FLEXIBLE DATA STRUCTURES AND INTERFACE RITUALS FOR RAPID DEVELOPMENT OF OSD APPLICATIONS

Caitlin Bestler, Manager Control Systems Design
Zenith Cable Products, Division of Zenith Electronics Corporation.

Abstract

On Screen Display (OSD) used in CATV subscriber set-top decoders can be used for many different interactive viewer information services such as Schedule Guides and Sports Scores. Allowing for the required flexibility and functionality of Interactive Information Services, an OSD decoder system must use flexible redefinable data structures and interfacing rituals. This mandates downloadable behavior and data , not just downloadable screen images.

Decades of Information Systems (IS) software development on mainframe and personal computers have shown that mere reprogramability is not enough. IS applications must evolve almost constantly. Staying responsive to user needs while avoiding development bottlenecks requires that IS systems be built from standard parts customized by parameterization and/or non-procedural specifications rather than custom hand-crafted code. Examples would include Relational Databases and Application Generators.

These IS productivity techniques can be applied directly in headend computers, and scaled to fit within the OSD decoder. Zenith's HT-2000 decoder system applies both techniques to rapidly develop and then deploy Interactive OSD Information applications.

SUPPORTING OSD INTERACTIVE INFORMATION APPLICATIONS

A typical first exposure to On Screen Display capability is a VCR. Cryptic flashing lights on a control panel are replaced by cryptic text instructions displayed on the screen.

To be fair, the text messages aren't all that cryptic. They are just unfamiliar, and expert consumers had already learned how to do everything with the control panel.

For non-expert users, the VCR's features are more accessible. There just aren't any new features. Existing features just had a new, more "user friendly", front end.

While an OSD set-top decoder can certainly be made more user friendly, the real potential is in entirely new features such as Schedule Guides. These new Interactive Information Services can be standalone, or integrated with video programming.

An interactive OSD information application allows the viewer to obtain specific information when they want it. Selection and timing is under viewer control. The requested data is presented on screen, possibly on top of specific video programming.

When designing Zenith's HT-2000 decoder and its headend computer, the OSD Information Gateway, several requirements were identified. Each is discussed in one of the following sections: The Need For Flexibility, Downloaded Data, not Images, and Integrated Control

These points led to the conclusion that a downloadable decoder was needed to allow easy development and evolution of new applications. It was also important to allow for easy deployment of new applications.

THE NEED FOR FLEXIBILITY

Correctly predicting exactly what information is required for Interactive Information services is next to impossible.

For example, a Schedule Guide application has far more open questions than you might expect:

- Will viewers want to find movies by theme and/or actor? Will the data be available?
- Should the schedule be presented in a two-dimensional grid, or a tabular listing?
- Will viewers want information on all channels, or only the movie channels, or only the PPV offerings?
- How much information do viewers want for each movie? The title, two lines, four pages, the actors, the director? Do they want to search by date, rating, price range, director, theme and/or copyright date?
- Would people prefer a long schedule with minimal detail, or a shorter one with more information? Which is worth more to them?
- Should PPV offerings be listed with Pay TV movies or separately? What about movies on non-pay channels?
- How would a staggered start Video on Demand channel be shown in the Schedule Guide? What about a pure Video on Demand service?
- Is the Schedule Guide a premium service, or a method of promoting PPV?

Moving past the schedule guide, what other services will be required: stock quotes, horoscopes, sport scores? Will Baseball fans want just the final score, or a complete box score?

Even more complex than identifying the information to be presented is deciding exactly how it will be presented and which keystrokes the viewers will use to access it.

A well defined user interface combines User Rituals with User Myths . The user rituals are patterns of input required to do certain things. Pressing backspace to erase the previously typed character is a common computer user ritual.

A user myth is an explanation, in user terms, of what each input key or sequence does. Clicking the left mouse button in a certain screen region is "pushing a toggle button".

Consistent user rituals and myths make an interface easy to work with and understand. An interface that requires raw memorization of arbitrary input and output sequences is very difficult to learn and user unfriendly.

Predicting in advance what rituals viewers will find difficult, and which they will find frustrating is even more difficult than knowing what information services they want.

It would be impossible to correctly define all of the information and interface rituals required for these various services. Even with something as 'obvious' as the schedule guide, the answers are just not reliably available.

You don't know. I don't know. That marketing consultant who wants to sell you the answers doesn't know either. None of us **can** know for the simple reason that the viewers themselves don't know.

Conducting a survey won't do any good, no matter how large your sample is. The viewers can only give you their **guesses**. The viewers themselves won't know until **after** they have started using these services.

A survey conducted before the introduction of the remote control might have projected little interest in remote controls, since people only change channels a few times an hour. "Zappers" did not exist.

We are attempting to provide tools for viewers. As with those who designed automotive or computer input devices, we can only propose options. We must wait to find out which options consumers will find useful and/or become accustomed to, and which they will find annoying.

Accommodating consumer interests requires flexibility, and constant adaptability. Would a supermarket manager buy a check-out system that dictated how merchandise be shelved for the next ten years? Why should a Cable Operator accept an OSD decoder that locks in the format of the Schedule Guide and other OSD services?

The nature of the information displayed, the format it is displayed in, and the interactions the viewer goes through to access them will all **need** to change during the lifetime of any OSD decoder.

To meet these needs we must be able to actually redefine the behavior of the decoder from the headend without modifying the decoder.

DOWNLOADED DATA, NOT IMAGES

How the OSD decoder receives and **uses** its information is critical to allowing flexible creation, and evolution, of these and other user-friendly features.

A Schedule Guide, for example, could be viewed as nothing more than many pages of schedule information. Rather than waiting for the information to scroll by, the viewer can now Page Up and Page Down on their own.

Doing so would sell the potential of an OSD decoder short. Separating data reception and storage from display allows flexible efficient implementation of many desirable features. Examples are given in the following sub-sections.

Tiering

Services such as Sports are likely to be tiered. Only subscribers to these services would be able to display this data.

Even within a given application, there could be levels of service offered by tiering. A "basic" Schedule Guide might only provide detailed movie descriptions for tonight's PPV offerings. A "premium" Schedule Guide tier would provide complete descriptions of all movies.

Since decoder RAM space will always be limited, it would be desirable to have the decoder only store data for which it was authorized. For a given RAM capacity the decoder would be limited in what tiers it could be authorized for, not in what tiers were available to it.

Conditional Display of Data / User Filtering

Unwanted information is clutter. It gets in the way of valuable information. The information displayed should adapt to individual viewer preferences. Insisting that every household receive detailed movie descriptions for an Adults Only service would probably not be desirable.

You may view a Schedule Guide as a value added service or as a promotional device. In either case information about channels a viewer will **never** want to watch is undesirable.

If the Schedule Guide is viewed as a premium service, an annoying one will not be worth as much. If the Schedule Guide is a promotional feature, you want the viewer to concentrate on promotions for things they are likely to buy.

Multiple Indexing of the Same Data

In many applications the same data could be found in different indexing orders or via alternate User Rituals.

The same movie may be found in a PPV index, in a Schedule Grid, a time-oriented listing, a channel-oriented listing, a theme index, an actor index, or even a Director index.

The box score for the Chicago White Sox vs. California Angels game could be reached via either "Chicago White Sox" or "California Angels".

Redundant Display for Convenience

Sometimes an application displays information it normally edits on another screen for the viewer's reference.

The fact that a channel is locked out via Parental Control should be displayed not only on the Parental Control screens, but on the Schedule Guide display as well.

Programs scheduled for automatic taping might be flagged in Schedule Guide grids and listings.

INTEGRATED CONTROL

Schedule Guide data should interact, not just be a passive display. The viewer should be able to do things with it. While browsing through a Schedule Guide the viewer should be able to select a given program and then do any of the following:

- Request a more detailed description.
- Tune to that channel immediately.
- Request an IPPV purchase of that program.
- Schedule an automatic taping of that program.

When tuning to a new channel, the decoder could display the channel number, name, and information about the current program.

Parental Control and Favorite Channel maps could reference channels by name. Parental Control could be extended to lock-out or exempt specific **programs**, rather than whole channels.

Opinion surveys and/or home shopping applications could use the two-way transmission capability to interact with the headend.

Application Co-existence

While working on each separate OSD Information Application it would be easy to forget that the Viewer is likely to view their set-top Decoder and television as one unit. And the primary purpose of that unit is to watch television, not check stock quotes.

One aspect of "integrated control" is the ability to toggle between the OSD Information Application and watching the viewer's program.

Remote control layout and software conventions should make it easy for the user to flip in and out without having to start over from scratch every time they re-enter an application.

Conditional Access

Any Schedule Guide application should be integrated with the decoder's Conditional Access system. It should be able to display IPPV ordering instructions, transmit two-way IPPV orders, and give viewer feedback on confirmed IPPV authorizations and authorized subscription channels.

Virtual Channels and Interactive Television

Many applications require "virtual channels". In these applications the decoder is tuned to an alternate channel without bothering the viewer with the details or changing the displayed channel number.

Examples would include Barkering, staggered start PPV Video "On Demand", interactive television, video back-drops for messaging systems, and targeted advertising. The ability to tune the decoder becomes just one part of the OSD applications "message".

DON'T RE-INVENT THE WHEEL

While the information, and viewer rituals for accessing it, are unstable, the possible sources of this information are even more varied and subject to change. A flexible OSD Decoder Information Service must display unknown data in unknown formats that is originally captured by Headend equipment from unknown sources.

Responding to rapidly changing highly flexible requirements is not an easy process. Making the OSD decoder downloadable allows new services to be developed after the decoder is already in the field. It does not make developing those services any easier.

We need a way of developing and evolving new decoder features. We have to be able to integrate the new features with existing features. Lastly, we have to be able to test and deploy the new features without constantly disrupting customer's use of the decoders and the features they already enjoy.

These are not new problems. They have plagued software development for several decades now. Methods of coping evolved to deal with these problems include Prototyping, Relational Database Management System (RDBMS), Report Generators GUI Screen Painters, and Revision Control.

Rather than re-invent the wheel, we should examine these solutions to see what lessons can be applied to the developing Interactive OSD Information Services.

Prototyping vs. Specification

As a rule, Software developers are much better at correctly implementing something than we are at knowing what it is we should be implementing.

The most serious "bugs" are not incorrect algorithms. They are incorrect features. The software functions "perfectly." It just doesn't do anything that is of any particular use to anyone.

The first attempt at solving this problem was **The Specification**. The Specification has taken many forms: long narrative descriptions ("Victorian Novels"), contract-like "rules," flowcharts, data-flow diagrams, structure charts, data structure diagrams, state transition diagrams and the latest fad: object oriented diagrams.

Software Developers can spend a great deal of time debating the relative merits of **Methodologies**. A Methodology specifies how The Specification is developed, what must be in it, and how it is translated into a working system.

Methodologies are most often compared and contrasted with a near religious fervor. However, they almost all make one critical assumption - the "User" already knows what is required.

Evaluating interfaces described on pieces of paper is a very difficult task. The format of the Specification, let alone which icons are used in what Diagramming conventions hardly matters if the user is only reporting their hunches.

When a user has never used a system exactly like this, there is really no alternative but to let the user actually test a prototype of the system. Nothing can match a 'hands-on' test drive of the actual interface for identifying problems with it.

Most interactive user interfaces are now prototyped and/or developed with tools that allow the displays and input rituals to be rapidly updated.

Prototyping and/or rapid deployment of modified interfaces is extremely vital to the development of Interactive OSD Information Services for several reasons:

- A screen that looks fine when sketched on a paper memo, or even drawn on a PC graphics program, may look terrible when shown a real television set.
- Viewers interact using a Remote Control and/or set-top keypad. These are very different from a PC keyboard.

Relational Database Management Systems

Relational Database Management Systems (RDBMS) manage data for IS applications. Professors build entire careers out of debating the fine points of Relational Database theory with each other, so I'm not certain how much can be explained in a page or so.

A RDBMS organizes data into Tables. Tables are said to have Rows and Columns.

Each Row is a record, or one instance of data. A "Programs" table would have one row for "The Empire Strikes Back".

Each Column represents one thing that is known about each instance. It is an attribute of each record. Columns for the "Programs" table could include "Title" or "MPAA Rating".

Tables do **not** include any "repeating" data. You cannot have a "Stars" column that lists up to six different stars. Instead a separate table lists **each** star for **each** movie.

Tables reference each other only by value. Some of the Columns in Table X can be used to search Table Y. Table X does not have anything like a pointer or Record Number for Table Y.

These conventions avoid data that is massively entangled both with code and itself. The dependencies have been limited and cataloged. This separation allows migration to new definitions of the systems data without having to rewrite **every** piece of code that uses that data.

An OSD decoder can benefit greatly from similarly standardized data structures. Headend computers supplying data to the OSD decoder, such as the HT-2000 system's OSD Information Gateway, can use an RDBMS to store the original data and to map its translation into the downloaded data.

Report Generators

Report Generators use RDBMS standardization to allow reports to be specified in a non-procedural fashion.

A Non-procedural report specification states what information should be in the report, and how it should be formatted. It does not specify **how** the data should be retrieved, sorted and formatted.

When applicable, specifying something in non-procedural format has proven to be far faster and reliable than writing procedural specifications.

Non-procedural specifications also allow hardware upgrades. The same results can be achieved under new hardware. The same procedures may not translate as easily. Later OSD decoders are likely to have more RAM and higher resolution display devices. These decoders would co-exist with older models. Non-procedural specifications will be shared.

GUI Application Generators

GUI (Graphical User Interface) applications that run under such platforms as Windows and Presentation Manager can be very difficult to code. GUI Application Generators slash development time by allowing non-procedural specifications to combine standardized components for a new interface.

Use of common building blocks is not just a convenience. It is **desired** for its own sake. Conventions such as list boxes, drop-down combo boxes, checklists, pull-down menus and radio buttons are valuable not only because they reduce coding time but because they make it easier for the computer user to learn how to use the application.

Maintaining a common "Look and Feel" for OSD Information applications is even more important than for PC applications. Viewers have less motivation to work their way through a strange input ritual. While being flexible, the development system for OSD Information applications should encourage use of standard input and display mechanisms.

Revision Control

Even before software developers identified the need to prototype systems, they knew that systems already in use had to modified.

Developing systems is hard enough. Modifying systems that are already in use is considerably harder. Many of these problems are still relevant to OSD information applications:

- Interfaces must evolve. Even new features should feel familiar to old users.
- User edited data should be maintained.

- For many systems operations must continue even while the system is upgraded.
- Many items may change for a new feature: the data sources, how data is stored, and the code to use the data. Changing all of them **at the same instant** is unrealistic.
- New code sometimes blows up, requiring an immediate **rollback**.

SCALING THE WHEEL TO FIT

RDBMSs and the other techniques described are all very powerful tools. Re-inventing the wheel is always a waste. But a set-top decoder with an 80386 is overkill, expensive overkill.

As currently defined and implemented, these tools would not fit in a set-top decoder. The wheel already exists, but it doesn't fit. However the approach does not have to be abandoned, just scaled to fit.

The HT-2000 decoder would need a processor capable of accessing considerably more than 64KB of RAM. No matter how you compress it, you can't fit much of a Schedule Guide in 64KB, let alone any other features.

Additionally this processor would have many other responsibilities:

- Accept input from keyboard.
- Accept input from IR remote.
- Control the OSD chip.
- Tune the decoder.
- Control other outputs: IR out, LEDs, Volume, video blanking, audio mute, any two-way transmitter and any expansion port.
- Manage a small amount of self-edited data. This data would include favorite channels and a user PIN for IPPV purchases and Parental Control.
- Interface with the Conditional Access system. To enhance security this was kept a separate module in the HT-2000 decoder.

The processor would not have to do anything that complex with the data. There are no square roots, or regression analysis to be performed. Vast processing speeds are not needed, just the ability to do simple things with vast amounts of data.

The most powerful affordable candidates were derivatives of 8-bit processors that could look at anywhere from 512KB to a few Megabytes. However it is typically only visible in 64KB chunks. This increases memory management complexity. Many had built-in help for IO interfacing, such as UARTs.

Downloading Safely

A downloadable OSD decoder's behavior is controlled by the data packets sent to it from the headend. The "code" it is executing is updated over the cable downstream, rather than by distributing new ROMs.

On the HT-2000 project the downloaded behavior is called the "Dialog". Once a Dialog has been written it would remain in use indefinitely. This might be a few days, a few weeks, or a few years.

The other data downloaded is the Dynamic Data. This data changes on a daily basis, or possibly more frequently. Schedule Guides, actual weather information and sports scores are all Dynamic Data.

By contrast, the Dialog is what knows how to use the Dynamic Data.

Downloading dynamic data is relatively simple and risk free. The worst that can happen is that the information is wrong. The information has a limited lifespan. It will be updated and/or deleted soon.

Errors are also relatively innocent. If a movie title is "Star Warx" a few viewers will notice and get a minor chuckle.

Downloading dialogs is considerably riskier. An undetected error might cause the decoder to do something various obnoxious such as jamming the volume to full or refusing to tune where the viewer want to tune.

It is imperative that Dialog downloads be as reliable and resilient as possible.

Downloading dialogs over a cable plant to thousands, or even hundreds of thousands, of decoders presents several problems.

In a one-way plant there is no way to acknowledge a successful download, or request a new download. Even in a two-way plant the upstream capacity might not allow each decoder to individually acknowledge the download.

Bad downloads are unavoidable. Whatever is downloaded can and eventually will be garbled at one or more points in the distribution chain:

- Error detection codes are very powerful, but only a partial solution. Fewer than 1 undetected error in 10,000,000 sounds very secure. But if you have 200,000 decoders talked to 1000 times a day it means 20 irate customer calls each and every day.
- Operational errors can occur. The dialog may have been restored onto the headend computer from a faulty tape or floppy disk.
- Software errors are inevitable.

If bad downloads are unavoidable the question becomes, can you fix it? The only way to **guarantee** that is to ensure that **nothing** you can download could prevent the decoder from accepting **another** download.

This requires that the OSD decoder essentially have "two minds". One accepts downloads. The other **is** downloaded. Nothing the second mind does can interfere with the first mind's ability to accept further downloads.

There are four valid approaches to this:

- Some processors allow two virtual programs in the same processors. Usually dubbed Supervisory and User mode, these processors ensure that User mode code cannot interfere with Supervisory code.

  This would be a perfect solution, except that this feature is generally not available in the processors affordable enough to place inside a set-top decoder.

- Two processors. This is essentially a hardware simulation of the above. It takes more power and is nearly as expensive.

- An External watchdog timer could force termination of RAM stored code and return to ROM based code. This is more feasible, but still costs money and takes up board space.

- Interpretive code. The downloaded behavior does not truly gain control of the processor. The ROM based code simulates a processor that executes the downloaded code.

Interpretive code was chosen for the HT-2000 decoder. In addition to its safety features, interpretive code offers other advantages:

- Processor independence. Later versions of the decoder can be implemented on a more powerful processor, possibly with extended instructions while maintaining full backwards compatibility at the binary level.

- More compact downloads. Because the "machine code" is designed for OSD applications it can be more compact than native machine code would have been.

- Interpretive code can be restricted from updating data under control of the headend computer. This would be more difficult under the other solutions.

- Interpretive code can have other safeguards, such as ensuring that it will listen for fresh user input regularly, in addition to being re-downloadable.

- Hardware dependent code, such as device drivers for handling input and output, can be placed in the interpreter. Essentially, the interpreter becomes the equivalent of a PC BIOS. Later revisions of the hardware can handle hardware interfacing differently without having to recode the downloaded applications.

State Driven Display Painting

The decoder's display painting primitives should support non-procedural generation of displays.

It should be optimized to allow static receptive elements to be specified easily and efficiently, while still allowing complex data dependent displays.

The HT-2000 decoder decides what to display by a current Dialog State. In a given state the decoder will have a recognizable display and respond to inputs in a specific way.

Making display logic state based helps reinforce an important principal of graphical user interfaces: if the program/system now **acts** differently it should **look** different.

Basically a state is what the viewer would think of as a given display. The data displayed there might change, but it is recognizable. One state can be "typical schedule guide page". All pages of the schedule guide act the same way, and display the same type of data in the same format. Only the specific data is different.

The viewer will recognize a Schedule Grid as a given display no matter which page they are on, or what day they tune to the Schedule Grid. It has a certain "look", and the viewers will know what they can do when the screen has this "look".

DATA MANAGEMENT PRIMITIVES

A full RDBMS is clearly beyond the horsepower of any 8-bit processor. A more realistic set of standard data management capabilities would have to be selected.

Many factors had to be considered:

- ROM space was even more tightly constrained. Device interfacing, such as IR input handling, would have first claim on ROM space. Standardized data primitives would have to be implemented in very compact simple code.
- There will always be more uses for RAM. Feature tiering will require different decoders to hold different data.
- The decoders were in a heavily distributed environment. Most customers would be running one-way Cable plants. This meant that updates would not be acknowledged.

Certain RDBMS features were too expensive to implement in the decoder. We found ways to do without those features, or to provide the same service in the OSD Information Gateway rather than the decoder itself.

An RDBMS allows the same data to be fetched in many different formats. While this allows portions of the application to view the data as appropriate to its needs and how the data was defined when it was coded it does carry a heavy run-time penalty.

This penalty is high enough that most RDBMSs allow the source code to be compiled to match the actual data format, rather than binding at run-time. There would certainly be no need for run-time binding within the OSD decoder.

RDBMSs also allow the data format to be redefined without losing existing data. You can add new columns to a table, or re-arrange existing columns, without losing any current data.

The code to support this in the decoder would simply be too complex. Instead we decided that the original data would always be stored in a RDBMS on the OSD Information Gateway. The new data would simply be re-downloaded in the new format.

Many applications have data that expires at a specified time. Schedule Guides are just the most obvious example.

When all the decoders are deleting the same records it actually makes more sense for the headend to tell them to delete those records.

In this way "garbage collection" code can execute in a more powerful headend computer, rather than in each and every separate processor.

Record Sets

An RDBMS minimizes inter-dependence of records by using "content" addressing. An employee record does not have a pointer to the physical location of their Department. Instead it has a key value that can be used to search for and find the Department record.

This sort of capability is crucial to a distributed database system. It was scaled down for the HT-2000 decoder by turning RDBMS tables into **Record Sets.**

Each Record Set contains many fixed length records. Each record is much like a record, or row, from a RDBMS table.

Each record would also contain many fields. These are essentially the same thing as a RDBMS column or attribute.

The downloaded dialog simply asks for a specific record from a specific record set. That record may be in different locations for different decoders. Indeed if memory capacities have been varied based on individual option tiering, they will be. The application code does not need to understand this.

The size of each record, and the number of records allowed is defined for each Record Set. Additionally the record set may have an optional key field used to sort all of its records with.

Sorted Record Sets are kept in sorted order. Updates are merged into the Record Set based on the key value at the beginning of the record. Old records, such as yesterday's schedule, are deleted by key range.

Sorted records sets are used to index with user meaningful data. Schedule records sorted by date, time and channel would be a prime example. The downloaded dialog looks for the Schedule Record for a specific channel and time, not a specific memory location.

Unsorted record sets are really just optimized sorted Record Sets. They have a one or two byte 'key value' that is not physically stored with the record. Instead it is implied by which slot of the record set the record is placed within.

Unsorted record sets are most useful when dealing with things that can be numbered in a tight range. Examples would include visible identifiers such as Channel numbers, and internal identifiers such as Program Numbers. Internal identifiers would be obtained from sorted Record Sets.

Splitting the schedule information like this into two separate Record Sets allows the same Program Information to be used for a single movie no matter how many times it is in the schedule.

Placing data in the database **once**, no matter how many places it is referenced from, is one of the most crucial aspects of Relational Database theory. It is critical to efficient data handling and standardized handling of distributed updates.

The Record Sets of the HT-2000 decoder maintain this essential simplification with only a handful of easily implemented data manipulation primitives: virtual memory, record sorting and direct indexing.

An Example Schedule Guide in Record Sets

Suppose that schedule information is available from an outside source. The records each describe all of the times a particular channel shows a given movie. A record might be formatted as follows:

- The Service Name.
- The Movie Title.
- A comma separated list of actors.
- The Movie Length.
- The dates and times it will be shown.

This data would be normalized into OSD Information Gateway RDBMS tables:

- Each **Service** row would specify an available service.

- Each **Movie** row would give information on one specific movie no matter how many times it was shown.
- A **Showing** row would specify that Movie **X** was being shown by Service **Y** at a specific time. If the same movie was shown seventeen times there would be seventeen Showing rows referencing it.
- A **Starring** row would specify that Actor **R** appeared in Movie **X**. If Movie X had five listed stars there would be five Starring rows referencing it.

During the final translation into Record Sets we would attempt to further compress the data. Long keys values used as foreign keys may be replaced by short keys that index into an unsorted Record Set, for example.

Referential Integrity

One of the key features of a RDBMS is **Referential Integrity**. This feature ensures that if a **Pending Order** record has a **Customer Num** and a **Part Num** that there are matching **Customer** and **Part** records identifying who that customer is and what the part is.

The OSD decoder's data primitives must ensure that a **Schedule Record** reference to a **Program Num** is not referencing a non-existent one.

The highly distributed nature of a one-way Cable system presents a major challenge here. Unlike an RDBMS, the OSD decoder cannot just reject a Schedule Record that references a bad Program Number.

In a one-way Cable plant the OSD decoder has no way to complain. Putting up an OSD error message to the effect of "Illegal foreign key in update at 13:47 on 4/21" would be extremely user unfriendly.

Instead, we have to ensure that the Program record is added before the Schedule record by controlling the order in which the OSD decoder will **accept** them.

INFORMATION DISTRIBUTION

The OSD Decoder is just the last step in a complete OSD Information Service. Many other components are involved in a complete system.

Zenith's HT-2000 system has the following components:

- The **OSD Information Gateway** is responsible for downloading the OSD decoders. This requires it to collect all of the data to be downloaded from various sources first.
- Various **Data Providers** supply data to the OSD Information Gateway. Many of them are supplying the data in a standard format, rather than talking specifically to the OSD Information Gateway.
- A separate **Conditional Access Controller** manages Pay TV security. It is likely to have associated equipment such as Encoders and Receivers. Because HT-2000 was built upon the existing Z-TAC system, the Controller had to remain separate to provide continued support for customers who had not yet fully converted to HT-2000 decoders.
- The Cable Operator presumably has some form of **Management System** which is typically a separate computer. Typically the Management Computer would be connected to **either** the OSD Information Gateway or the Conditional Access Controller, but not both. Commands would then be routed to the correct computer.

Data would first be captured or accepted from a Data Provider. Conceivably the Management System could also provide some data.

Some configuration data, such as Channel line-ups would be entered directly on the OSD Information Gateway using GUI front ends.

From either source the data would be placed into RDBMS tables.

Reports, developed using Report Generators, could be run on this data.

GUI front ends, developed using GUI Application Generators, allow review and correction of any errors in the data. While data would typically not be hand inspected, the Cable Operator should always retain the ability to review any information before it is sent out to their customers.

The data may then have to be **prepared** for download. This process may involve some compression of data. For example, the same phrases may be found in many different movie descriptions. In order to preserve decoder RAM space the OSD Information Gateway will attempt to minimize the number of separate times it downloads the text "thrilling adventure" or "heartwarming romance."

The data is then **exported** from the database. This involves final translation of the data into Record Sets and the required download packets.

The data is received by the decoder.

The viewer may then have the dialog display the desired data.

CREATING NEW SERVICES

Developing a new interactive application for an OSD Decoder would involve several steps. The exact ordering of these steps, and some of the boundaries between them could vary depending on the development tools and procedures adopted. The specific steps presented here are for Zenith's HT-2000 OSD Decoder and OSD Information Gateway.

Defining the Editable Data

All information downloaded is ultimately derived from the OSD Information Gateway's RDBMS tables. A new application will typically require the addition of new tables. Enhancements to existing applications may need no additional tables, or only additional columns for existing tables.

It may be desirable to allow high volume data with rapid turnover to bypass the Relational Database. Zenith's OSD Information Gateway allows this option, but still pretends the data came from the RDBMS.

This standardizes handling of the data, and allows the by-pass decision to be deferred until after the application has been developed and tested when solid performance data is available.

Defining the Downloadable Data

The data that will be available to the dialog must be designed. This requires deciding how the data will be formatted and sorted, and what Edited data it is derived from.

In addition to deciding what the data will look like, the Application Designer must decide **which** data will be available to which decoders.

In some cases the supporting data will already exist -it is just being indexed or ordered differently. In other cases new RDBMS tables and/or columns will have to be specified.

Each Record Set can have many Record Set Revisions. The first version of a Record Set may have been missing a field, or it may have included a field that turned out not to be required.

Both Record Set Revisions can co-exist in the same OSD Information Gateway. The "Data Design" details how each Record Set Revision is formed from RDBMS tables.

Record Sets changes may be related. Schedule records and Program Information records might both be changed at the same time. The old versions work together just fine; the new versions get along even better - but you can't mix the two.

The Data Design also specifies which Record Set Revisions can be used together. This specification is called a Record Set List.

The Data Design is itself stored in RDBMS tables.

Specifying the Downloaded Dialog

The **Dialog Editor** is a GUI Application Generator for HT-2000 dialogs. It allows a Dialog Author to develop an interactive application, dubbed a dialog, that works for a specific Record Set List.

The Dialog Editor allows screen images, called "Forms", to be organized in Sections and populated with input and output fields.

Each Form consists of a static screen image and dynamic displays associated with fields. Fields can display data from downloaded Record Sets and/or decoder variables.

These can be combined to form complex expressions. A field specification might be the equivalent of "The name of the Program shown on the Channel in variable X no earlier than then Time in variable Y."

The Dialog Editor also allows the Author to specify **Transitions**. Essentially each transition specifies what **Response** there is to each **Input** event when the dialog is 'focused' on each given input field.

A Dialog Editor 'input focus' matches the Decoder's concept of 'state'. A dialog is in a given 'state' when a given input field has focus. Some 'forms' have invisible 'input fields'. For example, we pretend there is an input field when the decoder is turned off.

Input events include IR Remote keypresses, keyboard input, Conditional Access status changes and time-outs.

A **Response** may invoke a routine that is allowed to modify decoder variables. It can also specify a new state/input focus.

The Dialog is then compiled. The compiled dialog can be distributed to many different OSD Information Gateways.

Dialog Preparation and Download

At each site the Dialog can be prepared for downloading. This step can allow optimization of downstream downloading capacity and decoder RAM to meet site priorities.

The prepared Dialog can then be downloaded to decoders. The new dialog "takes over" from the previous dialog in the decoder. It can then preserve as much of the old state as possible.

With minor dialog changes, there may be no disruption at all to the viewer. Even in more significant code changes the new dialog will be able to let the viewer keep watching and/or taping whatever they were watching.

A Complete Update Scenario

Suppose a new feature has been proposed. In order to explain it a Dialog using no new dynamic data is prototyped with the Dialog Editor.

This dialog could start with the current dialog as a base, or be nothing more than a stand-alone stub of this single proposed new feature for demonstration purposes.

Creating static forms and transitioning between them the Dialog Author "storyboards" the new application. No real data is displayed, but a user will be able to get the "look and feel" of the proposed feature.

The prototype dialog can be compiled, prepared and downloaded to a set of test decoders. Working with actual decoders the Dialog Author and/or his/her reviewers find changes they want to make to the dialog.

This process continues until the storyboarded version is considered polished enough for use.

The Dialog Author would then decide what downloaded data was required to support the application. In many cases the data will already be available in the pre-defined Record Sets.

In other cases new Record Set Revisions will have to be defined and placed in new Record Set Lists. This is when the development process shifts from Dialog Authoring to Data Designing.

As wonderful and powerful as RDBMS concepts are, they are still very tricky. A Data Designer will need far more training than a Dialog Author.

The Dialog Author would specify the new Record Set Revisions and how they are formed from existing edited tables.

Occasionally the required data will not already be part of a pre-existing edited data. In this case the new RDBMS tables and/or columns must be specified. Frequently the RDBMS can be used to fill in new columns with default values based on other columns.

Data capture routines must be developed and/or modified to capture the new data. This step can be very simple if the data source uses a standard interface. When capturing data from a pre-formatted source this step would take a bit longer.

GUI front-end programs that access the modified RDBMS tables will have to be modified to access the new data. However, because of the Client/Server architecture they can continue to use the data in the old format without modification.

Even after updating the format of the RDBMS tables, the current Record Set Revisions will still be supported. Their data will be downloaded without change; they simply ignore the new data.

Once the edited data is defined, the downloaded data can be defined and a new Record Set List exported to the Dialog Authoring system.

The Dialog Editor can now be used to refine the dialog to use the actual data rather than being a simple storyboard.

After extensive testing, the new feature would reach the point where a field trial can be attempted. If anything goes wrong with the field trial the system can be quickly rolled back to the prior dialog. The OSD Information gateway is still fully capable of downloading the old dialog and the data in the format that it requires. The system can be rolled back.

An Alternate Update Scenario

It would be just as valid to start with a new source of data that has become available.

After figuring out how to capture this data, the Data Designer would determine how to store this data in normalized RDBMS tables.

Data Designers could then enumerate useful Record Set Revisions using this data.

Dialog Authors would then attempt to write new Dialogs after selecting the Record Set List that seemed most useful to them.

SUMMARY

Development of Interactive Information Services for OSD Decoders will be an ongoing process as the market adjusts to what information consumers desire and how they wish to access it.

Staying responsive to these changing requirements mandates that the set-top OSD decoder be downloadable.

Downloadability alone merely allows new applications to be developed. More powerful features are required to support rapid development and non-intrusive deployment of new applications.

Zenith's HT-2000 decoder system is an example of a complete development system for Interactive OSD Applications.

The OSD Information Gateway is developed under a Client/Server architecture using GUI Application Generators and the OS/2 RDBMS.

Data is captured and normalized into RDBMS tables. Data can be merged from many sources and even operator edited. Other RDBMS tables control the translation of this data into downloadable Record Sets.

The HT-2000 decoder uses a state driven interpretive model with built-in primitives for screen painting and data manipulation. It accepts downloaded behavior without risk of becoming permanently frozen by a defective download or excessively disrupting viewers.