Proceedings of LISA-NT:
The 2nd Large Installation System Administration of Windows NT Conference

Seattle, Washington, USA, July 16–17, 1999

# DEPLOYMENT OF MICROSOFT WINDOWS NT IN A DESIGN ENGINEERING ENVIRONMENT

Jason Sampson, Elwood Coslett, Gary Washington,
Bob Paauwe, Russ Craft, and Kevin Wheeler

# Deployment of Microsoft Windows NT In a Design Engineering Environment.

Jason Sampson, Elwood Coslett, Gary Washington,
Bob Paauwe, Russ Craft, Kevin Wheeler

*Intel Corporation*

## Abstract

This paper details some of the experiences and issues Intel Corporation encountered when developing and deploying a Microsoft Windows NT 4.0* based environment for use in by our chip designers. We are deploying Intel* based Windows NT workstations to engineers designing Intel's next generation products.

We offer our experiences in hope that other groups using NT in a design engineering environment can benefit. Along the way we explain how we solved the problems associated with a tightly controlled design engineering workstation and environment presented us.

## 1. Introduction

In 1995-1996, Intel Corporation began to deploy Windows NT to design engineering groups at our sites in Folsom, CA, Hillsboro, OR and Haifa, Israel.

All of our design engineering sites have local administrative groups that support the local Design Engineering departments. The groups typically assist with normal day to day operations and maintenance of the environment while working with the Design groups to plan and develop engineering environments to meet their future needs.

The number of design engineers varies from site to site. Design engineering customers total about 5000-6000 individuals world wide, all of which run UNIX and/or NT. The need for a well-planned and functionally complete heterogeneous environment is imperative to the success of each project. The total number NT design engineering workstation number in the 1000-2000 systems worldwide.

## 2. Workstation Models

Our deployment of NT followed a progression from an PC running as an "Xterm" all the way up to a full-fledged engineering workstation that has full access to both NT and UNIX environments, either natively or through emulation.

## 2.1 Xterm Model

Our initial effort was to deploy Intel based NT workstations to the engineer's desk. The engineer could access our UNIX environment through an X Windows emulation product. The NT workstations effectively became a powerful X-Term.

By deploying Windows NT, our engineers had the convenience of having all of the productivity tools available on Windows NT (Word*, Excel*, Outlook*, etc) as well as complete access to the UNIX tools needed to design our products. This allowed us to reduce costs by providing a single desktop to meet all of our engineer's needs. It also offers us the ability to move on to an environment where some or most of the design engineer's tasks are done locally to the workstation.

## 2.2 Standard Infrastructure Release

When our design sites started with NT, they individually developed their own build. This presented problems when projects had to share data and scripts between sites using different builds. A significant amount of work would be necessary to port scripts to another NT environment.

To rectify this problem, we pulled together the best features of the existing NT 4.0 environments and built a standard build, called the Standard Infrastructure Release, for deployment at all of our engineering sites. This standardization not only helps with portability, but also allows us to share the expense of maintaining the environment across multiple sites. This becomes particularly important when we consider our application integration process.

## 2.3 Work Model

In our design engineering UNIX environments, the design engineers never have administrative (root) rights on their workstation. In addition, applications and data are not permanently stored on the workstation. The UNIX workstations are considered application-less and data-less. This offers the ability to replace systems quickly and have the user back up and running in short time. Should a system fail, there is no reason to worry about re-installing applications, re-enabling daemons, etc. because all workstations have exactly the same build and features enabled. Very rarely is one machine significantly different than another.

In our UNIX based design environments we have developed, and continue to develop, an environment that allows any engineer to run any tool on any workstation anywhere within the company. This "any-any-any" concept is a guiding principle that is also applied to our NT workstations. Paths to applications and tools are consistent across all sites. In addition, the exact same version of each tool or application is usually available at every site.

When developing our NT environments, we worked to apply these same principles. Our NT workstations are application-less and data-less. Also, the exact same version of each tool is available at every site and it is found in same location.

Like with our UNIX workstations, we are able to swap out replacement of defective systems without worry of data loss. The engineers are able to be back to work quickly without having to wait many hours to install all of his or her required applications.

The application-less workstation thus requires that most applications run from a file server. While this is common place in UNIX, it is less commonly found within NT environments. This has forced us to create tools and processes to re-engineer application installations to allow them to run from the network. However, we've also found that some applications just refuse to run from a network location. They usually want to write temp files to the directory they are installed in, instead of using %TEMP%. In that case, we are forced to install the application locally and open the ACLs to allow the application to run.

Since many CAD tools are very large, and the number of tools used by the engineers is extensive, their time is saved re-installing these tools by minimizing the installation process to include only the required local changes like DLLs, registry and profile changes. The bulk of the application is place on the network file server.

In an attempt to apply these effective practices employed to manage our UNIX environments, our NT workstations also have restricted file system permissions. Users generally are not granted administrative rights and can not install just any software package they desire. This places the burden of software installation on the administrative group.

In addition to preventing application installation by the user, we prohibit the permanent storage of data on workstations, thus workstations are not backed up. Personalized components of the application are stored on a personal drive instead of the profile. The data stored on the personal drive, which is a drive located on a network file server, could easily be 50 to 100MB in size. Should we store this in the user's profile, login times would be greatly increased.

Locking down the workstation and preventing users from installing their own application generally increases stability of the workstation. Workstation blue screens and application crashes are reduced.

Another driver for locking down the workstation is our requirement to run batch jobs. To be successful, a batch environment requires a highly reliable and consistent environment. Locking down the workstation and providing consistent tools and tool locations make for a reliable environment.

## 3. Building the OS Image

Our workstation builds utilize Microsoft's OEM installation process. Using this process, and a few custom scripts, we are able to create an image that is consistent. Any two systems using the same build should not have different components installed, and they are never installed in a different order.

## 3.1 Partitioning

Our workstations are split into three partitions, C: D: and E:. C: is for OS, system administration tools and the swap file. D: is for any local applications and E: is a user scratch area. By breaking the system into multiple partitions, we're able to reserve space on the system for specific functions. Since the applications are always installed on the D: drive, we can be assured that the C: drive will never fill up because only OS and system administration tools are found on that drive. Same goes for the E: drive, since the user has almost free reign on that drive, we do not have to worry about an application failing to install because the user has filled the application drive with temporary data.

With each install, the person building the system answers a few questions on the build diskette, including whether to FDISK the system, and then the install process takes over from there.

Partitioning is handled using freeware tools to wipe the partition table and scripted FDISK commands to create the C:, D: and E: partitions. Internally developed command line tools extend the D: and E: partitions beyond the 2GB limit presented by our DOS based build disk.

### 3.2 Disk Cloning and the OEM Build Process

The OEM build process is an effective means of creating consistent and fairly easy to maintain and extend build images but there are a few limitations.

One of the most serious limitations is that the OEM build process is time consuming; building a system from a boot disk can take a forty-five minutes to a hour depending on network speeds, size and number of applications installed. Our standard build installs the OS, Service Packs, Hotfixes and the productivity applications needed on every system.

To reduce the time taken to build a system, at some sites we've utilized disk cloning to build our systems. The original image is still built using the OEM build process, from which the clone is generated. The result is a workstation image than can be installed in about 10-15 minutes instead of a closer to one hour.

A disadvantage of disk cloning is that it requires a re-creation of the cloned images if you need to change something as simple as a device driver file. We have standardized on only a few workstation models to support, thus the number of images needed is very small. Also the time between releases of our OS image is relatively long, thus we do not have to recreate the cloned images several times. It is our opinion that the time benefits saved installing the cloned image outweighs the cost of creating and recreating the initial clone image.

### 3.3 ACLs

Once we established a process to install the OS, we extended it to customize the build Microsoft provides. First we used the Zero Admin Kit for Windows* to build our initial ACL settings. This kit, freely available from Microsoft, provides a script to set the ACLs on the local file system to remove Everyone:Read from the file system ACLs and to set them to Users:Read-Only for the C: partition. We applied the same ACL set to the D: partitions, where applications are installed.

Another tweak we made is to %Systemroot%\system32, which has the directory permissions set to Users:Add & Read, Creator-Owner:Full Control. The files provided at build time are set to Users:Read-only. With this setting, users could add new files to the directory, but couldn't change any of the files that were there at build time. The intent of this setting was to allow "good" programs to install to the system, letting them add files to the system32 partition, but not change anything that was already there. This proved in the end to be a very minor impact.

These ACL settings provided the greatest level of controversy with our customers. Many of our customers who were familiar with Microsoft Windows, Windows 95/98 and NT were accustomed to having full administrative rights or the ability in general to install any application they wished. To them, their NT system was their personal computer to do with as they pleased. Our wish was to treat these machines like tightly controlled UNIX workstations, not personal computers. This mindset change was difficult for some to get over, but once they realized their environment was more stable, they began to accept the locked down system.

When these same users began to use our locked down build, they quickly found that most applications could not install on the system. The installation programs would either complain about the inability to copy a DLL, or simply fail with an error message. They would even complain if the version of the file there were copying was exactly the same as what was on the system. The installation required copying the file no matter what. Some good applications would allow you to skip those types of errors and continue with the install, but unfortunately those were very far and few between.

Since the users were not able to install their own applications, it placed the entire burden of application installation on the heads of the administrative support team. We needed a means to install the application on the workstation remotely and consistently.

## 4. Software Distribution and Installation

The application-less workstation requires that we install as many tools as possible to run from a file server. The file server location also had to be read-only to all users.

We did not want users writing to temp files, etc to the application server.

These requirements resulted in an application installation process that is summarized to updating the local and user's registry, creating short cuts to the network location, and copying DLLs to the system32 directory.

Most of the applications we use do not provide an option of installing and running the application from a network file server. This means in order to make our locked down, application-less workstation model work, we were going to have to re-create the vendor installation that would allow the application to run from the network.

Since we also intended to run batch jobs on our NT workstations at any time of the day, we wanted to minimize job loss by minimizing the number of reboots of a system. Many vendor installs seem to require reboot of the system simply to start a service or perform some other simple function. However, most of the time a reboot was only necessary to update a locked DLL.

We further complicated the problem by requiring that no DLL could ever be downgraded by an application installation. Our experience has been that DLLs generally are backward compatible with an older version. A tool should run, with a high level of certainty, when a newer version of a DLL it depends on is found on the system. Microsoft has published guidelines and made tools available for ISVs to use to perform DLL version checking, but unfortunately, not all applications can go through these checks. Since it is sometimes difficult to tell if the vendor has been able to go through the DLL version checking, we were forced to implement our own DLL versioning system.

## 4.1 DLL Management

Combining the problem of a lack of network installs for applications with the questionable lack of DLL management in vendor installs, we were forced into creating our installation process and tools.

These internally developed tools perform most of the functions needed to install an application. Where they do not, we rely on NT 4.0 Resource Kit* tools like, SC, to handle specialized tasks.

Our tools also allow the administrator to create an application installation script that would allow the tool to run from a local drive or network location with a few simple variable changes. Every application installation script we make, called a module, is variable-ized. All path locations and site-specific settings are replaced with a variable, and then at installation time, those variables are resolved to values. The result is that one site can create a module specific to their site, and send it to another site, and with just a few changes in a text file, the module should be able to run at the new site.

Perhaps most importantly, our tools nearly give us complete control over what happens to the DLLs on the system. When we started, we used a PERL script to manage our DLLs on the system. When a new application image was created, the DLLs that application installed were checked against a central location. The versions were compared using FILEVER from the NT 4.0 Resource Kit. If the application wanted to install a newer version of the DLL, we would update the central location with that version. A table was kept to track all of the file versions of the DLLs found in the central location.

When an application installed, all of the DLLS provided by the original installation program are copied to the system. Then the PERL script would check whatever was currently on the system against the central location. If the local version were older, we would update the local version with what was found in the central location. If they were the same version, the file was not updated.

When this process was followed, we were able to test multiple applications against the same set of DLLs to validate functionality. However, due to time spans between releases, and the requirement to have site specific tools, the central directory quickly became out-dated.

To improve the DLL management process and to remove dependency on the central location, we built into our installation tool the same logic our PERL script had, excepting that we used native Win32 calls to get the file version. This greatly improved installation time by preventing unnecessary file copies and the execution of a PERL script. It removed the dependency on the central location by preventing the downgrade of the DLL in the first place.

What we found in using this process is that we were nearly able to remove any concern about DLL loading order because the newest version of the DLL should be found on the system. However, what we also found is that locked DLLs are more troublesome than originally thought.

When our installation tool encounters a locked DLL, we use a slightly different process than most vendor installs use. Most vendor installs set the system up to replace the locked DLL with the newer version at reboot. This forces the user to reboot the system once the application install is completed. The newly installed application cannot use the new DLL until the system reboots, even if the locked DLL becomes unlocked before reboot. Our process renames the locked DLL (e.g. ren msvcrt.dll msvcrt.dll.old23452) , copies in the new DLL (e.g. copy msvcrt.dll %Systemroot%\system32), and then use the Win32 API call to delete the locked file at the next reboot. This process allows an application to immediately use the new version of the DLL, should the locked version be released. Thus a reboot is not necessary.

However, DLLs locked by the OS are not released until reboot. This ended up being somewhat problematic.

When querying the file version table of a locked DLL, one that is in use by the OS or other application, we found that the FILEVER tool and the Win32 API calls always return the version of the file loaded into memory. Thus, even if the file has been replaced using our DLL update process, the version that is loaded into memory will be returned. The only way to combat this problem is to require the user to reboot the system before installing additional tools or to rename the locked DLL, check the version, and then rename it back. This of course happens fairly frequently and has made utilization of NT in a batch environment more difficult, since a rebooted system means lost batch jobs.

## 4.2 Application Installation and Differencing Technology

Our application installation requirements, the ability to run from the network or locally, DLL management, and minimization of reboots, has required us to develop our own tools for software distribution and installation.

As a basis for installs, we are using a process that uses a differencing tool to create a before and after snapshot of the system and then create a new installation script that meets our application requirements.

First we build a bare bones system that includes only the OS, service pack and hotfixes. This is done from an option on our build disk that prevents any of the standard applications from being installed. In effect, we have installed NT using the three boot disks and CD, and running the service pack and hotfix installs manually.

Next we use the differencing tool to take a pre-installation snapshot. The application is installed using the vendor installation program. If we are trying to create a network install, we install it to a simulated network drive. Using SUBST or a local share, we map a pseudo-application drive to X:, our drive letter for our applications. Then the application may also be configured with a customized set of default options. Finally, the differencing tool captures a post-installation snapshot and generates a listing of changes to the workstation that make up that instance of the application installation. We use the results of the differencing tool to generate our installation scripts using our internal tools.

The differencing technology has some distinctive advantages:

- Administrators know exactly what changes are made to the local DLLs, registry and the user's profile. A source of a problem can be tracked down by searching through all of the registry changes, which are kept in plaintext REGEDIT format. A more comprehensive workstation audit is also possible because we know what is supposed to be on the system and where to go to address deficiencies. Without a complete understanding of what files and registry keys are used by an application, problem determination is hampered.
- Conflicts between applications can be identified in advance. Some applications will set registry values to one value, another will want it with another value.
- Removes user interaction from install; installations are consistent. We can create an install that looks exactly the same as every other install.

One problem we've encountered when using this differencing technology is that the output of the tool may be only one possible installation scenario. The differencing tool is not able to identify and replicate the logic built into the install that may be triggered when a particular file or registry entry is present. Thus, the differencing technology is prone to bugs. We've attempted to minimize this problem by trying to guess what the vendor install may do in particular situations, or patch the install when we've encountered a problem with the original install.

In addition, many times registry keys are identified as being changed during the install that either are side effects of the original install or part of normal NT operation, like the Most Recently Used lists or last shutdown time key. Identification of these types of keys is not easy for the new administrator.

The administrator must learn through practice, extensive training and a little bit of art to determine which keys in a registry file are applied and which ones are removed. This learning curve further complicates the application integration process.

Invariably, the resulting application image has bugs or introduces problems through inconsistent application of standards. Many of these issues have been addressed through scripts we've developed to tackle the usual situations an application integrator may encounter, however, intimate knowledge of NT application installation and some good detective work are necessary to provide a high quality application installation.

Clearly the amount of effort needed to integrate an application is very high. In addition, because we force applications to run from locations where they were not tested or intended to run from, and because we are not using the vendor installation method, getting support from a vendor on application issues may be difficult or impossible.

## 4.3 Split Installs

With most application images, we create two installation scripts. One to apply the system registry and local file system changes, and another script to apply the user profile and personal drive changes. This split, without automation, can also be an intensive, error-prone task.

The system changes are performed by a service running in a local administrative context. Thus any DLLs or registry entries that require administrative rights to update can be handled by this service. The user does not need to have administrative rights to install the product.

The user changes are applied in user context. In our environment the LocalSystem account rarely has rights to access the user's personal drive, so it is necessary to install the user portion of an application by running as that user.

This split of the system and user portions of the application installation also allow us the luxury of being able to "push" the system portion, usually the most resource intensive aspect of an installation, to many systems in the off hours of the day. We are then able to schedule the installation to reduce impact on our file servers and networks. The engineers then simply have to run the user portion of the installation to complete the installation, which is usually very quick and less resource intensive.

We also have the ability to force the installation of the user portion of the application for all logged on users through a user process that is started from the All Users startup group. This user process checks the local system registry for jobs and will execute it.

Our tool suite also offers a pull mechanism. The user can open a GUI program select the application(s) he or she wishes to install and click one button to initiate the install. Depending on the size and location of the application, it will install in just a few minutes, up to a half-hour. Our underlying installation tool works with the GUI to let the user know whether to logoff and log back in, or to reboot the system. If no user action is necessary, the user should be able to immediately open the application and begin using the tool.

## 5.0 Conclusion

Recently we've taken a hard look at our processes and attempted to re-evaluate their effectiveness or worthiness. We've known all along that our application integration process, including the support for the internally developed tools, was an expensive cost to burden. We felt that the benefits achieved from the policies and processes we implemented outweighed the costs associated with support of the tools and processes we implemented.

Now with the introduction of Microsoft Windows Installer* technology based installation tools, and more ISVs using installation products that allow us to record installation options, many of our higher priority concerns have been addressed. We can now script our application installations, gather a fair amount of information about what changes on the system, and feel comfortable with the DLL management processes. It appears that the industry has made significant strides in addressing many of the issues that had no resolution when we began this process two to three years ago.

With these advances we're close to being able to retire our internally developed tools and rely more on third party solutions for most of our system administration needs.

However, that is not to say that there isn't anything left to fix. A short list of items we would have liked to have seen to make our implementation of NT more successful include:

- More command line tools, which means the ability to script administrative functions. We've had the luxury have having excellent resources internally who can write command line tools, but there are far more important items we should have to worry about.
- We need to minimize reboots. Applications have to be able to use the new version of a DLL without having to reboot.
- The option to install applications locally or on the network. Network and locally installed applications should use the Microsoft specified means to identify areas to write temp files to.
- A means of determining exactly what changes are made during the install both on the filesystem and the registry.

## 6.0 Acknowledgements

Throughout the past two to three years we've had a number of individuals that have made this paper possible. Thanks go out to the entire iA/NT development team for their hard work and dedication to developing a quality design engineering environment.

## 7.0 References

MS Windows NT Workstation Deployment Guide - Automating Windows NT Setup. Microsoft Corporation.

Microsoft Zero Administration Kit. Microsoft Corporation. http://www.microsoft.com/windows/zak/

Microsoft Windows NT Workstation 4.0 Resource kit, Microsoft Corporation, Microsoft Press, 1996.