttons on the decoder, or for desired events in the decoder, such as timers. The displayed form is created, designed by, and fully under the control of, the dialog author. The forms are treated as objects that consist of the display data (picture and literals), and the behavior, that is, the actions associated with each input key stroke or event. The user can define the action sequences for each of the input keys, as well as for predefined classes of events. For instance, this provides the flexibility of defining a single key to tune to a specific channel, start recording or any other actions that may be required; or to set an inactivity timer to return to a viewing state.

Due to the hierarchical nature of the dialog, sections, and forms, default behavior may be specified at each level, with local (i.e., form) responses differing from that defined at a higher level (i.e., section or dialog-wide).

Each dialog may have multiple action routines. An action routine is a common sequence of action statements. The action sequences can be in the form of the script language, which gets translated by the DC, or, it can have the native assembly source embedded for tasks too complicated to easily express in scripting, or to permit performance optimization. Despite the fact that use of embedded assembler code requires thorough knowledge and understanding of the assembler and decoder architecture, this capability provides the knowledgeable user with a great deal of flexibility.

A dialog can have internal action routines that are visible only to that dialog-- either attached to a specific display object, or defined as callable procedures. External assembler action routines may be imported via prototype definitions. (Future versions of the DE will be capable of importing/exporting script routines.) This allows the creation of common libraries of useful routines usable by multiple dialogs, reducing the per-dialog unique development necessary for ongoing maintenance and authoring.

Dialog Compiler

The Dialog Compiler accepts as input the binary data store created by the DE. It then reduces actions to executable code sequences, resolves the tokenized global and local data references from the intermediate form to absolute dataset type and field references, produces the final state transition dataset records, and emits an ASCII Dialog Assembler output data store. Any necessary information pertaining to external data references are provided by the Record Set Definition Export (RSDE) file.

The Dialog Compiler is wholly written in 'C' as a portable application--although currently an OS/2 application, it was successfully compiled and executed on UNIX platforms during its development. Its execution platform, and invocation, is usually totally transparent to the user. This is accomplished via a command interpreter script that compiles and assembles a dialog source file in a single invocation, much in the way the early 'cc' command in UNIX was implemented. This shell script invokes the appropriate components depending on the command switches, passing options along to the correct target component as necessary.

Dialog Assembler

The Dialog Assembler accepts as input ASCII files expressed in terms of the Dialog Processor Assembler Source. This is in the form of a traditional assembler, e.g., opcodes, operands, labels, pseudo-ops, etc. Relocatable and link-editable formats are not supported; all references must be satisfied by local declarations or globals provided from the RSDE data.

It is implemented as a one-pass non-relocatable assembler; as such, all definitions and references must be contained in the primary and included files at assembly time. It supports an indefinite number of forward references, however, which are resolved via a patch look-up scheme as opposed to the more traditional two-pass approach.

Input to the DA is either the output of a DC execution--which is therefore the result of a DE session--or explicitly coded assembler source. If the latter, it is the responsibility of the author to provide for any external definitions required as described in the RSDE file. Explicitly coded assembler source requires thorough knowledge and understanding of the assembler and decoder architecture.

Output is in the form of a binary file suitable for processing by the Download planning and transmitting components of the Information Gateway.

Support Libraries

The three components DE, DC and DA share the information concerning the dialog and data. As a result, there are well-defined in-memory and data store structures for each of these interface requirements. These interface definitions are expressed via common support routine libraries provided to simplify processing of shared data and to assure identical views of such data.

The Common File Format (CFF) library allows creation of arbitrary data sections within a binary file; the contents of each section is unknown by, and irrelevant to, the library manipulation routines. Thus, this mechanism provides an operating-system independent program-level means of archiving information that is logically related, but must be handled in discrete separate sets.

The routines provided by the CFF library, as might be expected, are oriented to providing easily-used performance of the following tasks:

- Creating the CFF file
- Retrieving and manipulating existing file sections and data
- Adding and deleting file sections
- Providing information about the current file status and contents

Dynamic Data

The RSDE library provides the callable routines necessary to extract the data associated with various fields, and to construct the ASCII datastore which permits

export of this information to the Dialog Editor. It also provides support routines to import data from this file into 'C' structures, and to populate the DE symbol table.

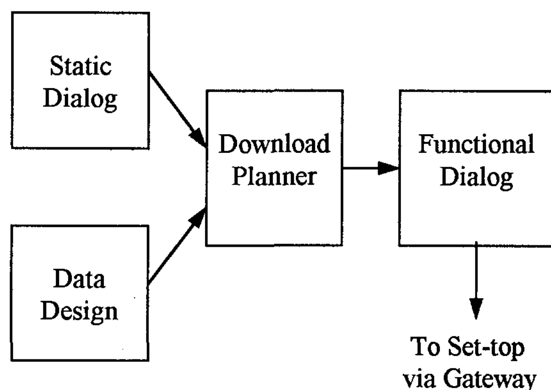

Figure 2. Preparation and Download

Data Design has the information on the dynamic data recordsets. Download planner, which is a part of the downloader, combines the static dialog created by the Authoring System, with the dynamic data and produces a downloadable fully functional dialog as shown in figure 2. The packet sequences are also prioritized.

Authoring System in the Headend

The Authoring System is implemented as a sibling to the Information Gateway System at the cable plant headend, as shown in figure 3. Information Gateway System is the main link for the Authoring System. Detailed description of the Information Gateway System is beyond the scope of this paper.

Other data sources typically include program guide data and sports and weather information. Once a dialog is created by using the Authoring System, the compiled output is sent to the Gateway, which then gets the dialog prepared and starts transmitting to the set-top boxes.

As and when the Gateway receives any new data, which may be an entirely new dialog or new dynamic data, the downloader transmits the incremental update, which is totally transparent to the user. That is, the subscriber of the set-top box is not aware of when and how the data is received. The Gateway also provides the capability to altogether reset the set-top boxes and start afresh with a new dialog.

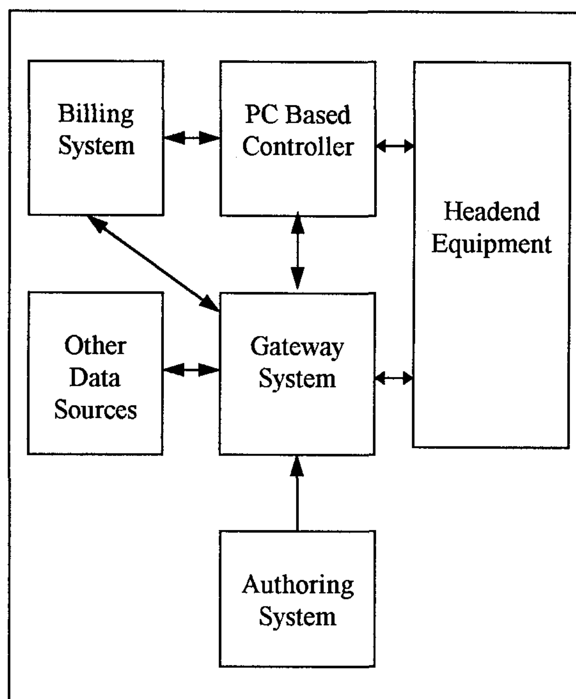

Figure 3. Typical Headend

ADVANTAGES AND LIMITATIONS

As with any software system, the experience of development and use has reinforced some of the early design decisions, and pointed out some of the oversights and shortcomings in others.

Advantages

- *Customer Acceptance*. We have had excellent response from our customers who are currently using the Authoring System.

- *Rapid Prototyping.* As hoped, even without underlying external data for dynamic displays, the tools have proven useful and usable by Sales or Marketing personnel. Non-programmers can, and do, create dialogs using their laptops running MS-Windows or OS/2 without having to compile and assemble the dialogs. These may be viewed for acceptance and modification by customers, and later compiled, assembled, and downloaded for on-set review.
- *Reduction in Required Developers Skill Sets.* Prior to the introduction of the Authoring System, dialog developers were required to be software developers. Experience has shown that personnel with minimal traditional software development experience can successfully produce functional dialogs. Despite some concerns that the state-driven model of the set-top decoder might present problems, this has not proven to be an obstacle. And for those tasks requiring greater skill sets, fewer skilled developers are required to intervene.
- *Flexibility.* The ready ability to define the keys to suit the needs of the headend, of different customers, and even of different dialogs for the same customer, has proved to be an easily exploited and useful capability. Rapid modification and extensibility of the dialog, even to the extent of major logic redefinition, has proved to be a benefit of the system.

Limitations

- *Hardware Dependence.* Despite the intention of virtualizing the interrelationships between hardware and the Authoring System, there are still too many dependencies at the Dialog Editor level on underlying capabilities of the display hardware.
- *Need expertise in MS-Windows API to maintain the Editor.* The time needed to develop, debug, and modify the MS-Windows code proved to be one of the greatest bottlenecks during the development effort; plus, skilled MS-Windows programmers are at a premium. In retrospect, a solid GUI application generator with support for multiple platforms would have reduced the workload significantly.
- *No support for relocatable and link editable formats.* This was designed as a follow-up capability to the basic tool; 'hooks' were built into the initial release to facilitate this. Development of the first few dialogs pointed out the importance of this capability; it should have been an integral feature of the initial release.
- *Script Language Implementation Incomplete.* In a common problem with object oriented systems allowing attachment of script or code to objects, it's difficult for authors unfamiliar with a dialog to garner a complete picture of the actions embedded in the dialog without extraction and printing tools. We realized the lack of this feature and plan to implement it in the upcoming version.
- *Lack of Emulation.* The debugging environment received far too little attention in the original plan. Particularly painful in its absence is an emulation environment which would permit debugging of complex dialogs without assembling the panoply of equipment necessary to support a functional download of static and dynamic data. We plan to overcome this by adding an emulation feature as part of the Authoring System.

SAMPLE SCREENS

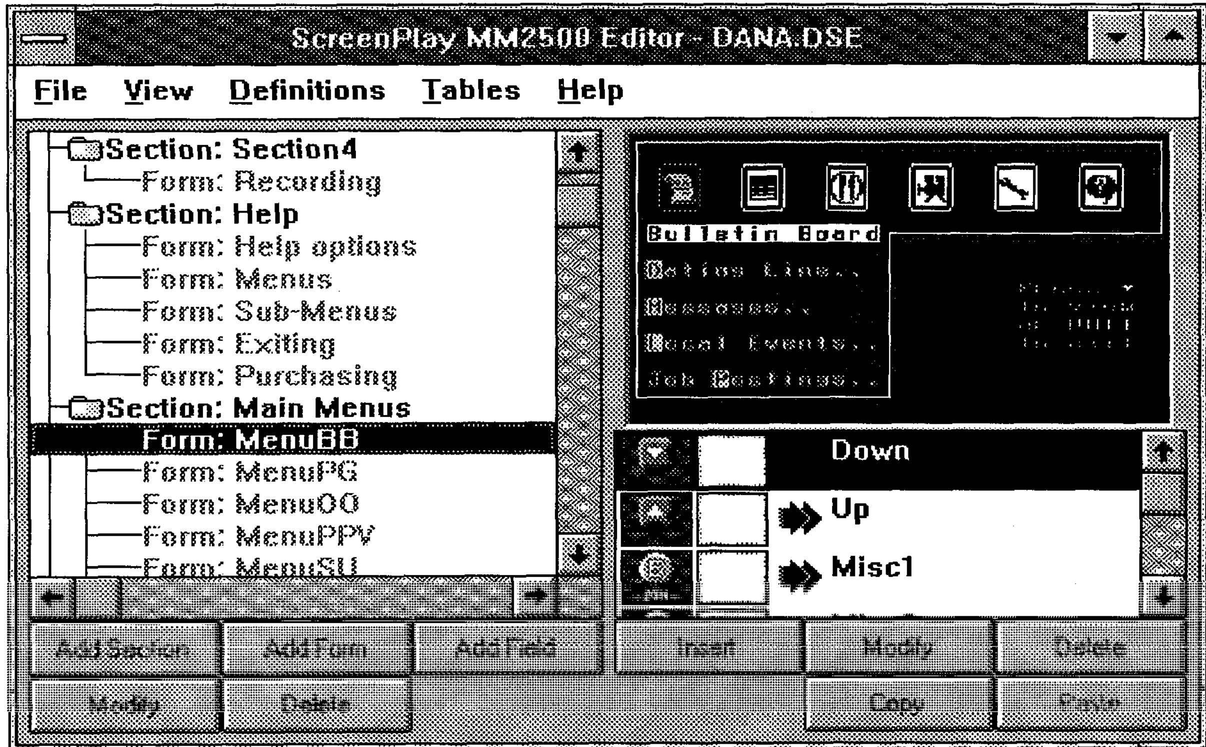

Figure 4. Main Screen

Figure 5. Form Editor

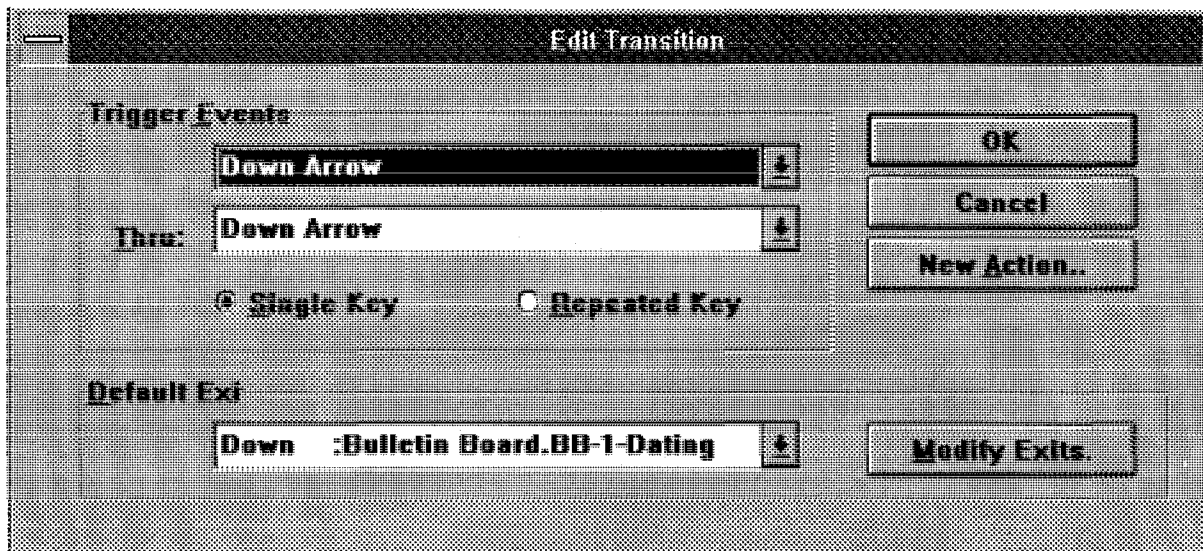

Figure 6. Actions for an Input Key

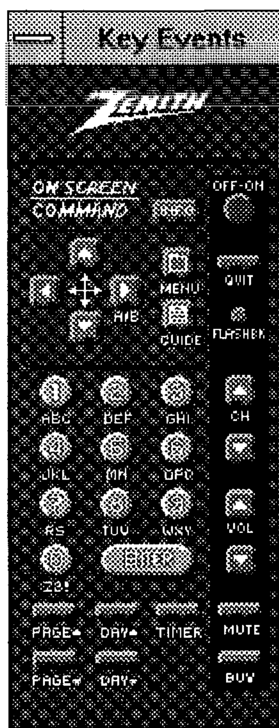

Figure 7. User Remote

Figure 4 shows the main screen of the Dialog Editor. It shows an opened dialog which has multiple sections and forms defined. The highlighted form MenuBB is being currently modified by the user. The user can design the screen display and define the transitions for the appropriate input keys. A transition is basically a sequence of actions to be executed for an input key. This may be as simple as switching to a different form or executing more instructions.

Figure 5 shows the form editor which lets the user paint the screen as it is intended to be displayed on the TV. The screen area is divided into multiple cells. Each cell may have its own attributes, like background color, foreground color, blinking etc. Each form may have the video on or off. Any lines on the form may selectively be enabled or disabled from being displayed.

Figure 6 shows the action(s) to be taken on a particular input key on the remote, in this case, the Down Arrow key on the remote. Figure 7 shows the replica of the remote control. The user just has to click on any key when defining an action sequence for that key.

CONCLUSIONS

The design approach is sound. An Authoring System should enable the users with less programming skills and minimum training, to create dialogs. The Screenplay Authoring System which keeps evolving, does achieve this primary goal. In this implementation process, a significant amount of custom software for the tools was developed , a major portion of which can be reused. In retrospect, the use of some off-the-shelf software packages would have saved some time and effort.

It is obvious that the interactive TV is the wave of the future. It is going to be increasingly important to provide the tools that are easy and flexible to develop the interactive TV applications. It should be possible for marketing experts to decide what they want the TV screen to display without having to become technology experts. By handing over the maintenance of these dialogs to the cable TV MSOs, the system provides them with the control and flexibility.

ACKNOWLEDGMENTS

Many thanks to our colleagues in the cable TV software engineering department for their review of the paper and valuable suggestions. Special thanks to Winston Tsao for his help in preparing the sample screens.

REFERENCES

[1] John Adam, "Interactive Multimedia", IEEE Spectrum, March, 1993, p23.

[2] David M Ihnat, "Authoring System Component Overview", July, 1992, Unpublished Technical Memorandum.