"BUT THE DEMO LOOKED GREAT"
A TECHNICAL PRIMER ON SETTOP SOFTWARE DEVELOPEMENT

Stephen Johnson
Coach Media

Abstract

Creating effectively designed software—for any platform—is a complicated and time-consuming business. Creating software for digital settops has encountered challenges unique to the cable industry: scarce applicable design precedence, a thankfully temporary mania of inflated expectations for "interactive television," and a paucity of basic technical knowledge about how applications are created and deployed. While the first two issues have largely faded, the last continues to hamper effective application deployment. Without a general understanding of the issues involved in settop software design, cable operators will inevitably experience frustration and disap pointment with their deployments.

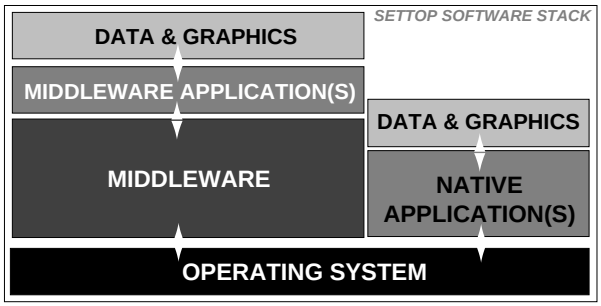
This paper addresses five topics in software design applicable to digital settops, providing a brief technical background of each issue followed by their individual effects on software performance—and ultimately viewer comprehension and acceptance. While illustrated by real-world examples, this discussion does not focus on a particular settop box manufacturer or set of features; rather, the principles discussed here apply to all platforms. The topics cover the following:

- *Middleware Usage*
- *Memory Allocation*
- *Settop Performance*
- *Viewer Interface Details*
- *TV Display Technology*

While not comprehensive, this list nevertheless covers a range of technical issues collectively having a large impact on successful software deployment. Armed with this knowledge, cable operators and other gatekeepers can ask the right questions and thereby create—or procure—better applications to meet their ever-expanding needs.

MIDDLEWARE: FLEXIBILITY VS. SPEED

Middleware refers to a software application that runs programs—but also runs on top of an existing operating system (OS). Think of a web browser, e.g., Internet Explorer™ running "on top of" Windows™ and you get the idea. A middleware application uses code—or even just files of text or formatted data—written according to its own languages and syntax rather than that of the OS. The diagram below illustrates how these various applications stack up:



Schematic diagram of typical digital settop box software stack

Current Examples

How and where is middleware used? The most prominent use today is probably the OpenCable™ Application Platform (OCAP), which uses middleware called the Java Virtual Machine™ (JVM). Java™ was originally developed by Sun Microsystems as an Internet-friendly programming language

that could execute code using a JVM on *any* operating system. OCAP applications, therefore, are written in Java and could potentially run on any settop operating system; the JVM would handle all the underlying operating system routines.

While OCAP remains the most prominent middleware example, the settop box development world has also seen the introduction of middleware applications from companies like OpenTV, Liberate, and Microsoft. With so many companies offering middleware solutions to settop development, why consider any competing strategy? Indeed, in addition to the promise of OCAP middleware offers some very tempting advantages.

The Best of All Worlds?

Start with labor. Given the latent popularity of Internet-based programming a relatively large pool of trained candidates has become available to code middleware applications—at least of the browser-based or JVM variety. Since middleware programmers don't write code directly to the operating system, developers are additionally free to "script" quick applications and test them without enduring the rigor of compiling and linking code. Think of an HTML scripter doing a quick web page layout compared to a C++ programmer writing proprietary code and you can appreciate this difference.

With deployment flexibility and an available labor pool on its side, what disadvantages could there be to middleware development? Why suffer through hiring programmers to work in a very specific and unforgiving development environment—on code that can't be used elsewhere?

In a word: speed. Programs written without middleware overhead simply run faster. Sometimes *much* faster. The reasons for this are simple: fewer instructions to translate and direct, customized code. Think of the stack diagram above; when a middleware program runs its code it has to be continually translated for the operating system. The difference is as dramatic as speaking English to an English speaker versus speaking Italian to the same speaker and attempting to translate it in real-time. While the translation will generally work, no one can reasonably argue it will be as fast or as clear.

While it's certainly the most important reason to avoid middleware development, lack of speed alone unfortunately doesn't exhaust the disadvantages. Middleware is also very large. Since it translates instructions for many different operating systems—and needs to be resident in the settop's memory to execute commands at anything approaching reasonable speed—by necessity it often requires generous amounts of precious memory. Many middleware applications simply cannot run on older digital boxes due to memory restrictions.

In addition to its other difficulties, middleware platforms also aren't—alas—as flexible as advertised. Even Sun's touted Java, once marketed as "write once, run anywhere" has occasionally been derided as "write once, debug everywhere." Even discounting the jokes, Java developers often find they have to adapt code for JVMs to their particular use.

So use of middleware—a key decision point in settop software development—really comes down to flexibility versus speed, with a sizeable caveat on the former and little argument over the latter.

MEMORY: BE LEAN, MEAN, AND PLAY WELL WITH OTHERS

Whether an application uses a middleware platform or not (see previous section), it still loads into the settop's memory when launched—not unlike how a program loads on your PC. Compared to an average PC, however, settop memory is severely limited. The first issue confronting application memory usage, therefore, is simple: how much room do you need? Many developers equivocate on this issue—and not without good reason. The raw code size specified for an executable application may be misleading for at least three (3) reasons:

- The additional *data* the application requires (e.g., program guide data) might be even larger than the code that manipulates it;
- *Graphics* and other large non-code components might not be included (for good reason) in an application's memory footprint; and
- An application may be sharing memory space with several *other* applications—about which it has no knowledge—and might be unstable at *any* size.

Why do developers need to address these issues? Program guide data, for example, can be enormous. *One day* of TV listing data uses up to 250 kilobytes of memory. Doing the math it's not hard to see why loading two weeks of guide data (14 x 250K, or 3.5 megabytes) into settops with free memory sizes of a few megabytes presents some difficulties. Graphics are arguable worse: *one* high-resolution (uncompressed) full screen background requires almost a megabyte of data. And lack of vigilance about several different applications (regardless of size) residing in memory has very high costs: settop crashes (and reboots) correlate very well with this situation.

Developers recognize these issues and devise appropriate strategies to save memory space, but the resulting tradeoffs are far from painless. Memory-related challenges confronting developers—along with accompanying strategies and tradeoffs—are noted in the table below:

Challenge	*Strategy*	*Tradeoff(s)*
Large application size (including supporting data) vs. limited memory	• Store unused components on server • Distinguish between launch-ready and full application	• Interactive performance when uploading new data
Large application graphics vs. limited memory	• Store unused components on server • Compress graphics • Use settop-based graphics	• Interactive performance when uploading new graphics • Graphic degradation • Customized graphic programmin g
Multiple applications residing in memory	• Limit number of simultaneous running applications • Certify deployed applications	• Application Swapping • Extensive Quality Assurance (QA)

To save memory many developers keep a subset of application code, data, and/or graphics stored on a remote server and only load it when required (e.g., on viewer request) to do so. While this strategy saves space, the process of swapping in code and data might create some awkward performance delays when contacting a remote server. Delays become acutely painful when loading graphics as viewers wait for an interface to "arrive" and a screen is not yet visually "complete."

A related memory-saving strategy effectively divides an application into parts, allowing a smaller version to launch (e.g., initially display on-screen) while the full version is stored remotely (e.g., on a server, as described above) and downloaded later. While requiring some programming subtlety, this strategy mitigates performance delays by: 1) launching application faster; and 2) allowing other applications parts to be loaded into memory while the initial launched application runs.

Compressing graphics also saves space, but tradeoffs beyond the obvious—potential visual degradation—should be considered. Depending on the compression schemes (e.g., MPEG, JPEG, BMP, IMG) supported by the settop operating system or middleware, graphics may require *decompression* time to be properly displayed—adding to performance delays. If compressed graphics can be shown as-is the inherent low-resolution NTSC display standard for television often hides ugly artifacts—to a point. The rapid deployment of High Definition (HD) capable settops—and their demand for high-resolution imagery—will likely eliminate this advantage very soon.

The process of creating graphics directly from code available in the settop's operating system (if available) offers the promise of avoiding speed and compression issues altogether. Already available from the OS, settop-generated graphics display very quickly and require no compression. Creating these graphics, however, requires hyper-specific programming skills and generous development time; working at the level of individual screen *pixels* is not uncommon.

Memory management issues unfortunately aren't limited to those relating to application size; keeping several applications—regardless of their size(s)—simultaneously in a settop's memory creates challenges of its own. Furthermore, these challenges directly affect the cable operator since individual developers may have no prior knowledge of other applications with which their programs need to co-exist.

As noted above, having a large number of applications in memory often creates settop crashes for a variety of technical reasons beyond the scope of this paper. With this unpleasant fact in mind, cable operators need to address two challenges in this area: 1) keeping applications from *running* simultaneously as much as possible; and/or 2) testing multiple application combinations before deployment. The first challenge places restrictions on application features and user interfaces while the second potentially imposes testing costs on the operator.

Settop boxes will probably never have enough memory for every deployed application—or combination of applications. Even with the current tradeoffs operators have deployed dozens of stable and robust interactive products. Intimate knowledge of these issues often makes the difference between deployable and "demo" applications.

SETTOP PERFORMANCE: FASTER IS GOOD, ROOMY IS BETTER

Many standards are available to gauge settop hardware performance. The previous section discussed memory management; this section covers two other issues: processor speed and permanent, or non-volatile, memory.

Compared to similar hardware in PCs, settop microprocessor speeds are frighteningly slow. Even accounting for settop box economics, the numbers are startling: for example, the MicroSparc

processor in a baseline Scientific Atlanta 3000-series settop clocks in at 166 Megahertz. In non-geekspeak that's about four to six *times* slower than an Intel or AMD microprocessor in a cheap PC.

Fortunately, raw processor speed pales in importance to how well a settop is optimized to use it—and how well developers make use of it. Current settops are generally optimized to play digital video; relatively speaking, the processing power devoted to running interactive applications doesn't *need* to be nearly as fast as that for a PC. Settop applications display graphics and data to viewers and respond to interactive commands but have little need for the number crunching power PCs require for their word processing and spreadsheet tasks.

The amount of available memory—both volatile (lost without power) and non-volatile (or NVRAM, maintained without power) in a settop actually has a stronger impact on application speed and stability than its microprocessor clock speed. NVRAM contains data that a developer doesn't want his application to lose if the settop loses power. Pay per View purchase (secured) data is an obvious example; settops have been storing this data since the days of analog boxes. Applications also use NVRAM for parameters like viewer preferences (e.g., favorite channels) and code-critical data (e.g., patches and updates).

NVRAM is also expensive and therefore scarce. The aforementioned Scientific Atlanta box contains all of 2000 *bytes* of it. Developers who run out of room are forced to store this important data on a server, via sending it at regular intervals and correlating the data with individual settops.

Settop hardware performance therefore boils down to both suggestive and definitive numbers. Raw processor speed can be overrated—but all the memory in the world probably isn't enough

.

VIEWER INTERFACE: THE DEVIL IN THE DETAILS

Even when a settop application runs quickly, uses memory efficiently, doesn't tax the network or local storage, it still may not *look* very good. Obviously this is often due to aesthetics, but this paper leaves that hornet's nest to the legion of TV graphic designers. Rather, the primary focus here is on the technical details of the viewer interface display. While this subject is nearly inexhaustible, a survey of three (3) general areas—graphics, video, and on-screen text—suffices as a primer.

Getting Graphic

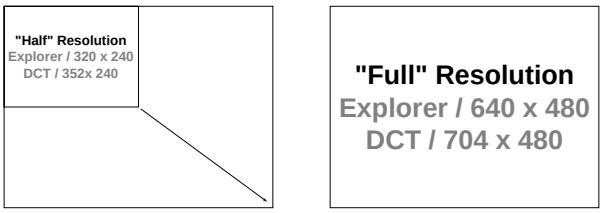
While graphics may be compressed in several creative ways, their resolution is ultimately dictated by the settop's operating system. Resolution is actually made up of two separate parameters: size and color depth. The supported sizes and color depths of some typical settop boxes are shown below:

Settop Class/ Manufacturer	*Graphic Resolution*	*Color Depth*
Explorer 2000/ Scientific Atlanta	320 x 240* at 72 dpi	16 bits
Explorer 8000/ Scientific Atlanta	640 x 480**	16 bits
DCT-2000/ Motorola	352 x 240	Up to 8 bits
DCT-5100/ Motorola	704 x 480	Up to 24 bits

*640 x 480 supported, but not typically used due to memory constraints** Typical use; up to 720 x 480 supported

(Motorola) settop boxes presents a developer with some intriguing differences. Notice from the table above that resolutions and color depth vary on both systems on almost a box-by-box basis. Developing for lower-end settops has attendant difficulties: low resolution graphics must be recreated for the same application on a higher-end box.

Graphic resolution is often dictated by the largest image size accommodated, then scaling back if necessary. For example, the settops listed above use so-called "full" and "half" resolutions, meaning the fullest resolution uses twice the horizontal and vertical pixel count as the lowest. By some simple visualization one can see these labels are somewhat misleading: a full resolution image (at the same color depth) actually requires *four times* the memory capacity as a half resolution image (see below).



Comparison of "Half" and "Full" Graphic Resolutions

Use of low resolution graphics is often an attractive tradeoff when memory space is a consideration. In this case graphics are expanded ("stretched") at twice their horizontal and vertical resolution to fill the screen. Fortunately the NTSC television display standard—which relies on flickering interlaced lines rather than pixels—often compensates for the low resolutions artifacts (or "stretch marks").

Video: Scale with Caution

Settop applications have little control over full-screen video playback resolution; the MPEG video compression standard (used in all digital settops) uses its own display parameters. However, when video plays back at less-than-full-screen, e.g., in a quarter screen "window", the settop application chooses the "scaled" video window size.

Since settops scale video by simply removing pixels, video playback in windows at non-fractional sizes can introduce visual anomalies: "pixelated" objects and jagged diagonal edges. In this case the relatively low resolution of NTSC television doesn't help and often hurts: degraded video images still move at 30 frames/second and are more likely than graphics to be noticed by viewers.

What's Your Type?

Like graphics and video, on-screen type must abide by resolutions largely set by the settop operating system. Unlike these other visual criteria, however, type resolution is determined by *how* it is rendered to the screen rather than the number of pixels or color depth of its characters.

Most operating systems support both vector-based and anti-aliased fonts; the difference can be seen in the images below:

Vector-based *Anti-aliased* *Bitmapped*

Vector-based fonts rely on mathematical formulas to print characters with smooth edges, not unlike a standard laser printer. Although these formulas take up very little space (approximately 32 kilobytes per typeface; bold and italic versions not included) many settops exclude them because developers don't tend to use the same typeface—and including more than a few creates storage difficulties.

Anti-aliased fonts typically take up less space than vector-based fonts—and surprisingly look better at some resolutions—but have two non-trivial drawbacks: 1) they must be adjusted—sometimes pixel-by-pixel—at low resolutions to read properly; and 2) the "fuzzy" edges of their characters must be legible against every background on which they're rendered.

If an application cannot use either vector-based or anti-aliased fonts, settops typically provide several bitmapped fonts as a fallback. Characters from this font type have very rough edges but some typefaces rendered in this fashion can be reasonably legible and acceptable for limited application use.

TV DISPLAY TECHNOLOGY: "ART IS NOTHING WITHOUT LIMITS"[i]

Since settop applications focus primarily on *displaying* images to viewers (and occasionally receiving feedback) developers need a thorough understanding of television visual technology. Ignorance of these constraints doesn't necessitate catastrophic consequences—but the resulting applications can look pretty ugly. Interfaces that flout technical constraints exhibit overly bright colors, distorted images, and illegible or off-screen text and graphics.

Two basic technical constraints on displays come straight from the arcane world of television post-production: title safety and color limits.

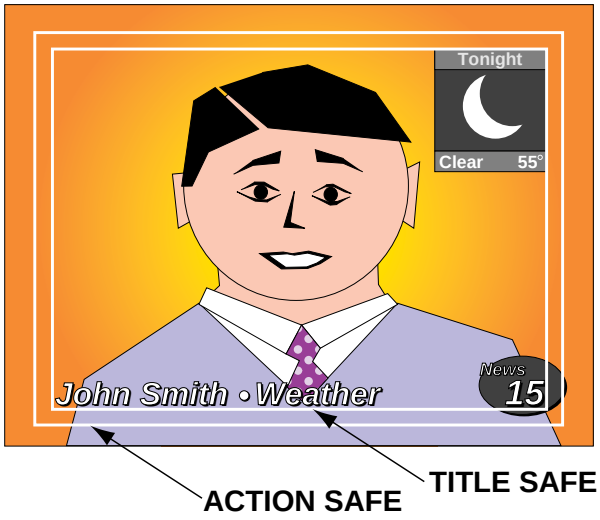
Better Safe Than (Really) Sorry

Commercial televisions[ii] do not show an entire video signal; rather, the TV monitor itself visually cuts off the outside edges. Images and colors at the edge of the display appear to "bleed off" the border of a TV. The reasons for this are complex, but this type of display has some advantages, e.g., allowing certain unsightly video artifacts like the vertical blanking interval, or VBI, (where closed-captioning information is transmitted) to be hidden off-screen.

To complicate matters, TVs differ—sometimes drastically—in the actual amount of signal they cut off; that is, two TVs may display different amounts of the same signal.

Televisions uses two cut-off display conventions: *title safe*, meaning the area where *no* TV will *ever* cut off any portion of an image and *action safe*, meaning the area where on-screen action (e.g., something moving) must be contained. Action safe is slightly larger than title safe since viewers are not especially bothered by moving objects occasionally disappearing at the edges of the screen; their eyes can compensate for the

movement. The image below shows how the safety areas work on a standard 4:3 television display.



Comparison of Full-Screen Signal, Action Safety, and Title Safety

So how much should images be reduced to "fit" into these safety areas? Many rules-of-thumb exist, but a good conservative standard is 20% less than full-screen for title safety and 15% less for action safety.

NTSC: Never Twice (the) Same Color

As for color limits, televisions display a rather narrow range. Historical precedent is at work here: color usage was nearly an afterthought when televisions were initially developed. When black-and-white television was the standard, video was recorded at a speed of 29.97 frames-per-second; the remaining 0.03 cycles (to fill-out the 30 frames/second standard) were reserved for the (future) color spectrum. This range—0.03 Hz—is not large, and favors blue and green hues at the expense of other colors.

Settop developers wrestling with color should be aware that televisions—unlike standard computer monitors—are not calibrated to saturation values. A PC display typically uses colors in mixed values of red, green, and blue (or R/G/B); these are color saturation values, describing a linear scale of how much of each color is included to create a final value. (This technology largely mimics the process used for printing, which uses a mixture of four colors: Cyan, Magenta, Yellow, and Black, or CMYK.)

Television, however, create colors based on chroma, luminance and other values that are *not* direct mixes of saturated colors. This process makes television sets very sensitive to sharp changes in brightness and contrast, especially with highly saturated colors, e.g., rich red, pure black and white. Offending this sensitivity leads to displays that bleed or "buzz" at the edges of colored areas of text. And strong contrasting colors aligned vertically create a bowing effect, e.g., the vertical separated image appears to curve inward or outward, depending on its on-screen placement.

Armed with technical knowledge about television display constraints, developers should also consider conforming to less-formal design limitations:

- Conform to standard television graphic conventions. These include how screen objects move or animate, how images are rendered against other images, and how screens of information or video transition from one to the next (e.g., via visual effects like "wipes" and "dissolves").
- Avoid inactivity. While an interface that must be addressed by a viewer may be "held" for a short time, never forget television is a visual, moving medium.
- Make text BIG and BOLD—and ensure messages stand out from the background on which they're rendered.

This paper merely scratches the surface of the rich subject of TV display technology; the principles delineated here only aim to starting a design on a firm technological foundation.

CONCLUSION: EMBRACE YOUR LIMITS—YOU'LL ALWAYS HAVE THEM

We can all dream of settops with unlimited memory and blisteringly fast processors supporting perfectly flexible middleware, full-resolution graphics and pixel-perfect video. But until then we have to make choices. The limitations involved in settop software development won't go away tomorrow. Even if they did—mirroring the evolution of the PC development world—new limitations would surely replace them.

Asking the right questions *before* development begins keeps application requirements and expectations in perspective. While demonstrations often provide a nice preview of application features, knowing the technical tradeoffs in *deployable* settop software is crucial now and will soon be indispensable. The number of settop technologies—and software to support them—shows no signs of abating.

---

[i] A paraphrased quote, ascribed variously to Beethoven, Picasso, and Goethe, among many others.

[ii] Note this discussion presumes a standard, interlace-scan, analog display television. Televisions displaying a true digital or high definition signal—whether progressive- or interlace-scan—are outside the scope of this discussion.

Stephen Johnson is technology consultant specializing in television interface design. He can be reached at steve@coachmedia.com.