Data Management in Set-Top Box Electronic Programming Guides

Abstract: The electronic programming guide (EPG) offers the most fully developed interactive browsing service on digital television, enabling users to search, filter and customize program listings and even control access to some content. These capabilities add significant data management considerations to the design of new set-top boxes. Seeking a proven solution, a handful of vendors have begun incorporating off-the-shelf database technology into their set-top boxes.

This paper explores EPG data management, with the goal of educating developers and improving their software results. It maps the set-top box technology environment, explores data management requirements, and presents typical data objects and interrelationships found in programming guides. In examining solutions, the paper focuses on one relatively new type of data management, the in-memory database system (IMDS), which improves data management performance and minimizes resource demands through a streamlined design. Code examples and sample database schema focus on efficiencies gained by implementing set-top box data management using an off-the-shelf database system.

McObject LLC
33309 1st Way South
Suite A-208
Federal Way, WA 98003

Phone: 425-888-8505
E-mail: info@mcobject.com
www.mcobject.com

Introduction.
Digital television has no doubt emerged as one of the most fully realized networked multimedia technologies. While other convergence applications strive to move into the mainstream, digital TV is already delivering hundreds of channels, a crisp clear picture, and better-than-CD-quality audio to millions of users. The period of transition from analog to digital broadcasting could last a decade or more, but eventually all broadcasters will move to the digital domain. Viewers accustomed to a rather passive video experience are often pleasantly startled by digital TV's interactive browsing of data services. Among the most useful of these are data services that describe audio-visual *content*.

The *electronic programming guide* (EPG) is one of the most important data containers of channel-rich digital TV. An EPG allows the user to scan available channel offerings and tune to current programs by pointing at listings with a remote control. The EPG informs the user of the most interesting programs that fit specified viewing criteria, and the user can check program availability over a series of days. EPG technology is no futuristic dream—cable and satellite receivers offer such guides *today*, allowing users to browse available programs by themes such as movies, sports, news/business, family/children and education. Favorite channel lists can be customized for easy program selection, and an *info* button on the remote instantly calls up program information. Some models even enable parents and others to block content, usually according to program ratings.

Systems designers will recognize these capabilities as entailing significant searching, filtering, sorting, selecting and storing tasks. In fact, the EPG, more than any other feature, has added a challenging data management component to the design of new digital television receivers (also known as Integrated Receiver Decoders, IRDs or set-top boxes). To support the growth in EPG features, developers increasingly must incorporate a database "layer" within receivers' embedded software that facilitates optimal data designs, supports transactions and data integrity, and minimizes impact of data management tasks on performance and on RAM and CPU usage.

Developers have typically responded to this challenge with self-developed data management components. This solution often meets the performance and small footprint requirements for set-top boxes. But when required to manage increased volumes of data or to adapt to new IRD features, the self-developed data management often proves difficult to upgrade, demanding inordinate QA and extending development cycles.

Increasingly, IRD developers are turning to commercial embedded database systems, which offer stability, extensibility and scalability gained from years of usage. However, commercial databases' lack of real-time performance often rule out their use in digital TV receivers—when surfing through program offerings, viewers expect the zero-latency responses associated with consumer electronics. These databases' demands on processing resources also make them a poor fit—in order to minimize costs, companies developing IRDs incorporate minimal RAM and choose less powerful processors.

One of the greatest challenges of using off-the-shelf data management in set-top boxes is the sheer wealth of available products and information. Set-top box designers are typically not database experts. To assist them in understanding this emerging aspect of set-top box development, this paper explores IRD data management requirements as well as typical data objects and interrelationships defined by the EPG database. It examines off-the-shelf solutions, focusing on one relatively new type of database, the in-memory database system, which meets EPG system demands by eliminating much of the unnecessary overhead associated with traditional (disk-based) databases.

The EPG as local data store.
The EPG data that consists of program descriptions is issued in real-time from broadcast or terrestrial sources. The information cycle rates in the EPG streams vary and can be relatively long (i.e. hours). Because of this, the set-top box cannot "wait" for data upon a user selection or some other query of EPG information. Rather, the set-top box receiver collects and retains the information beforehand. Once the EPG data is received and stored locally in the set-top DRAM, the viewer can search, filter, sort, and select programs for immediate or future viewing. Local storage of the program guide isolates the consumer from any transport delays.

Operating environment.
Cable, satellite, and over-the-air digital receivers are usually built as embedded systems with limited amounts of DRAM as the primary storage. Set-top boxes usually don't run traditional operating systems, such as those found in PCs and workstations. Only multi-tasking real-time operating systems can support the high data rates and performance demands of an IRD solution.

On the other hand, as set-top boxes evolve in their capabilities and features, their embedded software is becoming more robust and multiple applications now co-exist within the box. In addition to video display and basic television functions, the growing list of add-ins includes e-mail, video-on-demand, Personal Video Recorder (PVR), etc. Often these services are integrated with the EPG. Not only does the EPG have to store the data in a timely fashion, it must provide various search capabilities to these applications. Thus, the EPG needs a database system that will serve as underlying storage and expose an application programming interface (API) to EPG functions. This underlying database storage has to integrate seamlessly with the OS environment. It must be robust to handle complex requests for data storage and retrieval, but not monopolize the CPU, which is required to handle other tasks, such as PVR.

Storage management requirements: EPG data.
On-screen program guides are composed of two major parts: 1) the application software (often called the EPG presentation engine) that runs in the set-top and 2) the data that the application software receives and formats for display. The EPG-equipped set-top must be frequently updated with guide data via an external source. The IRD extracts the program guide data, stores it in memory (DRAM) and keeps it for display as either a full-screen grid guide or as an overlay to active video. The amount of future programming covered by a guide is determined by the amount of DRAM storage available in the set-top box. In this sense, the EPG storage management sub-system competes with EPG content for the use of these memory resources, so that minimizing the application's memory footprint enhances the end-user's experience by enabling richer, more complete content.

Based on the above standards, the following data objects are usually kept by the IRD to form the content of the EPG:

Channel-related data. A channel represents each satellite or cable channel and terrestrial broadcast channel. At a minimum, each channel is characterized by:

- The two-part (major.minor) channel number for access to the service
- Text name (up to seven characters)
- How the service is physically delivered (carrier frequency and modulation mode)
- The channel's "source ID" (referencing a unique name in a national database for program sources)
- The type of service (analog TV, digital TV, audio only or data)

Other types of data specific to each channel include a flag that tells whether the service is access-controlled or not, and an indication as to whether or not "extended text" is available to provide a text description of the service (optional descriptor).

Event data. Events carry program schedule information for each channel. Each event consists of:

- Event start time
- Event duration
- Event title (text)
- Pointer to optional additional descriptive text - *Program data* described below, which may contain a synopsis, cast, director, etc.
- Program content advisory data (optional)
- Caption service data (optional)
- Audio service descriptor, which can list available languages (optional)

Events are usually organized in *Schedules*. A schedule consists of a start time and a list of events. The complete schedule for a channel is made up of a number of schedules linked together.

Program data. As a minimum, program information contains program title, and optionally the program's description, category and *rating* information, and conditional access information.

Rating regions data define a "rating system" for a given region, characterized by a number of *rating dimension*s. Each dimension is composed of two or more *rating level*s. An example of a typical rating dimension used on cable is the Motion Picture Association of America (MPAA) system. The levels within the MPAA dimension include "G," "PG," "PG-13," "R," and "NC-17."

Category data. The category system defines a hierarchical structure and descriptions of program categories used by the EPG. The category system changes dynamically over the life of EPG. New or updated definitions can be sent and must be inserted into the EPG.

Additional data that is commonly present in modern IRDs include:

- Favorite channels lists – this feature allows the viewer to edit the EPG to show only those channels that are of interest. It also speeds up the sorting process by allowing the viewer to create a personalized guide that shows only favorite channels
- Help screens to guide the novice EPG user, updated as needed if new features or functions are downloaded
- Cable systems and local off-air programming. There are roughly 12,000 cable systems in the US, for which the IRD should be able to provide an adequate service
- Advertisement and other material that the IRD could be required to accept, including images and sound
- Wink support data

Storage management requirements: search capabilities.
Given these data objects and the receiver's required features, the underlying data management module should expose a number of search methods, including:

- Usually, the EPG data elements (objects) are assigned unique identifiers, which are used by the programming source to help combine relevant programming data. The data management module should provide a fast search mechanism based on such identifiers

- Source-id based search mechanism. Source-id identifies a unique programming source for a channel. A complete schedule for a channel is made up with programming information linked together by a common source-id. In order to obtain a channel's programming data, the IRD needs to locate all the programming sources with the given source-id within the EPG database
- Sequential fetch through objects of the same type, such as a list of all acquired programs or channels
- Channel sorting based on major and minor numbers, or by channel call sign (title). The storage management usually must provide a mechanism to change the channel sort order at runtime according to the user's preference to see channels listed numerically or alphabetically
- Alternative sort mechanisms, such as program show times across all channels

Storage management requirements: memory management.
The amount of memory available for local EPG data storage is usually quite limited. In general, the more RAM that a receiver devotes to processing and storing the EPG, the more capable the IRD will be. IRD capabilities that affect RAM usage include the number of days of schedule information to retain, use of short vs. long descriptions, which credits to retain, language(s) supported, and graphics. Therefore, it is imperative that the storage management module make efficient use of IRD memory. The storage management should be able to support various data alignments – EPG data elements commonly occupy just one byte, for example. About 90 percent of all EPG data is, in fact, dynamic text. The storage management should not store dynamic data, such as variable length strings, in data types demanding excessive memory. That could easily become prohibitive.

Service providers usually require the IRD to keep 3, 7 or 14 days of guide data. The 3-day schedule means that the IRD always carries 2 days of programming into the future, plus the current day's schedule. The EPG should be able to handle "boundary conditions" gracefully. When all the memory dedicated for storage is used up, the data management should allow the EPG to run "clean-up" procedures—remove expired data and return memory to the memory pool for reuse. Ideally, the EPG would implement OS-independent dynamic memory management, since embedded operating systems usually lack efficient memory managers.

Storage management requirements: multi-threaded access, and transaction mechanism.
The EPG database is accessed by multiple threads (tasks) simultaneously, including those dedicated to writing new data into the EPG database, user interfaces queries, conditional access queries, Wink support tasks, etc. Some of these tasks—such as writing new data to its permanent location as soon as it is available—should gain prioritized access to the database, while lower priority tasks can execute in the background during otherwise idle CPU time. The storage management module must, therefore, provide multi-threaded functionality to the EPG, and implement a transaction prioritization mechanism.

Storage management requirements: performance.
EPG data must be transmitted at a high rate. These rates vary from one service provider to another, but *all* providers try to utilize the available bandwidth to the greatest extent possible. IRD software processes received packets before storing this data in the EPG database. Such processing may include assembling EPG objects from frames, decompressing the data, and filtering. Generally, the transmission rate is in the range of tens of megabits per second. For the IRD to be accepted by service providers, the storage management software must be able to keep up with this or even higher rates on the order of hundreds of megabits per second. Accordingly, the storage management should use a specialized and extremely fast transaction control

mechanism as well as memory-oriented search algorithms. At the same time, it is important for the EPG database management module not to monopolize the CPU, because other important functions of the IRD, such as video recording, require its cycles. EPG storage algorithms should be optimized to minimize CPU cycles.

In-memory database systems: a recipe for EPG database management.

Maintaining an EPG database is a challenging and complex task. Historically, developers of traditional (non-embedded) applications have addressed mounting data complexity with database management systems that provide formalized methods for maintaining data integrity, constructing complex data relationships, and providing access to information quickly and efficiently. By replacing self-developed code with proven database APIs, DBMS technology reduces coding, debugging and maintenance requirements, decreasing the developer's burden. However, "traditional" databases make substantial memory and CPU demands and deliver decidedly non-real-time performance.

But a relatively young technology, the in-memory database system (IMDS), benefits from a design that eliminates certain unnecessary database functions and streamlines others to fit demanding embedded systems. Designed from the start for memory-only deployment, IMDSs accelerate processing through an architecture that eliminates disk I/O, caching and other high-overhead functions. One database in this category, McObject's *e*X*treme*DB®, was designed for memory-only deployment in device-based applications including set-top boxes. To provide realistic examples, *e*X*treme*DB will be used in the remainder of this paper to illustrate the advantages of integrating an IMDS as a set-top box's data management layer. *e*X*treme*DB provides additional development capabilities that prove valuable in addressing the EPG requirements described above, such as support for certain data types, object relationships and search methods. While not unique to in-memory database systems, such support provides compelling advantages for EPG development, and is discussed at some length below.

Memory management: main memory database.

As illustrated above, EPG data management demands support for a high transmission rate. The database must also be exceptionally frugal in its use memory and CPU resources. Compared to disk-bound database systems, *e*X*treme*DB eliminates memory demands and related overhead in the following ways:

- There is no application-database connection overhead – *e*X*treme*DB is tightly linked with the EPG code
- No extra layers - designed with the assumption that data is in memory, *e*X*treme*DB streamlines data management tremendously, removing the layers of overhead, such as caching, typical of disk-based DBMSs
- Search algorithms are highly optimized for memory access (disk-based databases are optimized to minimize disk I/O)
- Search translation – *e*X*treme*DB points directly to the memory location of data elements. In contrast, conventional DBMSs usually point to a block number and an offset. The database needs to locate the block, load it into memory and find the appropriate memory location in the memory buffer
- *e*X*treme*DB eliminates the need for buffer management. Conventional DBMSs assume that new data from disk will replace data in memory buffers, and therefore constantly write memory buffers to disk

- *e*X*treme*DB provides direct access to data. In a disk-based DBMS, an application never gets access to a data element in the memory buffer. Instead, the data is copied elsewhere to memory, adding more overhead yet

Memory management: memory managers.
*e*X*treme*DB is designed to operate with limited amounts of memory. *e*X*treme*DB's own memory managers are optimized to provide low overhead data and index layouts. The *e*X*treme*DB runtime does not use the operating system's malloc()/free() – which in the RTOS environments can be inefficient. Instead, it uses its own highly optimized memory managers. These managers provide packing of object data, keeping object sizes small and, to a degree, controllable from the application.

Transactions and multi-threaded access to data.
A database should make the most of its operating environment. While a data manager for an electronic programming guide must fully support the ACID principles, it should also take advantage of the embedded, limited multi-tasking nature of the set-top box. In this setting, only a few tasks execute simultaneously, and rarely do simultaneous tasks require write access to the object store. Therefore it is practical to minimize footprint and simplify the transaction manager by serializing write transactions, which eliminates the demand for complex transaction synchronization.

*e*X*treme*DB allows multiple simultaneous read requests to be executed at a time—i.e. multiple "read" transactions are allowed. "Write" transactions have exclusive access to the database runtime. This approach to coordination eliminates the need for locking mechanisms with their overhead and complex deadlock prevention problems, while maintaining database integrity. Serialization of write transactions is transparent to the EPG programmer. Because the transaction manager is "light," properly designed and implemented transactions execute swiftly and serialization is not a performance concern.

*e*X*treme*DB also supports transaction priorities – it is possible to assign a priority value to each transaction at runtime. Using the transaction priority mechanism, the EPG can also "boost" the execution of a transaction at runtime if necessary (for example, to clean up the transport buffer during high-speed transmissions).

***e*X*treme*DB mapping of EPG data.**
The availability of certain native data types, object relationships and search methods greatly simplifies construction of efficient electronic programming guides. For example, EPG data is often tree-structured. While "traditional" (disk-based) databases may address this by offering simple b-tree capabilities, *e*X*treme*DB provides rich tools for implementing such relationships through compound data types. Specifically, *e*X*treme*DB supports vectors (an arbitrarily large stream of typed data), structures (a structure declaration names a type and specifies elements of the structure that can have different types; structures and simple types are building blocks to construct object definitions; structures can be used as elements of other structures), and optional data fields.

These constructs greatly simplify the representation of native EPG data. With them, EPG information is typically separated into data objects, consolidated into classes. Object classes include *channels*, *schedule, programs*, etc. Each object is formatted according to its type, but in general, objects are comprised of *basic elements* and *additional elements,* sometimes called *descriptors*. The basic elements are defined in a rigid structure particular for that object type. Additional elements are typically attached to the basic portion of the object and there can be zero to many additional elements following the basic portion. Usually, for the purpose of identification, all EPG objects are assigned unique integer identifiers within a common pool. This number is called the *object_id* and is an object's serial number for the life of the object.

Let's consider a hypothetical description of a channel object. A channel is the destination, usually a number, name, and logo, that is recognized by the user as a single entity that will provide access to a TV program. The channel object shown in Example 1 identifies the channel and its contents.

```
channel_object
{
        object_id;
        time_acquired;

        source_id;
        major_number;
        minor_number;
        short_name_size;
        short_name();
        indicator;
        if (indicator == 1) {
              expression_size();
              expression();
        }
        descriptors()
        {
         [text | category | channel_content | audio_service ]
        }
}
```

Example 1. Typical channel object

The definition of fields in the above pseudo-code is as follows:

object_id	this 32-bit field uniquely identifies this object
time_acquired	time of the object's acquisition by the IDR
source_id	this 16-bit field identifies the programming source for the channel. The source_id allows an IRD to link a channel to its schedule
major_number	this 16-bit field identifies the major number for this channel
minor_number	this 16-bit field identifies the minor number for this channel
short_name	these are the bytes for the string that represent the channel's short name
indicator	a 1-bit field, if it is set it indicates that the object contains an expression associated with it. This expression can be used by a conditional access engine to determine whether to include or exclude the object into the guide
text	this field provides a textual description about this channel. May or may not be present for the channel
category	if present, this field provides the category classification for this channel
channel_content	if present this field provides alternate information about the content of this channel
audio_service	if present contains information on available languages for the channel

The *eXtreme*DB schema for the above fragment, shown in Example 2, defines the channel object:

```
struct ID {
      uint4 object_id
};
declare oid ID [200000];
/* Class Channels  */
struct Expression {
      string str_expression;
};
/* These descriptors are used in various objects */
struct Text {
      string str;
};
struct Category {
      . . .
};
struct AudioS {
      . . .
};
compact class channel {
      mco_time time_acquired;
      uint2 source_id;
      uint2 major_number;
      uint2 minor_number;
      string short_name;
      uint1 indicator;
      optional Expression opt_expression;

      /* descriptors */
      optional Text     this_text;
      optional Category this_category;
      optional Content  this_content;
      optional AudioS   this_audio;

      tree < major_number, minor_number> chan_number;
      tree <short_name> chan_name;
      oid;
};
```

Example 2.

The *eXtreme*DB Data Definition Language (DDL) supports C structures and dynamic strings, object identifiers, and indexes. Note how the *optional* modifier is used above. All the descriptors as well as the Expression field are declared as *optional*. An optional declaration means that the field may or may not be actually stored in the database by the application. If the application does not store the field, the runtime does not reserve (allocate) space for it within the data layout. The *compact* class qualifier limits the total size of the class' elements to 64K. That includes not just raw EPG data, but also the overhead required by *eXtreme*DB. However, the *compact* declaration significantly reduces the overhead that is necessary to maintain the class data. (Since it is known that the object fits within 64K, *eXtreme*DB can use 2-byte offsets instead of 4-byte offsets. The 2-byte savings multiplied by thousands of objects translates to meaningful savings.)

The EPG is usually required to display the list of channels sorted either by their number or call signs. The above code fragment defines two indexes for the channel class: *chan_number* index provides sorting/searching by major and minor number, while the *chan_name* index provides sort/searching channels based on their short names. The EPG can use the *eXtreme*DB cursor API to navigate the sorted channel list and also to quickly switch between the two indexes.

To avoid unnecessary transmissions and save resources, EPG objects should provide the capability to be stored once but referenced by many other EPG objects. Such content might be, for instance, a nonspecific *News* program without any detailed description of the topics covered from one showing to the next. The *News* program might be sent once in the EPG transmission but repeatedly referenced night after night at 6 o'clock. A unique object identifier is used to maintain this information referencing. *eXtreme*DB natively supports object identifiers, and provides a special "reference" - ***ref*** data type as well as extremely fast search capabilities based on ***oid***. In the above example, the *channel* class is declared with ***oid***. That corresponds to the unique object identifier assigned to all EPG objects by the service provider. The *eXtreme*DB runtime enforces the "uniqueness" of the object id at runtime, thus simplifying referencing channel objects by other EPG objects.

In a further elaboration of tree-structuring, EPG objects often include variable-length arrays of structures. For example, a *schedule object* such as the one in Example 3, below, defines a portion of a channel's schedule. The various schedule objects make up a complete schedule. The schedule objects are associated with their referencing channel(s) by means of the *source_id*. In addition, each schedule object carries start time, duration, and a number of events.

object_id	this 32-bit field uniquely identifies this object
time_acquired	time of the object's acquisition by the IRD
source_id	this 16-bit field identifies the programming source presented in this schedule object. The source_id allows an IRD to relate a channel object to this schedule object
start_time	this 32-bit field identifies the schedule start time
duration	this field indicates the total number of hours of scheduling information provided by this object
number_of_events	this 8-bit field indicates the number of events in this schedule
event_start_time	start time of the event
event_duration	this field indicates the duration of an event
program_object_id	this 32-bit field defines the object_id for the *program object* in this time period

```
schedule_object
{
      object_id;
      time_acquired;

      source_id;
      start_time;
      schedule_duration;
      number_of_events;
      for (i = 0; i < number_of_events; i++)
      {
            event_start_time;
            event_duration;
            ref program_object_id;
      }
}
```

Example 3.

*e*X*treme*DB represents the schedule by using *vectors*. Vectors are useful when the object model in an application already contains natively dynamically structured items such as events within a schedule. Example 4 illustrates *e*X*treme*DB's use of vectors in organizing scheduled events.

```
struct Event {
      time start_time;
      uint2 duration;
      ref program_id;

};
compact class schedule {
      time time_acquired;
      uint2 source_id;
      uint4 start_time;
      uint2 duration;
      vector <Event> events;
      tree <source_id, start_time> lineup;
      oid;
};
```

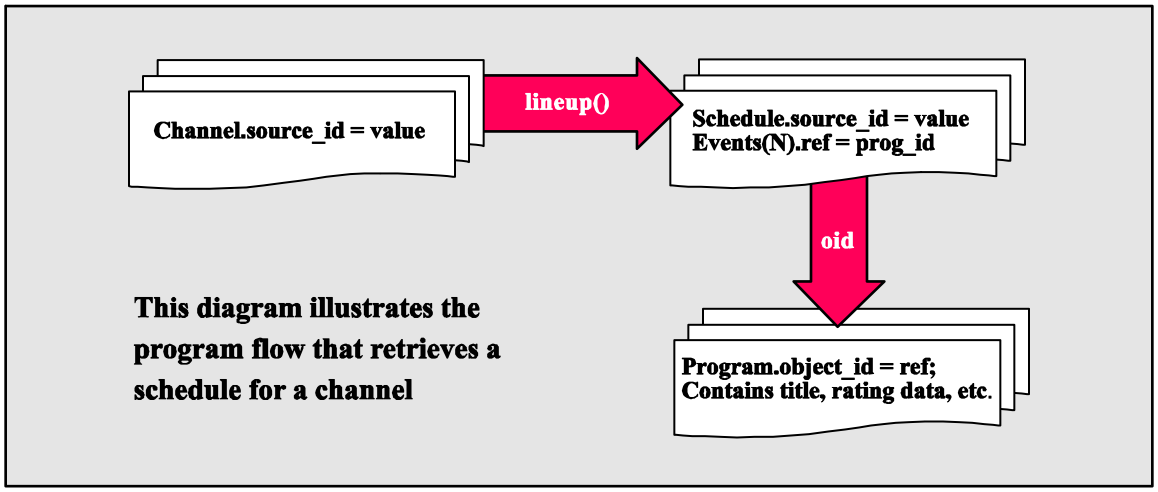
Example 4.

To represent the program data, *e*X*treme*DB defines a class *program* illustrated in the Example 5. Note that the Event structure references an oid of the corresponding program. Each program can be aired at different times on various channels, however, the data associated with the program description, program rating information, etc., is only stored once in the database. Also note that the description of the rating system is simplified for brevity. Please refer to the ATSC standard for the complete description of the rating system.

```
struct dimension {
   uint1 level;
   uint1 value;
};
struct rtt { /* rating region table */
   vector <dimension> dim;
    string text; /* the text that describes this rating dimension
*/
};
struct content {
  vector <rrt> region;
};
class program {
      time time_acquired;
      string title;
      optional content this_content;
      optional text    this_text;
      oid;
};
```

Example 5.

Very often the IRD needs to obtain a complete schedule for a channel. It is very straightforward to implement this request using the *eXtreme*DB API. The diagram in Figure 1 below illustrates the program flow.



Channel.source_id = value
lineup()
Schedule.source_id = value
Events(N).ref = prog_id
oid
This diagram illustrates the program flow that retrieves a schedule for a channel
Program.object_id = ref;
Contains title, rating data, etc.

Figure 1.

The EPG first locates all schedule objects associated with a channel by navigating schedules based on the *lineup* tree-based index. For each schedule in this group, the EPG reads the vector of events. Each event points out to a program_id, i.e. the object identifier of a program object that has all the textual information about the event. Using this reference, the EPG quickly locates the associated program_object and reads out the description and other related information of the event. The code fragment in example 6 below implements this query using the *eXtreme*DB application-specific application programming interface (API). *eXtreme*DB-based applications use an API generated by the *eXtreme*DB schema compiler to store, read, and manipulate objects in the database. This is in contrast to many database products that offer a static proprietary navigational API, or a static standard API (like SQL). The *eXtreme*DB API is always tailored to the specific application, resulting in an intuitive database/application integration—as if the database were written for the exact needs of the application.

```
uint1 channels_schedule(mco_db_h db, uint2 source_id_from_channel)
{
      /*
       * t   is a transaction handle
       * csr is a database cursor
       * sch is a handle to a schedule object
       * epgoid is the object id structure defined in the schema
       * event is a handle to an Event object.  Event is a part of a
         schedule that has descriptive information attached
       * prog is a handle to a program object - the descriptive
         information for the event could be shared between
         multiple events
       */
      mco_trans_h       t = 0;
      mco_cursor_t      csr;
      MCO_RET           rc;
      schedule          sch;
      MyEPG_oid         epgoid;
      Event             event;
      Program           prog;
      char              title[MAX_TITLE];
      uint2             real_size;
      uint2             schedule_source_id;
      uint2             number_of_events;
      uint2             I;

      rc =
       mco_trans_start(db, MCO_READ_WRITE, MCO_TRANS_FOREGROUND, &t );
      if(rc)
            goto err;
      /* instantiate a cursor for the 'lineup' index */
      rc = schedule_lineup_index_cursor(t, &csr);
      if (rc)
            goto err;
      /* search the 'lineup' index and position the cursor with the
         supplied argument 'source_id_from_channel' */
      rc =  schedule_lineup_search( t, &csr, MCO_GE,
                                    source_id_from_channel, 0 );

      /* set up a loop to interate over the 'lineup' index */
      for(; rc == MCO_S_OK; rc = mco_cursor_next(t, &csr))
      {
            /* obtain the schedule and... */
            schedule_from_cursor (t, &csr, &sch);
            /* ... read the source_id field out of it */
            schedule_source_id_get( &sch, &schedule_source_id);

            /* have we enumerated all the schedules for the channel */
            if (schedule_source_id != source_id_from_channel)
```

```
                  break;

            /* the current schedule has the same source as the channel,
             * so we read all events that make the schedule. */

            /* find out the size of the events vector */
            schedule_events_size (&sch, &number_of_events);

            /* scan through them */
            for (i = 0; i < number_of_events; i++)    {
                  /* obtain a handle to the event within the vector */
                  schedule_events_at (&sch, i, &event );
                  /* read the program's identifier */
                  Event_program_id_get ( &event, &epgoid );
                  /* locate the program that contains textual
                     information about the event. Use extremely fast
                     oid-based search */
                  rc = program_oid_find (t, &epgoid,  &prog );
                  if (rc)
                  /* no information available for this time period */
                        break;
                  /* read and print out the title. In reality this
                     information is forwarded to the presentation
                     engine for further processing and display */
                  program_title_get
                    (&prog, title, MAX_TITLE, &real_size);
                  printf("%s\n", title);
            }
err:  }
}
```

Example 6.

Summary

Electronic Program Guides are no longer just alternatives to printed media program guides. Enhanced with intelligently structured and frequently updated content, an EPG provides a way to gain detailed program information, sort through viewing choices, and even record a personal selection of viewing alternatives. Personal viewing services that couple EPG technology with digital video recorders further enhance the possibility of truly personal television.

The company selected to develop and maintain an EPG needs to have special knowledge and experience with the TV screen (as distinct from the PC screen) and the remote control unit. It also must have significant knowledge of data processing, especially given the memory constraints in the decoder. Developing the database management module for an EPG is extremely challenging by itself, and the IRD developer needs to adhere to the EPG operator rules.

In-memory databases systems (IMDSs) such as McObject's *eXtreme*DB provide an ideal underlying database management system for EPG applications. Offering support for the multi-threaded environment of many embedded operating systems, as well as a small footprint, the *eXtreme*DB runtime exemplifies database technology that is highly optimized for minimizing CPU and RAM usage. It also provides compact storage layout, supports advanced search methods, and natively implements data types required by the EPG, leading to more efficient development and maintenance of EPG code.