Programmers Guide For
MapuSoft Standalone Products

MapuSoft Technologies
1301 Azalea Road
Mobile, AL 36693

For additional assistance, please contact us at:
MapuSoft Technologies
1301 Azalea Road
Mobile, Alabama 36693
251.665.0280
251.660.0288 FAX
support@mapusoft.com
info@mapusoft.com
http://www.mapusoft.com
Last Revised: 14/05/2009

Table of Contents

Chapter 1. About this Guide

This chapter contains the following topics:

- Objectives
- Document Conventions
- MapuSoft Technologies and Related Documentation
- Requesting Support

Objectives

This manual contains instructions on how to get started with the Mapusoft products. The intention of the document is to guide the user to install, configure, build and execute the applications using Mapusoft products.

Document Conventions

Table 1 defines the notice icons used in this manual.

Table 1: Notice Icons

Icon	Meaning	Description
☞	Informational note	Indicates important features or icons.
⚠	Caution	Indicates a situation that might result in loss of data or software damage.

Table 2 defines the text and syntax conventions used in this manual.

Table 2: Text and Syntax Conventions

Convention	Description
Courier New	`Identifies Program listings and Program examples.`
Italic text like this	`Introduces important new terms.` • `Identifies book names` • `Identifies Internet draft titles.`
COURIER NEW, ALL CAPS	`Identifies File names.`
Courier New, Bold	`Identifies Interactive Command lines`

MapuSoft Technologies and Related Documentation

Document	**Description**
Programmers Guide to Mapusoft Products	Provides detailed description of how to get started with MapuSoft Abstraction frame work and porting applications. • Explains how to generate standalone OS Abstractor/OS Changer packages
OS Abstractor Reference Manual	Provides detailed description of how to do abstraction solution. This guide: • Explains how to develop code independent of the underlying OS • Explains how to make your software easily support multiple OS platforms
OS Changer Reference Manual	Provides detailed description of how to get started with OS Changer. This guide: • Explains how to port applications to target platforms
OS PAL User Guide	Provides detailed description of how to use OS PAL. This guide: • Explains how to port applications • Explains how to import legacy applications • Explains how to do code optimization
Release Notes	Provides the updated release information about MapuSoft Technologies new products and features for the latest release. This document: • Gives detailed information of the new products • Gives detailed information of the new features added into this release and their limitations, if required

All the documents are available at http://mapusoft.com/products/techdata/.

Requesting Support

Technical support is available through the MapuSoft Technologies Support Center. If you are a customer with an active MapuSoft support contract, or covered under warranty, and need post sales technical support, you can access our tools and resources online or open a ticket at http://mapusoft.com/support/.

To submit a ticket, you need to register for a new account.

Registering a New Account

To register:

1. From OS PAL main page, select **Support**.
2. Select **Register** and enter the required details.
3. After furnishing all your details, click **Submit**.

Submitting a Ticket

To submit a ticket:

1. From OS PAL main page, select **Support > Submit a Ticket.**
2. Select a department according to your problem, and click **Next.**
3. Fill in your details and provide detailed information of your problem.
4. Click **Submit.**

MapuSoft Support personnel will get back to you within 48 hours with a valid response.

Live Support Offline

MapuSoft Technologies also provides technical support through Live Support offline.

To contact live support offline:

1. From OS PAL main page, select Support > Live Support Offline.
2. Enter your personal details in the required fields. Enter a message about your technical query. One of our support personnel will get back to you as soon as possible.
3. Click **Send**.

You can reach us at our toll free number: 1-877-627-8763 for any urgent assistance.

Chapter 2. Introduction to OS Abstractor

This chapter contains the OS Abstractor framework with the following topics:

- Introduction to OS Abstractor
- Installing OS Abstractor Products
- Installing OS Abstractor
- How to Use OS Abstractor
- Building BASE OS Abstractor Library
- Building BASE OS Abstractor Demo Application
- Building POSIX OS Abstractor
- Building POSIX OS Abstractor Library
- Building POSIX OS Abstractor Demo Application
- Using OS Abstractor under GNU Makefile
- Building micro-ITRON OS Abstractor

OS Abstractor Frame Work

Introduction to OS Abstractor Products

The following are the OS Abstractor products:

- BASE OS Abstractor
- POSIX
- micro-ITRON
- VxWorks
- pSOS
- Nucleus

OS Abstractor is designed for use as a C library. Services used inside your application software are extracted from the OS Abstractor libraries and are then combined with the other application objects to produce the complete image. This image may be downloaded to the target system or placed in ROM on the target system. OS Abstractor will also function under various host environments.

Application developers need to specify the target operating system that the application and the libraries are to be built for inside the project build scripts. Application developers can also customize OS Abstractor to include only the components that are needed and exclude the ones that are not required for their application.

If the Application also uses OS Changer products, additional configuration may be necessary. Please refer to the individual OS Changer documents.

Installing OS Abstractor Products

To install OS Abstractor products:

1. From OS PAL main menu, click on the Generate Standalone product button or select **Tools > Generate Standalone** on OS PAL main page.
2. Select the Target OS from the list and click **Next**.
3. Select the OS Changer or OS Abstractor products needed to create the standalone project and click **Next**.
4. Select the destination path to save the generated package and click **Finish**.

The successful standalone generation is displayed on Generator Verification window.

How to Use OS Abstractor

The steps for using OS Abstractor are described in the following generic form:

1. Include osabstractor.h in all your application source files.
2. Set the appropriate compiler switches within the project build files to indicate the target OS and other target configurations
3. Configure the pre-processor defines found in the osabstractor_usr.h header file under each target OS folder to applications requirements
4. Initialize the OS Abstractor library by calling OS_Application_Init() function. If you are also using POSIX OS Abstractor, then also use OS_Posix_Init() function call to initialize the POSIX component as well. If you use OS Changer(s), you may need to call other appropriate initialization functions as well. After initialization, create your initial application resources and start the application's first task. After this and within the

main thread, call OS_Application_Wait_For_End() function to suspend the main thread and wait for application re-start or termination requests.
5. Compile and link your application using appropriate development tools.
6. Download the complete application image to the target system and let it run.

Refer to the sample demo applications provided with OS Abstractor as a reference point to start your application. Please review the target processor and appropriate development tools documentation for additional information, including specific details on how to use the compiler, assembler, and linker.

Building BASE OS Abstractor Library

Before using OS Abstractor, make sure the OS and tools are configured correctly for your target. To ensure this, compile, link and execute a native sample demo application that is provided by the OS vendor on your target. Refer to the OS vendor provided documentation on how to compile, link, download, and debug the demo applications for your specific target and toolset. After this step, you are ready to use the OS Abstractor library to develop your applications.

Building BASE OS Abstractor Demo Application

The demo application is located at the \mapusoft\demo\osabstractor directory location. From this location, you will find the make files or project files at the appropriate specific/<OS>/<tool>/<target> directory. For instance, if you need the demo application to be built for Nucleus PLUS OS using visual studio 6 tools and for x86 target, then the make file location will be at specific\nucleus\visual_studio_6\x86 directory.

Building POSIX OS Abstractor

Before building the POSIX OS Abstractor library and/or application, ensure that the flags INCLUDE_OS_POSIX and INCLUDE_OS_PROCESS are set to OS_TRUE in the osabstractor_usr.h configuration file.

Building POSIX OS Abstractor Library

The POSIX OS Abstractor library is located at `\mapusoft\osabstractor_posix` directory. From this location, you will find the make files or project files at the appropriate specific/<OS>/<tool>/<target> directory. For instance, if you need the demo application to be built for Nucleus PLUS OS using visual studio 6 tools and for x86 target, then the make file location will be at specific\nucleus\visual_studio_6\x86 directory.

Building POSIX OS Abstractor Demo Application

The demo application is located at the `\mapusoft\demo_osabstractor_posix` directory location. From this location, you will find the make files or project files at the appropriate specific/<OS>/<tools>/<target> directory. For instance, if you need the demo application to be built for Nucleus PLUS OS using visual studio 6 tools and for x86 target, then the make file location will be at specific\nucleus\visual_studio_6\x86 directory. We need to have the Base OS Abstractor Library. It has to be included in all the OS Changer/Abstractor demos.
After every demo application, include/link in the POSIX base Abstractor library.

Building micro-ITRON OS Abstractor

Before building the micro-ITRON OS Abstractor library and/or application, ensure that the flag INCLUDE_OS_UITRON is set to OS_TRUE in the osabstractor_usr.h configuration file.

Building micro-ITRON OS Abstractor Library

The micro-ITRON OS Abstractor library is located at `\mapusoft\ uitron_osabstractor` directory. From this location, you will find the make files or project files at the appropriate specific/<OS>/<tool>/<target> directory. For instance, if you need the demo application to be built for Nucleus PLUS OS using visual studio 6 tools and for x86 target, then the make file location will be at specific\nucleus\visual_studio_6\x86 directory.

Building micro-ITRON OS Abstractor Demo Application

The demo application is located at the \mapusoft\ demo_osabstractor_uitron directory location. From this location, you will find the make files or project files at the appropriate specific/<OS>/<tools>/<target> directory. For instance, if you need the demo application to be built for Nucleus PLUS OS using visual studio 6 tools and for x86 target, then the make file location will be at specific\nucleus\visual_studio_6\x86 directory.

Building VxWorks OS Changer

Before building the VxWorks OS Changer library and/or application, ensure that the flag INCLUDE_OS_VxWorks is set to OS_TRUE in the osabstractor_usr.h configuration file.

Building VxWorks OS Changer Library

The VxWorks OS Changer library is located at `\mapusoft\ VxWorks_osabstractor` directory. From this location, you will find the make files or project files at the appropriate specific/<OS>/<tool>/<target> directory. For instance, if you need the demo application to be built for Nucleus PLUS OS using visual studio 6 tools and for x86 target, then the make file location will be at specific\nucleus\visual_studio_6\x86 directory.

Building VxWorks OS Changer Demo Application

The demo application is located at the \mapusoft\ demo_osabstractor_VxWorks directory location. From this location, you will find the make files or project files at the appropriate specific/<OS>/<tools>/<target> directory. For instance, if you need the demo application to be built for Nucleus PLUS OS using visual studio 6 tools and for x86 target, then the make file location will be at specific\nucleus\visual_studio_6\x86 directory.

Building pSOS OS Changer

Before building the pSOS OS Changer library and/or application, ensure that the flag INCLUDE_OS_pSOS is set to OS_TRUE in the osabstractor_usr.h configuration file.

Building pSOS OS Changer Library

The pSOS OS Changer library is located at \mapusoft\ pSOS_osabstractor directory. From this location, you will find the make files or project files at the appropriate specific/<OS>/<tool>/<target> directory. For instance, if you need the demo application to be built for Nucleus PLUS OS using visual studio 6 tools and for x86 target, then the make file location will be at specific\nucleus\visual_studio_6\x86 directory.

Building pSOS OS Changer Demo Application

The demo application is located at the \mapusoft\ demo_osabstractor_pSOS directory location. From this location, you will find the make files or project files at the appropriate specific/<OS>/<tools>/<target> directory. For instance, if you need the demo application to be built for Nucleus PLUS OS using visual studio 6 tools and for x86 target, then the make file location will be at specific\nucleus\visual_studio_6\x86 directory.

Building Nucleus OS Changer

Before building the Nucleus OS Changer library and/or application, ensure that the flag INCLUDE_OS_Nucleus is set to OS_TRUE in the osabstractor_usr.h configuration file.

Building Nuceus OS Changer Library

The pSOS OS Changer library is located at \mapusoft\ Nucleus_osabstractor directory. From this location, you will find the make files or project files at the appropriate specific/<OS>/<tool>/<target> directory. For instance, if you need the demo application to be built for Nucleus PLUS OS using visual studio 6 tools and for x86 target, then the make file location will be at specific\nucleus\visual_studio_6\x86 directory.

Building Nucleus OS Changer Demo Application

The demo application is located at the \mapusoft\ demo_osabstractor_Nucleus directory location. From this location, you will find the make files or project files at the appropriate specific/<OS>/<tools>/<target> directory. For instance, if you need the demo application to be built for Nucleus PLUS OS using visual studio 6 tools and for x86 target, then the make file location will be at specific\nucleus\visual_studio_6\x86 directory.

Chapter 3. OS Changer Framework

This chapter contains the following topics:

- About OS Changer
- How to Use OS Changer
- Conditional Compilations
- Porting Applications from Legacy Code to Target OS
- OS Changer Defines
- API Variations

Introduction to OS Changer

OS Changer is designed for use as a C library. Services used inside your application software are extracted from the OS Changer and TARGET OS libraries, and, are then combined with the other application objects to produce the complete image.
For more information on OS Changer Frame work, refer to the OS Changers section of this document.

About OS Changer

OS Changer provides extensive support to various common proprietary libraries widely used by the application developers. Further, developers can utilize the native TARGET OS interface as well. This works toward getting the migration effort faster, much easier and greatly reduce time-to-market period.

OS Changer is optimized to take full advantage of the underlying TARGET RTOS features. It is built to be totally independent of the target hardware and all the development tools (like compilers and debuggers).

Please note that there may be some minor implementation differences in some of the OS Changer APIs when compared to the native API's. This may be as a result of any missing features within the underlying RTOS that OS Changer provides migration to.

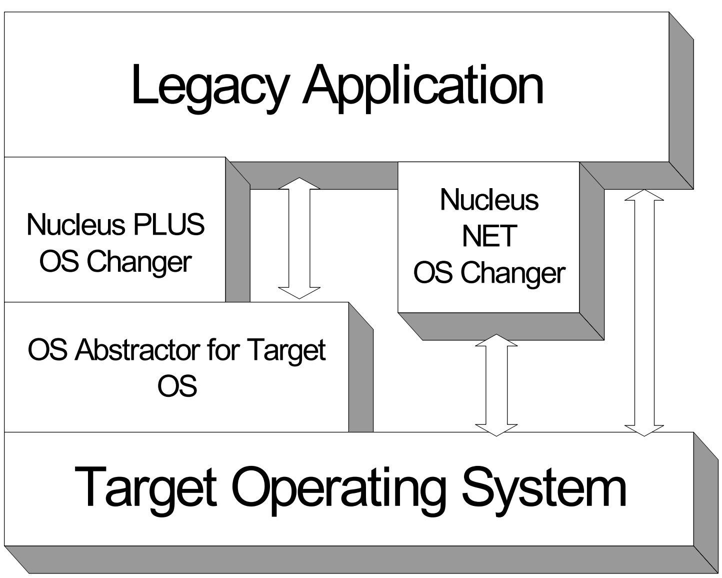


Figure 1: ***An example NUCLEUS OS Changer and Target OS Integration***

Your legacy application can be re-usable and also portable by the support provided by the OS Changer library and the OS Abstractor library. Applications can directly use the native target OS API, however doing so will not make your code portable across operating systems. We recommend that you use the optimized abstraction APIs for the features and support that are not provided by the OS Changer compatibility library.

NOTE: For more information on configuration and target OS specific information, see OS Abstractor Developer section of this document.

How to Use OS Changer

OS Changer is designed for use as a C library. Services used inside your application software are extracted from the OS Changer and TARGET OS libraries, and, are then combined with the other application objects to produce the complete image. This image can be loaded to the target system or placed in ROM on the target system.

The steps for using OS Changer are described in the following generic form:

- Remove the TARGET RTOS header file defines from all the TARGET RTOS source files.
- Remove definitions and references to all the TARGET RTOS configuration data structures in your application.
- Include the OSChanger_ TARGET RTOS.h (For example, OSChanger_Nucleus.h in case of OS Changer Nucleus) and osabstractor.h in the source files.
- Modify the OS Changer init code (see sample provided) and the TARGET RTOS root task of your application appropriately. (For example, Application_Initialize)
- Compile and link your application using appropriate development tools. Resolve all compiler and linker errors.
- Port the underlying low-level drivers to Target OS.
- Load the complete application image to the target system and run the application.
- Review the processor and development system documentation for additional information, including specific details on how to use the compiler, assembler, and linker.

Conditional Compilations

For more information on target specific configuration, refer to the OS Abstractor Developer section of this document.

Porting Applications from Legacy Code to Target OS

In most applications, using OS Changer is straight forward. The effort required in porting is mostly at the underlying driver layer. Since we do not have specific information about your application, it will be hard to tell how much work is required. However, we want you to be fully aware of the surrounding issues upfront so that necessary steps could be taken for a successful and timely porting. It is possible that we have not addressed all your application specific issues, so for further information, contact MapuSoft Technologies.

OS Changer Defines

The OS Changer library contains the following respective header files:

Module	Description
OSCHANGER_VXWORKS.H	This header file is required in all of the vxworks source modules. This header file provides the translation layer between the vxworks defines, APIs and parameters to OS Abstraction.
OSCHANGER_PSOS.H	This header file is required in all of the PSOS source modules. This header file provides the translation layer between the pSOS defines, APIs and parameters to OS Abstraction.
OSCHANGER_NUCLEUS.H	This header file is required in all of the Nucleus PLUS source modules. This header file provides the translation layer between the Nucleus PLUS defines, APIs and parameters to OS Abstraction.
UITRON_OSABSTRACTOR.H	This header file is required in all of the micro-ITRON source modules. This header file provides the translation layer between the micro-ITRON defines, APIs, and parameters to OS Abstraction

The OS Changer demo contains the following modules:

Module	Description
DEMO.C	Contains a sample demo application

You will find relevant make/project files for a specific RTOS in the specific RTOS directory following where you find the demo and the Changer library modules.

API Variations

Since API support is being added in each release, contact MapuSoft to get up-to-date support information for the latest OS Changer version.

Error Handling

Applications receive a run-time error via the OS_Fatal_Error() function on some occasions. This happens due to:

- Unsupported API function call, or
- Unsupported parameter value or flag option in a API call, or
- Error occurred on the target OS for which there are no matching error codes in OS Abstractor.

OS Changer calls OS_Fatal_Error and passes along an error code and error string. The OS_Fatal_Error handling function is fully customizable to the application needs. At the moment it prints the error message if the OS_DEBUG_INFO conditional compile option is set, then OS_Fatal_Error does not return. For more details on error handling and definition of this function, refer to the OS Abstractor Reference Guide. The non-zero value in the error code corresponds to the underlying RTOS API error. Refer to the target OS documentation for a better description of the error. Error Handling section lists the errors and the reasons for the occurrence.

Chapter 4. Using OS Abstractor with Native Tools

This chapter contains the information about the System Configuration with the following topics:

- OS Abstractor Tool Sets
- Using OS Abstractor under GNU Makefile Environment
- Building with Eclipse IDE**Error! Bookmark not defined.**
- Building with Windriver Workbench
- Building with QNX Momentics
- Building Visual Studio 6.0

OS Abstractor Tool Sets

OS Abstractor can be used in a multitude of toolsets. The distribution only includes project files for a small subset of the tools that OS Abstractor can be used with. If the project files for the tools you are using are not included, please contact MapuSoft to set up OS Abstractor for your tools.

Target Operating System	Project Files Included	Project File Paths
Windows	Eclipse	\osabstractor_windows\specific\windows_xp\
	Visual Studio 6.0	\osabstractor_windows\specific\windows_xp\x86\visual_studio_6
Linux	Eclipse	\osabstractor_linux\specific\linux\x86\eclips
	Make	\osabstractor_linux\specific\linux\x86\make
Solaris	Eclipse	\osabstractor_solaris\specific\solaris\x86\eclipse
	Make	\osabstractor_solaris\specific\solaris\x86\gnu
QNX	Momentics	\osabstractor_qnx\specific\qnx\x86\momentics
VxWorks	Windriver Workbench	\osabstractor_vxworks\specific\vxworks_rtp\x86\workbench_gnu \osabstractor_vxworks\specific\vxworks_kernel\x86\workbench_gnu
LynxOS	Make	\osabstractor_lynxos\specific\lynxos\x86\gnu
MQX	Metaware	\osabstractor_mqx\specific\mqx\arc\metaware
ThreadX	Eclipse	\osabstractor_threadx\specific\threadx\x86\
	Visual Studio 6.0	\osabstractor_threadx\specific\threadx\x86\visual_studio_6
Nucleus	Eclipse	\osabstractor_nucleus\specific\nucleus\mnt\
	Visual Studio 6.0	\osabstractor_nucleus\specific\nucleus\mnt\visual_studio_6
micro-ITRON	Renasas	\osabstractor_uitron\specific\sh\hew

The included project files for Windows, Linux, Solaris, QNX and LynxOS are setup to be used directly on the target operating system. The project files for VxWorks and MQX are setup to utilize the tools built in simulated environment. Nucleus, ThreadX, and micro-ITRON require separate OS files and simulators are provided in the following directories. These supporting projects need to be included in the workspace and built in conjunction with OS Abstractor.

Target Operating System	Supporting Files
Nucleus	\osabstractor_nucleus\specific\nucleus\mnt\mnt
ThreadX	\osabstractor_threadx\specific\threadx\x86\threadx_win32
micro-ITRON	\osabstractor_uitron\specific\sh\uitron_kernel

Using OS Abstractor under GNU Makefile Environment

Example: Build and execute application using OS Abstractor Library

NOTE: This example assumes all the source code, library, and makefile are in the following file structure:

```
OSabstractor_application
                                    → demo_osabstractor
                                          →include
                                          →source
                                          →specific
                                           →linux
                                                      → x86
                                                            → gnu
                                                                  → Makefile
                                    → include → include
                                    → osabstractor_linux
                                          → include
                                          → source
                                          →specific
                                           → linux
                                                      → x86
                                                            → gnu
                                                                  → Makefile
```

1. The rest of this topic will assume that your osabstractor_application directory is under the root directory.

2. To build the osabstractor library, open up a terminal and type:
   $cd /root/osabstractor_application/osabstractor_linux/specific/linux/x86/gnu
   $make clean all ROOT_DIR=/root/osabstractor_application/
   NOTE: After the compilation is completed, you should see a folder called "lib" under folder "osabstractor_application" which has the "libosabstractor_linux.a" file.

3. To build the osabstractor demo, open up a terminal and type:
   $cd /root/osabstractor_application/demo_osabstractor/specific/linux/x86/gnu/
   $make clean all ROOT_DIR=/root/osabstractor_application/

NOTE: After the compilation is completed, you should see "osabstractor_linux_demo" executable file under directory "/root/osabstractor_application/demo_osabstractor/specific/linux/x86/gnu/"

4. To execute/debug the demo executable, open up a terminal and type:
   $cd /root/ osabstractor_application / demo_osabstractor/ specific/linux/x86/gnu/
   $gdb osabstractor_linux_demo
   $run
   NOTE: If you need to modify the makefiles that build the demo application and the libraries, make sure you use an editor that will NOT add the carriage return character (each line should only have the line feed character), otherwise the 'make' utilities will not work correctly. To remove the carriage return character that was introduced by some editors, run the dos2unix utility to convert the dos format text file to unix format.

Building with Eclipse IDE

The eclipse specific project files are located in \<specific>\<OS>\<arch>\eclipse\ where "OS" is the corresponding target operating system and "arch" is corresponding architecture. For instance, if you need the demo application to be built for linux using eclipse tools x86 target, then the corresponding eclipse project file can be located in .\demo_osabstractor\specific\linux\x86\eclipse directory.
The Eclipse framework with CDT can be downloaded from http://www.eclipse.org/downloads/

To install Eclipse, follow the instructions at http://wiki.eclipse.org/Eclipse/Installation

To configure this macro in eclipse:
1. Select **Preferences** under the **Window** menu.
2. Expand **General > Workspace** and select **Linked Resources** node.
3. Click **New** and enter ROOT_DIR for the name and the full path to the workspace root.

To import the project files in Eclipse:
1. Select **Import** from **File** menu.
2. Expand **General** folder.
3. Select **Existing Projects into Workspace** and click **Next**.
4. Click **Browse** and navigate to the location of the project file.
5. The project name should appear under **Projects**.
6. Select the project to import and click **Next**.

To build the OS Abstractor library:
1. Select **OS Abstractor project** file.
2. Choose **Build Project** from the **Project** menu.

To build the OS Abstractor Demo:
1. Select **OS Abstractor Demo project** file.
2. Choose **Build Project** from the **Project** menu.

To debug the OS Abstractor Demo:
1. Select **OS Abstractor Demo project** file.
2. Choose **Open Debug Dialog** from the **Run** menu.
3. Select **C/C++ Local Application.**
4. Click **New Launch Configuration**.
5. Click **Run**.

Building with Windriver Workbench

The Windriver Workbench specific project files are of two types: kernel type projects and RTP type projects are located in .\<specific>\<OS>\<arch>\workbench_gnu. i.e, specific\vxworks_kernel\x86\workbench_gnu for kernel projects and.\specific\vxworks_rtp\x86\workbench_gnu\RTP respectively. For instance, if you need the demo application to build Kernel type projects, then the corresponding workbench project file can be located in \demo_osabstractor\specific\vxworks_rtp\x86\workbench_gnu directory.

The included project files require a path variable macro called ROOT_DIR to be defined.

To configure this macro in eclipse:

1. Select **Preferences** under the **Window** menu.
2. Then expand **General->Workspace** and select **Linked Resources** node.
3. Click **New** and enter ROOT_DIR for the name and the full path to the workspace root.

NOTE: Please refer Workbench documentation on how to build and debug.

Building with QNX Momentics

The QNX Momentics related project files are located in \<specific>\<OS>\<arch>\momentics where "OS" is the corresponding target operating system and "arch" is corresponding architecture. For instance, if you need the demo application to be built for QNX using Momentics tools and x86 target, then the corresponding Momentics project file can be located in \demo_osabstractor\specific\qnx\x86\momentics\ directory.

The included project files require a path variable macro called ROOT_DIR to be defined.

To configure this macro in eclipse:

1. Select **Preferences** under the **Window** menu.
2. Then expand **General->Workspace** and select **Linked Resources** node.
3. Click **New** and enter ROOT_DIR for the name and the full path to the workspace root.

To import the project files in Eclipse:

1. Select **Import** from **File** menu.
2. Expand **General** folder.
3. Select **Existing Projects into Workspace** and click **Next**.
4. Click **Browse** and navigate to the location of the project file.
5. The project name should appear under **Projects**.
6. Select the project to import and click **Next.**

NOTE: Please refer Momentics documentation on how to build and debug.

Building with Visual Studio 6.0

The Visual Studio 6.0 specific project files are located in `\<specific>\<OS>\<arch>\visual_studio_6\`.where `OS` is the corresponding target operating system and `arch` is corresponding architecture. For instance, if you need the demo application to be built for Windows XP using visual studio 6.0 tools and x86 target, and then the corresponding visual studio project files can be located in `\specific\windows_xp\x86\visual_studio_6` directory.

To import the project files in Visual Studio 6.0 do the following

1. Select **New** from **File** menu to create a new workspace.
2. Select **Workspaces** tab.
3. Enter a workspace name into the Workspace name text box.
4. Set the path to the root of location of the Mapusoft products.
5. Click **OK.**
6. In Workspace window choose **File View** tab.
7. Right click on **Workspace <project name>** tree node in the Workspace window and select **Insert Project into Workspace.**
8. Browse to the *.dsp you want to add to the project and click **OK**.

To build the OS Abstractor library:

1. Right click on the OS Abstractor project file.
2. Select **Build** from the pop-up menu.

To build the OS Abstractor Demo:

1. Right click on the OS Abstractor Demo project file.
2. Select **Build** from the pop-up menu.

To debug the OS Abstractor Demo:

1. Right click on the OS Abstractor Demo project file.
2. Select **Set as active project** from the pop-up menu.
3. Click **F5 key** on your keyboard.

Chapter 5. System Configuration

This chapter contains the information about the System Configuration with the following topics:

- System Configuration
- Target OS Selection
- OS HOST Selection
- Target 64 bit CPU Selection
- User Configuration File Location
- OS Changer Components Selection
- POSIX OS Abstractor Selection
- OS Abstractor Process Feature Selection
- OS Abstractor Task-Pooling Feature Selection
- OS Abstractor Profiler Feature Selection
- OS Abstractor Output Device Selection
- OS Abstractor Debug and Error Checking
- OS Abstractor ANSI API Mapping
- OS Abstractor Resource Configuration
- OS Abstractor Minimum Memory Pool Block Configuration
- OS Abstractor Application Shared Memory Configuration
- OS Abstractor Clock Tick Configuration
- OS Abstractor Device I/O Configuration
- OS Abstractor Target OS Specific Notes

System Configuration

The user configuration is done by setting up the appropriate value to the pre-processor defines found in the osabstractor_usr.h.

NOTE: Make sure the OS Abstractor libraries are re-compiled and newly built whenever configuration changes are made to the osabstractor_usr.h when you build your application. In order to re-build the library, you would actually require the full-source code product version (not the evaluation version) of OS Abstractor.

Applications can use a different output device as standard output by modifying the appropriate functions defines in osabstractor_usr.h along with modifying os_setup_serial_port.c module if they choose to use the format I/O calls provided by the OS Abstractor.

Target OS Selection

Based on the OS you want the application to be built, set the following pre-processor definition in your project setting or make files:

Flag and Purpose	Available Options
OS_TARGET To select the target operating system.	The value of the OS_Target should be for the OS Abstractor product that you have purchased. For Example, if you have purchased the license for : **OS_NUCLEUS** – Nucleus PLUS® from ATI **OS_THREADX** – ThreadX® from Express Logic **OS_VXWORKS** – VxWorks® from Wind River Systems **OS_ECOS** – eCOS standards from Red Hat **OS_MQX** - Precise/MQX® from ARC International **OS_UITRON** – micro-ITRON standard based OS **OS_PSOS** – pSOS systems from Wind River Systems **OS_LINUX** - Open-source/commercial Linux® distributions **OS_WINDOWS** – Windows 2000, Windows XP®, Windows CE, Windows Vista from Microsoft. If you need to use the OS Abstractor both under Windows and Windows CE platforms, then you will need to purchase additional target license. **OS_TKERNEL** – Japanese T-Kernel® standards based OS **OS_LYNXOS** - LynxOS® from LynuxWorks **OS_QNX** – QNX operating system from QNX **OS_LYNXOS** – LynxOS from Lynuxworks **OS_SOLARIS** – Solaris from SUN Microsystems For example, if you want to develop for ThreadX, you will define this flag as follows: OS_TARGET = OS_THREADX PROPRIETARY OS: If you are doing your own porting of OS Abstractor to your proprietary OS, you could add your own define for your OS and include the appropriate OS interface files within osabstractor.h file. MapuSoft can also add custom support and validate the OS Abstraction solution for your proprietary OS platform

OS HOST Selection

The flag has to be false for standalone generation.

OS_HOST To select the host operating system	This flag is used only in OS PAL environment. It is not used in the target environment. In Standalone products, this flag should be set to OS _FALSE.

Target 64 bit CPU Selection

Based on the OS you want the application to be built, set the following pre-processor definition in your project setting or make files:

Flag and Purpose	Available Options
OS_CPU_64BIT To select the target CPU type.	The value of OS_CPU_64BIT can be any ONE of the following: OS_TRUE – Target CPU is 64 bit type CPU OS_FALSE – Target CPU is 32 bit type CPU **NOTE**: This value cannot be set in the osabstractor_usr.h, instead it needs to be passed to compiler as –D macro either in command line for the compiler or set this pre-processor flag via the project settings. If this macro is not used, then the default value used will be OS_FALSE.

User Configuration File Location

The default directory location of the osabstractor_usr.h configuration file is given below:

Target OS	Configuration Files Directory Location
OS_NUCLEUS	`\mapusoft\osabstractor_nucleus\include`
OS_THREADX	`\mapusoft\osabstractor_threadx\include`
OS_VXWORKS	`\mapusoft\osabstractor_vxworks\include` Please make sure you specify the appropriate target OS versions that you use in the osabstractor_usr.h
OS_MQX	`\mapusoft\osabstractor_mqx\include`
OS_UITRON	`\mapusoft\osabstractor_uitron\include`
OS_LINUX	`\mapusoft\osabstractor_linux\include` Please make sure you specify the appropriate target OS versions that you use in the osabstractor_usr.h **NOTE**: RT Linux, for using rtlinux you need to select this option.
OS_SOLARIS	`\mapusoft\osabstractor_solaris\include`
OS_WINDOWS	`\mapusoft\osabstractor_windows\include` Any windows platform including Windows CE platform. If you use OS Abstractor under both Windows and Windows CE, then you would require additional target license. **NOTE**: Windows 2000, Windows XP®, Windows CE, Windows Vista from Microsoft
OS_ECOS	`\mapusoft\osabstractor_ecos\include`
OS_LYNXOS	`\mapusoft\osabstractor_lynxos\include`
OS_QNX	`\mapusoft\osabstractor_qnx\include`
OS_TKERNEL	`\mapusoft\osabstractor_tkernel\include`

If you have installed the MapuSoft's products in directory location other than mapusoft then refer the corresponding directory instead of \mapusoft for correct directory location.

OS Changer Components Selection

OS Abstractor optional comes with various OS Changer API solutions in addition to its BASE and POSIX API offerings. OS Changer APIs are used to port legacy code base from one OS to another. Select one or more OS Changer components depending on the type of code that you needed to port to one or more new operating system platforms. Set the pre-processor flag below to select the components needed by your application:

Flag and Purpose	Available Options
INCLUDE_OS_VXWORKS To include VxWorks OS Changer product. Refer to the appropriate OS Changer manual for more details.	OS_TRUE – Include support OS_FALSE – Do not include support The default is OS_FALSE
INCLUDE_OS_PSOS To include pSOS OS Changer product. Refer to the appropriate OS Changer manual for more details.	OS_TRUE – Include support OS_FALSE – Do not include support The default is OS_FALSE
INCLUDE_OS_PSOS_CLASSIC To include a very old version of pSOS OS Changer product. Refer to the appropriate OS Changer manual for more details.	OS_TRUE – Include support for pSOS 4.1 rev 3/10/1986 OS_FALSE – do not include pSOS 4.1 support The default is OS_FALSE
INCLUDE_OS_NUCLEUS To include Nucleus PLUS OS Changer product. Refer to the appropriate OS Changer manual for more details.	OS_TRUE – Include support OS_FALSE – Do not include support The default is OS_FALSE.
INCLUDE_OS_NUCLEUS_NET To include Nucleus NET OS Changer product. Refer to the appropriate OS Changer manual for more details.	OS_TRUE – Include support OS_FALSE – Do not include support The default is OS_FALSE.
INCLUDE_OS_UITRON To include micro-ITRON OS Abstractor product. Refer to the appropriate OS Abstractor manual for more details.	OS_TRUE – Include support OS_FALSE – Do not include support The default is OS_FALSE.
INCLUDE_OS_FILE To include ANSI file system API compliance for the vendor provided File Systems. Refer to the appropriate OS Changer manual for more details.	OS_TRUE – Include support OS_FALSE – Do not include support The default is OS_FALSE. This option is only available for Nucleus PLUS target OS

NOTE: For additional information regarding how to use any specific OS Changer product, refer to the appropriate reference manual or contact www.mapusoft.com.

POSIX OS Abstractor Selection

OS Abstractor optionally comes with POSIX support as well. Set the pre-processor flag provided below to select the POSIX component for application use as follows:

Flag and Purpose	Available Options
INCLUDE_OS_POSIX To include POSIX OS Abstractor product component.	**OS_TRUE** – Include support. You will need this option turned ON either if the underlying OS does not support POSIX (or) you need to POSIX provided by OS Abstractor instead of the POSIX provided natively by the target OS OS_FALSE – Do not include support The default is OS_FALSE.

NOTE: The above component can be used across POSIX based and non-POSIX based target OS for gaining full portability along with advanced real-time features. POSIX OS Abstractor library will provide the POSIX functionality instead of application using POSIX functionalities directly from the native POSIX from the OS and as a result this will ensure that your application code will work across various POSIX/UNIX based target OS and also its various versions while providing various real-time API and performance features. In addition, OS Abstractor will allow the POSIX application to take advantage of safety critical features like task-pooling, fixing boundary for application's heap memory use, self recovery from fatal errors, etc. (these features are defined else where in this document). For added flexibility, POSIX applications can also take advantage of using BASE OS Abstractor APIs non-intrusively for additional flexibility and features.

OS Abstractor Process Feature Selection

Flag and Purpose	Available Options
INCLUDE_OS_PROCESS	OS_TRUE – Include OS Abstractor process support APIs and track resources under each process and also allow multiple individually executable applications to use OS Abstractor OS_FALSE – Do not include process model support. Use this option for optimized OS Abstractor performance The default is OS_FALSE

The INCLUDE_OS_PROCESS option is useful when there are multiple developers writing components of the applications that are modular. The resource created by the process is automatically tracked and when the process goes away they also go away. One process can use another process resource, only if that process is created with "system" scope. A process cannot delete a resource that it did not create.

The INCLUDE_OS_PROCESS feature can also be used on target OS like VxWorks 5.x a non-process based operating system. In this case, the OS Abstractor provides software process protection. Under process-based OS like Linux, the processes created by the OS Abstractor will be an actual native system processes.

The INCLUDE_OS_PROCESS feature is also useful to simulate complex multiple embedded controller application on x86 single processor host platform. In this case, each individual process/application will represent individual controllers, which uses a shared memory region for inter-communication. This application could then be ported to the real multiple embedded controller environments with shared physical memory.

For more information regarding the process feature, refer to the section titled "Process Support" in the "Function Reference" chapter in this manual.

Process Feature use within OS Changer
It is possible for legacy applications to use the process feature along with OS Changer and take advantage of process protection mechanism and also have the ability to break down the complex application into multiple manageable modules to reduce complexity in code development. However, when porting legacy code, we recommend that the application be first ported to a single process successfully. Once this is completed, then the application can be modified to move the global data to shared memory and can be made to easily reside into individual process and or multiple executables.

To allow the legacy applications to be broken down into process modules and/or multiple applications the flag INCLUDE_OS_PROCESS needs to be set to OS_TRUE. Also the application needs to use OS_Create_Process envelopes to move the resources to appropriate processes. Legacy application can also make in multiple applications which then compile separately and can continue to use OS Changer APIs for inter-process communication. OS Changer APIs provides transparency to the application and allows the application to use the API among resources within a single process or multiple processes/applications.

OS Abstractor Task-Pooling Feature Selection

Task-Pooling feature enhances the performances and reliability of application. Creating a task (thread) at run-time require considerable system overhead and memory. The underlying OS thread creation function call can take considerable amount of time to complete the operation and could fail if there is not enough system memory. Enabling this feature, Applications can create OS Abstractor tasks during initialization and be able to re-use the task envelope again and again. To configure task-pooling, set the following pre-processor flag as follows:

Flag and Purpose	Available options
INCLUDE_OS_TASK_POOLING	OS_TRUE – Include OS Abstractor task pooling feature to allow applications to re-use task envelops from task pool created during initialization to eliminate run-time overhead with actual resource creation and deletion OS_FALSE – Do not include task pooling support The default is OS_FALSE

Except for the performance improvement, this behavior will be transparent to the application. Each process/application will contain its own individual task pool. Any process, which requires a task pool, must successfully add tasks to the pool before it can be used. Tasks can be added to (via OS_Add_To_Task_Pool function) or removed (via OS_Remove_From_Task_Pool function) from a task pool at anytime.

When an application makes a request to use a pool task, OS Abstractor will first search for a free task in the pool with an exact match based on stack size. If it does not find a match, then a free task with the next larger stack size that is available will be used. If there are multiple requests pending, a search will be made in FIFO order on the request list when a task is freed to the pool. The first request that matches or fulfills the stack requirement will then be fulfilled.

Refer to the MapuSoft supplied os_application_start.c file that came with the MapuSoft's demo application. The demo application pre-creates a bunch of fixed-stack-size (using STACK_SIZE as defined in osabstractor_def.h) task-pool-task as shown below:

```
#if (INCLUDE_OS_TASK_POOLING == OS_TRUE)
      for(i = 0; i < Max_Threads; i++)
      {
      OS_Add_To_Task_Pool(STACK_SIZE); /*this is a portion of code in
      init.c,
                                         STACK_SIZE should be changed
                                         according to the desired stack size
      }
#endif
```

Typically, applications would need a variety of threads with different stack size. If you would like to modify the demo application to use threads with larger or differing stack size, make sure you modify the os_application_start.c file according to your needs.

The OS_Create_Task function will be used to retrieve a task from the task pool. This will be accomplished by passing one of the flags OS_POOLED_TASK_WAIT or OS_POOLED_TASK_NOWAIT as a parameter to OS_Create_Task. When a task has completed and either exits, falls through itself or gets deleted by another task using the OS_Delete_Task function, the task will automatically be freed to be used again by the task pool. For further details, please refer to the OS_Create_Task specification defined in the following pages.

An Application can add or remove tasks with a specified stack size to the task pool at any time. The task pool will grow or shrink depending on each addition or deletion of tasks in the task pool. The Application cannot remove a valid task, which does not belong to the task pool. OS_Get_System_Info function can be used to retrieve the system configuration and run-time system status including information related to task pool.

If OS_TASK_POOLING is enabled, then all tasks POSIX threads created using the POSIX OS Abstractor POSIX APIs provided by POSIX OS Abstractor with POSIX and/or any task creation created using task create functions in any OS Changer products will automatically use the task pool mechanism with the flag option set to OS_POOLED_TASK_NOWAIT.

Warning: Your application will fail during task creation if OS_TASK_POOLING is enabled and you have not added any tasks to the task pool. Make sure you add tasks (via OS_Add_To_Task_Pool function) with all required stack sizes prior to creating pooled tasks (via OS_Create_Task function).

Special Notes: Task Pooling feature is not supported in ThreadX and Nucleus targets.

OS Abstractor Profiler Feature Selection

The following are the user configuration options that can be set in the osabstractor_usr.h:

Flag and Purpose	Available Options
OS_PROFILER Profiler feature allows applications running on the target to collect valuable performance data regarding the application's usage of the OS Abstractor APIs. Using the OS PAL tool, this data can then be loaded and analyzed in graphical format. You can find out how often a specific OS Abstractor API is called across the system or within a specific thread. You can also find out how much time the functions took across the whole system as well as within a specific thread Profiler feature uses high resolution clock counters to collect profiling data and this implementation may not be available for all target CPU and OS platforms. Please contact MapuSoft for any custom high resolution timer implementation required for the profiler for your target/OS environment. Refer to OS_Get_Hr_Clock_Freq() and OS_Read_Hr_Clock() for additional details on what target/OS platforms are currently supported by the profiler. The current release provides profiling capabilities for BASE OS Abstractor APIs only. The future releases will add support for POSIX OS Abstractor or OS Changer APIs.	Can either be: OS_TRUE – Profiler feature will be included. Profiling takes place with each BASE OS Abstractor API call. If profiler is turned on, also set the value for the following defines: **PROFILER_TASK_PRIORITY** The priority level (0 to 255) of the profiler thread.The profiler thread starts picking up the messages in the profiler queue, formats them into XML record and write to file. If the priority is set to the lowest (i.e, 255), then the profiler thread may not have an opportunity to pick the message from the queue in time and as such the queue gets filled up and as such the profiler will stop. The default profiler task priority value is set to 200. **NUM_OF_MSG_TO_HOLD_IN_MEMORY** This will be the depth of the profiler queue. The bigger the number, the more the memory is needed. A maximum of 30,000 profiler records can be created. Please make sure you increase you application's heap size by NUM_OF_MSG_TO_HOLD_IN_MEMORY times PROFILER_MSG_SIZE in the OS_Application_Init call. **PROFILER_DATAFILE_PATH** This will be the directory location where the profiler file will be created. The default location set is "/root". OS_FALSE – Profiler code will be excluded and the feature will be turned off. The default value is OS_FALSE.

If profiler feature is turned ON, then it needs to use the open/read/write calls to write to profiler data file. If you set OS_MAP_ANSI_IO to OS_TRUE then make sure you install the appropriate file device and driver.	

The profiler starts as soon as the application starts and will continue to collect performance data until the memory buffers in the profiler queue gets filled up. After, this the profiling stops and data is dumped into *.pal files at the user specified location. It is recommended that the profiler feature be turned off for the production release of your application.

If the profiler feature is turned OFF, then the profiler hooks disappear within the OS Abstractor and as such there are no impacts to the OS Abstractor API performance.

Special Notes: Profiler feature is not supported in ThreadX and Nucleus targets.

OS Abstractor Output Device Selection

The following are the user configuration options and their meanings:

Flag and Purpose	Available options
OS_STD_OUTPUT	Output device to print. OS_SERIAL_OUT – Print to serial OS_WIN_CONSOLE – Print to console User can print to other devices by modifying the appropriate functions within os_setup_serial_port.c in the OS Abstractor "source" directory and use OS Abstractor's format i/o calls. The default value is OS_WIN_CONSOLE

OS Abstractor Debug and Error Checking

Flag and Purpose	Available Options
OS_DEBUG_INFO	OS_TRUE – print debug info, fatal and compliance errors OS_FALSE – do not print debug info The default value is OS_TRUE
OS_ERROR_CHECKING	OS_TRUE – Check for API usage errors OS_FALSE – do not check for errors. Use this option to increase performance and reduce code size The default value is OS_TRUE
OS_IGNORE_FATAL_ERRO R	OS_TRUE – Return from OS_Fatal_Error() OS_FALSE – Stop execution when a fatal error occurs The default value is OS_FALSE

OS Abstractor ANSI API Mapping

OS Abstractor APIs can be mapped to exact ANSI names by turning on these features:

Flag and Purpose	Available options
MAP_OS_ANSI_MEMORY	OS_TRUE – map ANSI malloc() and free() to OS abstractor equivalent functions OS_FALSE – do not map functions. Also, when you call OS_Application_Free in this case, the memory allocated via malloc() calls will NOT be automatically freed. The default value is OS_TRUE **NOTE**: Refer to OS_USE_EXTERNAL_MALLOC define, if you want to connect your own memory management solution for use by OS Abstractor
MAP_OS_ANSI_FMT_IO	OS_TRUE – map ANSI printf() and sprintf() to OS abstractor equivalent functions OS_FALSE – do not map functions The default value is OS_FALSE
MAP_OS_ANSI_IO[1]	OS_TRUE – map ANSI device I/O functions like open(), close(), read(), write, ioctl(), etc. to OS abstractor equivalent functions NOTE: If your target OS is NOT a single-memory model based (e.g. Windows, Linux, QNX, etc.), then the OS Abstractor I/O functions are to be used within one single process/application.. If you need to use the I/O across multiple process, then set this define to OS_FALSE so that your application can use the native I/O APIs from the OS OS_FALSE – do not map functions The default value is OS_FALSE

NOTE: When you set MAP_OS_ANSI_IO to OS_TRUE, OS Abstractor automatically replaces open() calls to OS_open() during compile time when you include osabstractor.h in your source code. If you set MAP_OS_ANSI_IO to OS_FALSE, then in your source code when you include osabstractor.h, application can actually use both OS_open() and open() calls, where the OS_open will come from OS Abstractor library and open() will come from the native OS library. Given that OS Abstractor I/O APIs are similar to ANSI I/O, you probably can use the third option so that you eliminate some performance overhead going through OS Abstractor I/O wrappers if necessary. But, it is always recommended that application use BASE OS Abstractor or POSIX APIs instead of directly using native API calls from OS libraries for maximum portability.

OS Abstractor External Memory Allocation

OS Abstractor APIs can be mapped to exact ANSI names by turning on these features:

Flag and Purpose	Available options
OS_USE_EXTERNAL_MALLOC	OS_TRUE – OS abstractor can be configured to use an application defined external functions to allocate and free memory needed dynamically by the process. In this case, the OS Abstractor will use these function for allocating and freeing memory within OS_Allocate_Memory and OS_Deallocate_Memory functions These external functions needs to be similar to malloc() and free() and should be defined within osabstractor usr.h in order for OS Abstractor to successfully use them. This feature is useful if the application has it's own memory management schemes far better than what the OS has to offer for dynamic allocations. OS_FALSE – OS Abstractor will directly use the target OS system calls for allocating and freeing the memory The default value is OS_FALSE

OS Abstractor Resource Configuration

In addition to OS Abstractor resources used by application, there may be some additional resources required internally by OS Abstractor. The configuration should take into the account of these additional resources while configuring the system requirements. All or any of the configuration parameters set in osabstractor usr.h config file can be altered by OS_Application_Init function (refer to Chapter 3, Functional Reference for OS_Application_Init function specification) as well.

The following are the OS Abstractor system resource configuration parameters:

Flag and Purpose	Default Setting
OS_TOTAL_SYSTEM_PROCESSES The total number of processes required by the application	100 One control block will be used by the OS_Application_Init function when the INCLUDE_OS_PROCESS option is true
OS_TOTAL_SYSTEM_TASKS The total number of tasks required by the application	100 One control block will be used by the OS_Application_Init function when the INCLUDE_OS_PROCESS option is true.
OS_TOTAL_SYSTEM_PIPES The total number of pipes for message passing required by the application	100
OS_TOTAL_SYSTEM_QUEUES The total number of queues for message passing required by the application	100

OS_TOTAL_SYSTEM_MUTEXES The total number of mutex semaphores required by the application	100
OS_TOTAL_SYSTEM_SEMAPHORES The total number of regular (binary/count) semaphores required by the application	100
OS_TOTAL_SYSTEM_DM_POOLS The total number of dynamic variable memory pools required by the application	100 One control block will be used by the OS_Application_Init function when the INCLUDE_OS_PROCESS option is true.
OS_TOTAL_SYSTEM_PM_POOLS The total number of partitioned (fixed-size) memory pools required by the application	100
OS_TOTAL_SYSTEM_SM_POOLS The total number of shared partitioned (fixed-size) memory pools required by the application	100
OS_TOTAL_SYSTEM_EV_GROUPS The total number of event groups required by the application	100
OS_TOTAL_SYSTEM_TIMERS The total number of application timers required by the application	100

The following are the additional resources required internally by OS Abstractor:

Resources	Linux/Unix target	VxWorks Target
TASK	• 1 Event Group required by BASE OS Abstractor • 1 Event group required if application uses POSIX OS Abstractor and/or VxWorks OS Changer and/or pSOS OS Changer	1 Event group required if application uses POSIX POSIX OS Abstractor and/or VxWorks OS Changer and/or pSOS OS Changer
DM_POOL	• 1 Event Group required by BASE OS Abstractor	
QUEUE	• 2 Semaphores used by BASE OS Abstractor • 1 Semaphore used by POSIX OS Abstractor	1 Semaphore used by POSIX POSIX OS Abstractor
MUTEX		1 Semaphore used by BASE OS Abstractor
PROCESS	• 1 DM_POOL used by BASE OS Abstractor	1 DM_POOL used by BASE OS Abstractor
PM_POOL	• 1 Semaphore is used by BASE OS Abstractor	
Posix Condition Variable	• 1 Event Group required by POSIX OS Abstractor	1 Event Group required by POSIX OS Abstractor
Posix R/W Lock	• 1 Event Group required by POSIX OS Abstractor • 1 Semaphore required by POSIX OS Abstractor	• 1 Event Group required by POSIX OS Abstractor • 1 Semaphore required by POSIX OS Abstractor

If INCLUDE_OS_PROCESS feature is set to OS_FALSE, then the memory will be allocated from the individual application/process specific pool, which gets created during the OS_Application_Init function call.

If INCLUDE_OS_PROCESS is set to OS_TRUE, then the memory is allocated from a shared memory region to allow applications to communicate across multiple processes. Please note that in this case, the control block allocations cannot be done from the process specific dedicated memory pool since the control blocks are required to be shared across multiple applications.

For additional information related to memory definitions, please refer to Chapter 3, Functional Reference, section Process, and sub-section Memory.

OS Abstractor Minimum Memory Pool Block Configuration

Flag and Purpose	Default Setting
OS_MIN_MEM_FROM_POOL Minimum memory allocated by the malloc() and/or OS_Allocate_Memory() calls. This will be the memory allocated even when application requests a smaller memory size	16 (bytes) **NOTE**: Increasing this value further reduces memory fragmentation at the cost of more wasted memory.

OS Abstractor Application Shared Memory Configuration

Flag and Purpose	Default Setting
OS_USER_SHARED_REGION1_SIZE Application defined shared memory region usable across all process-based OS Abstractor and OS Changer processes/applications. Process-based applications are required to be built with OS_INCLUDE_PROCESS feature set to OS_TRUE	1024 (bytes)

OS Abstractor includes this shared user region in the memory area immediately following all the OS Abstractor control block allocations. Applications can access the shared memory via the System_Config->user_shared_region1 global variable. Also, access to shared memory region must be protected (i.e. use mutex locks prior to read/write by the application).

NOTE: The actual virtual address of the shared memory may be different across processes/application; however the OS Abstractor initialized the System_Config pointer correctly during OS_Application_Init function call. Applications should not pass the shared memory region address pointer from one process to another since the virtual address pointing to the shared region may differ from process to process (instead use the above global variable defined above for shared memory region access from each process/applications).

OS Abstractor Clock Tick Configuration

Flag and Purpose	Default Setting
OS_TIME_RESOLUTION This will be the system clock ticks (not hardware clock tick). For example, when you call OS_Task_Sleep(5), you are suspending task for a period (5* OS_TIME_RESOLUTION). See **NOTES** in this table.	10000 μ second (= 10milli sec) Normally this value is derived from the target OS. If you cannot derive the value then refer to the target OS reference manual and set the correct per clock tick value
OS_DEFAULT_TSLICE Default time slice scheduling window width among same priority pre-emptable threads when they are all in ready state.	10 Number of system ticks. If system tick is 10ms, then the threads will be schedule round-robin at the rate of every 100ms. **NOTE**: On Linux operating system, the time slice cannot be modified per thread. OS Abstractor ignores this setting and only uses the system default time slice configured for the Linux kernel. **NOTE**: Time slice option is NOT supported under micro-ITRON. **NOTE**: If the time slice value is non-zero, then under Linux the threads will use Round-Robin scheduling using the system default time slice value of Linux. If the Linux kernel support LINUX_ADV_REALTIME then the time slice value will be set accordingly.

NOTE: Since the system clock tick resolution may vary across different OS under different target. It is recommended that the application use the macro OS_TIME_TICK_PER_SEC to derive the timing requirement instead of using the raw system tick value in order to keep the application portable across multiple OS.

OS Abstractor Device I/O Configuration

Flag and Purpose	Default Setting
NUM_DRIVERS Maximum number of drivers allowed in the OS Abstractor driver table structure	20 **NOTE**: This excludes the native drivers the system, since they do not use the OS Abstractor driver table structure.
NUM_FILES Maximum number of files that can be opened simultaneously using the OS Abstractor file control block structure.	30 **NOTE**: One control block is used when an OS Abstractor driver is opened. This settings do not impact the OS setting for max number of files.
EMAXPATH Maximum length of the directory path name including the file name for OS Abstractor use excluding the null char termination	255 **NOTE**: This setting does not impact the OS setting for the max path/file name.

OS Abstractor Target OS Specific Notes

Nucleus PLUS Target

The following is the compilations define that has to be set when building the Nucleus PLUS library in order for the OS Abstractor to perform correctly:

Compilation Flag	Meaning
NU_DEBUG	Regardless of the target you build, the OS Abstractor library always requires this flag to be set in order to be able to access OS internal data structures. Without this flag, you will see a lot of compiler errors.

Precise/MQX Target

The following are the compilation defines that has to be set if you are using Precise/MQX as your target OS:

Compilation Flag	Meaning
MQX_TASK_DESTRUCTION	Set this macro to zero to allow OS Abstractor to manage destruction of MQX kernel objects such as semaphores.
BSP_DEFAULT_MAX_MSGPOOLS	Set this macro to match the maximum number of message queues and pipes required by your application at a given time. For example, if your application would need a max of 10 message queues and 10 pipes, then this macro needs to be set to 20.

The MQX_TASK_DESTRUCTION macro is located in source\include\mqx_cnfg.h in your MQX installation. Set it to zero as shown below (or pass it to compiler via pre-processor setting in your project make files):

```
#ifndef MQX_TASK_DESTRUCTION
#define MQX_TASK_DESTRUCTION 0
#endif
```

The BSP_DEFAULT_MAX_MSGPOOLS macro is located in source\bsp\bspname\bspname.h in your MQX installation, where bspname is the name of your BSP. Set the required value as follows:

```
#define BSP_DEFAULT_MAX_MSGPOOLS (20L)
```

Linux Target

User Vs ROOT Login

OS Abstractor internally checks the user ID to see if the user is ROOT or not. If the user is ROOT, then it will automatically utilize the Linux real time policies and priorities. It is always recommended that OS Abstractor application be run under ROOT user login. In this mode:

- OS Abstractor task priorities, time slice, pre-emption modes and critical region protection features will work properly.
- OS Abstractor applications will have better performance and be more deterministic behavior since the Linux scheduler is prevented to alter the tasks priorities behind the scenes.
- Also, when you load other Linux applications (that uses the default SCHED_OTHER policies), they will not impact the performance of the OS Abstractor applications that are running under real-time priorities and policies.

Under non-ROOT user mode, the task scheduling is fully under the mercy of the Linux scheduler. In this mode, the OS Abstractor does not utilize any real-time priorities and/or policies. It will use the SCHED_OTHER policy and will ignore the application request to set and/or change scheduler parameters like priority and such. OS Abstractor applications will run under the non-ROOT mode, with restrictions to the following OS Abstractor APIs:

- OS_Create_Task: The function parameters *priority, timeslice* and OS_NO_PREEMPT flag options are ignored
- OS_Set_Task_Priority: This function will have no effect and will be ignored
- OS_Set_Task_Preemption: Changing the task pre-emption to OS_NO_PREEMPT has no effect and will be ignored
- OS_Protect: Will offer NO critical region data protection and will be ignored. If you need protection, then utilize OS Abstractor mutex features
- OS_Create_Driver: The OS Abstractor driver task will NOT be run at a higher priority level that the OS Abstractor application tasks.

Though OS Abstractor applications may run under non-ROOT user mode, it is highly recommended that the real target applications be run under ROOT user mode.

Time Resolution

The value of the system clock ticks is defined by OS_TIME_RESOLUTION, which is retrieved from the Linux system. Under Red Hat®/GNU® Linux, this is actually 100 (this means every tick equals to 10ms). However, the OS_TIME_TICK_PER_SEC could be different under other real-time or proprietary Linux distributions.

Also, make sure you modify OS_DEFAULT_TSLICE value to match with your application needs if necessary. By default, this value is set for the time slice to be 100ms. If the Linux Advanced Real Time Feature is present (i.e the Linux kernel macro LINUX_ADV_REALTIME == 1), then OS Abstractor automatically takes advantage of this feature if present and uses the sched_rr_set_interval() function and sets the application required round-robin thread time-slice for the OB Abstractor thread. If this feature is not present, the the timeslice value for round-robin scheduling will be whatever the kernel is configured to.

Memory Heap

OS Abstractor uses the system heap directly to provide the dynamic variable memory allocation. The Memory management for the variable memory is best left for the Linux kernel to be handled, so OS Abstractor only does boundary checks to ensure that the application does not allocate beyond the pool size. The maximum memory the application can get from these pools will depend on the memory availability of the system heap.

Priority Mapping Scheme

The OS Abstractor uses priorities 0~255 plus one more for exclusivity which results in a total of 257 priorities. If the Linux that you use provides less than 257 priority values, then OS Abstractor maps its priority in a simple window-mapping scheme where a window of OS Abstractor priorities gets mapped to each individual Linux priority. If the Linux that you use provides more than 257 priority values, then the OS Abstractor maps it priority one-on-one somewhere in the middle of the range of Linux priorities. Please modify the priority scheme as necessary if required by your application. If you want to minimize the interruption of the external native Linux applications then you would want the OS Abstractor priorities to map to the higher end of the Linux priority window.

OS Abstractor priority value of 257 is reserved internally by OS Abstractor to provide the necessary exclusivity among the OS Abstractor tasks when they request no preemption or task protection. The exclusivity and protections are not guaranteed if the external native Linux application runs at a higher priority.

It is recommended that the Linux kernel be configured to have a priority of 512, so that the OS Abstractor priorities will use the window range in the middle and as such would not interfere with some of core Linux components. If your Linux kernel is configured to have less than 257 priorities, the OS Abstractor will automatically configuring a windowing scheme, where multiple number of OS Abstractor priorities will map to a single Linux priority. Because of this, the reported priority value could be slightly different than what was used during the task creating process. If your application uses the pre-processor called OS_DEBUG_INFO, then all the priority values and calculations will be printed to the standard output device.

Memory and System Resource Cleanup

OS Abstractor uses shared memory to support multiple OS Abstractor and OS Changer application processes that are built with OS_INCLUDE_PROCESS mode set to OS_TRUE.

Single-process Application Exit

This will apply to application that does not use the OS_PROCESS feature. Each application needs to call OS_Application_Free to unregister and free OS Abstractor resources used by the application. Under circumstances where the application terminates abnormally, the applications need to install appropriate signal handler and call OS_Application_Free within them.

Multi-process Application Exit

This will be the case where the applications are built with OS_PROCESS feature set to OS_TRUE. When the first multi-process application starts, shared memory is created to accommodate all the shared system resources for all the multi-process application. When subsequent multi-process application gets loaded, they will register and OS Abstractor will create all the local resources (memory heap) necessary for the application. Application's can also spawn new applications using OS_Create_Process and will result the same as if a new application get's loaded. Each application needs to call OS_Application_Free to unregister and free OS Abstractor resources used by the application. Under circumstances where the application terminates abnormally, the applications need to install appropriate signal handler and call OS_Application_Free within them. When the last application calls OS_Application_Free, then OS Abstractor frees the resources used by the application and also deletes the shared memory region.

Manual Clean-up

If application terminates abnormally and for any reason and it was not possible to call OS_Application_Free, then it is recommended that you execute the provide **cleanup.pl** script manually before starting to load applications. Users can query the interprocess shared resources status by typing ipcs in the command line.

Multi-process Zombie Cleanup

There are circumstances where a multi-process application terminates abnormally and was not able to call OS_Application_Free. In this case, the shared memory region would be left with a zombie control block (i.e there is no native process associated with the OS Abstractor process control block). Whenever, a new multi-process application get's loaded, OS Abstractor automatically checks the shared memory region for zombie control blocks. If it finds any, it will take the following action:
Free and initialize all the control blocks that belong to the zombie process (this could even be the zombie process of the same application currently being loaded but was previously terminated abnormally).

Task's Stack Size

The stack size has to be greater than PTHREAD_STACK_MIN defined by Linux, otherwise, any OS Abstractor or OS Changer task creation will return success, but the actual task (pthread) will never get launched by the target OS. It is also safe to use a value greater than or equal to OS_MIN_STACK_SIZE defined in def.h. OS Abstractor ensures that OS_STACK_SIZE_MIN is always greater that the minimum stack size requirement set by the underlying target OS.

SMP Flags

The following is the compilation defines that can be set when building the OS Abstractor library for Linux SMP kernel target OS:

Compilation Flag	Meaning
OS_BUILD_FOR_SMP Support for Symmetric Multi-Processors **(SMP)**	Specify the SMP or non-SMP kernel. The value can be: **OS_TRUE** SMP enabled **OS_FALSE** SMP disabled

Windows Target

OS_Relinquish_Task API uses Window's sleep() to relinquish task control. However, the sleep() function does not relinquish control when stepping through code in the debugger, but behaves correctly when executed. This is a problem inherent in the OS itself.

QNX Target

User Vs ROOT Login

OS Abstractor internally checks the user ID to see if the user is ROOT or not. If the user is ROOT, then it will automatically utilize the Linux real time policies and priorities. It is always recommended that OS Abstractor application be run under ROOT user login. In this mode:

- OS Abstractor task priorities, time slice, pre-emption modes and critical region protection features will work properly.
- OS Abstractor applications will have better performance and be more deterministic behavior since the Linux scheduler is prevented to alter the tasks priorities behind the scenes.
- Also, when you load other Linux applications (that uses the default SCHED_OTHER policies), they will not impact the performance of the OS Abstractor applications that are running under real-time priorities and policies.

Under non-ROOT user mode, the task scheduling is fully under the mercy of the Linux scheduler. In this mode, the OS Abstractor does not utilize any real-time priorities and/or policies. It will use the SCHED_OTHER policy and will ignore the application request to set and/or change scheduler parameters like priority and such. OS Abstractor applications will run under the non-ROOT mode, with restrictions to the following OS Abstractor APIs:

- OS_Create_Task: The function parameters priority, timeslice and OS_NO_PREEMPT flag options are ignored
- OS_Set_Task_Priority: This function will have no effect and will be ignored
- OS_Set_Task_Preemption: Changing the task pre-emption to OS_NO_PREEMPT has no effect and will be ignored
- OS_Protect: Will offer NO critical region data protection and will be ignored. If you need protection, then utilize OS Abstractor mutex features
- OS_Create_Driver: The OS Abstractor driver task will NOT be run at a higher priority level that the OS Abstractor application tasks.

Though OS Abstractor applications may run under non-ROOT user mode, it is highly recommended that the real target applications be run under ROOT user mode.

Time Resolution
The value of the system clock ticks is defined by `OS_TIME_RESOLUTION`, which is retrieved from the Linux system. Under Red Hat®/GNU® Linux, this is actually 100 (this means every tick equals to 10ms). However, the `OS_TIME_TICK_PER_SEC` could be different under other real-time or proprietary Linux distributions.

Also, make sure you modify `OS_DEFAULT_TSLICE` value to match with your application needs if necessary. By default, this value is set for the time slice to be 100ms.

Memory Heap
OS Abstractor uses the system heap directly to provide the dynamic variable memory allocation. The Memory management for the variable memory is best left for the Linux kernel to be handled, so OS Abstractor only does boundary checks to ensure that the application does not allocate beyond the pool size. The maximum memory the application can get from these pools will depend on the memory availability of the system heap.

Priority Mapping Scheme
QNX native priority value of 255 will be reserved for OS Abstractor Exclusivity. The rest of the 255 QNX priorities will be mapped as follows:
0 to 253 OS Abstractor priorities -> 254 to 1 QNX priorities
254 and 255 OS Abstractor priorities -> 0 QNX priority
The OS Abstractor uses priorities 0~255 plus one more for exclusivity which results in a total of 257.

Memory and System Resource Cleanup
Please refer to the same section under target specific notes for Linux operating system.

Task's Stack Size
The stack size has to be greater than PTHREAD_STACK_MIN defined by Linux, otherwise, any OS Abstractor or OS Changer task creation will return success, but the actual task (pthread) will never get launched by the target OS. It is also safe to use a value greater than or equal to OS_STACK_SIZE_MIN defined in def.h. OS Abstractor ensures that OS_STACK_SIZE_MIN is always greater that the minimum stack size requirement set by the underlying target OS.

VxWorks Target

Version Flags

The following is the compilation defines that has to be set when building the OS Abstractor library for VxWorks target OS:

Compilation Flag	Meaning
OS_VERSION	Specify the VxWorks version. The value can be: **OS_VXWORKS_5X** – VxWorks 5.x or older **OS_VXWORKS_6X** – Versions 6.x or higher
OS_KERNEL_MODE	Set this value to **OS_TRUE** if the OS Abstractor is required to run as a kernel module. Under OS_VXWORKS_5X, the OS_KERNEL_MODE flag is ignored. The library is built to run as a kernel module. Under OS_VXWORKS_6X, you have the option to create the library for either as a kernel module or a user application as below: OS_KERNEL_MODE = OS_TRUE for kernel module OS_KERNEL_MODE = OS_FALSE for user application.

Unsupported OS Abstractor APIs

The following OS Abstractor APIs are not supported as shown below:

Compilation Flag	Unsupported APIs
OS_VERSION = OS_VXWORKS_5X	OS_Delete_Partion_Pool OS_Delete_Memory_Pool OS_Get_Semaphore_Count
OS_VERSION = OS_VXWORKS_6X and OS_KERNEL_MODE = OS_TRUE	OS_Set_Clock_Ticks
OS_VERSION = OS_VXWORKS_6X and OS_KERNEL_MODE = OS_FALSE	OS_Get_Semaphore_Count

Application Initialization

Once you have configured the OS Abstractor (refer to chapter OS Abstractor Configuration), now you are ready to create a sample demo application.

Application needs to initialize the OS Abstractor library by calling the OS_Application_Init() function prior to using any of the OS Abstractor function calls. Please refer to subsequent pages for more info on the usage and definition of OS_Application_Init function.

The next step would be is to create the first task and then within the new task context, application needs to call other initializations functions if required. For example, to use the POSIX OS Abstractor component, application need to call OS_Posix_Init() function within an OS Abstractor task context prior to using the POSIX APIs. The OS_Posix_Init() function initializes the POSIX library and makes a function call to px_main() function pointer that is passed along within OS_Posix_Init() call. Please note that the px_main() function is similar to the main() function that is typically found in posix code. Please refer to the example initialization code shown at the end of this section.

If the application also uses OS Changer components, then the appropriate OS Changer library initialization calls need to be made in addition to POSIX initialization. Please refer to the appropriate OS Changer reference manual for more details.

Please refer to the init.c module provided with the sample demo application for the specific OS, tools and target for OS Abstractor initialization and on starting the application.

If you need to re-configure your board differently or would like to use a custom board, or would like to re-configure the OS directly, then refer to the appropriate documentations provided by the OS vendor.

Example: BASE OS Abstractor for Windows Initialization

```
int main(int   argc,
         LPSTR argv[])
{
    OS_Main();

    return (OS_SUCCESS);
} /* main */

VOID OS_Main()
{
    OS_TASK          task;
    OS_APP_INIT_INFO info;

    /* set the OS_APP_INIT_INFO structure with the actual number of resources 
we will use.  If we set all the Variables to -1, the default values would be 
used. On ThreadX and Nucleus, we must pass an OS_APP_INIT_INFO structure with 
at least first_available set to the first unused memory. Other OS's can pass 
NULL to OS_Application_Init and all defaults would be used.  */
```

```
#if ((OS_TARGET == OS_THREADX) || (OS_TARGET == OS_NUCLEUS))
    info.first_available          = first_unused_memory; /* required for
ThreadX */
#endif
    info.debug_info               = OS_DEBUG_VERBOSE;
    info.task_pool_enabled        = OS_TRUE;
    info.task_pool_timeslice      = -1;
    info.task_pool_timeout        = -1;
    info.root_process_preempt     = -1;
    info.root_process_priority    = -1;
    info.root_process_stack_size  = -1;
    info.root_process_heap_size   = -1;
    info.default_timeslice        = -1;

    info.max_tasks                = 6;
    info.max_timers               = 3;
    info.max_mutexes              = 0;
    info.max_pipes                = 1;
#if (INCLUDE_OS_PROCESS == OS_TRUE)
    info.max_processes            = 2;
#else
    info.max_processes            = 0;
#endif
    info.max_queues               = 1;
    info.user_shared_region1_size = 0;
    info.max_partition_mem_pools  = 0;
    info.max_dynamic_mem_pools    = 1;
    info.max_event_groups         = 2;
    info.max_semaphores           = 1;

    OS_Application_Init("DEMO", HEAP_SIZE, &info);

    OS_Create_Task(&task,
                   "APPSTART",
                   OS_Application_Start,
                   0,
                   STACK_SIZE,
                   1,
                   0,
                   OS_NO_PREEMPT | OS_START);

    OS_Application_Wait_For_End();
} /* OS_Main */

VOID OS_Application_Start(UNSIGNED argv)
{
/*User application code*/
}
```

Example: POSIX OS Abstractor for Windows Target Initialization

```
int main(int   argc,
         LPSTR argv[])
{
    OS_Main();

    return (OS_SUCCESS);
} /* main */

VOID OS_Main()
{
    OS_TASK          task;
    OS_APP_INIT_INFO info;

    /* set the OS_APP_INIT_INFO structure with the actual
     * number of resources we will use.  If we set all the
     * variables to -1, the default values would be used.
     * On ThreadX and Nucleus, we must pass an OS_APP_INIT_INFO
     * structure with at least first_available set to the first
     * unused memory.  Other OS's can pass NULL to OS_Application_Init
     * and all defaults would be used */
#if ((OS_TARGET == OS_THREADX) || (OS_TARGET == OS_NUCLEUS))
    info.first_available          = first_unused_memory; /* required for
ThreadX */
#endif
    info.debug_info               = OS_DEBUG_VERBOSE;
    info.task_pool_enabled        = OS_TRUE;
    info.task_pool_timeslice      = -1;
    info.task_pool_timeout        = -1;
    info.root_process_preempt     = -1;
    info.root_process_priority    = -1;
    info.root_process_stack_size  = -1;
    info.root_process_heap_size   = -1;
    info.default_timeslice        = -1;

    info.max_tasks                = 6;
    info.max_timers               = 3;
    info.max_mutexes              = 0;
    info.max_pipes                = 1;
#if (INCLUDE_OS_PROCESS == OS_TRUE)
    info.max_processes            = 2;
#else
    info.max_processes            = 0;
#endif
    info.max_queues               = 1;
    info.user_shared_region1_size = 0;
    info.max_partition_mem_pools  = 0;
    info.max_dynamic_mem_pools    = 1;
    info.max_event_groups         = 2;
    info.max_semaphores           = 1;

    OS_Application_Init("DEMO", HEAP_SIZE, &info);

    OS_Create_Task(&task,
                   "APPSTART",
```

```
                   OS_Application_Start,
                   0,
                   STACK_SIZE,
                   1,
                   0,
                   OS_NO_PREEMPT | OS_START);

    OS_Application_Wait_For_End();
} /* OS_Main */

VOID OS_Application_Start(UNSIGNED argv)
{
    pthread_t task;

/* posix compatibility initialization.  create the main process
    *  and pass in the osc posix main entry function px_main.*/
    OS_Posix_Init();

    pthread_create(&task, NULL, (void*)px_main, NULL);
    pthread_join(task, NULL);

    OS_Application_Free(OS_APP_FREE_EXIT);
} /* OS_Application_Start */

int px_main(int   argc,
            char* argv[])
{
            /*User application code*/
}
`
```

Runtime Memory Allocations

OS Abstractor

Some of the allocations for this product will be dependant on the native os. Some of these may be generic among all products. The thread stacks should come from the process heap. This is only being done on the OS Abstractor for QNX product at the moment.

- Message in int_os_send_to_pipe.
- Device name in os_creat
- Partitions in os_create_partition_pool
- Device name in os_device_add
- File structures in os_init_io
- Driver structures in os_init_io
- Device header for null device in os_init_io
- Device name for the null device in os_init_io
- Device name in os_open
- Environment structure in os_put_environment
- Environment variable in os_put_environment
- Memory for profiler messages if profiler feature is turned ON
- Thread stack (only under QNX)

POSIX OS Abstractor

All of the following allocations use OS_Allocate_Memory using the System_Memory pool. Thus, all these allocations come from the calling processes memory pool:

- Pthread key lists and values
- Stack item in pthread_cleanup_push
- Sem_t structures created by sem_open.
- Timer_t structures created by timer_create.
- mqueue_t structures created by mq_open.
- Message in mq_receive. This is deallocated before leaving the function call.
- Message in mq_send. This is deallocated before leaving the function call.
- Message in mq_timedreceive. This is deallocated before leaving the function call.
- Message in mq_timedsend. This is deallocated before leaving the function call.

All of the following are specific to the TKernel OS and use the SMalloc api call. These will not be accounted for in the process memory pool:

- Parameter list for execve
- INT_PX_FIFO_DATA structure in fopen

All of the following are specific to the TKernel OS and use os_malloc_external API call. These will not be accounted for in the process memory pool.

- Buffer for getline
- Globlink structure in int_os_glob_in_dir
- Globlink name in int_os_glob_in_dir
- Directory in int_o_prepend_dir

micro-ITRON OS Abstractor

All of the following allocations use OS_Allocate_Memory using the System_Memory pool. Thus, all these allocations come from the calling processes memory pool.

- Message in snd_dtq. This is deallocated before leaving the function call.
- Message in psnd_dtq. This is deallocated before leaving the function call.
- Message in tsnd_dtq. This is deallocated before leaving the function call.
- Message in fsnd_dtq. This is deallocated before leaving the function call.
- Message in rcv_dtq. This is deallocated before leaving the function call.
- Message in prcv_dtq. This is deallocated before leaving the function call.
- Message in trcv_dtq. This is deallocated before leaving the function call.
- Message in snd_mbf. This is deallocated before leaving the function call.
- Message in psnd_mbf. This is deallocated before leaving the function call.
- Message in tsnd_mbf. This is deallocated before leaving the function call.
- Message in rcv_mbf. This is deallocated before leaving the function call.
- Message in prcv_mbf. This is deallocated before leaving the function call.
- Message in trcv_mbf. This is deallocated before leaving the function call.

OS Changer VxWorks

All of the following allocations use OS_Allocate_Memory using the System_Memory pool. Thus, all these allocations come from the calling processes memory pool.

- Wdcreate allocates memory for an OS_TIMER control block .
- Message in msgqsend. This is deallocated before leaving the function call.
- Message in msgqreceive. This is deallocated before leaving the function call

OS Changer pSOS

All of the following allocations use OS_Allocate_Memory using the System_Memory pool. Thus, all these allocations come from the calling processes memory pool.

- Rn_getseg will allocate from the System_Memory if a pool is not specified.
- Message in q_vsend. This is deallocated before leaving the function call.
- Message in q_vrecieve. This is deallocated before leaving the function call.
- Message in q_vurgent. This is deallocated before leaving the function call.

All of the following allocations use malloc. Depending on the setting of OS_MAP_ANSI_MEM these may or may not be accounted for in the process memory pool.

- IOPARMS structure in de_close
- IOPARMS structure in de_cntrl
- IOPARMS structure in de_init
- IOPARMS structure in de_open
- IOPARMS structure in de_read

OS Changer Nucleus

All of the following allocations use OS_Allocate_Memory using the System_Memory pool. Thus, all these allocations come from the calling processes memory pool.

- Message in nu_receive_from_pipe. This is deallocated before leaving the function call
- Message in nu_receive_from_queue. This is deallocated before leaving the function call
- Message in nu_send_to_front_of_pipe. This is deallocated before leaving the function call
- Message in nu_send_to_front_of_queue. This is deallocated before leaving the function call
- Message in nu_send_to_pipe. This is deallocated before leaving the function call
- Message in nu_send_to_queue. This is deallocated before leaving the function call

OS Abstractor Process Feature

An OS Abstractor process or an application ("process") is an individual module that contains one or more tasks and other resources. A process can be looked as a container that provides encapsulation from other process. The OS Abstractor processes only have a peer-to-peer relationship (and not a parent/child relationship).

An OS Abstractor process comes into existence in two different ways. Application registers a new OS Abstractor process when it calls OS_Application_Init function. Application also launches a new process when it calls the OS_Create_Process function. In the later case, the newly launched process does not automatically inherit the open handles and such; however they can access the resources belonging to the other process if they are created with "system" scope.

Under process-based operating system like Linux, this will be an actual process with virtual memory addressing. As such the level of protection across individual application will be dependent on the underlying target OS itself.

Under non-process-based operating system like Nucleus PLUS, a process will be a specialized task (similar to a main() thread) owning other tasks and resources in a single memory model based addressing. The resources are protected via OS Abstractor software. This protection offered by OS Abstractor is software protection only and not to be confused with MMU hardware protection in this case.

OS Abstractor automatically tracks all the resources (tasks, threads, semaphores, etc.) and associates them with the process that created them. All the memory requirements come from its own process dedicated memory pool called "process system pool". Upon deletion of the process, all these resources will automatically become freed.

Depending on whether the resource needs to be shared across other processes, they can be created with a scope of either OS_SCOPE_SYSTEM or OS_SCOPE_PROCESS. The resources with system scope can be accessible or usable by the other processes. However, the process that creates them can only do deletion of these resources with system scope.

A new process will be created as a "new entity" and not a copy of the original. As such, none of the resources that are open becomes immediately available to the newly created process. The new created process can use the resources which were created with system scope by first retrieving their ID through their name. For this purpose, the application should create the resources with unique names. OS Abstractor will all resource creation with duplicate names, however the function that returns the resource ID from name will provide the ID of only the first entry.

Direct access to any OS Abstractor resource control blocks are prohibited by the application. In other words, the resource Ids does not directly point to the addresses of the control blocks.

Simple (single-process) Versus Complex (multiple-process) Applications

An OS Abstractor application can be simple (i.e. single-process application) or complex (multi-process application). Complex and large applications will greatly benefit in using the OS_INCLUDE_PROCESS feature support offered by OS Abstractor.

OS_INCLUDE_PROCESS = OS_FALSE (Simple OR Single-process Application)	**OS_INCLUDE_PROCESS = OS_TRUE (Complex OR multi-process Application)**
OS Abstractor applications are independent from each other and are complied and linked into a separate executables. There is no need for the OS Abstractor and/or OS Changer APIs to work across processes.	OS Abstractor applications can share the OS Abstractor resources (as long as they are created with system scope) between them even though they may be complied and linked separately. The OS Abstractor and/or OS Changer APIs works across processes.
Many independent or even clones of OS Abstractor single-process applications can be hosted on the OS platform.	In addition to independent single-process applications, the current release of OS Abstractor allows to host one multi-process application.
OS Abstractor applications do NOT spawn new processes via the OS_Create_Process function. In fact, any APIs with the name “process” in them are not available for a single-process application.	OS Abstractor applications can spawn new processes via the OS_Create_Process function.
Each application uses its own user configuration parameters set in the osabstractor usr.h file.	Each application has to have the same set of shared resources defined in the osabstractor_usr.h (e.g. max number of tasks/threads across all multi-process applications). When the first multi-process application gets loaded, the OS Abstractor uses the values defined in osabstractor_usr.h or the over-ride values passed along its call to OS_Application_Init function to create all the shared system resources. When subsequent multi-process application gets loaded, OS Abstractor ignores the values defined in the osabstractor_usr.h or the values passed in the OS_Application_Init call. Please note that the shared resources are only gets created during the load time of the first application and they gets deleted when the last multi-process application exits.
OS Abstractor creates all the resource control blocks within the process memory individually for each application.	OS Abstractor creates all the resource control blocks in shared memory during the first OS_Application_Init function call. In other words, when the first application gets loaded, it will initialize the OS Abstractor library. After this,

	every subsequent OS_Application_Init call will register and adds the application as a new OS Abstractor process and also creates the memory pool for the requested heap memory. An application can delete or free or re-start itself with a call to OS_Application_Free. An application can delete or re-start another application via OS_Delete_Process. Also, it is up to the application to provide the necessary synchronization during loading individual applications so that the complex application will start to run only in the preferred sequence.

Memory Usage

The memory usage depends on whether your application is built in single process mode (i.e OS_INCLUDE_PROCESS set to false) or multi-processes mode (i.e OS_INCLUDE_PROCESS set to true).

The memory usage also depends on whether the target OS supports single memory model or a virtual memory model. Operating systems such as LynxOS, Linux, Windows XP, etc. are based on virtual memory model where each application are protected from each other and run under their own virtual memory address space. Operating systems like Nucleus PLUS, ThreadX, MQX, etc. are based on single memory model where each application shares the same address space and there is no protection from each other.

In general, OS Abstractor applications require memory to store the system configuration and also to meet the application heap memory needs.

Memory Usage under Virtual memory model based OS

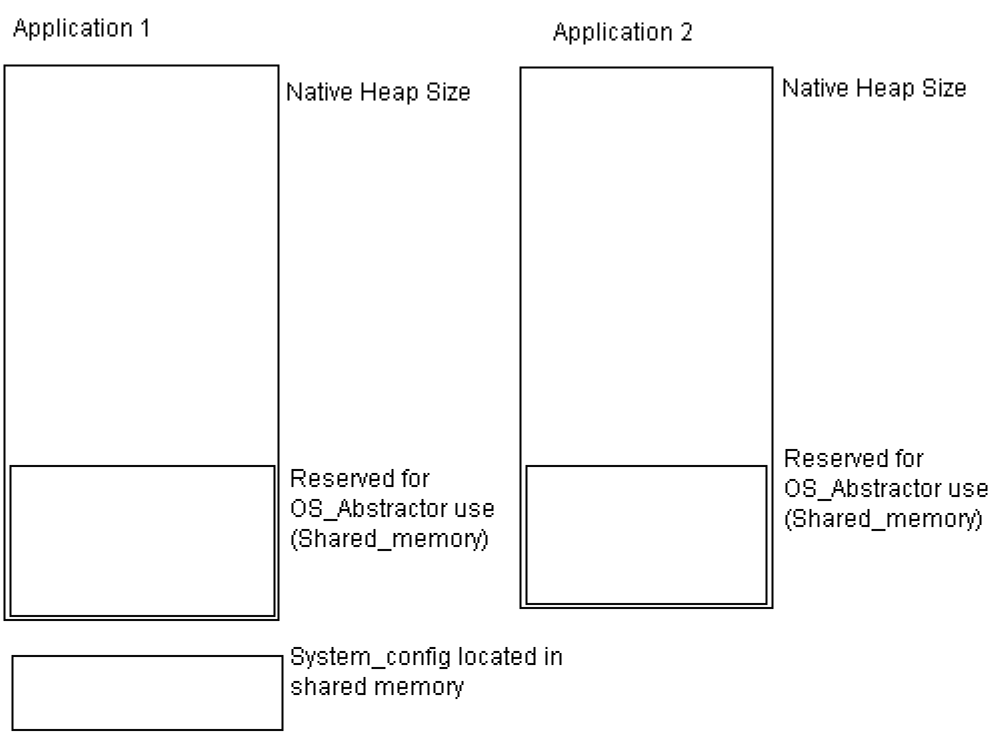
Multi-process Application

System_Config: The system config structure will be allocated from shared memory. The size will be returned to the user for informational use via the OS_SYSTEM_OVERHEAD macro.

OS_Application_Init: the memory value passed into this API by memory_pool_size will be the heap size for this particular process. In this type of system, it is possible to have multiple applications, all of which will call this API. This API will create an OS Abstractor dynamic memory pool the size of the heap. The global variable System_Memory will be set to the id of this pool.

OS_Create_Process: The memory value passed into this API by process_pool_size will be the heap size for this particular process. This API will create an OS Abstractor dynamic memory pool the size of the heap. The global variable System_Memory will be set to the id of this pool.

System_Memory: This will be set to the pool id of the process memory pool.

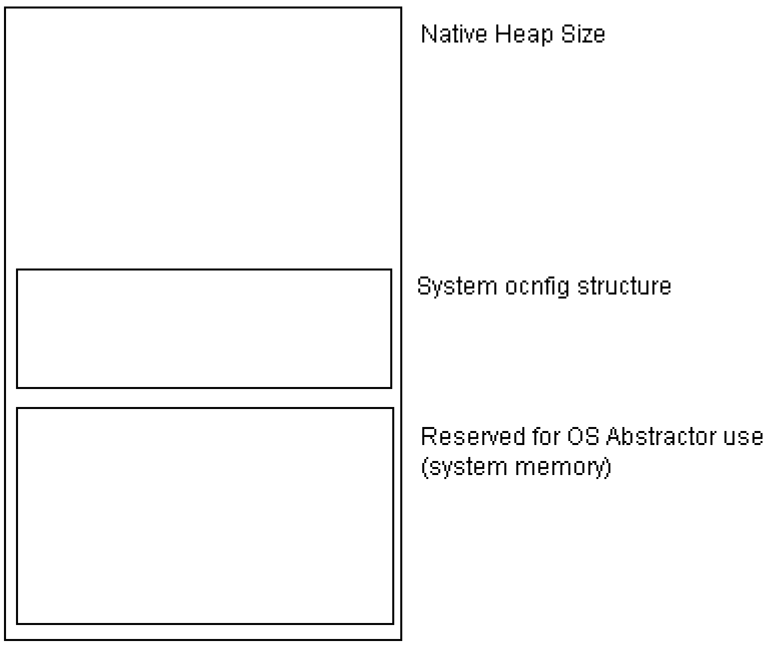


Single-process Application

System_Config: The system config structure will be allocated from the process heap. The size will be returned to the user for informational use only by calling OS_System_Overhead();

OS_Application_Init: the memory value passed into this API by memory_pool_size will be the amount of memory available to the system. This API will create an OS Abstractor dynamic memory pool this size. The memory for System_Config does not come from this pool. So the total memory requirements will be OS_SYSTEM_OVERHEAD + memory_pool_size.

System_Memory: This will be set to 0. Since there are no processes, the first pool will always be the system memory pool.



Native process heap size: We are not adjusting the native process heap size, so it could be possible that there is an inconsistency between the amount of memory reserved by OS Abstractor and the amount of memory reserved for the actual heap of the native process.
There is no upper bounds limit to the system wide memory use while in process mode. We will create processes without regard to the actual size of the physical memory.

Memory Usage under Single memory model based OS

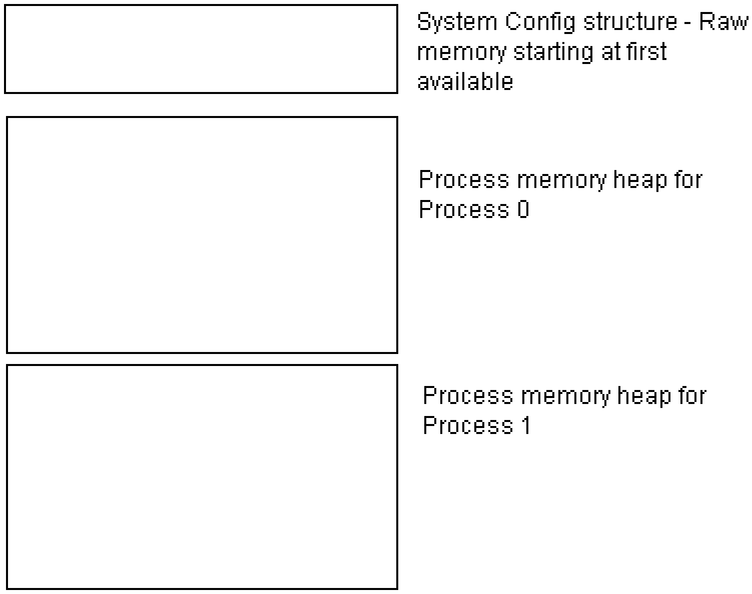
Multi-process Application

System_Config: The first available memory will be set in the OS_APP_INFO structure and will be adjusted the size of the system_config structure.

OS_Application_Init: The memory value passed into this API by memory_pool_size will be the heap size for this particular process. This API can only be called once since it is not possible to have multiple applications natively. This API will create an OS Abstractor dynamic memory pool the size of the heap.

OS_Create_Process: The memory value passed into this API by process_pool_size will be the heap size for this particular process. This API will create an OS Abstractor dynamic memory pool the size of the heap.

System_Memory: This will always be set to 0. When we get a pool id of 0 in any of the allocation APIs we will know to allocate from the current process memory pool. This means that the dynamic memory pool control block at index 0 is not to be used.

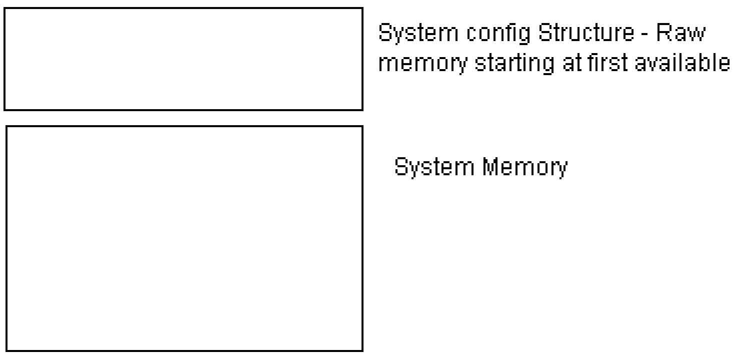

System Config structure - Raw memory starting at first available
Process memory heap for Process 0
Process memory heap for Process 1

Single-process Application

System_Config: The first available memory will be set in the OS_APP_INFO structure and will be adjusted the size of the system_config structure.

OS_Application_Init: the memory value passed into this API by memory_pool_size will be the amount of memory available to the system. This API will create an OS Abstractor dynamic memory pool this size. The memory for System_Config does not come from this pool. So the total memory requirements will be OS_SYSTEM_OVERHEAD + memory_pool_size.

System_Memory: This will always be set to 0. Since we are not in process mode, there should not be any other OS Abstractor memory pools created.



There is no upper bounds limit to the system wide memory use while in process mode. Also, it cannot be guaranteed that there will be enough memory to create all the processes of the application since there is no total memory being reserved.

POSIX OS Abstractor Configuration

When the INCLUDE_OS_POSIX option is set to OS_TRUE, the OS Abstractor also includes POSIX APIs in addition to the BASE OS Abstractor APIs available to the application.

Inclusion of osabstractor.h will ensure that all the POSIX API calls in the application are automatically re-mapped to OS Abstractor libraries. Applications can also selectively exclude individual modules of POSIX OS Abstractor APIs, if required.

Current release does not support including or excluding Individual modules within POSIX OS Abstractor.

Porting POSIX Legacy Code with OS Abstractor

The first step in porting any POSIX legacy code base using POSIX OS Abstractor component would be to rename the application main() function to px_main(). Then this function can be started via the OS_Posix_Init() call. Please refer to the list of POSIX APIs that are supported by the POSIX OS Abstractor component. If the application requires a specific POSIX function which is not support by OS Abstractor, then there are two options:

1. Re-write the application with BASE OS Abstractor function calls for all the unsupported POSIX APIs needed by your application.
2. Check if the target OS offers support to this function and if so, you can directly use those functions (however, in this case, the OS Abstraction will not be there). In this case, make sure you include all the relevant POSIX header files provided by the target OS before including osabstractor.h. This way, the POSIX calls used by the application will get mapped to the POSIX equivalent calls from the OS Abstractor library.

If applications need to use the POSIX APIs offered by the target OS (or) tools in addition to what is offered with POSIX OS Abstractor, then you need to do it by including additional POSIX header files provided by the target OS. However, these headers files are required to be included prior to osabstractor.h within the application source code.

POSIX OS Abstractor – API Deviations

POSIX API available on some selected OS and also support for new APIs are constantly added in newer releases.

- Contact MapuSoft to find out the latest POSIX API support for your target OS platform.
- Refer to the POSIX standards reference documents for the specifications for all the above POSIX APIs.

NOTE: Extensive POSIX level and other standard's compliance is provided on VxWorks 6.x OS platform. Additional POSIX support is available on T-Kernel platform

Chapter 6. OS Changer Porting Examples

This chapter contains the following topics:

Sample Porting of pSOS Application to Linux with OS Changer

In most applications, using OS Changer is straightforward. The effort required in porting is mostly at the underlying driver layer. Since we do not have specific information about your application, it will be hard to tell how much work is required. However, we want you to be fully aware of the surrounding issues upfront so that necessary steps could be taken for a successful and timely porting.

This section provides porting guidelines in two different flow charts. Contact MapuSoft Technologies for further information on your application specific issues.

Chart A covers issues relating with OS Changer, device drivers, interrupt service routines, etc.

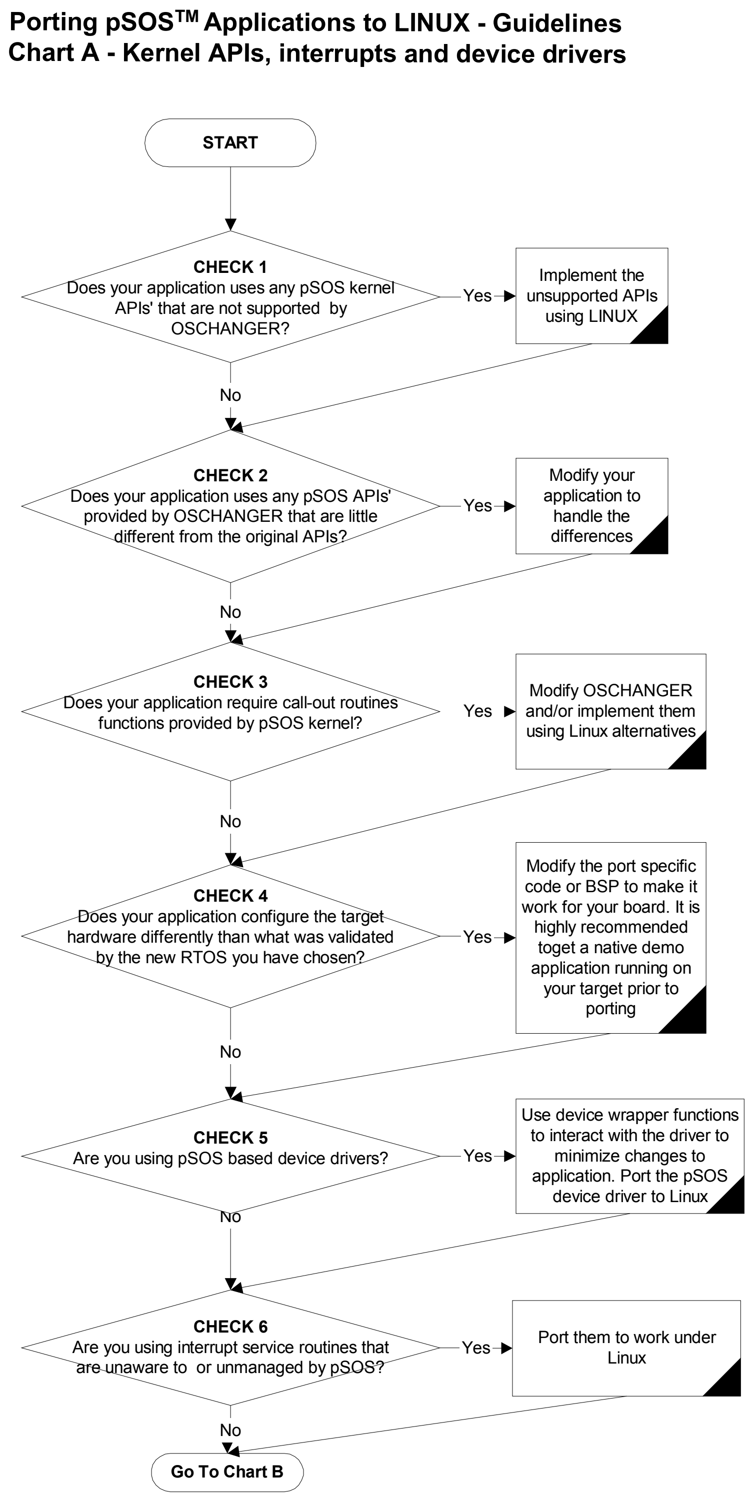
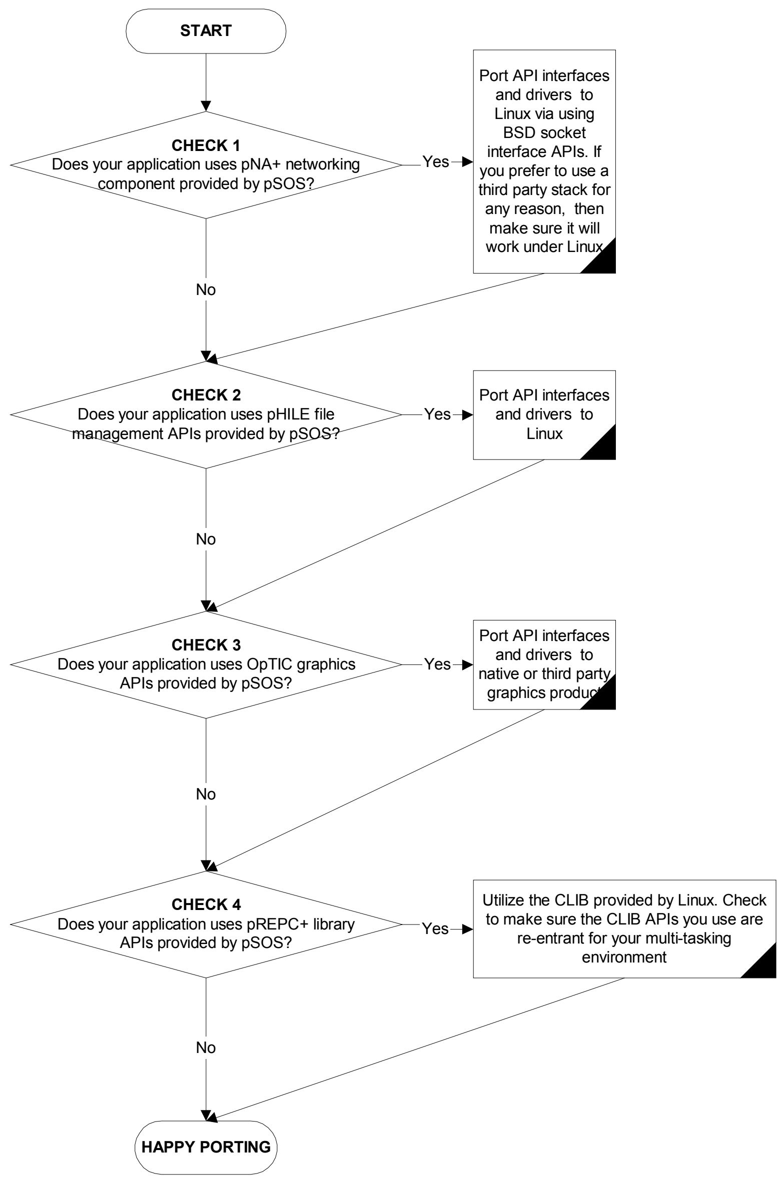


Chart B covers issues relating to other add-on components (like pHILE) that application may use.

Porting pSOS™ Applications to Linux - Guidelines
Chart B - Other Components



OS Changer Overview

The OS Changer contains the following modules, which can be found at the installation directory:

Module	Description
OSCHANGER.H	This header files include RTOS specific components and also components that is required for the application
PS_OSCHANGER\PS_OSCHANGER.H	This header file provides the translation layer between the pSOS™ defines, APIs and parameters to OS Changer's virtual abstraction definition, which then re-maps to Linux equivalents
PS_OSCHANGER\PS_I.C	Provides the pSOS OS Changer initialize function
PS_OSCHANGER\PS_AS.C	Provides pSOS™ signal handling APIs
PS_OSCHANGER\PS_EV.C	Provides pSOS™ event handling APIs
PS_OSCHANGER\PS_T.C	Provides pSOS™ task handling APIs
PS_OSCHANGER\PS_PT.C	Provides pSOS™ partition memory management APIs
PS_OSCHANGER\PS_RN.C	Provides pSOS™ memory region management APIs
PS_OSCHANGER\PS_Q.C	Provides pSOS™ fixed and variable queue APIs
PS_OSCHANGER\PS_SM.C	Provides pSOS™ semaphore handling APIs
PS_OSCHANGER\PS_TM.C	Provides pSOS™ timer, time and date APIs
PS_OSCHANGER\PS_DE.C	Provides pSOS™ of device and driver APIs
OSC_LX\OSC_LL.C	Provides link list manipulation
OSC_LX\OSC_LX.H	OSC to Linux compile time mapping module
OSC_LX\OSC_LX.C	OSC to Linux function mapping module
OSC_LX\OSC_LX_INIT.C	OS initialization to start application - function main() User configurable module.
OSC_NU\OSC_LX_USR.C	Configure fatal error handler rountines to your needs
DEMO\PS_OSCHANGER\PS_LX_INIT.C	User configurable Linux initialization module User condigurable module.
DEMO\PS_OSCHANGER\PS_DEMO.C	Sample pSOS demo application that runs on Linux
DEMO\PS_OSCHANGER\PS_DVSERIAL.C	Sample pSOS device driver code
DEMO\PS_OSCHANGER\PS_USR.C	User configurable module to setup pSOS drivers' init configurations.

NOTE: Install OS Changer in the root file system (Rfs) under the folder called 'opt', in a directory called 'mapusoft'. Please be aware that the Rfs path location would be different depending on if you are working or doing a cross-compiling.

About pSOS OS Changer

OS Changer makes it easy to transition applications developed using pSOS™ kernel APIs to the Linux operating system. This product comes in the form of a library providing support for pSOS™ kernel APIs integrated and optimized for Linux operating system.
Porting is done in the following three steps:

- Remove references to the pSOS™ header files and the pSOS™ configuration tables within your application.
- Set pre-processor defines to indicate OS selection and also the OS Changer APIs that you require to use
- Include the Linux and OS Changer libraries and insert oschanger.h in your application.
- Compile, link and download your application to the target. Resolve compiler or linker or run-time errors as appropriate

NOTE: The pSOS™ APIs have gone through very little change over the past years and as a result this product should work with all pSOS™ versions. We also support older versions of pSOS APIs, so please contact MapuSoft for further help.

OS Changer and Linux OS Integration

The library mostly uses POSIX API functions and may accesses Linux OS's internal data structures to provide you further optimization under selected Linux Distribution. OS Changer may also be integrated with selected Linux vendors tools and IDE to provide you a out-of-the box solution. Some of the pSOS kernel APIs may be using more than one or more Linux equivalent APIs in order to provide the required pSOS API support. The OS Changer should work with all the versions of Linux that support POSIX 1003.1a, 1003.1b, and 1003.1c API compliance in the field because there were no specific changes are required or made to the underlying Linux product

How to Use pSOS OS Changer

OS Changer is designed for use as a C library. Services used inside your application software are extracted from the OS Changer and Linux libraries, and, are then combined with the other application objects to produce the complete image. This image may be downloaded to the target system or placed in ROM on the target system. Please refer to appropriate documentation for help with compiling, debugging and downloading your application to target.
The steps for using OS Changer are described in the following generic form:

- Remove the pSOS™ header file include defines from all your source files.
- Remove definitions and references to all the pSOS™ configuration data structures in your application.
- Include the OS Changer header file oschanger.h in all of the source files.
- In your project make file, define the RTOS for Linux, if you are using advanced real-time options of Linux, the appropriate compiler tool environment if required and other pre-processor options to build the OS Changer libraries and your application.
- Under Linux, the OSC_Application_Start function will be your main() routine. This function calls ps_Initialize which creates the pSOS root task. If some things need to be done for your application prior to root task creation, then place those codes between OSC_RTOS_Init and ps_Initialize functions.

- Modify the Linux BSPs to match with your development board configurations (see appropriate Linux documentation).
- Customize the priority mapping if necessary within OSC_RTOS_Init function. OS Changer does an automatic mapping of the required 257 priorities to Linux somewhere in the middle of Native Linux's lowest and highest priorities.
- Resolve the compiler and linker errors.
- Download the complete application image to the target system and resolve all the OS Changer generated run-time errors.

Please review the processor and development system documentation for additional information, including specific details on how to use the compiler, assembler, and linker. Please refer to the underlying Linux documentation to make the necessary changes to the BSP.

It is recommended that you first bring up the standard OS demo application provided by the Linux product for your target first, prior to trying out porting applications via OS Changer.

OS Changer is designed to be independent of the underlying hardware and operating system itself. It does not contain any assembly code. If you need any specific features of pSOS functions that is required but not provided by the standar OS Changer release, please contact MapuSoft Technologies. In most cases, we will be able to provide an easy work-around or may have an updated release covering those required functionalities.

OS Changer Library Initialization

After, Linux initializes itself, your applications main() entry point is mapped directly to OS Changer's OSC_Application_Start function where you will initialize your application if required. This function also provides a memory pointer for application's run time memory needs. But under Linux OS, since all OS Changer's and applications memory requirement are directly derived from the system heap, so you can safely ignore this parameter. There are four steps needed to be performed within the OSC_Application_Start() function located in PS_LX_INIT.C file module prior to using any of the OS Changer libraries:

1. Intialize OS Changer RTOS specific library by calling the OSC_RTOS_Init()
2. Insert any application specific code if necessary, before the root task get's spawned
3. Initialize the OS Changer pSOS library by calling ps_Initialize().
4. Just Idle or sleep so that your linux program will not exit.

```
#include "oschanger.h"      /* remove psos header file includes and use
oschanger.h */

void Function_Root(UNSIGNED);  /* root task - prototype definition */

ulong tRoot;  /* Define the Root Task ID, this is initialized in ps_initialize
func */

void OSC_Application_Start(VOID *first_available_memory)
{
       OSC_RTOS_Init();

       /* insert your code here !!!! */

       /* OS Changer psos library initialzation and root task creation */

       ps_Initialize(STACK_SIZE, &tRoot, Function_Root);

       for(;;)
         OSC_Sleep_Task(10000);  /* just Idle here, after starting root,
                             Otherwise your program will EXIT !!! */
}
```

When there is a fatal system error and the pre-processor flag OSC_DEBUG is set, the execution will stop inside the OSC_Fatal_Error function define in osc_lx\osc_lx_usr.c file. To handle the error differently, insert your code within OSC_Fatal_Error.

Device Drivers Initialization

The device drivers and the interrupt service routines needed to be ported to work under Linux. OS Changer provides the necessary application level API interface via the functions like de_init, de_open, and others. For each device that you access via the de_xxx interface, you will need to provide corresponding wrapper device driver routines for that device. The functions SetUpDrivers (see ps_usr.c module) will setup and install your driver. Setup installs the driver by calling InstallDriver along with providing required wrapper function pointers of the device specific routines. Within the wrapper rountines, use the device I/O routines to connect to the device and upon return, you can provide the driver response in the form how the application expects. This will greatly minimize changes to the application interacting with the Linux devices. When adding a driver, there are three steps:
Modify ps_oschanger.h to add the unique device ID with device major & minor values set accordingly.
NOTE: The device major ID cannot exceed the define SC_DEVMAX value.
If you need more drivers, then modify the SC_DEVMAX value accordingly. Code sample given below (refer to ps_oschanger.h also) adds device DEV_SERIAL_DEMO_DRV with the major number defined as SC_DEV_SERIAL_DEMO_DRV (equals to value 14, which is less than SC_DEVMAX) to the I/O system.

```
/* Device major ID value defined below  */
#define SC_DEV_SERIAL_DEMO_DRV 14  /* major number */

/* Device unique ID (major ID value = 14; minor ID value = 0).
Note that the major value is the most significant 16bits and minor value is the
least significant 16bits */
#define DEV_SERIAL_DEMO_DRV (SC_DEV_SERIAL_DEMO_DRV << 16)

Modify the SetUpDriver  to install and setup the device driver. See code sample
below:

/* Install the DEMO SERIAL DRIVER  */
/* Make sure you're the major value of the device ID does not exceed the dev
max value */
    #if(SC_DEV_SERIAL_DEMO_DRV > SC_DEVMAX)
        #error "SC_DEV_SERIAL_DEMO_DRV cannot be > SC_DEVMAX"
    #endif

/* sample installation and setup for the  serial driver */
InstallDriver(SC_DEV_SERIAL_DEMO_DRV, DevSerialInit, DevSerialOpen,
              DevSerialClose, DevSerialRead, DevSerialWrite,
              DevSerialCntrl, 0, 0);
```

Develop your device specific routines. See dvserial.c module in the demo directory for sample device specific routines. Every device specific routine should return two values (errcode and retval) to the de_xxxx api interface as shown below prior to their function return:
/* set driver return value */

```
   iopb->out_retval = 0;
iopb->err                                =                                    EOK;
```

NOTE: Please note that the return values are returned differently unlike how it pSOS does it. In pSOS, the return values are normally set in specific registers instead of how it is done above for OS Changer. However, this is much more convenient way since we are not reading writing to registers via assembly code.

Linux Time and Clock Initialization

In this release, tm_set and tm_get calendar time API calls are currently not supported. The number of clock ticks is defined by OSC_TIME_TICK_PER_SEC, which is retrieved from the Linux system. Under Red Hat®/GNU® Linux, this is actually 100 (this means every tick equals to 10ms). However, the OSC_TIME_TICK_PER_SEC could be different under other real-time or proprietary Linux.

Setting the task Time-Slice value while creating pSOS tasks with the time slice option set, will use the value called OSC_DEFAULT_TSLICE, which is defined in osc_lx.h. By default, this value is set for the time slice to be 100ms. Make sure you modify this value to match with your application needs if necessary.

Memory Usage

OS Changer libraries used the system heap directly to provide the dynamic and partition pool memory. The Memory management and garbage collection is best left for the Linux kernel to be handled, so OS Changer does not restrict application with memory request from partition and/or dynamic memory pools. The maximum memory the application can use will depend on the memory availability of the system heap.

Priority Mapping from pSOS to Linux

OS Changer first maps the pSOS priorities “0 to 255” to “255 to 0” OS Changer’s internal abstraction priority values. The abstraction priorities 256 plus one more for exclusivity are mapped to Linux utilizing a simple scheme (please refer to OSC_RTOS_INIT function defined in osc_lx.c). OS Changer queries to kernel to find out the min and max priorities to first calculate the linux priority window. Then it maps the abstraction priorities one on one to Linux priorities by picking up a range exactly in the middle of the linux priority window. Please modify the priority scheme as necessary for your application. If you want to minimize the interruption of the external native linux applications then you would want the OS Changer abstraction priorities to map to the higher end of the linux priority window.

OS Changer abstraction priority value of 257 is reserved internally by OS Changer to provide the necessary exclusivity among the OS Changer tasks when they request no preemption or task protection. The exclusivity and protections are not guaranteed if the external native Linux application runs at a higher priority.

It is recommended that the Linux kernel be configured to have a priority of 512, so that the OS Changer priorities will use the window range in the middle and as such would not interfere with some of core Linux components. If your Linux kernel is configured to have less than 257 priorities, the OS Changer will automatically configuring a windowing scheme, where multiple number of OSC Changer priorities will map to a single Linux priority. Because of this, the reported priority value could be slightly different than what was used during the task creating process. If your application uses the pre-processor called OSC_DEBUG, then all the priority values and calculations will be printed when you call the OSC_RTOS_Init function.

Conditional Compilations

Select the RTOS by setting the following compiler definition as follows:

Compilation Flag	Meaning
RTOS	The value of this flag indicates the RTOS selection defined in osc_changer.h: OSC_NUCLEUS – Nucleus PLUS from ATI OSC_THREADX – ThreadX® from Express Logic OSC_VXWORKS – VxWorks® from Wind River Systems OSC_MQX – Precise/MQX® from ARC® International OSC_ITRON – ITRON based operating system OSC_LINUX - Linux® OS If you are doing your own porting either to another commercial or proprietary RTOS, you could add your own define and include appropriate interface files. For Linux, define as RTOS = OSC_LINUX.

Compilation Flag	Meaning
LINUX_ADV_REALTIME	The value is to be used only when RTOS selection is OSC_LINUX. If your Linux distribution supports LINUX_ADV_REALTIME then you would want to set this define to 1 as shown below: LINUX_ADV_REALTIME = 1 This would provide a better performance and timer resolution and also will take advantage of the advanced real-time extensions offered under some Linux distributions.

Based on the compiler tools that you use, please select any one of the following definitions below (if your choice is not listed, you can ignore this pre-processor flag):

Compilation Flag	Meaning
ARM_TOOLS	Using ADS tools from ARM® Ltd
GNU_TOOLS	Using GNU Tools
MQX_TOOLS	Using Metaware® Tools from ARC® International

Select the OS Changer components for your application use as follows:

Compilation Flag	**Meaning**
INCLUDE_OSC_ANSI	This flag is NOT supported under LINUX OS
INCLUDE_OSC_IO	Define this flag if your application needs the OS Changer I/O API support
INCLUDE_OSC_PSOS	Define this flag if your application needs to use the pSOS compatibility APIs (optional product)
INCLUDE_OSC_VXWORKS	Define this flag if your application needs to use the VxWorks compatibility APIs (optional product)
INCLUDE_OSC_POSIX	Define this flag if your application needs to use the POSIX compatibility APIs (optional product)

Select if running under windows emulation and prototyping environment:

Compilation Flag	**Meaning**
BUILDING_ON_WIN32	This option is NOT supported under RTOS = LINUX at the moment mainly because Cygwin does not support all the required posix APIs that OS Changer needs. If you are building on Windows computer using RTOS prototyping environment (NOT instruction set simulator) then define this flag. Also you should not define this flag if you are building the application for a specific target.

Select the following definition if you want to OS Changer to enable error checking for debugging purposes:

Compilation Flag	**Meaning**
OSC_DEBUG_INFO	Enable error checking for debugging

Sample Porting of VxWorks Application with OS Changer using OSPAL

OS Changer is designed to be used as a C library. Services used inside your application software are extracted from the OS Changer and TARGET OS libraries. They are then combined with the other application objects to produce the complete image. You can download this image to the target system, or place it in ROM on the target system.

To start using *VxWorks*™ OS Changer, do the following:

Create a New Project

You have to create a new project in OS PAL for the application.

To create a new project:

1. From OS PAL main window, select any project under C/C++ **Projects** tab on the left pane.
2. Select **File** > **Porting** > **VxWorks** > **Import Workbench Project.** You can also click on the Porting icon from the task bar.
3. On OS PAL Import window, select a workspace directory to search for existing workbench projects by clicking on **Browse** button next to the text box, and click **Next.**
4. In the Projects in Workspace window, the projects list is displayed in a Checkbox Tree. Applications and Libraries are separated into respective categories.
5. Select or deselect any one or all of the projects by selecting the check box next to the project name and click **Finish** to import the project.
6. If you select any application type project, provide the inputs for the project and click **OK.** If you do not want to provide the inputs, you can just click **Cancel.**
7. If you select an application project and if it contains any referenced projects not selected by you, then a Confirmation dialogue box is displayed on your screen to ask if you want to port the project.
8. After the porting is successfully done, the porting report page is displayed. Click **Done** to complete the process.
9. The ported projects are displayed in OS PAL projects perspective.

You have successfully imported your VxWorks application to OS PAL.

Link-in MapuSoft Technologies Products with the Application

Now that you have your application is in OS PAL, you are ready to link-in MapuSoft products.

To link-inMT's products with the application:

1. Double click **os_application_start.c** in the **Source** folder in your project to open it.
2. Replace the contents by copying all of the content from **os_application_start.txt** (found in the folder with the sample VxWorks application files) and pasting it over everything in the original file and click **Save**. **Note**: You have replaced the template file created by OS PAL with code customized for your application.
3. Double click on the **windDemo.c** file in the **Source** folder in your project to open it.
4. Comment out the #include directives by adding /* at the beginning and */ at the end since the application will not need them anymore.
   NOTE: The text should turn green once the comment is active.

```
/*
#include "vxWorks.h"
#include "semLib.h"
#include "taskLib.h"
#include "msgQLib.h"
#include "wdLib.h"
#include "logLib.h"
#include "tickLib.h"
#include "sysLib.h"
#include "stdio.h"
*/
```

5. Link-in MT's header files with the application by adding the following right below where you typed */ and click **Save**.

```
#include  "osabstractor.h"
#include  "oschanger_vxworks.h"
```

Build the Application to Include MT's Products

You have to rebuild the application to include MT's products.

To build the application:

- Select the top level (the project name) of the project that you have created, right click and select **Build Project**.

Run the Application on the Host in OS PAL

Now that your application is using MapuSoft's products, you can run this real-time VxWorks application on a host for simulation and debugging. MapuSoft provides the best possible simulation because we do not add a scheduler which would cause a performance strain. The only constraint for this application is the non real-time OS, Windows, being used as a host. Also, debugging on a readily available host machine, such as the Windows computer is much easier than debugging directly in the target environment.

To run the application on the Host in OS PAL:

1. Select the project that you have created, right click and select **Debug As > Open Debug Dialog**.
2. Click on **New** icon on the top left corner (first icon, blank page with a plus).
3. Click on **Debugger** tab.
4. From the Debugger drop down menu, select **OS PAL Supplied GDB.**
5. When the Debug perspective is open, click **Debug** and click **Resume** (yellow and green play arrow). The debugger console (black box) should automatically appear in Windows task bar. Open it to show the application's execution.
6. Your VxWorks application is now running on the host. When finished, close the console to stop it from running.

Generate Code on the New Target OS

You can now move your VxWorks application to your target OS, for example Linux*.

*MapuSoft Technologies support the following targets: Threadx, Nucleus, Solaris, Windows XP, micro-ITRON, VxWorks, MQX, Linux, and QNX, LynxOS.

To generate code on the new target OS:

1. Click **OS PAL Projects Perspective** button to get back to your project.
2. Select the project that you have created and click on the **Optimizer** button.
3. Select the target OS you want to run this application now from the drop down menu.
4. Select the check box next to **Generate Project File.**
5. Choose a folder to save the files (make sure the folder has no spaces in the name) and click **Next.**
6. In the **File Path to Store Profiler Data** box, type the path to your OS PAL project "/folder name/project name".
7. Enter 500 in the **Number of Messages to Hold in Memory** box (replace default).
8. Enter 500 in the **Number of Profiler Messages** box (replace default).
9. Click on **Platform API Profiling** tab.
10. Select the box next to **Enable Platform Profiling.** This provides you with data concerning utilization of MapuSoft's APIs in your application. You can also view graphs and charts that detail performance data such as API execution time.
11. Click on **Application Functions Profiling** tab. This provides you with data concerning the functions in your application. This data is presented in charts and graphs to analyze and identify bottlenecks which are slowing down your application.
12. Select **Enable Application Function(s) Profiling.**
13. Enter the name "taskHighPri" in the **Application Function** box and click **Add.**
14. Enter the name "taskLowPri" in the **Application Function** box and click **Add.**
15. Click **Next**.
16. Show the Inline Feature, but keep it as default and click **Next.**
17. Show each configuration tab (leave all options as default with Task Pooling and Process Features turned off – they won't work with this sample application).
18. Click Finish.

Run the Application on the Target OS

Now that MapuSoft's products have been generated for your application, you are now ready to run the legacy VxWorks application on Linux.

Note: For the file coping to work, you must use Ethernet on the LAN, not wireless. You may also need to disable the firewalls on your computer (anti-virus and Windows).

To run the application on the Target OS:

1. Browse to the folder on your computer where you choose to save the generated files.
2. Copy the folder and paste it into your Shared Documents Folder.
3. Start the Microsoft Virtual PC program.
4. Double click on **CENTOS.**
5. Click on Applications > Network Servers.
6. Double click on the share with your name (you might have to browse to where you have saved your generated folder on your shared drive).
7. Copy the folder and paste it into the **Root** folder (**Root's home** icon on desktop).
8. Browse into the generated folder until you see the **makefile,** make a note of the path (if you cannot see the path, click **edit > preferences** and navigate to the second tab **Behavior**, and select the check box next to **Always open in browser windows** box. Exit and return to your folder).
9. Right click on the blank space on the desktop and select **Open Terminal.**
10. Enter cd /"path that is displayed when you browsed to the makefile in Step 9" (For example, **cd /root/example_folder**), and click **Enter.**
11. Enter "make clean all ROOT_DIR=$PWD", and click Enter.
12. You can see some Warnings. It is OK to view the warnings but be careful with the **Errors.**
13. Enter "/your-project-name_out", and click Enter.
14. Click **Control, C** to stop the application.

Now your VxWorks application is running on Linux.

If you wish to port this application to a different OS, you only need to repeat the code generation steps (Step 6 and 7) and choose a different OS. This provides true cross-OS development.

Revision History

Document Title: Programmers Guide for MapuSoft Standalone Products in MS Word
Release Number: 1.3.5

Release	Revision	Orig. of Change	Description of Change
1.3.5	0.1	Vv	• New document • Updated UITRON with micro-ITRON • Added revision history • Renamed Getting started to Programmers Guide • Changed the Programmers Guide description on page 8