

SysLink 02.00.00.68_beta1

Install Guide

Contents

Articles

SysLink 02.00.00.68 beta1 InstallGuide	1
SysLink 02.00.00.68 beta1 InstallGuide Linux TI81XX	2
SysLink 02.00.00.68 beta1 InstallGuide Linux OMAP3530	61
SysLink 02.00.00.68 beta1 InstallGuide Linux OMAPL1XX	84

References

Article Sources and Contributors	106
----------------------------------	-----

SysLink 02.00.00.68 beta1 InstallGuide

SYS/Link 02.00.00.68_beta1 Install Guide

Install Guide

This release is intended to be used on:

- TI81XX
- OMAP3530
- OMAPL1XX

Introduction

The SysLink install guide provides information on how to install SysLink and its dependencies (where relevant). It also provides information on how to build SysLink, its dependencies (where relevant), and SysLink sample applications. Additionally, information is provided on how to run the sample applications provided with the SysLink product package.

Installation

Basic installation

SysLink is made available as a tar.gz file. To install the product follow the steps below:

- Unzip and untar the file sysLink_<Version>_<ReleaseQualifier>.tar.gz.

Note:

- This document assumes the install path to be in the user home directory if working on a Linux PC. This path will be used in remainder of this document.
- If the installation was done at different location, make appropriate changes to the commands listed in the document.

It is advisable to archive the released sources in a configuration management system. This will help in merging:

- The updates delivered in the newer releases of SysLink.
- The changes to the product, if any, done by the users.

Dependencies

The dependencies, along with their download paths are given in the Release Notes.

IPC

Please follow the instructions given in <IPC_install>/ipc_<version>/docs/ User_install.pdf

Platform-specific Install Guides

Platform-specific install Guides for the SysLink product are available for all supported platforms:

- → TI81XX Linux
 - → OMAP3530 Linux
 - → OMAPL1XX Linux
-

SysLink 02.00.00.68 beta1 InstallGuide Linux TI81XX

SYS/Link 02.00.00.68_beta1 Linux TI81XX

SysLink Install Guide for TI81XX running Linux on the ARM Cortex A8.

Introduction

This document gives information about SysLink installation for using SysLink with the TI81XX platform, where the ARM Cortex A8 is running Linux OS. It also gives detailed build instructions for SysLink as well as the sample applications. In addition, instructions are also provided for running the sample applications provided within the SysLink release.

Dependencies

The dependencies, along with their versions are given in the Release Notes.

Installation

Setting up Linux Workstation

The description in this section is based on the following assumptions:

- The workstation is running on a Linux workstation
- Services telnetd, nfsd, ftpd are configured on this workstation.
- The workstation is assigned a fixed IP address.

A fixed IP address is preferred, as the IP address needs to be specified while booting the target board. This allows the boot loader configuration to be saved in flash.

Enable TFTP for downloading the kernel image to target

U-boot can be configured to download the kernel onto the target by various mechanisms:

- TFTP
- Serial Port

This section configures the Linux development host as a TFTP server. Modify the 'xinet.d/tftp' file to enable TFTP:

- Make the following changes:

```
disable      = no
```

```
server_args = -s /tftpboot
```

- Restart the network service

```
$ /etc/init.d/xinetd restart
```

The above configuration assumes that a directory 'tftpboot' has been created at the root '/' directory. The files in this directory are exposed through the TFTP protocol.

Building the Linux kernel

Kernel

Untar the kernel that comes with the Linux PSP TI81XX release. Build the kernel according to the PSP Linux User Guide document. Refer to the PSP u-boot related documentation on how to build the mkimage utility which is needed for creation of uImage.

U-Boot boot-loader

Please refer to the Linux PSP user guide to load the u.boot and x-loader to the target.

Create target file system

The target device needs a file system to boot from.

Code Composer Studio

Code composer studio can be used to connect to the DSP, run applications and debug the DSP side code

- Verified using CCS v4.2.07000
- Verified using CCS v5.0

EZSDK - EZSDK configurations cover Integra SDK device configurations i.e. DSP only use cases

Build

Editing files for Linux headers and tool chain paths

- Set SYSLINK_ROOT environment variable. This should be set to the base of the SysLink repository, i.e. the folder containing the `ti` directory.

```
$ export SYSLINK_ROOT=~/.syslink_<VERSION>
```

- set SYSLINK_PKGPATH environment variable for IPC package path.

```
export
SYSLINK_PKGPATH="/db/psp_git/syslink_toolchains/IPC/ipc_<version>/packages;
```

Replace ipc_<version> above with the actual version of the IPC.

- To build kernel side syslink
- Modify the KDIR variable in `SYSLINK_ROOT/ti/syslink/buildutils/hlos/knl/Makefile.inc` to point to the built kernel source.

Replace the path in the Makefile.inc with the actual installation path. Alternatively, it can be overridden by passing to the make command.

For TI814X

```
.....
ifeq ("$(SYSLINK_VARIANT)", "TI814X")
KDIR :=
/db/psp_git/syslink_toolchains/TI814X-LINUX-PSP-04.01.00.03/src/kernel/linux-04.01.00.03
else # ifeq ("$(SYSLINK_VARIANT)", "TI814X")
.....
```


For TI816X

```
.....
else # ifeq ("$(SYSLINK_VARIANT)", "TI814X")
KDIR :=
/db/psp_git/syslink_toolchains/TI816X-LINUX-PSP-04.00.00.10/src/kernel/linux-04.00.00.10
endif # ifeq ("$(SYSLINK_VARIANT)", "TI814X")
.....
```

Actual path in your build environment must be given for the KDIR path.

- Alternatively, the paths can be passed to the make command as a ';' separated set of paths. For example:

```
$ make ARCH=arm
CROSS_COMPILE=/toolchains/git/arm-2009q1-203/bin/arm-none-linux-gnueabi-
SYSLINK_PKGPATH="/toolchains/IPC/ipc_<version>/packages;$HOME/syslink"
```

Type the above command in a single line

- To build user syslink
- Modify the TOOLCHAIN_PREFIX variable in \$SYSLINK_ROOT/ti/syslink/buildutils/hlos/usr/Makefile.inc to point to arm toolchain. Replace the path in the Makefile.inc with the actual installation path. Alternatively, it can be overridden by passing to the make command.

```
.....
ifeq ("$(SYSLINK_PLATFORM)", "TI81XX")
TOOLCHAIN_PREFIX :=
/db/psp_git/syslink_toolchains/arm-2009q1/bin/arm-none-linux-gnueabi-
endif # ifeq ("$(SYSLINK_PLATFORM)", "TI81XX")
.....
```

- Alternatively, the paths can be passed to the make command as a ';' separated set of paths. For example:

```
$ make
TOOLCHAIN_PREFIX=/db/psp_git/syslink_toolchains/arm-2009q1/bin/arm-none-linux-gnueabi-
SYSLINK_PKGPATH="/toolchains/IPC/ipc_<version>/packages;$HOME/syslink"
```

Type the above command in a single line

- Update CGTOOLS tool chain path in the following files:

```
$(SYSLINK_ROOT)/config.bld
```

For example:

```
var rootDirPre = "/toolchains/ti-tools/";
var rootDirPost = "";
C674.rootDir = rootDirPre + "c6000_7.2.0A10232" + rootDirPost;
```

- If build is required for only a specific configuration, comment the other configurations in Build.targets. e.g. set configuration to build only for TI816X device.

```
//list interested targets in Build.targets array
Build.targets = [
```



```

        //C28_large,
        //C64,
        //C64P_COFF,
        //C64P_ELF,
        //C67P,
        //C674_COFF,
        C674_ELF,
        //Arm9,
        //M3_ELF,
        //Win32,
        //A8_ELF
    ];

```

Building SysLink HLOS driver/library and sample applications

The default configuration includes Ipc in the build for the drivers and the sample applications.

Switch to bash shell to build HLOS side drivers and samples. The default configuration of kernel driver builds for TI816X. To build for a different device, the SYSLINK_PLATFORM build parameter needs to be given.

Note:

- *SYSLINK_VARIANT option is available to distinguish between TI816X and TI814X. If you don't specify this it will build for TI816X by default. This option can be used only for TI81XX platform (default) and its usage for TI816X is kept optional for backward compatibility.*
- *When you are building for TI814X you have to explicitly specify **SYSLINK_VARIANT=TI814X** in all build commands below.*

Following are the steps to build the SysLink HLOS driver/library and sample applications for TI81XX:

- Set SYSLINK_ROOT environment variable

```
$ export SYSLINK_ROOT=~/.syslink_<VERSION>/
```

- Add the toolchain path to PATH variable

```
$ export PATH=$PATH:/toolchains/git/arm-2009q1-203/bin
```

- Build the SysLink kernel module

```
$ cd $SYSLINK_ROOT/ti/syslink/utis/hlos/knl/Linux
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
```

- Build SysLink kernel samples. Run make command in the respective sample directories.

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/notify/knl/Linux
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- SYSLINK_SDK=EZSDK
```

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/messageQ/knl/Linux
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- SYSLINK_SDK=EZSDK
```

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/heapBufMP/knl/Linux
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- SYSLINK_SDK=EZSDK
```

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/heapMemMP/knl/Linux
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- SYSLINK_SDK=EZSDK
```



```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/frameq/knl/Linux
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- SYSLINK_SDK=EZSDK

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/listMP/knl/Linux
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- SYSLINK_SDK=EZSDK

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/ringIO/knl/Linux
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- SYSLINK_SDK=EZSDK

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/ringIO_gpp/knl/Linux
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- SYSLINK_SDK=EZSDK

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/sharedRegion/knl/Linux
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- SYSLINK_SDK=EZSDK

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/gateMP/knl/Linux
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- SYSLINK_SDK=EZSDK
```

- The Build System supports two different profiles on user side for library as well as the sample applications as follows:

```
* debug
* release
* both - debug & release (default)
```

- Build the SysLink user library (**debug & release - both profiles together**)

```
$ cd $SYSLINK_ROOT/ti/syslink/utls/hlos/usr/Linux
$ make
```

- Build the SysLink user library (**debug profile**)

```
$ cd $SYSLINK_ROOT/ti/syslink/utls/hlos/usr/Linux
$ make debug
```

- Build the SysLink user library (**release profile**)

```
$ cd $SYSLINK_ROOT/ti/syslink/utls/hlos/usr/Linux
$ make release
```

Note: For building the library no need to specify platform as the syslink user library is *independent of platform*.

- Build SysLink user samples (**debug & release - both profiles together**). Run make command in the respective sample directories.

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/common/usr/Linux
$ make SYSLINK_SDK=EZSDK

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/procMgr/usr/Linux
$ make SYSLINK_SDK=EZSDK

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/notify/usr/Linux
```



```
$ make SYSLINK_SDK=EZSDK

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/messageQ/usr/Linux
$ make SYSLINK_SDK=EZSDK

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/listMP/usr/Linux
$ make SYSLINK_SDK=EZSDK

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/heapBufMP/usr/Linux
$ make SYSLINK_SDK=EZSDK

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/heapMemMP/usr/Linux
$ make SYSLINK_SDK=EZSDK

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/frameq/usr/Linux
$ make SYSLINK_SDK=EZSDK

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/ringIO/usr/Linux
$ make SYSLINK_SDK=EZSDK

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/ringIO_gpp/usr/Linux
$ make SYSLINK_SDK=EZSDK

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/sharedRegion/usr/Linux
$ make SYSLINK_SDK=EZSDK

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/gateMP/usr/Linux
$ make SYSLINK_SDK=EZSDK
```

- **Build SysLink user samples (debug profile).** Run make command in the respective sample directories.

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/common/usr/Linux
$ make debug SYSLINK_SDK=EZSDK

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/procMgr/usr/Linux
$ make debug SYSLINK_SDK=EZSDK

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/notify/usr/Linux
$ make debug SYSLINK_SDK=EZSDK

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/messageQ/usr/Linux
$ make debug SYSLINK_SDK=EZSDK

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/listMP/usr/Linux
$ make debug SYSLINK_SDK=EZSDK

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/heapBufMP/usr/Linux
$ make debug SYSLINK_SDK=EZSDK
```



```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/heapMemMP/usr/Linux
$ make debug SYSLINK_SDK=EZSDK

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/frameq/usr/Linux
$ make debug SYSLINK_SDK=EZSDK

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/ringIO/usr/Linux
$ make debug SYSLINK_SDK=EZSDK

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/ringIO_gpp/usr/Linux
$ make debug SYSLINK_SDK=EZSDK

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/sharedRegion/usr/Linux
$ make debug SYSLINK_SDK=EZSDK

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/gateMP/usr/Linux
$ make debug SYSLINK_SDK=EZSDK
```

- **Build SysLink user samples (release profile).** Run make command in the respective sample directories.

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/common/usr/Linux
$ make release SYSLINK_SDK=EZSDK

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/procMgr/usr/Linux
$ make release SYSLINK_SDK=EZSDK

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/notify/usr/Linux
$ make release SYSLINK_SDK=EZSDK

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/messageQ/usr/Linux
$ make release SYSLINK_SDK=EZSDK

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/listMP/usr/Linux
$ make release SYSLINK_SDK=EZSDK

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/heapBufMP/usr/Linux
$ make release SYSLINK_SDK=EZSDK

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/heapMemMP/usr/Linux
$ make release SYSLINK_SDK=EZSDK

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/frameq/usr/Linux
$ make release SYSLINK_SDK=EZSDK

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/ringIO/usr/Linux
$ make release SYSLINK_SDK=EZSDK
```



```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/ringIO_gpp/usr/Linux
$ make release SYSLINK_SDK=EZSDK

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/sharedRegion/usr/Linux
$ make release SYSLINK_SDK=EZSDK

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/gateMP/usr/Linux
$ make release SYSLINK_SDK=EZSDK
```

Building RTOS SysLink and sample applications

- Set SYSLINK_ROOT environment variable.

```
$ export SYSLINK_ROOT=~/.syslink_<VERSION>
```

- Set XDC in path.

```
$ export PATH=$PATH:/toolchains/xdc/xdctools_<version>
```

Alternatively, use the full path while making calls to xdc in the commands given later in this section

- Set XDC related environment variables.set XDCPATH= <IPC>;<BIOS>;<SYSLINK>

```
$
export XDCPATH="/users/ipc/ipc_<version>/packages;/toolchains/bios6/bios_<version>/packages;/toolchains/syslink_<version>/packages;/toolchains/rtos_<version>/packages"
export SYSLINK_ROOT;
```

Type the above command in a single line Replace <version> above with the specific version for IPC, BIOS & XDC tools

- Following statement needs to be added to all application config(.cfg) files

```
xdc.loadPackage ('ti.syslink.ipc.rtos');
```

This ensures that all required linker options are included by default

User needs to add the above mentioned statement to application config (.cfg) files in order to include all required linker options by default

- Build RTOS SysLink. This builds RTOS FrameQ and RingIO

```
$ cd $SYSLINK_ROOT/ti/syslink/
$ xdc all XDCARGS="SDK=EZSDK" XDCBUILDCFG="$SYSLINK_ROOT/config.bld"
-PR .
```

alternatively "XDCARGS" can be exported

```
$ export XDCARGS="SDK=EZSDK"
$ xdc all XDCBUILDCFG="$SYSLINK_ROOT/config.bld" -PR .
```

- Default build profile is "whole_program_debug" to change this to "**debug**" run the following

```
$ xdc all XDCARGS="profile=debug SDK=EZSDK"
XDCBUILDCFG="$SYSLINK_ROOT/config.bld" -PR .
```

alternatively "XDCARGS" can be exported

```
$ export XDCARGS="profile=debug SDK=EZSDK"
$ xdc all XDCBUILDCFG="$SYSLINK_ROOT/config.bld" -PR .
```


- Default build profile is "whole_program_debug" to change this to "**release**" run the following

```
$ xdc all XDCARGS="profile=release SDK=EZSDK"
XDCBUILDCFG="$SYSLINK_ROOT/config.bld" -PR .
```

alternatively "XDCARGS" can be exported

```
$ export XDCARGS="profile=release SDK=EZSDK"
$ xdc all XDCBUILDCFG="$SYSLINK_ROOT/config.bld" -PR .
```

- If you want to exclude Samples and build rest of the libraries, use following command

```
xdc XDCARGS="SDK=EZSDK" XDCBUILDCFG="$SYSLINK_ROOT/config.bld" -PR
'xdcpkg syslink | grep -v /samples'
```

The step to build SysLink RTOS side also builds all sample applications. If the syslink/ipc and sample applications are to be built independently, this can be done through separate commands as given below.

- To optimize the build time by utilizing "parallel build feature of GNU make" pass command line option **--jobs** as shown below

```
$ xdc all --jobs=4 XDCARGS="SDK=EZSDK"
XDCBUILDCFG="$SYSLINK_ROOT/config.bld" -PR .
```

--jobs= <number of seperate jobs to be run> //usually 1.5 times no of cpu cores available on the build system

- To build only the sample applications independently, use the following commands:

```
$ cd $SYSLINK_ROOT/ti/syslink/ipc
$ xdc all XDCARGS="SDK=EZSDK" -PR .

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/platforms
$ xdc all XDCARGS="SDK=EZSDK" -PR .

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/notify
$ xdc all XDCARGS="SDK=EZSDK"

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/messageQ
$ xdc all XDCARGS="SDK=EZSDK"

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/frameq
$ xdc all XDCARGS="SDK=EZSDK"

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/heapBufMP
$ xdc all XDCARGS="SDK=EZSDK"

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/heapMemMP
$ xdc all XDCARGS="SDK=EZSDK"

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/listMP
$ xdc all XDCARGS="SDK=EZSDK"

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/ringIO
```



```
$ xdc all XDCARGS="SDK=EZSDK"

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/ringIO_gpp
$ xdc all XDCARGS="SDK=EZSDK"

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/sharedRegion
$ xdc all XDCARGS="SDK=EZSDK"

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/gateMP
$ xdc all XDCARGS="SDK=EZSDK"
```

Build Notes

The default build configuration for SysLink and sample applications assumes TI816X. If no specific `SYSLINK_PLATFORM` value is specified, TI816X is assumed by default.

TI81XX supports the following loader configurations:

- DSP: ELF

ELF is the default loader format for the slave executables, to change to COFF format user needs to pass **SYSLINK_LOADER=COFF** compile time option when building SysLink HLOS kernel

```
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- SYSLINK_SDK=EZSDK
SYSLINK_LOADER=COFF
```

For additional information on different build parameters, please refer to the UserGuide document.

Configuring Kernel Parameters

SysLink requires a few specific arguments to be passed to the Linux kernel during boot up. For running the sample applications, 3MB of memory is used by SysLink for communication between GPP and DSP, and for DSP external memory for placing its code/data. This must be reserved by specifying 3MB less as available for the Linux kernel for its usage.

- For example, with available memory 256M, memory required for shared regions/other utils 5M and syslink 3M,

```
bootargs console=ttyS2,115200n8 root=/dev/nfs
nfsroot=HOST:nfs_root,nolock rw mem=248M ip=dhcp
```

This is just an example, bootargs may vary depending on available setup

Depending on the memory map used for the final system configuration, the memory to be reserved from Linux may be different.

Running the sample applications

Sample applications are provided with SysLink for the supported platforms. This section describes the way to execute the sample applications.

The steps for execution of the samples are given below for execution with Linux running on the GPP.

For all kernel-side applications, `procmgrapp` is expected to be run before/after running the specific application to load, start and stop the slave processor.

For all user-side applications, if no command line arguments are provided while running the sample application, the `procmgrapp` application is expected to be run before/after running the specific application to load, start and stop the slave processor.

If the following command line arguments are provided, then the user-side application also loads/starts/stops the slave processor in the application context itself. In this case, procmgrapp application is not required to be run before/after running the specific application:

Arguments: <numProcs> <procId1> <Path including name of the slave executable>

Copying files to target file system

The generated binaries on the HLOS side and RTOS side must be copied to the target directory.

For executing the sample applications, follow the steps below to copy the relevant binaries:

- Copy the syslink.ko kernel module into the target file system

```
$ cp ${SYSLINK_ROOT}/ti/syslink/bin/TI816X/*.ko
${HOME}/ti81xx/target/opt/syslink/.
```

- Copy the user and kernel sample application binaries into the target file system

```
cp ${SYSLINK_ROOT}/ti/syslink/bin/TI816X/samples/*
${HOME}/ti81xx/target/opt/syslink/.
```

- Copy RTOS binaries into the target file system

Type the below commands each in a single line :: \$cp <src> <dst>

- To copy the DSP executables use the following steps:

```
$ cd ${SYSLINK_ROOT}/ti/syslink/samples/rtos/
```

```
$ cp notify/ti_syslink_samples_rtos_platforms_ti81xx_dsp/
whole_program_debug/notify_ti81xx_dsp.xe674
${HOME}/ti81xx/target/opt/syslink/
```

```
$ cp messageQ/ti_syslink_samples_rtos_platforms_ti81xx_dsp/
whole_program_debug/messageq_ti81xx_dsp.xe674
${HOME}/ti81xx/target/opt/syslink/
```

```
$ cp frameq/ti_syslink_samples_rtos_platforms_ti81xx_dsp/
whole_program_debug/frameq_ti81xx_dsp.xe674
${HOME}/ti81xx/target/opt/syslink/
```

```
$ cp listMP/ti_syslink_samples_rtos_platforms_ti81xx_dsp/
whole_program_debug/listmp_ti81xx_dsp.xe674
${HOME}/ti81xx/target/opt/syslink/
```

```
$ cp heapBufMP/ti_syslink_samples_rtos_platforms_ti81xx_dsp/
whole_program_debug/heapbufmp_ti81xx_dsp.xe674
${HOME}/ti81xx/target/opt/syslink/
```

```
$ cp heapMemMP/ti_syslink_samples_rtos_platforms_ti81xx_dsp/
whole_program_debug/heapmemmp_ti81xx_dsp.xe674
${HOME}/ti81xx/target/opt/syslink/
```

```
$ cp ringIO/ti_syslink_samples_rtos_platforms_ti81xx_dsp/
whole_program_debug/ringio_ti81xx_dsp.xe674
```



```

${HOME}/ti81xx/target/opt/syslink/

$ cp ringIO_gpp/ti_syslink_samples_rtos_platforms_ti81xx_dsp/
whole_program_debug/ringiogpp_ti81xx_dsp.xe674
${HOME}/ti81xx/target/opt/syslink/

$ cp sharedRegion/ti_syslink_samples_rtos_platforms_ti81xx_dsp/
whole_program_debug/sharedregion_ti81xx_dsp.xe674
${HOME}/ti81xx/target/opt/syslink/

$ cp gateMP/ti_syslink_samples_rtos_platforms_ti81xx_dsp/
whole_program_debug/gatemp_ti81xx_dsp.xe674
${HOME}/ti81xx/target/opt/syslink/
```

Type the above commands each in a single line :: \$cp <src> <dst>

Running applications

In the below execute instructions, note that the DSP executables have extension .xe674 since they are built for ELF mode. COFF format produces extensions .x674 for DSP

Loading syslink kernel module

- The syslink kernel module must be inserted before running any sample applications.

```
$ insmod syslink.ko TRACE=1 TRACEFAILURE=1
```

Enabling TRACEFAILURE puts out prints in case any failure occurs during SysLink/sample application runs.

- Export TRACE and TRACEFAILURE environment variables so that any user side failures are also printed out.

```
export TRACE=1
export TRACEFAILURE=1
```

Unloading syslink kernel module

- The SysLink kernel module can be unloaded when it is no longer required, i.e. after all applications that use SysLink have exited. Unloading the SysLink kernel module reclaims to the HLOS, the resources that might have been consumed by SysLink.
- The SysLink Kernel module can be unloaded using:

```
$ rmmod syslink
```

Running the sample applications using scripts

Scripts have been provided for running the sample applications within the ti/syslink/tools/scripts folder.

Copying the scripts to target file system

Copy the following scripts into the target file system.

- Copy the top-level scripts into the target file system

```
$ cp ${SYSLINK_ROOT}/ti/syslink/tools/scripts/*.sh
${HOME}/ti81xx/target/opt/syslink/.
```

- Copy the scripts for each sample application for EZSDK into the target file system


```
$ cp ${SYSLINK_ROOT}/ti/syslink/tools/scripts/ti81xx/ezsdk/*.sh  
${HOME}/ti81xx/target/opt/syslink/.
```

Run the sample applications using the scripts

- To run the applications in debug build:

```
$ ./runsamples_debug.sh
```

- To run the applications in release build:

```
$ ./runsamples_release.sh
```

Running the ProcMgr sample application

ProcMgr user-side sample application

- To invoke the application enter the following commands:
- Execute the following to load, start and stop all slave processors. You can replace path and executables as per your setup:

Note: *procmgrapp_debug* takes two parameters *remote procId* and *executable name*.

```
$ procmgrapp_debug 0 /opt/syslink/messageq_ti81xx_dsp.xe674
```

- To use the procmgrapp as a loader for running other sample applications or as a standalone user-space loader to load any slave executable, use a final optional parameter
 - On providing the final parameter as 0, the procmgrapp only powers up, loads & starts the slave core
 - On providing the final parameter as 1, the procmgrapp only stops and powers down the slave core
- Load & start the slave core by invoking the procmgrapp application with final parameter as 0

```
$ procmgrapp_debug 0 /opt/syslink/messageq_ti81xx_dsp.xe674 0
```

- Run the sample application by inserting its kernel module. For example, messageqapp

```
$ insmod messageqapp.ko
```

- For clean up:

```
$ rmmod messageqapp
```

- Stop the slave core by invoking the procmgrapp application with final parameter as 1

```
$ procmgrapp_debug 0 /opt/syslink/messageq_ti81xx_dsp.xe674 1
```

Note: Procedure for running **procmgrapp_release** sample application is same as above.

Running the Notify sample application

Notify kernel-side sample application

- To invoke the application enter the following commands:
 - Execute the following to load and start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/notify_ti81xx_dsp.xe674 0
```

- Run the notifyapp sample application by inserting its kernel module

```
$ insmod notifyapp.ko
```

- For clean up:

```
$ rmmod notifyapp
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/notify_ti81xx_dsp.xe674 1
```

Notify user-side sample application

- To invoke the application standalone, enter the following commands:

```
$ ./notifyapp_debug 1 0 ./notify_ti81xx_dsp.xe674
```

- Press 'enter' to exit and cleanup.

Alternatively:

- To invoke the application with procmgrapp, enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/notify_ti81xx_dsp.xe674 0
```

- Run the notifyapp sample application

```
$ ./notifyapp_debug
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/notify_ti81xx_dsp.xe674 1
```

Note: Procedure for running **notifyapp_release** sample application is same as above.

Running the MessageQ sample application

MessageQ kernel-side sample application

- To invoke the application enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/messageq_ti81xx_dsp.xe674 0
```

- Run the messageqapp sample application by inserting its kernel module

```
$ insmod messageqapp.ko
```

- For clean up:

```
$ rmmod messageqapp
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/messageq_ti81xx_dsp.xe674 1
```

MessageQ user-side sample application

- To invoke the application standalone, enter the following commands:

```
$ ./messageqapp_debug 1 0 ./messageq_ti81xx_dsp.xe674
```

Alternatively:

- To invoke the application with procmgrapp, enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/messageq_ti81xx_dsp.xe674 0
```

- Run the messageqapp sample application

```
$ ./messageqapp_debug
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/messageq_ti81xx_dsp.xe674 1
```

Note: Procedure for running **messageqapp_release** sample application is same as above.

Running the GateMP sample application

GateMP kernel-side sample application

- To invoke the application enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/gatemp_ti81xx_dsp.xe674 0
```

- Run the gatempapp sample application by inserting its kernel module

```
$ insmod gatempapp.ko
```

- For clean up:

```
$ rmmod gatempapp
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/gatemp_ti81xx_dsp.xe674 1
```

GateMP user-side sample application

- To invoke the application standalone, enter the following commands:

```
$ ./gatempapp_debug 1 0 ./gatemp_ti81xx_dsp.xe674
```

Alternatively:

- To invoke the application with procmgrapp, enter the following commands:
 - Execute the following to load & start all slave processors. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/gatemp_ti81xx_dsp.xe674 0
```

- Run the gatempapp sample application

```
$ ./gatempapp_debug
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/gatemp_ti81xx_dsp.xe674 1
```

Note: Procedure for running **gatempapp_release** sample application is same as above.

Running the ListMP sample application

ListMP kernel-side sample application

- To invoke the application enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/listmp_ti81xx_dsp.xe674 0
```

- Run the listmpapp sample application by inserting its kernel module

```
$ insmod listmpapp.ko
```

- For clean up:

```
$ rmmod listmpapp
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/listmp_ti81xx_dsp.xe674 1
```

ListMP user-side sample application

- To invoke the application standalone, enter the following commands:

```
$ ./listmpapp_debug 1 0 ./listmp_ti81xx_dsp.xe674
```

Alternatively:

- To invoke the application with procmgrapp, enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/listmp_ti81xx_dsp.xe674 0
```

- Run the listmpapp sample application

```
$ ./listmpapp_debug
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/listmp_ti81xx_dsp.xe674 1
```

Note: Procedure for running **listmpapp_release** sample application is same as above.

Running the HeapBufMP sample application

HeapBufMP kernel-side sample application

- To invoke the application enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/heapbufmp_ti81xx_dsp.xe674 0
```

- Run the heapbufmpapp sample application by inserting its kernel module

```
$ insmod heapbufmpapp.ko
```

- For clean up:

```
$ rmmod heapbufmpapp
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/heapbufmp_ti81xx_dsp.xe674 1
```

HeapBufMP user-side sample application

- To invoke the application standalone, enter the following commands:

```
$ ./heapbufmpapp_debug 1 0 ./heapbufmp_ti81xx_dsp.xe674
```

Alternatively:

- To invoke the application with procmgrapp, enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/heapbufmp_ti81xx_dsp.xe674 0
```

- Run the heapbufmpapp sample application

```
$ ./heapbufmpapp_debug
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/heapbufmp_ti81xx_dsp.xe674 1
```

Note: Procedure for running **heapbufmpapp_release** sample application is same as above.

Running the HeapMemMP sample application

HeapMemMP kernel-side sample application

- To invoke the application enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/heapmemmp_ti81xx_dsp.xe674 0
```

- Run the heapmemmpapp sample application by inserting its kernel module

```
$ insmod heapmemmpapp.ko
```

- For clean up:

```
$ rmmod heapmemmpapp
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/heapmemmp_ti81xx_dsp.xe674 1
```

HeapMemMP user-side sample application

- To invoke the application standalone, enter the following commands:

```
$ ./heapmemmpapp_debug 1 0 ./heapmemmp_ti81xx_dsp.xe674
```

Alternatively:

- To invoke the application with procmgrapp, enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/heapmemmp_ti81xx_dsp.xe674 0
```

- Run the heapmemmpapp sample application

```
$ ./heapmemmpapp_debug
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/heapmemmp_ti81xx_dsp.xe674 1
```

Note: Procedure for running **heapmemmpapp_release** sample application is same as above.

Running the FrameQ sample application

FrameQ kernel-side sample application

- To invoke the application enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/frameq_ti81xx_dsp.xe674 0
```

- Run the frameqapp sample application by inserting its kernel module

```
$ insmod frameqapp.ko
```

- For clean up:

```
$ rmmod frameqapp
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/frameq_ti81xx_dsp.xe674 1
```

FrameQ user-side sample application

- To invoke the application standalone, enter the following commands:

```
$ ./frameqapp_debug 1 0 ./frameq_ti81xx_dsp.xe674
```

Alternatively:

- To invoke the application with procmgrapp, enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/frameq_ti81xx_dsp.xe674 0
```

- Run the frameqapp sample application

```
$ ./frameqapp_debug
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/frameq_ti81xx_dsp.xe674 1
```

Note: Procedure for running **frameqapp_release** sample application is same as above.

Running the RingIO sample application

RingIO kernel-side sample application

- To invoke the application enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/ringio_ti81xx_dsp.xe674 0
```

- Run the ringioapp sample application by inserting its kernel module

```
$ insmod ringioapp.ko
```

- For clean up:

```
$ rmmod ringioapp
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/ringio_ti81xx_dsp.xe674 1
```

RingIO user-side sample application

- To invoke the application standalone, enter the following commands:

```
$ ./ringioapp_debug 1 0 ./ringio_ti81xx_dsp.xe674
```

Alternatively:

- To invoke the application with procmgrapp, enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/ringio_ti81xx_dsp.xe674 0
```

- Run the ringioapp sample application

```
$ ./ringioapp_debug
```

- For clean up:

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/ringio_ti81xx_dsp.xe674 1
```

Note: Procedure for running **ringioapp_release** sample application is same as above.

Running the RingIO GPP sample application

RingIO GPP kernel-side sample application

- To invoke the application enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/ringiogpp_ti81xx_dsp.xe674 0
```

- Run the ringioapp sample application by inserting its kernel module

```
$ insmod ringiogppapp.ko
```

- For clean up:

```
$ rmmod ringiogppapp
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/ringiogpp_ti81xx_dsp.xe674 1
```

RingIO GPP user-side sample application

- To invoke the application standalone, enter the following commands:

```
$ ./ringiogppapp_debug 1 0 ./ringiogpp_ti81xx_dsp.xe674
```

Alternatively:

- To invoke the application with procmgrapp, enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/ringiogpp_ti81xx_dsp.xe674 0
```

- Run the ringiogppapp sample application

```
$ ./ringiogppapp_debug
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/ringiogpp_ti81xx_dsp.xe674 1
```

Note: Procedure for running **ringiogppapp_release** sample application is same as above.

Running the SharedRegion sample application

SharedRegion kernel-side sample application

- To invoke the application enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/sharedregion_ti81xx_dsp.xe674 0
```

- Run the sharedregionapp sample application by inserting its kernel module

```
$ insmod sharedregionapp.ko SHAREDMEM=0x8FD00000
```

- For clean up:

```
$ rmmmod sharedregionapp
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/sharedregion_ti81xx_dsp.xe674 1
```

Note: If your memory map is different, you may need to specify a different physical address value for SHAREDMEM.

SharedRegion user-side sample application

- To invoke the application standalone, enter the following commands:

```
$ ./sharedregionapp_debug 0x8FD00000 1 0 ./sharedregion_ti81xx_dsp.xe674
```

Alternatively:

- To invoke the application with procmgrapp, enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/sharedregion_ti81xx_dsp.xe674 0
```

- Run the sharedregionapp sample application

```
$ ./sharedregionapp_debug 0x8FD00000
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/sharedregion_ti81xx_dsp.xe674 1
```

Note: If your memory map is different, you may need to specify a different physical address value as the first parameter.

Note: Procedure for running **sharedregionapp_release** sample application is same as above.

Other non-EZSDK use-cases - Non EZSDK configurations cover non Integra SDK device configurations

Build

Editing files for Linux headers and tool chain paths

- Set SYSLINK_ROOT environment variable. This should be set to the base of the SysLink repository, i.e. the folder containing the `ti` directory.

```
$ export SYSLINK_ROOT=~/.syslink_<VERSION>
```

- To build kernel side syslink
- Modify the KDIR variable in `$SYSLINK_ROOT/ti/syslink/buildutils/hlos/knl/Makefile.inc` to point to the built kernel source.

Replace the path in the `Makefile.inc` with the actual installation path. Alternatively, it can be overridden by passing to the `make` command.

For TI814X

```
.....
ifeq ("$(SYSLINK_VARIANT)", "TI814X")
KDIR :=
/db/psp_git/syslink_toolchains/TI814X-LINUX-PSP-04.01.00.01/src/kernel/linux-04.01.00.01
else # ifeq ("$(SYSLINK_VARIANT)", "TI814X")
.....
```

For TI816X

```
.....
else # ifeq ("$(SYSLINK_VARIANT)", "TI814X")
KDIR :=
/db/psp_git/syslink_toolchains/TI816X-LINUX-PSP-04.00.00.09/src/kernel/linux-04.00.00.09
endif # ifeq ("$(SYSLINK_VARIANT)", "TI814X")
.....
```

Actual path in your build environment must be given for the KDIR path.

- Similarly change compile flag in the same file to point to ipc installation

```
.....
COMPILE_FLAGS +=
-I//db/psp_git/syslink_toolchains/IPC/ipc_<version>/packages/
COMPILE_FLAGS += -I$(SYSLINK_ROOT)
.....
```

Replace ipc_<version> above with the actual version of the IPC.

- Alternatively, the paths can be passed to the `make` command as a ';' separated set of paths. For example:

```
$ make ARCH=arm
CROSS_COMPILE=/toolchains/git/arm-2009q1-203/bin/arm-none-linux-gnueabi-

SYSLINK_PKGPATH="/toolchains/IPC/ipc_<version>/packages; $HOME/syslink"
```


Type the above command in a single line

- To build user syslink
- Modify the TOOLCHAIN_PREFIX variable in \$SYSLINK_ROOT/ti/syslink/buildutils/hlos/usr/Makefile.inc to point to arm toolchain. Replace the path in the Makefile.inc with the actual installation path. Alternatively, it can be overridden by passing to the make command.

```
.....
ifeq ("$(SYSLINK_PLATFORM)", "TI81XX")
TOOLCHAIN_PREFIX :=
/db/psp_git/syslink_toolchains/arm-2009q1/bin/arm-none-linux-gnueabi-
endif # ifeq ("$(SYSLINK_PLATFORM)", "TI81XX")
.....
```

- Similarly change compile flag in the same file to point to ipc installation

```
.....
COMPILE_FLAGS +=
-I/db/psp_git/syslink_toolchains/IPC/ipc_<version>/packages/
COMPILE_FLAGS += -I$(SYSLINK_ROOT)
.....
```

Replace ipc_<version> above with the actual version of the IPC.

- Alternatively, the paths can be passed to the make command as a ';' separated set of paths. For example:

```
$ make
TOOLCHAIN_PREFIX=/db/psp_git/syslink_toolchains/arm-2009q1/bin/arm-none-linux-gnueabi-
SYSLINK_PKGPATH="/toolchains/IPC/ipc_<version>/packages; $HOME/syslink"
```

Type the above command in a single line

- Update CGTOOLS tool chain path in the following files:

```
$(SYSLINK_ROOT)/config.bld
```

For example:

```
var rootDirPre = "/toolchains/ti-tools/";
var rootDirPost = "";
C674.rootDir = rootDirPre + "c6000_7.2.0A10232" + rootDirPost;
```

- If build is required for only a specific configuration, comment the other configurations in Build.targets. e.g. set configuration to build only for TI816X device.

```
//list interested targets in Build.targets array
Build.targets = [
    //C28_large,
    //C64,
    //C64P_COFF,
    //C64P_ELF,
    //C67P,
    //C674_COFF,
    C674_ELF,
```



```

        //Arm9,
        M3_ELF,
        //Win32,
        //A8_ELF
    ];

```

Important Points

- The Ducati AMMU configuration must be done from Video-M3 side. Applications **must** add the AMMU configuration code in the VIDEO-M3's config file e.g.

```

var AMMU = xdc.useModule('ti.sysbios.hal.ammu.AMMU');

/* Map the DDR using large page*/
AMMU.largePages[1].pageEnabled      = AMMU.Enable_YES;
AMMU.largePages[1].logicalAddress    = 0x80000000;
AMMU.largePages[1].translatedAddress = 0x80000000;
AMMU.largePages[1].translationEnabled = AMMU.Enable_YES;
AMMU.largePages[1].size              = AMMU.Large_512M;
AMMU.largePages[1].volatileQualifier = AMMU.Volatile_FOLLOW;
AMMU.largePages[1].L1_cacheable      = AMMU.CachePolicy_CACHEABLE;
AMMU.largePages[1].L1_posted         = AMMU.PostedPolicy_NON_POSTED;
AMMU.largePages[1].L2_cacheable      = AMMU.CachePolicy_NON_CACHEABLE;
AMMU.largePages[1].L2_posted         = AMMU.PostedPolicy_NON_POSTED;

/* Maps the Mailbox, Spinlocks using large page*/
AMMU.largePages[2].pageEnabled      = AMMU.Enable_YES;
AMMU.largePages[2].logicalAddress    = 0x40000000;
AMMU.largePages[2].translatedAddress = 0x40000000;
AMMU.largePages[2].translationEnabled = AMMU.Enable_YES;
AMMU.largePages[2].size              = AMMU.Large_512M;

/* Enables the unicache */
var Cache = xdc.useModule('ti.sysbios.hal.unicache.Cache');
Cache.enableCache = true;

```

Building SysLink HLOS driver/library and sample applications

The default configuration includes Ipc in the build for the drivers and the sample applications.

Switch to bash shell to build HLOS side drivers and samples. The default configuration of kernel driver builds for TI816X. To build for a different device, the SYSLINK_PLATFORM build parameter needs to be given.

Note:

- *SYSLINK_VARIANT* option has been added to distinguish between TI816X and TI814X. If you don't specify this it will build for TI816X by default. This option can be used only with TI81XX platform and its usage for TI816X is kept optional for backward compatibility.
- When you are building for TI814X you have to explicitly specify **SYSLINK_VARIANT=TI814X** in all build commands below.

Following are the steps to build the SysLink HLOS driver/library and sample applications for TI81XX:

- Set SYSLINK_ROOT environment variable


```
$ export SYSLINK_ROOT=~/.syslink_<VERSION>/
```

- Add the toolchain path to PATH variable

```
$ export PATH=$PATH:/toolchains/git/arm-2009q1-203/bin
```

- Build the SysLink kernel module

```
$ cd $SYSLINK_ROOT/ti/syslink/utils/hlos/knl/Linux
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
```

- Build SysLink kernel samples. Run make command in the respective sample directories.

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/notify/knl/Linux
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
```

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/messageQ/knl/Linux
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
```

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/heapBufMP/knl/Linux
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
```

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/heapMemMP/knl/Linux
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
```

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/frameq/knl/Linux
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
```

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/listMP/knl/Linux
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
```

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/ringIO/knl/Linux
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
```

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/ringIO_gpp/knl/Linux
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
```

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/sharedRegion/knl/Linux
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
```

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/gateMP/knl/Linux
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
```

- Build System now supports two different profiles on user side for library as well as the sample applications as follows:

```
* debug
* release
* both - debug & release (default)
```

- Build the SysLink user library (**debug & release - both profiles together**)


```
$ cd $SYSLINK_ROOT/ti/syslink/utils/hlos/usr/Linux
$ make
```

- Build the SysLink user library (**debug profile**)

```
$ cd $SYSLINK_ROOT/ti/syslink/utils/hlos/usr/Linux
$ make debug
```

- Build the SysLink user library (**release profile**)

```
$ cd $SYSLINK_ROOT/ti/syslink/utils/hlos/usr/Linux
$ make release
```

Note: For building the library no need to specify platform as the syslink user library is *independent of platform*.

- Build SysLink user samples (**debug & release - both profiles together**). Run make command in the respective sample directories.

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/procMgr/usr/Linux
$ make
```

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/notify/usr/Linux
$ make
```

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/messageQ/usr/Linux
$ make
```

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/listMP/usr/Linux
$ make
```

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/heapBufMP/usr/Linux
$ make
```

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/heapMemMP/usr/Linux
$ make
```

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/frameq/usr/Linux
$ make
```

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/ringIO/usr/Linux
$ make
```

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/ringIO_gpp/usr/Linux
$ make
```

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/sharedRegion/usr/Linux
$ make
```

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/gateMP/usr/Linux
$ make
```


- Build SysLink user samples (**debug profile**). Run make command in the respective sample directories.

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/procMgr/usr/Linux
$ make debug

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/notify/usr/Linux
$ make debug

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/messageQ/usr/Linux
$ make debug

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/listMP/usr/Linux
$ make debug

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/heapBufMP/usr/Linux
$ make debug

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/heapMemMP/usr/Linux
$ make debug

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/frameq/usr/Linux
$ make debug

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/ringIO/usr/Linux
$ make debug

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/ringIO_gpp/usr/Linux
$ make debug

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/sharedRegion/usr/Linux
$ make debug

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/gateMP/usr/Linux
$ make debug
```

- Build SysLink user samples (**release profile**). Run make command in the respective sample directories.

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/procMgr/usr/Linux
$ make release

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/notify/usr/Linux
$ make release

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/messageQ/usr/Linux
$ make release

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/listMP/usr/Linux
$ make release
```



```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/heapBufMP/usr/Linux
$ make release

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/heapMemMP/usr/Linux
$ make release

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/frameq/usr/Linux
$ make release

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/ringIO/usr/Linux
$ make release

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/ringIO_gpp/usr/Linux
$ make release

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/sharedRegion/usr/Linux
$ make release

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/gateMP/usr/Linux
$ make release
```

FrameQ-Tiler sample application

Tiler module is part of the kernel so kernel build will include the tiler functionality.

To build tiler daemon sample application:

- Make sure user side syslink library on user side and syslink package are built first.

```
$ cd $SYSLINK_ROOT/ti/syslink/utls/hlos/usr/Linux/
$ make TOOLCHAIN_PREFIX=arm-nonlinux-gnueabi- SYSLINK_VARIANT=TI816X
$ cd $SYSLINK_ROOT/ti/syslink/
$ xdc all
```

- Make sure rcm package in <framework_components> is rebuilt for GCArmv5T target in config.bld and specified <ipc> package. Comment all other targets in config.bld

```
$ cd framework_components_3_21_00_15_eng/packages/ti/sdo/rcm
$ xdc all
```

```
$ cd framework_components_3_21_00_15_eng/packages/ti/sdo/tiler
$ xdc all
```

- Now build the tilerDaemon

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/tilerDaemon
$ xdc all
```

- Rename the tiler daemon

```
$ cd ./host_platforms_arm/debug/
$ mv tiler_daemon.xv5T tiler_daemon.exe
```


It is mandatory to rename tiler daemon executable as .exe and then copy to target.

Building RTOS SysLink and sample applications

- Set SYSLINK_ROOT environment variable.

```
$ export SYSLINK_ROOT=~/.syslink_<VERSION>
```

- Set XDC in path.

```
$ export PATH=$PATH:/toolchains/xdc/xdctools_<version>
```

Alternatively, use the full path while making calls to xdc in the commands given later in this section

- Set XDC related environment variables.set XDCPATH= <IPC>;<BIOS>;<SYSLINK>

```
$
export XDCPATH="/users/ipc/ipc_<version>/packages;/toolchains/bios6/bios_<version>/packages;/toolchains/syslink_<version>/packages"
$SYSLINK_ROOT;
```

Type the above command in a single line Replace <version> above with the specific version for IPC, BIOS & XDC tools

- Following statement needs to be added to all application config(.cfg) files

```
xdc.loadPackage ('ti.syslink.ipc.rtos');
```

This ensures that all required linker options are included by default

User needs to add the above mentioned statement to application config (.cfg) files in order to include all required linker options by default

- Build RTOS SysLink. This builds RTOS FrameQ and RingIO

```
$ cd $SYSLINK_ROOT/ti/syslink/
$ xdc all XDCBUILDCFG="$SYSLINK_ROOT/config.bld" -PR .
```

- Default build profile is "whole_program_debug" to change this to "**debug**" run the following

```
$ xdc all XDCARGS="profile=debug"
XDCBUILDCFG="$SYSLINK_ROOT/config.bld" -PR .
```

alternatively "XDCARGS" can be exported

```
$ export XDCARGS="profile=debug"
$ xdc all XDCBUILDCFG="$SYSLINK_ROOT/config.bld" -PR .
```

- Default build profile is "whole_program_debug" to change this to "**release**" run the following

```
$ xdc all XDCARGS="profile=release"
XDCBUILDCFG="$SYSLINK_ROOT/config.bld" -PR .
```

alternatively "XDCARGS" can be exported

```
$ export XDCARGS="profile=release"
$ xdc all XDCBUILDCFG="$SYSLINK_ROOT/config.bld" -PR .
```

- If you want to exclude Samples and build rest of the libraries, use following command

```
xdc XDCBUILDCFG="$SYSLINK_ROOT/config.bld" -PR 'xdcpkg syslink | grep
-v /samples'
```


The step to build SysLink RTOS side also builds all sample applications. If the syslink/ipc and sample applications are to be built independently, this can be done through separate commands as given below.

- To optimize the build time by utilizing "parallel build feature of GNU make" pass command line option **--jobs** as shown below

```
$ xdc all --jobs=4 XDCBUILDCFG="$SYSLINK_ROOT/config.bld" -PR .
```

--jobs= <number of seperate jobs to be run> //usually 1.5 times no of cpu cores available on the build system

- To build only the sample applications independently, use the following commands:

```
$ cd $SYSLINK_ROOT/ti/syslink/ipc
$ xdc all -PR .

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/platforms
$ xdc all -PR .

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/notify
$ xdc all

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/messageQ
$ xdc all

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/frameq
$ xdc all

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/heapBufMP
$ xdc all

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/heapMemMP
$ xdc all

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/listMP
$ xdc all

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/ringIO
$ xdc all

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/ringIO_gpp
$ xdc all

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/sharedRegion
$ xdc all

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/gateMP
$ xdc all

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/ti81xx/ringIO_ti81xx
$ xdc all
```



```
$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/ti81xx/frameq_ti81xx
$ xdc all

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/ti81xx/frameq_ti814x
$ xdc all

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/ti81xx/ringIO_ti814x
$ xdc all

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/ti81xx/singlecore/frameq
$ xdc all

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/ti81xx/singlecore/ringio
$ xdc all
```

- If frameq_ti81xx sample is to be built for tiled buffers:

Enable M3 target in config.bld

```
$cd framework_components_3_21_00_15_eng/packages/ti/sdo/rcm
$xdc all
```

```
$cd framework_components_3_21_00_15_eng/packages/ti/sdo/tiler
$xdc all
```

Update FrameQ_ti81xx_videom3.cfg and FrameQ_ti81xx_vpssm3.cfg to make
<code>ipcWithHLOS = 1</code>.

Update config.bld to define compiler option SYSLINK_USE_TILER

```
$cd $SYSLINK_ROOT/ti/syslink/samples/rtos/ti81xx/frameq_ti81xx
$xdc all
```

- If frameq_ti81xx sample is to be built for shared memory buffers:

Retain the default configuration in FrameQ_ti81xx_videom3.cfg and
FrameQ_ti81xx_vpssm3.cfg to have <code>ipcWithHLOS =
0</code>.

Retain the default config.bld to keep compiler option
SYSLINK_USE_TILER commented.

Build Notes

The default build configuration for SysLink and sample applications assumes TI816X. If no specific `SYSLINK_PLATFORM` value is specified, TI816X is assumed by default.

TI81XX supports the following loader configurations:

- VIDEO-M3: ELF
- VPSS-M3: ELF
- DSP: ELF

ELF is the default loader format for the slave executables, to change to COFF format user needs to pass **SYSLINK_LOADER=COFF** compile time option when building SysLink HLOS kernel

```
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
SYSLINK_LOADER=COFF
```

For additional information on different build parameters, please refer to the UserGuide document.

Configuring Kernel Parameters

SysLink requires a few specific arguments to be passed to the Linux kernel during boot up. For running the sample applications, 3MB of memory is used by SysLink for communication between GPP and DSP, and for DSP external memory for placing its code/data. This must be reserved by specifying 3MB less as available for the Linux kernel for its usage.

- For example, with available memory 256M, memory required for shared regions/other utils 5M and syslink 3M,

```
bootargs console=ttyS2,115200n8 root=/dev/nfs
nfsroot=HOST:nfs_root,nolock rw mem=248M ip=dhcp
```

This is just an example, bootargs may vary depending on available setup

Depending on the memory map used for the final system configuration, the memory to be reserved from Linux may be different.

Running the sample applications

Sample applications are provided with SysLink for the supported platforms. This section describes the way to execute the sample applications.

The steps for execution of the samples are given below for execution with Linux running on the GPP.

For all kernel-side applications, `procmgrapp` is expected to be run before/after running the specific application to load, start and stop the slave processor.

For all user-side applications, if no command line arguments are provided while running the sample application, the `procmgrapp` application is expected to be run before/after running the specific application to load, start and stop the slave processor.

If the following command line arguments are provided, then the user-side application also loads/starts/stops the slave processor in the application context itself. In this case, `procmgrapp` application is not required to be run before/after running the specific application:

```
Arguments: <numProcs> <procId1> <Path including name of
the slave executable>
```


Copying files to target file system

The generated binaries on the HLOS side and RTOS side must be copied to the target directory.

For executing the sample applications, follow the steps below to copy the relevant binaries:

- Copy the syslink.ko kernel module into the target file system

```
$ cp ${SYSLINK_ROOT}/ti/syslink/bin/TI816X/*.ko  
${HOME}/ti81xx/target/opt/syslink/.
```

- Copy the user and kernel sample application binaries into the target file system

```
cp ${SYSLINK_ROOT}/ti/syslink/bin/TI816X/samples/*  
${HOME}/ti81xx/target/opt/syslink/.
```

- Copy RTOS binaries into the target file system

Type the below commands each in a single line :: \$cp <src> <dst>

- To copy the DSP executables use the following steps:

```
$ cd ${SYSLINK_ROOT}/ti/syslink/samples/rtos/
```

```
$ cp notify/ti_syslink_samples_rtos_platforms_ti81xx_dsp/  
whole_program_debug/notify_ti81xx_dsp.xe674  
${HOME}/ti81xx/target/opt/syslink/
```

```
$ cp messageQ/ti_syslink_samples_rtos_platforms_ti81xx_dsp/  
whole_program_debug/messageq_ti81xx_dsp.xe674  
${HOME}/ti81xx/target/opt/syslink/
```

```
$ cp frameq/ti_syslink_samples_rtos_platforms_ti81xx_dsp/  
whole_program_debug/frameq_ti81xx_dsp.xe674  
${HOME}/ti81xx/target/opt/syslink/
```

```
$ cp listMP/ti_syslink_samples_rtos_platforms_ti81xx_dsp/  
whole_program_debug/listmp_ti81xx_dsp.xe674  
${HOME}/ti81xx/target/opt/syslink/
```

```
$ cp heapBufMP/ti_syslink_samples_rtos_platforms_ti81xx_dsp/  
whole_program_debug/heapbufmp_ti81xx_dsp.xe674  
${HOME}/ti81xx/target/opt/syslink/
```

```
$ cp heapMemMP/ti_syslink_samples_rtos_platforms_ti81xx_dsp/  
whole_program_debug/heapmemmp_ti81xx_dsp.xe674  
${HOME}/ti81xx/target/opt/syslink/
```

```
$ cp ringIO/ti_syslink_samples_rtos_platforms_ti81xx_dsp/  
whole_program_debug/ringio_ti81xx_dsp.xe674  
${HOME}/ti81xx/target/opt/syslink/
```

```
$ cp ringIO_gpp/ti_syslink_samples_rtos_platforms_ti81xx_dsp/  
whole_program_debug/ringiogpp_ti81xx_dsp.xe674  
${HOME}/ti81xx/target/opt/syslink/
```



```
$ cp sharedRegion/ti_syslink_samples_rtos_platforms_ti81xx_dsp/
whole_program_debug/sharedregion_ti81xx_dsp.xe674
${HOME}/ti81xx/target/opt/syslink/
```

```
$ cp gateMP/ti_syslink_samples_rtos_platforms_ti81xx_dsp/
whole_program_debug/gatemp_ti81xx_dsp.xe674
${HOME}/ti81xx/target/opt/syslink/
```

Type the above commands each in a single line :: \$cp <src> <dst>

- Similarly copy VIDEO-M3 and VPSS-M3 executables to the target file system. e.g.

```
$ cd ${SYSLINK_ROOT}/ti/syslink/samples/rtos/
```

```
$ cp notify/ti_syslink_samples_rtos_platforms_ti81xx_videom3/
whole_program_debug/notify_ti81xx_videom3.xem3
${HOME}/ti81xx/target/opt/syslink/
```

```
$ cp notify/ti_syslink_samples_rtos_platforms_ti81xx_vpssm3/
whole_program_debug/notify_ti81xx_vpssm3.xem3
${HOME}/ti81xx/target/opt/syslink/
```

Running applications

In the below execute instructions, note that the DSP executables have extension .xe674, VPSS-M3 and VIDEO-M3 executables to be used have extension .xem3 since they are built for ELF mode. COFF format produces extensions .x674 and .xm3 for DSP and M3s respectively

Loading syslink kernel module

- The syslink kernel module must be inserted before running any sample applications.

```
$ insmod syslink.ko TRACE=1 TRACEFAILURE=1
```

Enabling TRACEFAILURE puts out prints in case any failure occurs during SysLink/sample application runs.

- Export TRACE and TRACEFAILURE environment variables so that any user side failures are also printed out.

```
export TRACE=1
export TRACEFAILURE=1
```

Unloading syslink kernel module

- Kernel module can be unloaded using

```
$ rmmod syslink
```

Running the sample applications using scripts

Scripts have been provided for running the sample applications within the ti/syslink/tools/scripts folder.

Copying the scripts to target file system

Copy the following scripts into the target file system.

- Copy the top-level scripts into the target file system

```
$ cp ${SYSLINK_ROOT}/ti/syslink/tools/scripts/*.sh
${HOME}/ti81xx/target/opt/syslink/.
```


- Copy the scripts for each sample application into the target file system

```
$ cp ${SYSLINK_ROOT}/ti/syslink/tools/scripts/ti81xx/*.sh
${HOME}/ti81xx/target/opt/syslink/.
```

Run the sample applications using the scripts

- To run the applications in debug build:

```
$ ./runsamples_debug.sh
```

- To run the applications in release build:

```
$ ./runsamples_release.sh
```

Running the ProcMgr sample application

ProcMgr user-side sample application

- To invoke the application enter the following commands:
- Execute the following to load, start and stop all slave processors. You can replace path and executables as per your setup:

Note: *procmgrapp_debug* takes two parameters *remote procId* and *executable name*.

```
$ procmgrapp_debug 0 /opt/syslink/messageq_ti81xx_dsp.xe674
```

- To use the procmgrapp as a loader for running other sample applications or as a standalone user-space loader to load any slave executable, use a final optional parameter
 - On providing the final parameter as 0, the procmgrapp only powers up, loads & starts the slave core
 - On providing the final parameter as 1, the procmgrapp only stops and powers down the slave core
- Load & start all the slave cores by invoking the procmgrapp application with final parameter as 0

```
$ procmgrapp_debug 0 /opt/syslink/messageq_ti81xx_dsp.xe674 0
```

```
$ procmgrapp_debug 1 /opt/syslink/messageq_ti81xx_videom3.xem3 0
```

```
$ procmgrapp_debug 2 /opt/syslink/messageq_ti81xx_vpssm3.xem3 0
```

- Run the sample application by inserting its kernel module. For example, messageqapp

```
$ insmod messageqapp.ko
```

- For clean up:

```
$ rmmod messageqapp
```

- Stop all the slave cores by invoking the procmgrapp application with final parameter as 1

```
$ procmgrapp_debug 0 /opt/syslink/messageq_ti81xx_dsp.xe674 1
```

```
$ procmgrapp_debug 1 /opt/syslink/messageq_ti81xx_videom3.xem3 1
```

```
$ procmgrapp_debug 2 /opt/syslink/messageq_ti81xx_vpssm3.xem3 1
```

Note: Procedure for running **procmgrapp_release** sample application is same as above.

Running the Notify sample application

Notify kernel-side sample application

- To invoke the application enter the following commands:
 - Execute the following to load and start all slave processors. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/notify_ti81xx_dsp.xe674 0
```

```
$ procmgrapp_debug 1 /opt/syslink/notify_ti81xx_videom3.xem3 0
```

```
$ procmgrapp_debug 2 /opt/syslink/notify_ti81xx_vpssm3.xem3 0
```

- Run the notifyapp sample application by inserting its kernel module

```
$ insmod notifyapp.ko
```

- For clean up:

```
$ rmmmod notifyapp
```

- Execute the following to stop all slave processors.

```
$ procmgrapp_debug 0 /opt/syslink/notify_ti81xx_dsp.xe674 1
```

```
$ procmgrapp_debug 1 /opt/syslink/notify_ti81xx_videom3.xem3 1
```

```
$ procmgrapp_debug 2 /opt/syslink/notify_ti81xx_vpssm3.xem3 1
```

Notify user-side sample application

- To invoke the application standalone, enter the following commands:

```
$ ./notifyapp_debug 3 0 /opt/syslink/notify_ti81xx_dsp.xe674  
1 /opt/syslink/notify_ti81xx_videom3.xem3 2  
/opt/syslink/notify_ti81xx_vpssm3.xem3
```

- Command above should be entered in single line
- Press 'enter' to exit and cleanup.

Alternatively:

- To invoke the application with procmgrapp, enter the following commands:
 - Execute the following to load & start all slave processors. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/notify_ti81xx_dsp.xe674 0
```

```
$ procmgrapp_debug 1 /opt/syslink/notify_ti81xx_videom3.xem3 0
```

```
$ procmgrapp_debug 2 /opt/syslink/notify_ti81xx_vpssm3.xem3 0
```

- Run the notifyapp sample application

```
$ ./notifyapp_debug
```

- Execute the following to stop all slave processors.

```
$ procmgrapp_debug 0 /opt/syslink/notify_ti81xx_dsp.xe674 1
```



```
$ procmgrapp_debug 1 /opt/syslink/notify_ti81xx_videom3.xem3 1
```

```
$ procmgrapp_debug 2 /opt/syslink/notify_ti81xx_vpssm3.xem3 1
```

Note: Procedure for running **notifyapp_release** sample application is same as above.

Running the MessageQ sample application

MessageQ kernel-side sample application

- To invoke the application enter the following commands:
 - Execute the following to load & start all slave processors. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/messageq_ti81xx_dsp.xe674 0
```

```
$ procmgrapp_debug 1 /opt/syslink/messageq_ti81xx_videom3.xem3 0
```

```
$ procmgrapp_debug 2 /opt/syslink/messageq_ti81xx_vpssm3.xem3 0
```

- Run the messageqapp sample application by inserting its kernel module

```
$ insmod messageqapp.ko
```

- For clean up:

```
$ rmmod messageqapp
```

- Execute the following to stop all slave processors.

```
$ procmgrapp_debug 0 /opt/syslink/messageq_ti81xx_dsp.xe674 1
```

```
$ procmgrapp_debug 1 /opt/syslink/messageq_ti81xx_videom3.xem3 1
```

```
$ procmgrapp_debug 2 /opt/syslink/messageq_ti81xx_vpssm3.xem3 1
```

MessageQ user-side sample application

- To invoke the application standalone, enter the following commands:

```
$ ./messageqapp_debug 3 0 /opt/syslink/messageq_ti81xx_dsp.xe674  
1 /opt/syslink/messageq_ti81xx_videom3.xem3 2  
/opt/syslink/messageq_ti81xx_vpssm3.xem3
```

- Command above should be entered in single line

Alternatively:

- To invoke the application with procmgrapp, enter the following commands:
 - Execute the following to load & start all slave processors. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/messageq_ti81xx_dsp.xe674 0
```

```
$ procmgrapp_debug 1 /opt/syslink/messageq_ti81xx_videom3.xem3 0
```

```
$ procmgrapp_debug 2 /opt/syslink/messageq_ti81xx_vpssm3.xem3 0
```

- Run the messageqapp sample application


```
$ ./messageqapp_debug
```

- Execute the following to stop all slave processors.

```
$ procmgrapp_debug 0 /opt/syslink/messageq_ti81xx_dsp.xe674 1
```

```
$ procmgrapp_debug 1 /opt/syslink/messageq_ti81xx_videom3.xem3 1
```

```
$ procmgrapp_debug 2 /opt/syslink/messageq_ti81xx_vpssm3.xem3 1
```

Note: Procedure for running **messageqapp_release** sample application is same as above.

Running the GateMP sample application

GateMP kernel-side sample application

- To invoke the application enter the following commands:
 - Execute the following to load & start all slave processors. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/gatemp_ti81xx_dsp.xe674 0
```

```
$ procmgrapp_debug 1 /opt/syslink/gatemp_ti81xx_videom3.xem3 0
```

```
$ procmgrapp_debug 2 /opt/syslink/gatemp_ti81xx_vpssm3.xem3 0
```

- Run the gatempapp sample application by inserting its kernel module

```
$ insmod gatempapp.ko
```

- For clean up:

```
$ rmmod gatempapp
```

- Execute the following to stop all slave processors.

```
$ procmgrapp_debug 0 /opt/syslink/gatemp_ti81xx_dsp.xe674 1
```

```
$ procmgrapp_debug 1 /opt/syslink/gatemp_ti81xx_videom3.xem3 1
```

```
$ procmgrapp_debug 2 /opt/syslink/gatemp_ti81xx_vpssm3.xem3 1
```

GateMP user-side sample application

- To invoke the application standalone, enter the following commands:

```
$ ./gatempapp_debug 3 0 /opt/syslink/gatemp_ti81xx_dsp.xe674  
1 /opt/syslink/gatemp_ti81xx_videom3.xem3 2  
/opt/syslink/gatemp_ti81xx_vpssm3.xem3
```

- Command above should be entered in single line

Alternatively:

- To invoke the application with procmgrapp, enter the following commands:
 - Execute the following to load & start all slave processors. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/gatemp_ti81xx_dsp.xe674 0
```



```
$ procgrapp_debug 1 /opt/syslink/gatemp_ti81xx_videom3.xem3 0
```

```
$ procgrapp_debug 2 /opt/syslink/gatemp_ti81xx_vpssm3.xem3 0
```

- Run the gatempapp sample application

```
$ ./gatempapp_debug
```

- Execute the following to stop all slave processors.

```
$ procgrapp_debug 0 /opt/syslink/gatemp_ti81xx_dsp.xe674 1
```

```
$ procgrapp_debug 1 /opt/syslink/gatemp_ti81xx_videom3.xem3 1
```

```
$ procgrapp_debug 2 /opt/syslink/gatemp_ti81xx_vpssm3.xem3 1
```

Note: Procedure for running **gatempapp_release** sample application is same as above.

Running the ListMP sample application

ListMP kernel-side sample application

- To invoke the application enter the following commands:
 - Execute the following to load & start all slave processors. You can replace path and executables as per your setup:

```
$ procgrapp_debug 0 /opt/syslink/listmp_ti81xx_dsp.xe674 0
```

```
$ procgrapp_debug 1 /opt/syslink/listmp_ti81xx_videom3.xem3 0
```

```
$ procgrapp_debug 2 /opt/syslink/listmp_ti81xx_vpssm3 0
```

- Run the listmpapp sample application by inserting its kernel module

```
$ insmod listmpapp.ko
```

- For clean up:

```
$ rmmod listmpapp
```

- Execute the following to stop all slave processors.

```
$ procgrapp_debug 0 /opt/syslink/listmp_ti81xx_dsp.xe674 1
```

```
$ procgrapp_debug 1 /opt/syslink/listmp_ti81xx_videom3.xem3 1
```

```
$ procgrapp_debug 2 /opt/syslink/listmp_ti81xx_vpssm3 1
```


ListMP user-side sample application

- To invoke the application standalone, enter the following commands:

```
$ ./listmpapp_debug 3 0 /opt/syslink/listmp_ti81xx_dsp.xe674
1 /opt/syslink/listmp_ti81xx_videom3.xem3 2
/opt/syslink/listmp_ti81xx_vpssm3
```

- Command above should be entered in single line

Alternatively:

- To invoke the application with procmgrapp, enter the following commands:
 - Execute the following to load & start all slave processors. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/listmp_ti81xx_dsp.xe674 0
```

```
$ procmgrapp_debug 1 /opt/syslink/listmp_ti81xx_videom3.xem3 0
```

```
$ procmgrapp_debug 2 /opt/syslink/listmp_ti81xx_vpssm3 0
```

- Run the listmpapp sample application

```
$ ./listmpapp_debug
```

- Execute the following to stop all slave processors.

```
$ procmgrapp_debug 0 /opt/syslink/listmp_ti81xx_dsp.xe674 1
```

```
$ procmgrapp_debug 1 /opt/syslink/listmp_ti81xx_videom3.xem3 1
```

```
$ procmgrapp_debug 2 /opt/syslink/listmp_ti81xx_vpssm3 1
```

Note: Procedure for running **listmpapp_release** sample application is same as above.

Running the HeapBufMP sample application**HeapBufMP kernel-side sample application**

- To invoke the application enter the following commands:
 - Execute the following to load & start all slave processors. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/heapbufmp_ti81xx_dsp.xe674 0
```

```
$ procmgrapp_debug 1 /opt/syslink/heapbufmp_ti81xx_videom3.xem3 0
```

```
$ procmgrapp_debug 2 /opt/syslink/heapbufmp_ti81xx_vpssm3.xem3 0
```

- Run the heapbufmpapp sample application by inserting its kernel module

```
$ insmod heapbufmpapp.ko
```

- For clean up:

```
$ rmmod heapbufmpapp
```

- Execute the following to stop all slave processors.

```
$ procmgrapp_debug 0 /opt/syslink/heapbufmp_ti81xx_dsp.xe674 1
```



```
$ procmgrapp_debug 1 /opt/syslink/heapbufmp_ti81xx_videom3.xem3 1
$ procmgrapp_debug 2 /opt/syslink/heapbufmp_ti81xx_vpssm3.xem3 1
```

HeapBufMP user-side sample application

- To invoke the application standalone, enter the following commands:

```
$ ./heapbufmpapp_debug 3 0 /opt/syslink/heapbufmp_ti81xx_dsp.xe674
1 /opt/syslink/heapbufmp_ti81xx_videom3.xem3 2
/opt/syslink/heapbufmp_ti81xx_vpssm3.xem3
```

- Command above should be entered in single line

Alternatively:

- To invoke the application with procmgrapp, enter the following commands:
 - Execute the following to load & start all slave processors. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/heapbufmp_ti81xx_dsp.xe674 0
$ procmgrapp_debug 1 /opt/syslink/heapbufmp_ti81xx_videom3.xem3 0
$ procmgrapp_debug 2 /opt/syslink/heapbufmp_ti81xx_vpssm3.xem3 0
```

- Run the heapbufmpapp sample application

```
$ ./heapbufmpapp_debug
```

- Execute the following to stop all slave processors.

```
$ procmgrapp_debug 0 /opt/syslink/heapbufmp_ti81xx_dsp.xe674 1
$ procmgrapp_debug 1 /opt/syslink/heapbufmp_ti81xx_videom3.xem3 1
$ procmgrapp_debug 2 /opt/syslink/heapbufmp_ti81xx_vpssm3.xem3 1
```

Note: Procedure for running **heapbufmpapp_release** sample application is same as above.

Running the HeapMemMP sample application

HeapMemMP kernel-side sample application

- To invoke the application enter the following commands:
 - Execute the following to load & start all slave processors. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/heapmemmp_ti81xx_dsp.xe674 0
$ procmgrapp_debug 1 /opt/syslink/heapmemmp_ti81xx_videom3.xem3 0
$ procmgrapp_debug 2 /opt/syslink/heapmemmp_vpssm3.xem3 0
```

- Run the heapmemmpapp sample application by inserting its kernel module

```
$ insmod heapmemmpapp.ko
```

- For clean up:


```
$ rmmod heapmemmpapp
```

- Execute the following to stop all slave processors.

```
$ procmgrapp_debug 0 /opt/syslink/heapmemmp_ti81xx_dsp.xe674 1
```

```
$ procmgrapp_debug 1 /opt/syslink/heapmemmp_ti81xx_videom3.xem3 1
```

```
$ procmgrapp_debug 2 /opt/syslink/heapmemmp_vpssm3.xem3 1
```

HeapMemMP user-side sample application

- To invoke the application standalone, enter the following commands:

```
$ ./heapmemmpapp_debug 3 0 /opt/syslink/heapmemmp_ti81xx_dsp.xe674  
1 /opt/syslink/heapmemmp_ti81xx_videom3.xem3 2  
/opt/syslink/heapmemmp_vpssm3.xem3
```

- Command above should be entered in single line

Alternatively:

- To invoke the application with procmgrapp, enter the following commands:

- Execute the following to load & start all slave processors. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/heapmemmp_ti81xx_dsp.xe674 0
```

```
$ procmgrapp_debug 1 /opt/syslink/heapmemmp_ti81xx_videom3.xem3 0
```

```
$ procmgrapp_debug 2 /opt/syslink/heapmemmp_vpssm3.xem3 0
```

- Run the heapmemmpapp sample application

```
$ ./heapmemmpapp_debug
```

- Execute the following to stop all slave processors.

```
$ procmgrapp_debug 0 /opt/syslink/heapmemmp_ti81xx_dsp.xe674 1
```

```
$ procmgrapp_debug 1 /opt/syslink/heapmemmp_ti81xx_videom3.xem3 1
```

```
$ procmgrapp_debug 2 /opt/syslink/heapmemmp_vpssm3.xem3 1
```

Note: Procedure for running **heapmemmpapp_release** sample application is same as above.

Running the FrameQ sample application

FrameQ kernel-side sample application

- To invoke the application enter the following commands:
 - Execute the following to load & start all slave processors. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/frameq_ti81xx_dsp.xe674 0
```

```
$ procmgrapp_debug 1 /opt/syslink/frameq_ti81xx_videom3.xem3 0
```

```
$ procmgrapp_debug 2 /opt/syslink/frameq_ti81xx_vpssm3.xem3 0
```

- Run the frameqapp sample application by inserting its kernel module

```
$ insmod frameqapp.ko
```

- For clean up:

```
$ rmmod frameqapp
```

- Execute the following to stop all slave processors.

```
$ procmgrapp_debug 0 /opt/syslink/frameq_ti81xx_dsp.xe674 1
```

```
$ procmgrapp_debug 1 /opt/syslink/frameq_ti81xx_videom3.xem3 1
```

```
$ procmgrapp_debug 2 /opt/syslink/frameq_ti81xx_vpssm3.xem3 1
```

FrameQ user-side sample application

- To invoke the application standalone, enter the following commands:

```
$ ./frameqapp_debug 3 0 /opt/syslink/frameq_ti81xx_dsp.xe674  
1 /opt/syslink/frameq_ti81xx_videom3.xem3 2  
/opt/syslink/frameq_ti81xx_vpssm3.xem3
```

- Command above should be entered in single line

Alternatively:

- To invoke the application with procmgrapp, enter the following commands:
 - Execute the following to load & start all slave processors. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/frameq_ti81xx_dsp.xe674 0
```

```
$ procmgrapp_debug 1 /opt/syslink/frameq_ti81xx_videom3.xem3 0
```

```
$ procmgrapp_debug 2 /opt/syslink/frameq_ti81xx_vpssm3.xem3 0
```

- Run the frameqapp sample application

```
$ ./frameqapp_debug
```

- Execute the following to stop all slave processors.

```
$ procmgrapp_debug 0 /opt/syslink/frameq_ti81xx_dsp.xe674 1
```

```
$ procmgrapp_debug 1 /opt/syslink/frameq_ti81xx_videom3.xem3 1
```



```
$ procmgrapp_debug 2 /opt/syslink/frameq_ti81xx_vpssm3.xem3 1
```

Note: Procedure for running **frameqapp_release** sample application is same as above.

Running the RingIO sample application

RingIO kernel-side sample application

- To invoke the application enter the following commands:
 - Execute the following to load & start all slave processors. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/ringio_ti81xx_dsp.xe674 0
```

```
$ procmgrapp_debug 1 /opt/syslink/ringio_ti81xx_videom3.xem3 0
```

```
$ procmgrapp_debug 2 /opt/syslink/ringio_ti81xx_vpssm3.xem3 0
```

- Run the ringioapp sample application by inserting its kernel module

```
$ insmod ringioapp.ko
```

- For clean up:

```
$ rmmod ringioapp
```

- Execute the following to stop all slave processors.

```
$ procmgrapp_debug 0 /opt/syslink/ringio_ti81xx_dsp.xe674 1
```

```
$ procmgrapp_debug 1 /opt/syslink/ringio_ti81xx_videom3.xem3 1
```

```
$ procmgrapp_debug 2 /opt/syslink/ringio_ti81xx_vpssm3.xem3 1
```

RingIO user-side sample application

- To invoke the application standalone, enter the following commands:

```
$ ./ringioapp_debug 3 0 /opt/syslink/ringio_ti81xx_dsp.xe674  
1 /opt/syslink/ringio_ti81xx_videom3.xem3 2  
/opt/syslink/ringio_ti81xx_vpssm3.xem3
```

- Command above should be entered in single line

Alternatively:

- To invoke the application with procmgrapp, enter the following commands:
 - Execute the following to load & start all slave processors. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/ringio_ti81xx_dsp.xe674 0
```

```
$ procmgrapp_debug 1 /opt/syslink/ringio_ti81xx_videom3.xem3 0
```

```
$ procmgrapp_debug 2 /opt/syslink/ringio_ti81xx_vpssm3.xem3 0
```

- Run the ringioapp sample application

```
$ ./ringioapp_debug
```


- For clean up:
 - Execute the following to stop all slave processors.

```
$ procmgrapp_debug 0 /opt/syslink/ringio_ti81xx_dsp.xe674 1
```

```
$ procmgrapp_debug 1 /opt/syslink/ringio_ti81xx_videom3.xem3 1
```

```
$ procmgrapp_debug 2 /opt/syslink/ringio_ti81xx_vpssm3.xem3 1
```

Note: Procedure for running **ringioapp_release** sample application is same as above.

Running the RingIO GPP sample application

RingIO GPP kernel-side sample application

- To invoke the application enter the following commands:
 - Execute the following to load & start all slave processors. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/ringiogpp_ti81xx_dsp.xe674 0
```

```
$ procmgrapp_debug 1 /opt/syslink/ringiogpp_ti81xx_videom3.xem3 0
```

```
$ procmgrapp_debug 2 /opt/syslink/ringiogpp_ti81xx_vpssm3.xem3 0
```

- Run the ringioapp sample application by inserting its kernel module

```
$ insmod ringiogppapp.ko
```

- For clean up:

```
$ rmmod ringiogppapp
```

- Execute the following to stop all slave processors.

```
$ procmgrapp_debug 0 /opt/syslink/ringiogpp_ti81xx_dsp.xe674 1
```

```
$ procmgrapp_debug 1 /opt/syslink/ringiogpp_ti81xx_videom3.xem3 1
```

```
$ procmgrapp_debug 2 /opt/syslink/ringiogpp_ti81xx_vpssm3.xem3 1
```

RingIO GPP user-side sample application

- To invoke the application standalone, enter the following commands:

```
$ ./ringiogppapp_debug 3 0 /opt/syslink/ringiogpp_ti81xx_dsp.xe674  
1 /opt/syslink/ringiogpp_ti81xx_videom3.xem3 2  
/opt/syslink/ringiogpp_ti81xx_vpssm3.xem3
```

- Command above should be entered in single line

Alternatively:

- To invoke the application with procmgrapp, enter the following commands:
 - Execute the following to load & start all slave processors. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/ringiogpp_ti81xx_dsp.xe674 0
```

```
$ procmgrapp_debug 1 /opt/syslink/ringiogpp_ti81xx_videom3.xem3 0
```



```
$ procgrapp_debug 2 /opt/syslink/ringiogpp_ti81xx_vpssm3.xem3 0
```

- Run the ringiogppapp sample application

```
$ ./ringiogppapp_debug
```

- Execute the following to stop all slave processors.

```
$ procgrapp_debug 0 /opt/syslink/ringiogpp_ti81xx_dsp.xe674 1
```

```
$ procgrapp_debug 1 /opt/syslink/ringiogpp_ti81xx_videom3.xem3 1
```

```
$ procgrapp_debug 2 /opt/syslink/ringiogpp_ti81xx_vpssm3.xem3 1
```

Note: Procedure for running **ringiogppapp_release** sample application is same as above.

Running the SharedRegion sample application

SharedRegion kernel-side sample application

- To invoke the application enter the following commands:
 - Execute the following to load & start all slave processors. You can replace path and executables as per your setup:

```
$ procgrapp_debug 0 /opt/syslink/sharedregion_ti81xx_dsp.xe674 0
```

```
$ procgrapp_debug 1 /opt/syslink/sharedregion_ti81xx_videom3.xem3 0
```

```
$ procgrapp_debug 2 /opt/syslink/sharedregion_ti81xx_vpssm3.xem3 0
```

- Run the sharedregionapp sample application by inserting its kernel module

```
$ insmod sharedregionapp.ko SHAREDMEM=0x8FD00000
```

- For clean up:

```
$ rmmod sharedregionapp
```

- Execute the following to stop all slave processors.

```
$ procgrapp_debug 0 /opt/syslink/sharedregion_ti81xx_dsp.xe674 1
```

```
$ procgrapp_debug 1 /opt/syslink/sharedregion_ti81xx_videom3.xem3 1
```

```
$ procgrapp_debug 2 /opt/syslink/sharedregion_ti81xx_vpssm3.xem3 1
```

Note: If your memory map is different, you may need to specify a different physical address value for SHAREDMEM.

SharedRegion user-side sample application

- To invoke the application standalone, enter the following commands:

```
$ ./sharedregionapp_debug 0x8FD00000 3 0
/opt/syslink/sharedregion_ti81xx_dsp.xe674
1 /opt/syslink/sharedregion_ti81xx_videom3.xem3 2
/opt/syslink/sharedregion_ti81xx_vpssm3.xem3
```

- Command above should be entered in single line

Alternatively:

- To invoke the application with procmgrapp, enter the following commands:
 - Execute the following to load & start all slave processors. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/sharedregion_ti81xx_dsp.xe674 0
```

```
$ procmgrapp_debug 1 /opt/syslink/sharedregion_ti81xx_videom3.xem3 0
```

```
$ procmgrapp_debug 2 /opt/syslink/sharedregion_ti81xx_vpssm3.xem3 0
```

- Run the sharedregionapp sample application

```
$ ./sharedregionapp_debug 0x8FD00000
```

- Execute the following to stop all slave processors.

```
$ procmgrapp_debug 0 /opt/syslink/sharedregion_ti81xx_dsp.xe674 1
```

```
$ procmgrapp_debug 1 /opt/syslink/sharedregion_ti81xx_videom3.xem3 1
```

```
$ procmgrapp_debug 2 /opt/syslink/sharedregion_ti81xx_vpssm3.xem3 1
```

Note: If your memory map is different, you may need to specify a different physical address value as the first parameter.

Note: Procedure for running **sharedregionapp_release** sample application is same as above.

RTOS-only inter-Ducati and Single core sample applications

Following applications are to be executed using CCS. Inter-Ducati applications involve VIDEO-M3 and VPSS-M3 cores on TI81XX.

GEL files required to execute inter-ducati/Single core samples

- For TI816X DDR2,

```
** TI816X_DDR2_A8_Gel [1] -> to be loaded on Cortex-A8
** TI816X_DDR2_Ducati_Gel [2] -> to be loaded on VIDEO-M3/VPSS-M3
```

- For TI816X DDR3,

```
** TI816X_DDR3_667MHz_Gel [3] -> if DDR3 is running at 667 MHz, to be
loaded on to Cortex-A8
```

```
** TI816X_DDR3_796MHz_Gel [4] -> if DDR3 is running at 796 MHz, to be
loaded on to Cortex-A8
```

- For TI814X,

```
** TI814X_A8_Gel [5] -> to be loaded on to Cortex-A8
** TI814X_GEL [6] -> should be kept in the same folder as the above
gel file, this will be auto-loaded by CCS.
```


RingIO RTOS-only inter-Ducati sample application

This example has VIDEO-M3 sending data to VPSS-M3.

- Copy following binaries to the host machine:

```
cd $SYSLINK_ROOT\ti\syslink\samples\rtos\ringIO_ti81xx\
cp
ti_syslink_samples_rtos_platforms_ti81xx_videom3\debug\ringio_ti81xx_videom3.xem3
${HOME}/ti81xx/target/opt/syslink/

cp
ti_syslink_samples_rtos_platforms_ti81xx_vpssm3\debug\ringio_ti81xx_vpssm3.xem3
${HOME}/ti81xx/target/opt/syslink/
```

On TI816X EVM with DDR2 ,

- Launch CCS and choose the target configuration for TI816X
- Connect to Cortex-A8 and load the gel file
- In the "scripts" menu select "NETRA External Memories -> doall()"
- Once initialization successfully completed, In the "scripts" menu select "NETRA CPUS BRINGUP -> Ducati()"
- Connect to VIDEO-M3 and VPSS-M3
- load the gel file on VIDEO-M3 or VPSS-M3 and select "scripts -> UniCacheEnableDisable -> Ducati_Cache_Enable()"

```
Load VIDEO-M3 processor with ringio_ti81xx_videom3.xem3 and run.
Load and run ringio_ti81xx_vpssm3.xem3 on VPSS-M3
```

On TI816X EVM with DDR3,

- Launch CCS and choose the target configuration for TI816X
- Connect to Cortex-A8 and load the gel file
- In the "scripts" menu select "NETRA Omx Init -> OmxInit()"
- Once initialization successfully completed, Connect to VIDEO-M3 and VPSS-M3
- load and run the samples like below:

```
Load VIDEO-M3 processor with ringio_ti81xx_videom3.xem3 and run.
Load and run ringio_ti81xx_vpssm3.xem3 on VPSS-M3
```

On TI814X EVM,

- Launch CCS and choose the target configuration for TI814X/Centaurus
- Connect to Cortex-A8 and load the gel file, it auto loads the second gel file and runs "OnTargetConnect()" function.
- connect to VIDEO-M3 and VPSS-M3
- load and run the samples like below:

```
Load VIDEO-M3 processor with ringio_ti81xx_videom3.xem3 and run.
Load and run ringio_ti81xx_vpssm3.xem3 on VPSS-M3
```

- Config parameters through Program.global are provided to choose IPC between processors. The following config values are provided:

- On DSP Processor:

```
ipcWithVIDEOM3 - not validated
ipcWithVPSSM3  - not validated
```


- On Video Processor:

```
ipcWithDSP      - not validated
ipcWithVPSSM3   - supported
```

- On VPSS Processor:

```
ipcWithDSP      - not validated
ipcWithVIDEOM3  - supported
```

- For example:

- On VIDEO-M3, if you selected both ipcWithDSP and ipcWithVPSSM3 you must run executable on DSP with ipcWithVIDEOM3 enabled and on VPSS with ipcWithVIDEOM3 enabled.

RingIO RTOS-only single-core sample application

The example application runs on single core. It supports application running on VIDEO-M3, VPSS-M3 and DSP on TI816X platform. It demonstrates RingIO writer and RingIO reader exchanging buffers on single core.

- Copy binaries to the host machine:

```
$cd $SYSLINK_ROOT/ti/syslink/samples/rtos/singlecore/ringio/
$cp
ti_syslink_samples_rtos_platforms_ti81xx_videom3/whole_program_debug/ringio_ti81xx_videom3.xem3
${HOME}/ti81xx/target/opt/syslink/
$cp
ti_syslink_samples_rtos_platforms_ti81xx_vpssm3/whole_program_debug/ringio_ti81xx_vpssm3.xem3
${HOME}/ti81xx/target/opt/syslink/
$cp
ti_syslink_samples_rtos_platforms_ti81xx_dsp/whole_program_debug/ringio_ti81xx_dsp.xe674
${HOME}/ti81xx/target/opt/syslink/
```

On TI816X EVM with DDR2 ,

- Launch CCS and choose the target configuration for TI816X
- Connect to Cortex-A8 and load the gel file
- In the "scripts" menu select "NETRA External Memories -> doall()"
- Once initialization successfully completed, In the "scripts" menu select "NETRA CPUS BRINGUP -> Ducati()"
- or "NETRA CPUS BRINGUP -> C674X" for DSP
- Connect to VIDEO-M3 or VPSS-M3 or DSP

```
single core VIDEO :Load only VIDEO-M3 processor with
ringio_ti81xx_videom3.xem3 and run.
Singlecore VPSS: Load only VPSS-M3 with ringio_ti81xx_vpssm3.xem3 and
run
Singlecore DSP: Load only DSP with ringio_ti81xx_dsp.xe674 and run
```

On TI816X EVM with DDR3,

- Launch CCS and choose the target configuration for TI816X
- Connect to Cortex-A8 and load the gel file
- In the "scripts" menu select "NETRA Omx Init -> OmxInit()"
- Once initialization successfully completed, Connect to VIDEO-M3 or VPSS-M3 or DSP
- load and run the samples like below:


```
single core VIDEO :Load only VIDEO-M3 processor with
ringio_ti81xx_videom3.xem3 and run.
Singlecore VPSS: Load only VPSS-M3 with ringio_ti81xx_vpssm3.xem3 and
run
Singlecore DSP: Load only DSP with ringio_ti81xx_dsp.xe674 and run
```

On TI814X EVM,

- Launch CCS and choose the target configuration for TI814X/Centaurus
- Connect to Cortex-A8 and load the gel file, it auto loads the second gel file and runs "OnTaragetConnect()" function.
- connect to VIDEO-M3 or VPSS-M3 or DSP
- load and run the samples like below:

```
single core VIDEO :Load only VIDEO-M3 processor with
ringio_ti81xx_videom3.xem3 and run.
Singlecore VPSS: Load only VPSS-M3 with ringio_ti81xx_vpssm3.xem3 and
run
Singlecore DSP: Load only DSP with ringio_ti81xx_dsp.xe674 and run
```

- Note:

For TI814X all four cores can only be connected using xds510 JTAG for which you need emulation driver update on CCS installation. Once you have connectivity with each processor you can use following gel files and run single core applications

TI814X_20Mhz_Si.gel ^[5] evmTI814X.gel ^[6]

Please Note DSP of TI814X can not be connected using xds560v2. It is a known problem.

FrameQ RTOS-only inter-Ducati sample application

The example has VIDEO-M3 sending frames to VPSS-M3. This sample supports shared memory buffers or tiler memory buffers depending on configuration.

- Copy binaries to the host machine:

```
$cd $SYSLINK_ROOT/ti/syslink/samples/rtos/frameq_ti81xx/
$cp
ti_sdo_syslink_samples_rtos_platforms_ti81xx_videom3/whole_program_debug
/frameq_ti81xx_videom3.xem3
${HOME}/ti81xx/target/opt/syslink/
$cp
ti_sdo_syslink_samples_rtos_platforms_ti81xx_vpssm3/whole_program_debug/frameq_ti81xx_vps
${HOME}/ti81xx/target/opt/syslink/
```

On TI816X EVM with DDR2 ,

- Launch CCS and choose the target configuration for TI816X
- Connect to Cortex-A8 and load the gel file
- In the "scripts" menu select "NETRA External Memories -> doall()"
- Once initialization successfully completed, In the "scripts" menu select "NETRA CPUS BRINGUP -> Ducati()"
- Connect to VIDEO-M3 and VPSS-M3
- load the gel file on VIDEO-M3 or VPSS-M3 and select "scripts -> UniCacheEnableDisable -> Ducati_Cache_Enable()"


```
Load VIDEO-M3 processor with frameq_ti81xx_videom3.xem3 and run.
Load and run frameq_ti81xx_vpssm3.xem3 on VPSS-M3
```

On TI816X EVM with DDR3,

- Launch CCS and choose the target configuration for TI816X
- Connect to Cortex-A8 and load the gel file
- In the "scripts" menu select "NETRA Omx Init -> OmxInit()"
- Once initialization successfully completed, Connect to VIDEO-M3 and VPSS-M3
- load and run the samples like below:

```
Load VIDEO-M3 processor with frameq_ti81xx_videom3.xem3 and run.
Load and run frameq_ti81xx_vpssm3.xem3 on VPSS-M3
```

On TI814X EVM,

- Launch CCS and choose the target configuration for TI814X/Centaurus
- Connect to Cortex-A8 and load the gel file, it auto loads the second gel file and runs "OnTaragetConnect()" function.
- connect to VIDEO-M3 and VPSS-M3
- load and run the samples like below:

```
Load VIDEO-M3 processor with frameq_ti81xx_videom3.xem3 and run.
Load and run frameq_ti81xx_vpssm3.xem3 on VPSS-M3
```

- Config parameters through Program.global are provided to choose IPC between processors. The following config values are provided:

- On DSP Processor:

```
ipcWithVIDEOM3    - not validated
ipcWithVPSSM3     - not validated
```

- On Video Processor:

```
ipcWithDSP        - not vaildated
ipcWithVPSSM3     - supported
ipcWithHLOD       - supported
```

- On VPSS Processor:

```
ipcWithDSP        - not validated
ipcWithVIDEOM3    - supported
ipcWithHLOD       - supported
```

- For example:
 - On VIDEO-M3, if you selected both ipcWithDSP and ipcWithVPSSM3 you must run executable on DSP with ipcWithVIDEOM3 enabled and on VPSS with ipcWithVIDEOM3 enabled.

Running FrameQ samples with tiled memory

For build procedure refer to Building FrameQ tiler sample application ^[7]

It is important to rename the tilerDaemon as .exe.

- Load HOST ARM with uImage and uboot
- Insert the syslink driver

```
$ insmod syslink.ko
```

- Select VIDEO-M3 and press Run
- Load the VIDEO-M3 using procmgrapp (user side procmgr application)

```
$ procmgrapp 1 frameq_ti81xx_videom3.xem3 0
```

- Suspend HOST ARM and VIDEO-M3 through ccs and load the symbols on VIDEO-M3 (This is necessary to get prints on CIO)
- Now select HOST ARM and VPSS-M3 and press Run
- Load VPSS-M3 using procmgrapp

```
$ procmgrapp 2 frameq_ti81xx_vpssm3.xem3 0
```

- Suspend HOST ARM and VPSS-M3 through ccs and load the symbols on VPSS-M3 (This is necessary to get prints on CIO)
- Run Host ARM and run Tiler daemon

```
$ ./tiler_daemon.exe
```

- Now run VIDEO-M3 and VPSS-M3 and press enter on console (There are getchars in tiler daemon)
- Pressing enter again will shut down the tiler Daemon
- For clean up, execute the procmgrapp to shutdown the slaves:

```
$ procmgrapp 1 frameq_ti81xx_videom3.xem3 1
```

```
$ procmgrapp 2 frameq_ti81xx_vpssm3.xem3 1
```

- Remove the syslink kernel module

```
$ rmmod syslink
```

FrameQ multiQ RTOS-only inter-Ducati sample application

The example supports VIDEO-M3 sending frames to VPSS. It demonstrates FrameQ reader with multi queues to receive frames. FrameQ writer sends frames to these queues using "v" variant APIs. Reader on VPSS receives frames using getv API.

It also demonstrates FrameQBufMgr with two free Frame Pools (free frame queues).

Type below commands in a single line \$cp <src> <dst>

- Copy binaries to the host machine:

```
$cd $SYSLINK_ROOT/ti/syslink/samples/rtos/frameq_multiQ_ti81xx/
$cp
ti_sdo_syslink_samples_rtos_platforms_ti81xx_videom3/whole_program_debug/
frameq_ti81xx_videom3.xem3
${HOME}/ti81xx/target/opt/syslink/
$cp ti_sdo_syslink_samples_rtos_platforms_ti81xx_vpssm3/
whole_program_debug/
```



```
frameq_ti81xx_vpssm3.xem3
${HOME}/ti81xx/target/opt/syslink/
```

On TI816X EVM with DDR2 ,

- Launch CCS and choose the target configuration for TI816X
- Connect to Cortex-A8 and load the gel file
- In the "scripts" menu select "NETRA External Memories -> doall()"
- Once initialization successfully completed, In the "scripts" menu select "NETRA CPUS BRINGUP -> Ducati()"
- Connect to VIDEO-M3 and VPSS-M3
- load the gel file on VIDEO-M3 or VPSS-M3 and select "scripts -> UniCacheEnableDisable -> Ducati_Cache_Enable()"

```
Load VIDEO-M3 processor with frameq_ti81xx_videom3.xem3 and run.
Load and run frameq_ti81xx_vpssm3.xem3 on VPSS-M3
```

On TI816X EVM with DDR3,

- Launch CCS and choose the target configuration for TI816X
- Connect to Cortex-A8 and load the gel file
- In the "scripts" menu select "NETRA Omx Init -> OmxInit()"
- Once initialization successfully completed, Connect to VIDEO-M3 and VPSS-M3
- load and run the samples like below:

```
Load VIDEO-M3 processor with frameq_ti81xx_videom3.xem3 and run.
Load and run frameq_ti81xx_vpssm3.xem3 on VPSS-M3
```

On TI814X EVM,

- Launch CCS and choose the target configuration for TI814X/Centaurus
- Connect to Cortex-A8 and load the gel file, it auto loads the second gel file and runs "OnTaragetConnect()" function.
- connect to VIDEO-M3 and VPSS-M3
- load and run the samples like below:

```
Load VIDEO-M3 processor with frameq_ti81xx_videom3.xem3 and run.
Load and run frameq_ti81xx_vpssm3.xem3 on VPSS-M3
```

- Config parameters through Program.global are provided to choose IPC between processors. The following config values are provided:

- On VIDEO-M3 Processor:

```
ipcWithDSP      - not validated
ipcWithVPSSM3   - supported
```

- On VPSS-M3 Processor:

```
ipcWithDSP      - not validated
ipcWithVIDEOM3  - supported
```

The Host processor is not included in this sample application.

FrameQ RTOS-only single-core sample application

The example application runs on single core. It supports application running on VIDEO-M3, VPSS-M3 and DSP on TI81XX platform. It demonstrates FrameQ writer writer and FrameQ reader exchanging frames on single core.

It also demonstrates FrameQBufMgr with two free Frame Pools (free frame queues).

- Copy binaries to the host machine:

```
$cd $SYSLINK_ROOT/ti/syslink/samples/rtos/singlecore/frameq/
$cp
ti_syslink_samples_rtos_platforms_ti81xx_videom3/whole_program_debug/
frameq_ti81xx_videom3.xem3
${HOME}/ti81xx/target/opt/syslink/
$cp
ti_syslink_samples_rtos_platforms_ti81xx_vpssm3/whole_program_debug/
frameq_ti81xx_vpssm3.xem3
${HOME}/ti81xx/target/opt/syslink/
$cp ti_syslink_samples_rtos_platforms_ti81xx_dsp/whole_program_debug/
frameq_ti81xx_dsp.xe674
${HOME}/ti81xx/target/opt/syslink/
```

On TI816X EVM with DDR2 ,

- Launch CCS and choose the target configuration for TI816X
- Connect to Cortex-A8 and load the gel file
- In the "scripts" menu select "NETRA External Memories -> doall()"
- Once initialization successfully completed, In the "scripts" menu select "NETRA CPUS BRINGUP -> Ducati()"
- or "NETRA CPUS BRINGUP -> C674X" for DSP
- Connect to VIDEO-M3 or VPSS-M3 or DSP

```
single core VIDEO :Load only VIDEO-M3 processor with
frameq_ti81xx_videom3.xem3 and run.
Singlecore VPSS: Load only VPSS-M3 with frameq_ti81xx_vpssm3.xem3 and
run
Singlecore DSP: Load only DSP with frameq_ti81xx_dsp.xe674 and run
```

On TI816X EVM with DDR3,

- Launch CCS and choose the target configuration for TI816X
- Connect to Cortex-A8 and load the gel file
- In the "scripts" menu select "NETRA Omx Init -> OmxInit()"
- Once initialization successfully completed, Connect to VIDEO-M3 or VPSS-M3 or DSP
- load and run the samples like below:

```
single core VIDEO :Load only VIDEO-M3 processor with
frameq_ti81xx_videom3.xem3 and run.
Singlecore VPSS: Load only VPSS-M3 with frameq_ti81xx_vpssm3.xem3 and
run
Singlecore DSP: Load only VPSS-M3 with frameq_ti81xx_dsp.xe674 and run
```

On TI814X EVM,

- Launch CCS and choose the target configuration for TI814X/Centaurus

- Connect to Cortex-A8 and load the gel file, it auto loads the second gel file and runs "OnTargetConnect()" function.
- connect to VIDEO-M3 or VPSS-M3 or DSP
- load and run the samples like below:

```
single core VIDEO :Load only VIDEO-M3 processor with
frameq_ti81xx_videom3.xem3 and run.
Singlecore VPSS: Load only VPSS-M3 with frameq_ti81xx_vpssm3.xem3 and
run
Singlecore DSP: Load only VPSS-M3 with frameq_ti81xx_dsp.xe674 and run
```

- Note:

For TI814X all four cores can only be connected using xds510 JTAG for which you need emulation driver update on CCS installation. Once you have connectivity with each processor you can use following gel files and run single core applications

TI814X_20Mhz_Si.gel ^[5] evmTI814X.gel ^[6]

Please Note DSP of TI814X can not be connected using xds560v2. It is a known problem.

- More details on running on TI814X,

Steps to run four cores with A8+bios on TI814x

- Dsp of TI814X can be only connected by XDS510USB. The issue with this JTAG is if you use this you won't be able to load VIDEO-M3 or VPSS-M3 through CCS. The error that you get is "Core locked up in NMI". In order to overcome this you must follow these steps and install additional emulation drivers. After this you will be able to load and run all the 4 cores on TI814X
- Step 1 : Update the emulation driver.

```
Go to Help -> Software Updates -> Find and Install
```

- Step 2 : Select "Search for new features to install" and click Next

```
Click the New Remote Site button.
```

- Step 3 : For the name enter "Spectrum test site" and for the url enter

```
http://support.spectrumdigital.com/ccs40/PrivateUpdates/
```

- Step 4 : Click finish to have it find the new package on this site.

```
When it finds the update check the box beside it to install it and
click next.
```

- Step 5 : Agree to the license, click next
- Step 6 : Click finish

Steps to run frameQ sample on all four cores with A8+bios on TI814x

- Got to directory where executables are generated

```
cd $SYSLINK_ROOT/ti/syslink/samples/rtos/ti81xx/frameq_ti814x/
```

- Copy binaries following to the host machine:

```
$cp ti_syslink_samples_rtos_platforms_ti814x_dsp/whole_program_debug/
frameq_ti814x_dsp.xe674
${HOME}/ti81xx/target/opt/syslink/
$cp
ti_syslink_samples_rtos_platforms_ti814x_videom3/whole_program_debug/
frameq_ti814x_videom3.xem3
${HOME}/ti81xx/target/opt/syslink/
$cp
ti_syslink_samples_rtos_platforms_ti814x_vpssm3/whole_program_debug/
frameq_ti814x_vpssm3.xem3
${HOME}/ti81xx/target/opt/syslink/
$cp ti_syslink_samples_rtos_platforms_ti814x_arm/whole_program_debug/
frameq_ti814x_arm.xea8f
${HOME}/ti81xx/target/opt/syslink/
```

- Launch TI814X target configuration in CCS
- Use following gel files

TI814X_20Mhz_Si.gel ^[5] evmTI814X.gel ^[6]

- Download both the gel files and keep it in same directory
- Select A8 and Load only evmTI814X.gel
- Connect A8. This step will automatically invoke OnConnectTarget() function from gel file which will do all the initialization. Wait till initialization completes
- Now connect VIDEO-M3, VPSS-M3 and Dsp
- Reset A8
- Load A8 using frameq_ti814x_arm.xea8f. It will go in running state.
- Halt A8 and reload with same executable. This is a known issue with A8 + XDS510 combination
- Load VIDEO-M3, VPSS-M3 and Dsp with respective executables
- Pin the console for each core so that you get print for each core.
- Run all the cores together.

Steps to run ringIO sample on all four cores with A8+bios on TI814x

- Got to directory where executables are generated

```
$cd $SYSLINK_ROOT/ti/syslink/samples/rtos/ti81xx/ringio_ti814x/
```

- Copy binaries following to the host machine:

```
$cp ti_syslink_samples_rtos_platforms_ti814x_dsp/whole_program_debug/
ringio_ti814x_dsp.xe674
${HOME}/ti81xx/target/opt/syslink/
$cp
ti_syslink_samples_rtos_platforms_ti814x_videom3/whole_program_debug/
ringio_ti814x_videom3.xem3
```



```

${HOME}/ti81xx/target/opt/syslink/
$cp
ti_syslink_samples_rtos_platforms_ti814x_vpssm3/whole_program_debug/ringio_ti814x_vpssm3.
${HOME}/ti81xx/target/opt/syslink/
$cp
ti_syslink_samples_rtos_platforms_ti814x_arm/whole_program_debug/ringio_ti814x_arm.xea8f
${HOME}/ti81xx/target/opt/syslink/

```

- Launch TI814X target configuration in CCS
- Use following gel files

TI814X_20Mhz_Si.gel^[5] evmTI814X.gel^[6]

- Download both the gel files and keep it in same directory
- Select A8 and Load only evmTI814X.gel
- Connect A8. This step will automatically invoke OnConnectTarget() function from gel file which will do all the initialization. Wait till initialization completes
- Now connect VIDEO-M3, VPSS-M3 and Dsp
- Reset A8
- Load A8 using ringio_ti814x_arm.xea8f. It will go in running state.
- Halt A8 and reload with same executable. This is a known issue with A8 + XDS510 combination
- Load VIDEO-M3, VPSS-M3 and Dsp with respective executables
- Pin the console for each core so that you get print for each core.
- Run all the cores together.

References

- [1] http://www.india.ti.com/~pspcm/syslink/internalDocs/misc/netra_vdb_ddr2.gel
- [2] http://www.india.ti.com/~pspcm/syslink/internalDocs/misc/netra_sw_ducati.gel
- [3] http://www.india.ti.com/~pspcm/syslink/internalDocs/misc/syslink_Netra_MicronDDR666MHz.gel
- [4] http://www.india.ti.com/~pspcm/syslink/internalDocs/misc/syslink_Netra_ddr3_796MHz.gel
- [5] http://www.india.ti.com/~pspcm/syslink/internalDocs/misc/Centaurus_20Mhz_Si.gel
- [6] <http://www.india.ti.com/~pspcm/syslink/internalDocs/misc/evmDM8148.gel>
- [7] http://ap-fpdsp-swapps.dal.design.ti.com/index.php/SysLink_02.00.00.68_beta1_InstallGuide_Linux_TI81XX#FrameQ-Tiler_sample_application

SysLink 02.00.00.68 beta1 InstallGuide Linux OMAP3530

SYS/Link 02.00.00.68_beta1 Linux OMAP3530

SysLink Install Guide for OMAP3530 running Linux on the ARM Cortex A8

Introduction

This document gives information about SysLink installation for using SysLink with the OMAP3530 platform, where the ARM Cortex A8 is running Linux OS. It also gives detailed build instructions for SysLink as well as the sample applications. In addition, instructions are also provided for running the sample applications provided within the SysLink release.

Dependencies

The dependencies, along with their versions are given in the Release Notes.

Installation

Setting up Linux Workstation

The description in this section is based on the following assumptions:

- The workstation is running on a Linux workstation
- Services telnetd, nfsd, ftpd are configured on this workstation.
- The workstation is assigned a fixed IP address.

A fixed IP address is preferred, as the IP address needs to be specified while booting the target board. This allows the boot loader configuration to be saved in flash.

Enable TFTP for downloading the kernel image to target

U-boot can be configured to download the kernel onto the target by various mechanisms:

- TFTP
- Serial Port

This section configures the Linux development host as a TFTP server. Modify the 'xinet.d/tftp' file to enable TFTP:

- Make the following changes:

```
disable      = no
```

```
server_args = -s /tftpboot
```

- Restart the network service

```
$ /etc/init.d/xinetd restart
```

The above configuration assumes that a directory 'tftpboot' has been created at the root '/' directory. The files in this directory are exposed through the TFTP protocol.

Building the Linux kernel

This section assumes that Linux release is installed on the development workstation at /toolchains/git.

Tool chain

Untar the arm-2009q1-203 tool chain and add the tool chain directory in your path.

```
$ tar -xjvf
arm-2009q1-203-arm-none-linux-gnueabi-i686-pc-linux-gnu.tar.bz2.tar
$ export PATH=$PATH:/toolchains/git/arm-2009q1-203/bin
```

Kernel

Untar the kernel that comes with the Linux release. Build the kernel according to the PSP Linux User Guide document. Refer to the PSP u-boot related documentation on how to build the mkimage utility which is needed for creation of uImage. Along with the tool chain, location of mkimage utility should also be there in the path.

```
$ tar -zxvf AM35x-OMAP35x-PSP-SDK-03.00.00.03.tgz
$ cd AM35x-OMAP35x-PSP-SDK-03.00.00.03/src/kernel
$ tar -zxvf linux-03.00.00.03.tar.gz
```

Patching the kernel

Download a kernel patch required (see Dependencies) to enable iommu for iva2 subsystem and apply it on the kernel as shown below:

```
#Copy the patch to kernel source directory
$ cp enable-iommu-for-iva2.patch
AM35x-OMAP35x-PSP-SDK-03.00.00.03/src/kernel/linux-03.00.00.03
$ cd AM35x-OMAP35x-PSP-SDK-03.00.00.03/src/kernel/linux-03.00.00.03
$ patch -p0 < enable-iommu-for-iva2.patch
```

Ensure that this patch is applied before creating uImage. Since syslink uses iommu internally, it is mandatory to have this patch applied.

U-Boot boot-loader

Please refer the Linux PSP user guide to load the u.boot and x-loader to the target.

Create target file system

The target device needs a file system to boot from. The file system can be exported to the target through NFS.

Exporting target file system through NFS

- A directory on the development host can be setup and exported for this purpose.
- AM35x-OMAP35x-PSP-SDK-03.00.00.03/images/fs/omap3530 contains the NFS file system nfs.tar.gz.

```
$ tar -xzvf nfs.tar.gz
```

You need to be 'root' to successfully execute this command.

The file system can be copied to a different location. In such a case ~/omap3530/target can be a soft link to the actual location.

- libpthread libraries required for SysLink may not be available by default within the target file system. Copy this from the tool-chain.


```
$ cp
/toolchains/git/arm-2009q1-203/arm-none-linux-gnueabi/libc/lib/libpthread*
~/omap3530/target/lib
```

This step is not required if libpthread libraries are available by default in the target file system.

- The directory ~/omap3530/target will be mounted as root directory on the target through NFS.

To do so, add the following line to the file /etc/exports.

```
/home/<user>/omap3530/target *(rw,no_root_squash)
```

Replace "/home/<user>" in the path above with the actual path of your home directory on the development workstation.

Code Composer Studio

CCS v5.0 (supports CCS 4.2.07000 or later) can be used. To use CCS for debugging the DSP side application, CCS needs to be configured to use both ARM and DSP with OMAP3530.

CCS can attach to only ARM in the beginning. It can attach to the DSP only after the ARM-side application releases DSP from reset.

Build

Editing files for Linux headers and tool chain paths

- Set SYSLINK_ROOT environment variable. This should be set to the base of the SysLink repository, i.e. the folder containing the ti directory.

```
$ export SYSLINK_ROOT=~/.syslink_<VERSION>
```

- set SYSLINK_PKGPATH environment variable for IPC package path.

```
export
SYSLINK_PKGPATH="/db/psp_git/syslink_toolchains/IPC/ipc_<version>/packages;
```

Replace ipc_<version> above with the actual version of the IPC.

- Update KDIR variable in the base Makefile with the Linux source path in the following files. Alternatively, it can be overridden by passing to the make command.

```
$(SYSLINK_ROOT)/ti/syslink/buildutils/hlos/knl/Makefile.inc
KDIR :=
/toolchains/omap3530/AM35x-OMAP35x-PSP-SDK-03.00.00.03/src/kernel/linux-03.00.00.03
```

Actual path in your build environment must be given for the KDIR path.

- Update TOOLCHAIN_PREFIX variable in the base Makefile with the Linux tool chain in the following files. Alternatively, it can be overridden by passing to the make command.

```
$(SYSLINK_ROOT)/ti/syslink/buildutils/hlos/usr/Makefile.inc
TOOLCHAIN_PREFIX :=
/toolchains/git/arm-2009q1-203/bin/arm-none-linux-gnueabi-
```

- Alternatively, the paths can be passed to the make command as a ';' separated set of paths. For example:

```
$ make ARCH=arm
CROSS_COMPILE=/toolchains/git/arm-2009q1-203/bin/arm-none-linux-gnueabi-
```



```
SYSLINK_PKGPATH="/toolchains/IPC/ipc_<version>/packages;$HOME/syslink"
```

Type the above command in a single line Replace ipc_<version> above with the actual version of the IPC.

- Update CGTOOLS tool chain path in the following files:

```
$(SYSLINK_ROOT)/config.bld
```

For example:

```
var rootDirPre = "/toolchains/ti-tools/";
var rootDirPost = "";
C64P.rootDir = rootDirPre + "c6000_7.2.0A10232" + rootDirPost;
```

- If build is required for only a specific configuration, comment the other configurations in Build.targets. e.g. set configuration to build only for OMAP3530 device.

```
//list interested targets in Build.targets array
Build.targets = [
    //C28_large,
    //C64,
    C64P_COFF,
    C64P_ELF,
    //C67P,
    //C674_COFF,
    //C674_ELF,
    //Arm9,
    //M3_ELF,
    //Win32,
    //A8_ELF
];
```

Building SysLink HLOS driver/library and sample applications

The default configuration includes Ipc in the build for the drivers and the sample applications.

Switch to bash shell to build HLOS side drivers and samples. The default configuration builds for TI816X. To build for a different device, the SYSLINK_PLATFORM build parameter needs to be given.

Following are the steps to build the SysLink HLOS driver/library and sample applications in the default configuration:

- Set SYSLINK_ROOT environment variable. This should be set to the base of the SysLink repository, i.e. the folder containing the ti directory.

```
$ export SYSLINK_ROOT=~/.syslink_<VERSION>
```

- Add the toolchain path to PATH variable

```
$ export PATH=$PATH:/toolchains/git/arm-2009q1-203/bin
```

- Build the SysLink kernel module

```
$ cd $SYSLINK_ROOT/ti/syslink/utils/hlos/knl/Linux
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
SYSLINK_PLATFORM=OMAP3530
```


- Build SysLink kernel samples. Run make command in the respective sample directories.

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/notify/knl/Linux
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
SYSLINK_PLATFORM=OMAP3530

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/messageQ/knl/Linux
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
SYSLINK_PLATFORM=OMAP3530

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/heapBufMP/knl/Linux
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
SYSLINK_PLATFORM=OMAP3530

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/heapMemMP/knl/Linux
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
SYSLINK_PLATFORM=OMAP3530

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/frameq/knl/Linux
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
SYSLINK_PLATFORM=OMAP3530

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/listMP/knl/Linux
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
SYSLINK_PLATFORM=OMAP3530

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/ringIO/knl/Linux
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
SYSLINK_PLATFORM=OMAP3530

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/ringIO_gpp/knl/Linux
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
SYSLINK_PLATFORM=OMAP3530

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/sharedRegion/knl/Linux
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
SYSLINK_PLATFORM=OMAP3530

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/gateMP/knl/Linux
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
SYSLINK_PLATFORM=OMAP3530
```

- The Build System supports three different profiles on user side for library as well as the sample applications as follows:

```
* debug
* release
* both - debug & release (default)
```


- Build the SysLink user library (**debug & release - both profiles together**)

```
$ cd $SYSLINK_ROOT/ti/syslink/utils/hlos/usr/Linux
$ make
```

- Build the SysLink user library (**debug profile**)

```
$ cd $SYSLINK_ROOT/ti/syslink/utils/hlos/usr/Linux
$ make debug
```

- Build the SysLink user library (**release profile**)

```
$ cd $SYSLINK_ROOT/ti/syslink/utils/hlos/usr/Linux
$ make release
```

Note: For building the library no need to specify platform as the syslink user library is **independent of platform**.

- Build SysLink user samples (**debug & release - both profiles together**). Run make command in the respective sample directories.

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/common/usr/Linux
$ make SYSLINK_PLATFORM=OMAP3530
```

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/procMgr/usr/Linux
$ make SYSLINK_PLATFORM=OMAP3530
```

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/notify/usr/Linux
$ make SYSLINK_PLATFORM=OMAP3530
```

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/messageQ/usr/Linux
$ make SYSLINK_PLATFORM=OMAP3530
```

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/listMP/usr/Linux
$ make SYSLINK_PLATFORM=OMAP3530
```

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/heapBufMP/usr/Linux
$ make SYSLINK_PLATFORM=OMAP3530
```

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/heapMemMP/usr/Linux
$ make SYSLINK_PLATFORM=OMAP3530
```

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/frameq/usr/Linux
$ make SYSLINK_PLATFORM=OMAP3530
```

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/ringIO/usr/Linux
$ make SYSLINK_PLATFORM=OMAP3530
```

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/ringIO_gpp/usr/Linux
$ make SYSLINK_PLATFORM=OMAP3530
```

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/sharedRegion/usr/Linux
$ make SYSLINK_PLATFORM=OMAP3530
```



```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/gateMP/usr/Linux
$ make SYSLINK_PLATFORM=OMAP3530
```

- **Build SysLink user samples (debug profile).** Run make command in the respective sample directories.

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/common/usr/Linux
$ make SYSLINK_PLATFORM=OMAP3530 debug

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/procMgr/usr/Linux
$ make SYSLINK_PLATFORM=OMAP3530 debug

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/notify/usr/Linux
$ make SYSLINK_PLATFORM=OMAP3530 debug

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/messageQ/usr/Linux
$ make SYSLINK_PLATFORM=OMAP3530 debug

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/listMP/usr/Linux
$ make SYSLINK_PLATFORM=OMAP3530 debug

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/heapBufMP/usr/Linux
$ make SYSLINK_PLATFORM=OMAP3530 debug

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/heapMemMP/usr/Linux
$ make SYSLINK_PLATFORM=OMAP3530 debug

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/frameq/usr/Linux
$ make SYSLINK_PLATFORM=OMAP3530 debug

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/ringIO/usr/Linux
$ make SYSLINK_PLATFORM=OMAP3530 debug

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/ringIO_gpp/usr/Linux
$ make SYSLINK_PLATFORM=OMAP3530 debug

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/sharedRegion/usr/Linux
$ make SYSLINK_PLATFORM=OMAP3530 debug

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/gateMP/usr/Linux
$ make SYSLINK_PLATFORM=OMAP3530 debug
```

- **Build SysLink user samples (release profile).** Run make command in the respective sample directories.

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/common/usr/Linux
$ make SYSLINK_PLATFORM=OMAP3530 release

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/procMgr/usr/Linux
$ make SYSLINK_PLATFORM=OMAP3530 release
```



```

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/notify/usr/Linux
$ make SYSLINK_PLATFORM=OMAP3530 release

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/messageQ/usr/Linux
$ make SYSLINK_PLATFORM=OMAP3530 release

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/listMP/usr/Linux
$ make SYSLINK_PLATFORM=OMAP3530 release

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/heapBufMP/usr/Linux
$ make SYSLINK_PLATFORM=OMAP3530 release

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/heapMemMP/usr/Linux
$ make SYSLINK_PLATFORM=OMAP3530 release

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/frameq/usr/Linux
$ make SYSLINK_PLATFORM=OMAP3530 release

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/ringIO/usr/Linux
$ make SYSLINK_PLATFORM=OMAP3530 release

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/ringIO_gpp/usr/Linux
$ make SYSLINK_PLATFORM=OMAP3530 release

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/sharedRegion/usr/Linux
$ make SYSLINK_PLATFORM=OMAP3530 release

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/gateMP/usr/Linux
$ make SYSLINK_PLATFORM=OMAP3530 release

```

Building RTOS SysLink and sample applications

- Set SYSLINK_ROOT environment variable. This should be set to the base of the SysLink repository, i.e. the folder containing the ti directory.

```
$ export SYSLINK_ROOT=~/.syslink_<VERSION>
```

- Set XDC in path.

```
$ export PATH=$PATH:/toolchains/xdc/xdctools_<version>
```

Alternatively, use the full path while making calls to xdc in the commands given later in this section

- Set XDC related environment variables.

```
$ export
XDCPATH="/users/ipc/ipc_<version>/packages;/toolchains/bios6/bios_<version>/packages"
```

Replace <version> above with the specific version for IPC, BIOS & XDC tools

- Following statement needs to be added to all application config(.cfg) files


```
xdc.loadPackage ('ti.syslink.ipc.rtos');
```

this ensures that all required linker options are included by default

User needs to add the above mentioned statement to application config (.cfg) files in order to include all required linker options by default

- Build RTOS SysLink. This builds RTOS FrameQ and RingIO

```
$ cd $SYSLINK_ROOT/ti/syslink/
$ xdc all XDCBUILDCFG="$SYSLINK_ROOT/config.bld" -PR .
```

- Default build profile is "whole_program_debug" to change this to "**debug**" run the following

```
$ xdc all XDCARGS="profile=debug"
XDCBUILDCFG="$SYSLINK_ROOT/config.bld" -PR .
```

alternatively "XDCARGS" can be exported

```
export XDCARGS=profile=debug

$ xdc all XDCBUILDCFG="$SYSLINK_ROOT/config.bld" -PR .
```

- Default build profile is "whole_program_debug" to change this to "**release**" run the following

```
$ xdc all XDCARGS="profile=release"
XDCBUILDCFG="$SYSLINK_ROOT/config.bld" -PR .
```

alternatively "XDCARGS" can be exported

```
export XDCARGS=profile=release

$ xdc all XDCBUILDCFG="$SYSLINK_ROOT/config.bld" -PR .
```

- If you want to exclude Samples and build rest of the libraries, run following command in directory "\$SYSLINK_ROOT/ti"

```
xdc XDCBUILDCFG="$SYSLINK_ROOT/config.bld" -PR 'xdcpkg syslink | grep
-v /samples'
```

The step to build SysLink RTOS side also builds all sample applications. If the syslink/ipc and sample applications are to be built independently, this can be done through separate commands as given below.

- To optimize the build time by utilizing "parallel build feature of GNU make" pass command line option **--jobs** as shown below

```
$ xdc all --jobs=4 XDCBUILDCFG="$SYSLINK_ROOT/config.bld" -PR .
```

--jobs= <number of seperate jobs to be run> //usually 1.5 times no of cpu cores available on the build system

- To build only the sample applications independently, use the following commands:

```
$ cd $SYSLINK_ROOT/ti/syslink/ipc
$ xdc all -PR .

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/platforms
$ xdc all -PR .

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/notify
```



```
$ xdc all

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/messageQ
$ xdc all

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/frameq
$ xdc all

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/heapBufMP
$ xdc all

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/heapMemMP
$ xdc all

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/listMP
$ xdc all

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/ringIO
$ xdc all

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/ringIO_gpp
$ xdc all

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/sharedRegion
$ xdc all

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/gateMP
$ xdc all
```

Build Notes

The default build configuration for SysLink and sample applications assumes TI816X. If no specific `SYSLINK_PLATFORM` value is specified, TI816X is assumed by default.

OMAP3530 supports both COFF & ELF build for the slave executables. The choice of which loader is to be used can be done through a build flag `SYSLINK_LOADER`, which can be passed to the make command for the syslink kernel module. If no flag is specified, the default is COFF. If `SYSLINK_LOADER=ELF` is used, the ELF loader is used.

```
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
SYSLINK_PLATFORM=OMAP3530 SYSLINK_LOADER=ELF
```

For additional information on different build parameters, please refer to the UserGuide document.

Configuring Kernel Parameters

SysLink requires a few specific arguments to be passed to the Linux kernel during boot up. For running the sample applications, 3MB of memory is used by SysLink for communication between GPP and DSP, and for DSP external memory for placing its code/data. This must be reserved by specifying 3MB less as available for the Linux kernel for its usage.

Depending on the memory map used for the final system configuration, the memory to be reserved from Linux may be different.

Running the sample applications

Sample applications are provided with SysLink for the supported platforms. This section describes the way to execute the sample applications.

The steps for execution of the samples are given below for execution with Linux running on the GPP.

For all kernel-side applications, ProcMgrApp is expected to be run before running the specific application to load and start the slave processor.

For all user-side applications, if no command line arguments are provided while running the sample application, the ProcMgrApp application is expected to be run before running the specific application to load and start the slave processor.

If the following command line arguments are provided, then the user-side application also loads/starts/stops the slave processor in the application context itself. In this case, ProcMgrApp application is not required to be run before running the specific application:

```
Arguments: <numProcs> <procId1> <Path including name of
the slave executable>
```

Copying files to target file system

The generated binaries on the HLOS side and RTOS side must be copied to the target directory.

When built for COFF mode, the generated executables have an extension .x64P. ELF mode executables have an extension .xe64P. In the below instructions, copy instructions are given for COFF mode RTOS executables. To copy ELF mode executables, replace the .x64P extensions with .xe64P below.

For executing the sample applications, follow the steps below to copy the relevant binaries:

- Copy the syslink.ko kernel module into the target file system

```
$ cp ${SYSLINK_ROOT}/ti/syslink/bin/OMAP3530/*.ko
${HOME}/omap3530/target/opt/syslink/.
```

- Copy the user and kernel sample application binaries into the target file system

```
$ cp ${SYSLINK_ROOT}/ti/syslink/bin/OMAP3530/samples/*
${HOME}/omap3530/target/opt/syslink/
```

To copy RTOS binaries, use the following steps:

Type the below commands each in a single line :: \$cp <src> <dst>

- Copy the rtos executables into the target file system

```
$ cd ${SYSLINK_ROOT}/ti/syslink/samples/rtos/

$ cp notify/evm3530_dsp/whole_program_debug/notify_omap3530_dsp.x64P
${HOME}/omap3530/target/opt/syslink/
```



```
$ cp
messageQ/evm3530_dsp/whole_program_debug/messageq_omap3530_dsp.x64P
${HOME}/omap3530/target/opt/syslink/

$ cp frameq/evm3530_dsp/whole_program_debug/frameq_omap3530_dsp.x64P
${HOME}/omap3530/target/opt/syslink/

$ cp listMP/evm3530_dsp/whole_program_debug/listmp_omap3530_dsp.x64P
${HOME}/omap3530/target/opt/syslink/

$ cp
heapBufMP/evm3530_dsp/whole_program_debug/heapbufmp_omap3530_dsp.x64P
${HOME}/omap3530/target/opt/syslink/

$ cp
heapMemMP/evm3530_dsp/whole_program_debug/heapmemmp_omap3530_dsp.x64P
${HOME}/omap3530/target/opt/syslink/

$ cp ringIO/evm3530_dsp/whole_program_debug/ringio_omap3530_dsp.x64P
${HOME}/omap3530/target/opt/syslink/

$ cp
ringIO_gpp/evm3530_dsp/whole_program_debug/ringiogpp_omap3530_dsp.x64P
${HOME}/omap3530/target/opt/syslink/

$ cp
sharedRegion/evm3530_dsp/whole_program_debug/sharedregion_omap3530_dsp.x64P
${HOME}/omap3530/target/opt/syslink/

$ cp gateMP/evm3530_dsp/whole_program_debug/gatemp_omap3530_dsp.x64P
${HOME}/omap3530/target/opt/syslink/
```

Type the above command in a single line

Running applications

When built for COFF mode, the generated executables have an extension .x64P. ELF mode executables have an extension .xe64P. In the below instructions, copy instructions are given for COFF mode RTOS executables. To copy ELF mode executables, replace the .x64P extensions with .xe64P below.

Loading syslink kernel module

- The syslink kernel module must be inserted before running any sample applications.

```
$ insmod syslink.ko TRACE=1 TRACEFAILURE=1
```

Enabling TRACEFAILURE puts out prints in case any failure occurs during SysLink initialization.

Unloading syslink kernel module

- The SysLink kernel module can be unloaded when it is no longer required, i.e. after all applications that use SysLink have exited. Unloading the SysLink kernel module reclaims to the HLOS, the resources that might have been consumed by SysLink.
- The SysLink Kernel module can be unloaded using:

```
$ rmmod syslink
```

Running the sample applications using scripts

Scripts have been provided for running the sample applications within the ti/syslink/tools/scripts folder.

Copying the scripts to target file system

Copy the following scripts into the target file system.

- Copy the top-level scripts into the target file system

```
$ cp ${SYSLINK_ROOT}/ti/syslink/tools/scripts/*.sh
${HOME}/omap3530/target/opt/syslink/.
```

- Copy the scripts for each sample application into the target file system

```
$ cp ${SYSLINK_ROOT}/ti/syslink/tools/scripts/omap3530/*.sh
${HOME}/omap3530/target/opt/syslink/.
```

Run the sample applications using the scripts

- To run the applications in debug build:

```
$ ./runsamples_debug.sh
```

- To run the applications in release build:

```
$ ./runsamples_release.sh
```

Running the ProcMgr sample application

ProcMgr user-side sample application

- To invoke the application enter the following commands:
- Execute the following to load, start and stop all slave processors. You can replace path and executables as per your setup:

Note: *procmgrapp_debug* takes two parameters *remote procId* and *executable name*.

```
$ procmgrapp_debug 0 /opt/syslink/messageq_omap3530_dsp.x64P
```

- To use the procmgrapp as a loader for running other sample applications or as a standalone user-space loader to load any slave executable, use a final optional parameter
 - On providing the final parameter as 0, the procmgrapp only powers up, loads & starts the slave core
 - On providing the final parameter as 1, the procmgrapp only stops and powers down the slave core
- Load & start the slave core by invoking the procmgrapp application with final parameter as 0

```
$ procmgrapp_debug 0 /opt/syslink/messageq_omap3530_dsp.x64P 0
```

- Run the sample application by inserting its kernel module. For example, messageqapp

```
$ insmod messageqapp.ko
```


- For clean up:

```
$ rmmod messageqapp
```

- Stop the slave core by invoking the procmgrapp application with final parameter as 1

```
$ procmgrapp_debug 0 /opt/syslink/messageq_omap3530_dsp.x64P 1
```

Note: Procedure for running **procmgrapp_release** sample application is same as above.

Running the Notify sample application

Notify kernel-side sample application

- To invoke the application enter the following commands:
 - Execute the following to load and start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/notify_omap3530_dsp.x64P 0
```

- Run the notifyapp sample application by inserting its kernel module

```
$ insmod notifyapp.ko
```

- For clean up:

```
$ rmmod notifyapp
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/notify_omap3530_dsp.x64P 1
```

Notify user-side sample application

- To invoke the application standalone, enter the following commands:

```
$ ./notifyapp_debug 1 0 ./notify_omap3530_dsp.x64P
```

- Press 'enter' to exit and cleanup.

Alternatively:

- To invoke the application with procmgrapp, enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/notify_omap3530_dsp.x64P 0
```

- Run the notifyapp sample application

```
$ ./notifyapp_debug
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/notify_omap3530_dsp.x64P 1
```

Note: Procedure for running **notifyapp_release** sample application is same as above.

Running the MessageQ sample application

MessageQ kernel-side sample application

- To invoke the application enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/messageq_omap3530_dsp.x64P 0
```

- Run the messageqapp sample application by inserting its kernel module

```
$ insmod messageqapp.ko
```

- For clean up:

```
$ rmmod messageqapp
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/messageq_omap3530_dsp.x64P 1
```

MessageQ user-side sample application

- To invoke the application standalone, enter the following commands:

```
$ ./messageqapp_debug 1 0 ./messageq_omap3530_dsp.x64P
```

Alternatively:

- To invoke the application with procmgrapp, enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/messageq_omap3530_dsp.x64P 0
```

- Run the messageqapp sample application

```
$ ./messageqapp_debug
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/messageq_omap3530_dsp.x64P 1
```

Note: Procedure for running **messageqapp_release** sample application is same as above.

Running the GateMP sample application

GateMP kernel-side sample application

- To invoke the application enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/gatemp_omap3530_dsp.x64P 0
```

- Run the gatempapp sample application by inserting its kernel module

```
$ insmod gatempapp.ko
```

- For clean up:

```
$ rmmod gatempapp
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/gatemp_omap3530_dsp.x64P 1
```

GateMP user-side sample application

- To invoke the application standalone, enter the following commands:

```
$ ./gatempapp_debug 1 0 ./gatemp_omap3530_dsp.x64P
```

Alternatively:

- To invoke the application with procmgrapp, enter the following commands:
 - Execute the following to load & start all slave processors. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/gatemp_omap3530_dsp.x64P 0
```

- Run the gatempapp sample application

```
$ ./gatempapp_debug
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/gatemp_omap3530_dsp.x64P 1
```

Note: Procedure for running **gatempapp_release** sample application is same as above.

Running the ListMP sample application

ListMP kernel-side sample application

- To invoke the application enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/listmp_omap3530_dsp.x64P 0
```

- Run the listmpapp sample application by inserting its kernel module

```
$ insmod listmpapp.ko
```

- For clean up:

```
$ rmmod listmpapp
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/listmp_omap3530_dsp.x64P 1
```

ListMP user-side sample application

- To invoke the application standalone, enter the following commands:

```
$ ./listmpapp_debug 1 0 ./listmp_omap3530_dsp.x64P
```

Alternatively:

- To invoke the application with procmgrapp, enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/listmp_omap3530_dsp.x64P 0
```

- Run the listmpapp sample application

```
$ ./listmpapp_debug
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/listmp_omap3530_dsp.x64P 1
```

Note: Procedure for running **listmpapp_release** sample application is same as above.

Running the HeapBufMP sample application

HeapBufMP kernel-side sample application

- To invoke the application enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/heapbufmp_omap3530_dsp.x64P 0
```

- Run the heapbufmpapp sample application by inserting its kernel module

```
$ insmod heapbufmpapp.ko
```

- For clean up:

```
$ rmmod heapbufmpapp
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/heapbufmp_omap3530_dsp.x64P 1
```

HeapBufMP user-side sample application

- To invoke the application standalone, enter the following commands:

```
$ ./heapbufmpapp_debug 1 0 ./heapbufmp_omap3530_dsp.x64P
```

Alternatively:

- To invoke the application with procmgrapp, enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/heapbufmp_omap3530_dsp.x64P 0
```

- Run the heapbufmpapp sample application

```
$ ./heapbufmpapp_debug
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/heapbufmp_omap3530_dsp.x64P 1
```

Note: Procedure for running **heapbufmpapp_release** sample application is same as above.

Running the HeapMemMP sample application

HeapMemMP kernel-side sample application

- To invoke the application enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/heapmemmp_omap3530_dsp.x64P 0
```

- Run the heapmemmpapp sample application by inserting its kernel module

```
$ insmod heapmemmpapp.ko
```

- For clean up:

```
$ rmmod heapmemmpapp
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/heapmemmp_omap3530_dsp.x64P 1
```

HeapMemMP user-side sample application

- To invoke the application standalone, enter the following commands:

```
$ ./heapmemmpapp_debug 1 0 ./heapmemmp_omap3530_dsp.x64P
```

Alternatively:

- To invoke the application with procmgrapp, enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/heapmemmp_omap3530_dsp.x64P 0
```

- Run the heapmemmpapp sample application

```
$ ./heapmemmpapp_debug
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/heapmemmp_omap3530_dsp.x64P 1
```

Note: Procedure for running **heapmemmpapp_release** sample application is same as above.

Running the FrameQ sample application

FrameQ kernel-side sample application

- To invoke the application enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/frameq_omap3530_dsp.x64P 0
```

- Run the frameqapp sample application by inserting its kernel module

```
$ insmod frameqapp.ko
```

- For clean up:

```
$ rmmod frameqapp
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/frameq_omap3530_dsp.x64P 1
```

FrameQ user-side sample application

- To invoke the application standalone, enter the following commands:

```
$ ./frameqapp_debug 1 0 ./frameq_omap3530_dsp.x64P
```

Alternatively:

- To invoke the application with procmgrapp, enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/frameq_omap3530_dsp.x64P 0
```

- Run the frameqapp sample application

```
$ ./frameqapp_debug
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/frameq_omap3530_dsp.x64P 1
```

Note: Procedure for running **frameqapp_release** sample application is same as above.

Running the RingIO sample application

RingIO kernel-side sample application

- To invoke the application enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/ringio_omap3530_dsp.x64P 0
```

- Run the ringioapp sample application by inserting its kernel module

```
$ insmod ringioapp.ko
```

- For clean up:

```
$ rmmod ringioapp
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/ringio_omap3530_dsp.x64P 1
```

RingIO user-side sample application

- To invoke the application standalone, enter the following commands:

```
$ ./ringioapp_debug 1 0 ./ringio_omap3530_dsp.x64P
```

Alternatively:

- To invoke the application with procmgrapp, enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/ringio_omap3530_dsp.x64P 0
```

- Run the ringioapp sample application

```
$ ./ringioapp_debug
```

- For clean up:

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/ringio_omap3530_dsp.x64P 1
```

Note: Procedure for running **ringioapp_release** sample application is same as above.

Running the RingIO GPP sample application

RingIO GPP kernel-side sample application

- To invoke the application enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/ringiogpp_omap3530_dsp.x64P 0
```

- Run the ringioapp sample application by inserting its kernel module

```
$ insmod ringiogppapp.ko
```

- For clean up:

```
$ rmmod ringiogppapp
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/ringiogpp_omap3530_dsp.x64P 1
```

RingIO GPP user-side sample application

- To invoke the application standalone, enter the following commands:

```
$ ./ringiogppapp_debug 1 0 ./ringiogpp_omap3530_dsp.x64P
```

Alternatively:

- To invoke the application with procmgrapp, enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/ringiogpp_omap3530_dsp.x64P 0
```

- Run the ringiogppapp sample application

```
$ ./ringiogppapp_debug
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/ringiogpp_omap3530_dsp.x64P 1
```

Note: Procedure for running **ringiogppapp_release** sample application is same as above.

Running the SharedRegion sample application

SharedRegion kernel-side sample application

- To invoke the application enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/sharedregion_omap3530_dsp.x64P 0
```

- Run the sharedregionapp sample application by inserting its kernel module

```
$ insmod sharedregionapp.ko SHAREDMEM=0x87C00000
```

- For clean up:

```
$ rmmod sharedregionapp
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/sharedregion_omap3530_dsp.x64P 1
```

Note: If your memory map is different, you may need to specify a different physical address value for SHAREDMEM.

SharedRegion user-side sample application

- To invoke the application standalone, enter the following commands:

```
$ ./sharedregionapp_debug 0x87C00000 1 0  
./sharedregion_omap3530_dsp.x64P
```

Alternatively:

- To invoke the application with procmgrapp, enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/sharedregion_omap3530_dsp.x64P 0
```

- Run the sharedregionapp sample application

```
$ ./sharedregionapp_debug 0x87C00000
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/sharedregion_omap3530_dsp.x64P 1
```

Note: If your memory map is different, you may need to specify a different physical address value as the first parameter.

Note: Procedure for running **sharedregionapp_release** sample application is same as above.

SysLink 02.00.00.68 beta1 InstallGuide Linux OMAPL1XX

SYS/Link 02.00.00.68_beta1 Linux OMAPL1XX

SysLink Install Guide for OMAPL1XX running Linux on the ARM.

Introduction

This document gives information about SysLink installation for using SysLink with the OMAPL1XX platform, where the ARM is running Linux OS. It also gives detailed build instructions for SysLink as well as the sample applications. In addition, instructions are also provided for running the sample applications provided within the SysLink release.

Dependencies

The dependencies, along with their versions are given in the Release Notes document.

Installation

Setting up Linux Workstation

The description in this section is based on the following assumptions:

- The workstation is running on a Linux workstation
- Services telnetd, nfsd, ftpd are configured on this workstation.
- The workstation is assigned a fixed IP address.

A fixed IP address is preferred, as the IP address needs to be specified while booting the target board. This allows the boot loader configuration to be saved in flash.

Enable TFTP for downloading the kernel image to target

U-boot can be configured to download the kernel onto the target by various mechanisms:

- TFTP
- Serial Port

This section configures the Linux development host as a TFTP server. Modify the 'xinet.d/tftp' file to enable TFTP:

- Make the following changes:

```
disable      = no
```

```
server_args = -s /tftpboot
```

- Restart the network service

```
$ /etc/init.d/xinetd restart
```

The above configuration assumes that a directory 'tftpboot' has been created at the root '/' directory. The files in this directory are exposed through the TFTP protocol.

Building the Linux kernel

This section assumes that Linux release is installed on the development workstation at `/toolchains/git`.

Tool chain

Untar the arm-2009q1-203 tool chain and add the tool chain directory in your path.

```
$ tar -xjvf
arm-2009q1-203-arm-none-linux-gnueabi-i686-pc-linux-gnu.tar.bz2.tar
$ export PATH=$PATH:/toolchains/git/arm-2009q1-203/bin
```

Kernel

Untar the kernel that comes with the Linux release. Build the kernel according to the PSP Linux User Guide document. Refer to the PSP u-boot related documentation on how to build the mkimage utility which is needed for creation of uImage. Along with the tool chain, location of mkimage utility should also be there in the path.

U-Boot boot-loader

Please refer to the Linux PSP user guide to load the u.boot and x-loader to the target.

Create target file system

The target device needs a file system to boot from. The file system can be exported to the target through NFS.

Exporting target file system through NFS

- A directory on the development host can be setup and exported for this purpose.
- AM35x-OMAP35x-PSP-SDK-03.00.00.03/images/fs/omapl1xx contains the NFS file system nfs.tar.gz.

```
$ tar -xzvf nfs.tar.gz
```

You need to be 'root' to successfully execute this command.

The file system can be copied to a different location. In such a case `~/omapl1xx/target` can be a soft link to the actual location.

- libpthread libraries required for SysLink may not be available by default within the target file system. Copy this from the tool-chain.

```
$ cp
/toolchains/git/arm-2009q1-203/arm-none-linux-gnueabi/libc/lib/libpthread*
~/omapl1xx/target/lib
```

This step is not required if libpthread libraries are available by default in the target file system.

- The directory `~/omapl1xx/target` will be mounted as root directory on the target through NFS.

To do so, add the following line to the file `/etc/exports`.

```
/home/<user>/omapl1xx/target *(rw,no_root_squash)
```

Replace `"/home/<user>"` in the path above with the actual path of your home directory on the development workstation.

Code Composer Studio

CCS v5.0 (supports CCS 4.2.07000 or later) can be used. To use CCS 4.2 for debugging the DSP side application, you CCS needs to be configured to use both ARM and DSP with OMAPL1XX.

CCS can attach to only ARM in the beginning. It can attach to the DSP only after the ARM-side application releases DSP from reset.

Build

Editing files for Linux headers and tool chain paths

- Set SYSLINK_ROOT environment variable. This should be set to the base of the SysLink repository, i.e. the folder containing the `ti` directory.

```
$ export SYSLINK_ROOT=~/.syslink_<VERSION>
```

- set SYSLINK_PKGPATH environment variable for IPC package path.

```
export
SYSLINK_PKGPATH="/db/psp_git/syslink_toolchains/IPC/ipc_<version>/packages;
```

Replace ipc_<version> above with the actual version of the IPC.

- Update KDIR variable in the base Makefile with the Linux source path in the following files. Alternatively, it can be overridden by passing to the make command.

```
$(SYSLINK_ROOT)/ti/syslink/buildutils/hlos/knl/Makefile.inc
KDIR :=
/toolchains/omapl1xx/DaVinci-PSP-SDK-03.20.00.11/src/kernel/linux-03.20.00.11
```

Actual path in your build environment must be given for the KDIR path.

- Update TOOLCHAIN_PREFIX variable in the base Makefile with the Linux tool chain in the following files. Alternatively, it can be overridden by passing to the make command.

```
$(SYSLINK_ROOT)/ti/syslink/buildutils/hlos/usr/Makefile.inc
TOOLCHAIN_PREFIX :=
/toolchains/git/arm-2009q1-203/bin/arm-none-linux-gnueabi-
```

- Alternatively, the paths can be passed to the make command as a ';' separated set of paths. For example:

```
$ make ARCH=arm
CROSS_COMPILE=/toolchains/git/arm-2009q1-203/bin/arm-none-linux-gnueabi-
SYSLINK_PKGPATH="/toolchains/IPC/ipc_<version>/packages; $HOME/syslink"
```

Type the above command in a single line

- Update CGTOOLS tool chain path in the following files:

```
$(SYSLINK_ROOT)/config.bld
```

For example:

```
var rootDirPre = "/toolchains/ti-tools/";
var rootDirPost = "";
C674.rootDir = rootDirPre + "c6000_7.2.0A10232" + rootDirPost;
```


- If build is required for only a specific configuration, comment the other configurations in Build.targets. e.g. set configuration to build only for OMAPL1XX device.

```
//list interested targets in Build.targets array
Build.targets = [
    //C28_large,
    //C64,
    //C64P_COFF,
    //C64P_ELF,
    //C67P,
    C674_COFF,
    C674_ELF,
    //Arm9,
    //M3_ELF,
    //Win32,
    //A8_ELF
];
```

Building SysLink HLOS driver/library and sample applications

The default configuration includes Ipc in the build for the drivers and the sample applications.

Switch to bash shell to build HLOS side drivers and samples. The default configuration builds for TI816X. To build for a different device, the SYSLINK_PLATFORM build parameter needs to be given.

Following are the steps to build the SysLink HLOS driver/library and sample applications for OMAPL1XX:

- Set SYSLINK_ROOT environment variable. This should be set to the base of the SysLink repository, i.e. the folder containing the ti directory.

```
$ export SYSLINK_ROOT=~/syslink_<VERSION>
```

- Add the toolchain path to PATH variable

```
$ export PATH=$PATH:/toolchains/git/arm-2009q1-203/bin
```

- Build the SysLink kernel module

```
$ cd $SYSLINK_ROOT/ti/syslink/utils/hlos/knl/Linux
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
SYSLINK_PLATFORM=OMAPL1XX
```

- Build SysLink kernel samples. Run make command in the respective sample directories.

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/notify/knl/Linux
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
SYSLINK_PLATFORM=OMAPL1XX
```

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/messageQ/knl/Linux
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
SYSLINK_PLATFORM=OMAPL1XX
```

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/heapBufMP/knl/Linux
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
SYSLINK_PLATFORM=OMAPL1XX
```



```

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/heapMemMP/knl/Linux
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
SYSLINK_PLATFORM=OMAPL1XX

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/frameq/knl/Linux
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
SYSLINK_PLATFORM=OMAPL1XX

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/listMP/knl/Linux
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
SYSLINK_PLATFORM=OMAPL1XX

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/ringIO/knl/Linux
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
SYSLINK_PLATFORM=OMAPL1XX

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/ringIO_gpp/knl/Linux
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
SYSLINK_PLATFORM=OMAPL1XX

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/sharedRegion/knl/Linux
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
SYSLINK_PLATFORM=OMAPL1XX

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/gateMP/knl/Linux
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
SYSLINK_PLATFORM=OMAPL1XX

```

- The Build System supports three different profiles on user side for library as well as the sample applications as follows:

```

* debug
* release
* both - debug & release (default)

```

- Build the SysLink user library (**debug & release - both profiles together**)

```

$ cd $SYSLINK_ROOT/ti/syslink/utls/hlos/usr/Linux
$ make

```

- Build the SysLink user library (**debug profile**)

```

$ cd $SYSLINK_ROOT/ti/syslink/utls/hlos/usr/Linux
$ make debug

```

- Build the SysLink user library (**release profile**)

```

$ cd $SYSLINK_ROOT/ti/syslink/utls/hlos/usr/Linux
$ make release

```

Note: For building the library no need to specify platform as the syslink user library is independent of platform.

- Build SysLink user samples (**debug & release - both profiles together**). Run make command in the respective sample directories.

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/common/usr/Linux
$ make SYSLINK_PLATFORM=OMAPL1XX

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/procMgr/usr/Linux
$ make SYSLINK_PLATFORM=OMAPL1XX

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/notify/usr/Linux
$ make SYSLINK_PLATFORM=OMAPL1XX

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/messageQ/usr/Linux
$ make SYSLINK_PLATFORM=OMAPL1XX

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/listMP/usr/Linux
$ make SYSLINK_PLATFORM=OMAPL1XX

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/heapBufMP/usr/Linux
$ make SYSLINK_PLATFORM=OMAPL1XX

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/heapMemMP/usr/Linux
$ make SYSLINK_PLATFORM=OMAPL1XX

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/frameq/usr/Linux
$ make SYSLINK_PLATFORM=OMAPL1XX

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/ringIO/usr/Linux
$ make SYSLINK_PLATFORM=OMAPL1XX

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/ringIO_gpp/usr/Linux
$ make SYSLINK_PLATFORM=OMAPL1XX

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/sharedRegion/usr/Linux
$ make SYSLINK_PLATFORM=OMAPL1XX

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/gateMP/usr/Linux
$ make SYSLINK_PLATFORM=OMAPL1XX
```

- Build SysLink user samples (**debug profile**). Run make command in the respective sample directories.

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/common/usr/Linux
$ make SYSLINK_PLATFORM=OMAPL1XX debug

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/procMgr/usr/Linux
$ make SYSLINK_PLATFORM=OMAPL1XX debug

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/notify/usr/Linux
$ make SYSLINK_PLATFORM=OMAPL1XX debug
```



```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/messageQ/usr/Linux
$ make SYSLINK_PLATFORM=OMAPL1XX debug

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/listMP/usr/Linux
$ make SYSLINK_PLATFORM=OMAPL1XX debug

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/heapBufMP/usr/Linux
$ make SYSLINK_PLATFORM=OMAPL1XX debug

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/heapMemMP/usr/Linux
$ make SYSLINK_PLATFORM=OMAPL1XX debug

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/frameq/usr/Linux
$ make SYSLINK_PLATFORM=OMAPL1XX debug

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/ringIO/usr/Linux
$ make SYSLINK_PLATFORM=OMAPL1XX debug

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/ringIO_gpp/usr/Linux
$ make SYSLINK_PLATFORM=OMAPL1XX debug

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/sharedRegion/usr/Linux
$ make SYSLINK_PLATFORM=OMAPL1XX debug

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/gateMP/usr/Linux
$ make SYSLINK_PLATFORM=OMAPL1XX debug
```

- **Build SysLink user samples (release profile).** Run make command in the respective sample directories.

```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/common/usr/Linux
$ make SYSLINK_PLATFORM=OMAPL1XX release

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/procMgr/usr/Linux
$ make SYSLINK_PLATFORM=OMAPL1XX release

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/notify/usr/Linux
$ make SYSLINK_PLATFORM=OMAPL1XX release

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/messageQ/usr/Linux
$ make SYSLINK_PLATFORM=OMAPL1XX release

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/listMP/usr/Linux
$ make SYSLINK_PLATFORM=OMAPL1XX release

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/heapBufMP/usr/Linux
$ make SYSLINK_PLATFORM=OMAPL1XX release
```



```
$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/heapMemMP/usr/Linux
$ make SYSLINK_PLATFORM=OMAPL1XX release

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/frameq/usr/Linux
$ make SYSLINK_PLATFORM=OMAPL1XX release

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/ringIO/usr/Linux
$ make SYSLINK_PLATFORM=OMAPL1XX release

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/ringIO_gpp/usr/Linux
$ make SYSLINK_PLATFORM=OMAPL1XX release

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/sharedRegion/usr/Linux
$ make SYSLINK_PLATFORM=OMAPL1XX release

$ cd $SYSLINK_ROOT/ti/syslink/samples/hlos/gateMP/usr/Linux
$ make SYSLINK_PLATFORM=OMAPL1XX release
```

Building RTOS SysLink and sample applications

- Set SYSLINK_ROOT environment variable. This should be set to the base of the SysLink repository, i.e. the folder containing the ti directory.

```
$ export SYSLINK_ROOT=~/.syslink_<VERSION>
```

- Set XDC in path.

```
$ export PATH=$PATH:/toolchains/xdc/xdctools_<version>
```

Alternatively, use the full path while making calls to xdc in the commands given later in this section

- Set XDC related environment variables.

```
$ export
XDCPATH="/users/ipc/ipc_<version>/packages;/toolchains/bios6/bios_<version>/packages"
```

Replace <version> above with the specific version for IPC, BIOS & XDC tools

- Following statement needs to be added to all application config(.cfg) files

```
xdc.loadPackage ('ti.syslink.ipc.rtos');
```

this ensures that all required linker options are included by default

User needs to add the above mentioned statement to application config (.cfg) files in order to include all required linker options by default

- Build RTOS SysLink. This builds RTOS FrameQ and RingIO

```
$ cd $SYSLINK_ROOT/ti/syslink/
$ xdc all XDCBUILDCFG="$SYSLINK_ROOT/config.bld" -PR .
```

- Default build profile is "whole_program_debug" to change this to "**debug**" run the following

```
$ xdc all XDCARGS="profile=debug"
XDCBUILDCFG="$SYSLINK_ROOT/config.bld" -PR .
```


alternatively "XDCARGS" can be exported

```
export XDCARGS=profile=debug
```

```
$ xdc all XDCBUILDCFG="$SYSLINK_ROOT/config.bld" -PR .
```

- Default build profile is "whole_program_debug" to change this to "**release**" run the following

```
$ xdc all XDCARGS="profile=release"
XDCBUILDCFG="$SYSLINK_ROOT/config.bld" -PR .
```

alternatively "XDCARGS" can be exported

```
export XDCARGS=profile=release
```

```
$ xdc all XDCBUILDCFG="$SYSLINK_ROOT/config.bld" -PR .
```

- If you want to exclude Samples and build rest of the libraries, run following command in directory "\$SYSLINK_ROOT/ti"

```
xdc XDCBUILDCFG="$SYSLINK_ROOT/config.bld" -PR 'xdcpkg syslink | grep
-v /samples'
```

The step to build SysLink RTOS side also builds all sample applications. If the syslink/ipc and sample applications are to be built independently, this can be done through separate commands as given below.

- To optimize the build time by utilizing "parallel build feature of GNU make" pass command line option **--jobs** as shown below

```
$ xdc all --jobs=4 XDCBUILDCFG="$SYSLINK_ROOT/config.bld" -PR .
```

--jobs= <number of separate jobs to be run> //usually 1.5 times no of cpu cores available on the build system

- To build only the sample applications independently, use the following commands:

```
$ cd $SYSLINK_ROOT/ti/syslink/ipc
$ xdc all -PR .

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/platforms
$ xdc all -PR .

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/notify
$ xdc all

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/messageQ
$ xdc all

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/frameq
$ xdc all

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/heapBufMP
$ xdc all

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/heapMemMP
$ xdc all
```



```
$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/listMP
$ xdc all

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/ringIO
$ xdc all

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/ringIO_gpp
$ xdc all

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/sharedRegion
$ xdc all

$ cd $SYSLINK_ROOT/ti/syslink/samples/rtos/gateMP
$ xdc all
```

Build Notes

The default build configuration for SysLink and sample applications assumes TI816X. If no specific SYSLINK_PLATFORM value is specified, TI816X is assumed by default.

Only COFF format is supported for the DSP.

For additional information on different build parameters, please refer to the UserGuide document.

Configuring Kernel Parameters

SYSLINK requires a few specific arguments to be passed to the Linux kernel during boot up. 3MB of memory is used by SYSLINK for communication between GPP and DSP, and for DSP external memory. This must be reserved by specifying 3MB less as available for the Linux kernel for its usage.

Running the sample applications

Sample applications are provided with SysLink for the supported platforms. This section describes the way to execute the sample applications.

The steps for execution of the samples are given below for execution with Linux running on the GPP.

For all kernel-side applications, ProcMgrApp is expected to be run before running the specific application to load and start the slave processor.

For all user-side applications, if no command line arguments are provided while running the sample application, the ProcMgrApp application is expected to be run before running the specific application to load and start the slave processor.

If the following command line arguments are provided, then the user-side application also loads/starts/stops the slave processor in the application context itself. In this case, ProcMgrApp application is not required to be run before running the specific application:

```
Arguments: <numProcs> <procId1> <Path including name of
the slave executable>
```


Copying files to target file system

The generated binaries on the HLOS side and RTOS side must be copied to the target directory.

For executing the sample applications, follow the steps below to copy the relevant binaries:

- Copy the syslink.ko kernel module into the target file system

```
$ cp ${SYSLINK_ROOT}/ti/syslink/bin/OMAPL1XX/*.ko
${HOME}/omapl1xx/target/opt/syslink/

* Copy the user and kernel sample application binaries into the target
file system


```
$ cp ${SYSLINK_ROOT}/ti/syslink/bin/OMAPL1XX/samples/*
${HOME}/omapl1xx/target/opt/syslink/
```


```

- To copy RTOS binaries, use the following steps:

Type the below commands each in a single line :: \$cp <src> <dst>

- Copy the rtos executables into the target file system

```
$cd ${SYSLINK_ROOT}/ti/syslink/samples/rtos

$ cp notify/evmDA830_dsp/whole_program_debug/notify_omapl1xx_dsp.x674
${HOME}/omapl1xx/target/opt/syslink/

$ cp
messageQ/evmDA830_dsp/whole_program_debug/messageq_omapl1xx_dsp.x674
${HOME}/omapl1xx/target/opt/syslink/

$ cp frameq/evmDA830_dsp/whole_program_debug/frameq_omapl1xx_dsp.x674
${HOME}/omapl1xx/target/opt/syslink/

$ cp listMP/evmDA830_dsp/whole_program_debug/listmp_omapl1xx_dsp.x674
${HOME}/omapl1xx/target/opt/syslink/

$ cp
heapBufMP/evmDA830_dsp/whole_program_debug/heapbufmp_omapl1xx_dsp.x674
${HOME}/omapl1xx/target/opt/syslink/

$ cp
heapMemMP/evmDA830_dsp/whole_program_debug/heapmemmp_omapl1xx_dsp.x674
${HOME}/omapl1xx/target/opt/syslink/

$ cp ringIO/evmDA830_dsp/whole_program_debug/ringio_omapl1xx_dsp.x674
${HOME}/omapl1xx/target/opt/syslink/

$ cp
ringIO_gpp/evmDA830_dsp/whole_program_debug/ringiogpp_omapl1xx_dsp.x674
${HOME}/omapl1xx/target/opt/syslink/

$ cp
sharedRegion/evmDA830_dsp/whole_program_debug/sharedregion_omapl1xx_dsp.x674
```



```
${HOME}/omapl1xx/target/opt/syslink/
```

```
$ cp gateMP/evmDA830_dsp/whole_program_debug/gatemp_omapl1xx_dsp.x674  
${HOME}/omapl1xx/target/opt/syslink/
```

Type above commands in single line *\$sp <src> <dst>*

Running applications

Loading syslink kernel module

- The syslink kernel module must be inserted before running any sample applications.

```
$ insmod syslink.ko TRACE=1 TRACEFAILURE=1
```

Enabling TRACEFAILURE puts out prints in case any failure occurs during SysLink initialization.

Unloading syslink kernel module

- The SysLink kernel module can be unloaded when it is no longer required, i.e. after all applications that use SysLink have exited. Unloading the SysLink kernel module reclaims to the HLOS, the resources that might have been consumed by SysLink.
- The SysLink Kernel module can be unloaded using:

```
$ rmmod syslink
```

Running the sample applications using scripts

Scripts have been provided for running the sample applications within the ti/syslink/tools/scripts folder.

Copying the scripts to target file system

Copy the following scripts into the target file system.

- Copy the top-level scripts into the target file system

```
$ cp ${SYSLINK_ROOT}/ti/syslink/tools/scripts/*.sh  
${HOME}/omapl1xx/target/opt/syslink/.
```

- Copy the scripts for each sample application into the target file system

```
$ cp ${SYSLINK_ROOT}/ti/syslink/tools/scripts/omapl1xx/*.sh  
${HOME}/omapl1xx/target/opt/syslink/.
```

Run the sample applications using the scripts

- To run the applications in debug build:

```
$ ./runsamples_debug.sh
```

- To run the applications in release build:

```
$ ./runsamples_release.sh
```

Running the ProcMgr sample application

ProcMgr user-side sample application

- To invoke the application enter the following commands:

- Execute the following to load, start and stop all slave processors. You can replace path and executables as per your setup:

Note: *procmgrapp_debug* takes two parameters *remote procId* and *executable name*.

```
$ procmgrapp_debug 0 /opt/syslink/messageq_omapl1xx_dsp.x674
```

- To use the procmgrapp as a loader for running other sample applications or as a standalone user-space loader to load any slave executable, use a final optional parameter
 - On providing the final parameter as 0, the procmgrapp only powers up, loads & starts the slave core
 - On providing the final parameter as 1, the procmgrapp only stops and powers down the slave core
- Load & start the slave core by invoking the procmgrapp application with final parameter as 0

```
$ procmgrapp_debug 0 /opt/syslink/messageq_omapl1xx_dsp.x674 0
```

- Run the sample application by inserting its kernel module. For example, messageqapp

```
$ insmod messageqapp.ko
```

- For clean up:

```
$ rmmod messageqapp
```

- Stop the slave core by invoking the procmgrapp application with final parameter as 1

```
$ procmgrapp_debug 0 /opt/syslink/messageq_omapl1xx_dsp.x674 1
```

Note: Procedure for running **procmgrapp_release** sample application is same as above.

Running the Notify sample application

Notify kernel-side sample application

- To invoke the application enter the following commands:
 - Execute the following to load and start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/notify_omapl1xx_dsp.x674 0
```

- Run the notifyapp sample application by inserting its kernel module

```
$ insmod notifyapp.ko
```

- For clean up:

```
$ rmmod notifyapp
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/notify_omapl1xx_dsp.x674 1
```


Notify user-side sample application

- To invoke the application standalone, enter the following commands:

```
$ ./notifyapp_debug 1 0 ./notify_omapl1xx_dsp.x674
```

- Press 'enter' to exit and cleanup.

Alternatively:

- To invoke the application with procmgrapp, enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/notify_omapl1xx_dsp.x674 0
```

- Run the notifyapp sample application

```
$ ./notifyapp_debug
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/notify_omapl1xx_dsp.x674 1
```

Note: Procedure for running **notifyapp_release** sample application is same as above.

Running the MessageQ sample application**MessageQ kernel-side sample application**

- To invoke the application enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/messageq_omapl1xx_dsp.x674 0
```

- Run the messageqapp sample application by inserting its kernel module

```
$ insmod messageqapp.ko
```

- For clean up:

```
$ rmmod messageqapp
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/messageq_omapl1xx_dsp.x674 1
```

MessageQ user-side sample application

- To invoke the application standalone, enter the following commands:

```
$ ./messageqapp_debug 1 0 ./messageq_omapl1xx_dsp.x674
```

Alternatively:

- To invoke the application with procmgrapp, enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/messageq_omapl1xx_dsp.x674 0
```


- Run the messageqapp sample application

```
$ ./messageqapp_debug
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/messageq_omapl1xx_dsp.x674 1
```

Note: Procedure for running **messageqapp_release** sample application is same as above.

Running the GateMP sample application

GateMP kernel-side sample application

- To invoke the application enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/gatemp_omapl1xx_dsp.x674 0
```

- Run the gatempapp sample application by inserting its kernel module

```
$ insmod gatempapp.ko
```

- For clean up:

```
$ rmmod gatempapp
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/gatemp_omapl1xx_dsp.x674 1
```

GateMP user-side sample application

- To invoke the application standalone, enter the following commands:

```
$ ./gatempapp_debug 1 0 ./gatemp_omapl1xx_dsp.x674
```

Alternatively:

- To invoke the application with procmgrapp, enter the following commands:
 - Execute the following to load & start all slave processors. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/gatemp_omapl1xx_dsp.x674 0
```

- Run the gatempapp sample application

```
$ ./gatempapp_debug
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/gatemp_omapl1xx_dsp.x674 1
```

Note: Procedure for running **gatempapp_release** sample application is same as above.

Running the ListMP sample application

ListMP kernel-side sample application

- To invoke the application enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/listmp_omapl1xx_dsp.x674 0
```

- Run the listmpapp sample application by inserting its kernel module

```
$ insmod listmpapp.ko
```

- For clean up:

```
$ rmmod listmpapp
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/listmp_omapl1xx_dsp.x674 1
```

ListMP user-side sample application

- To invoke the application standalone, enter the following commands:

```
$ ./listmpapp_debug 1 0 ./listmp_omapl1xx_dsp.x674
```

Alternatively:

- To invoke the application with procmgrapp, enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/listmp_omapl1xx_dsp.x674 0
```

- Run the listmpapp sample application

```
$ ./listmpapp_debug
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/listmp_omapl1xx_dsp.x674 1
```

Note: Procedure for running **listmpapp_release** sample application is same as above.

Running the HeapBufMP sample application

HeapBufMP kernel-side sample application

- To invoke the application enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/heapbufmp_omapl1xx_dsp.x674 0
```

- Run the heapbufmpapp sample application by inserting its kernel module

```
$ insmod heapbufmpapp.ko
```

- For clean up:

```
$ rmmod heapbufmpapp
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/heapbufmp_omapl1xx_dsp.x674 1
```

HeapBufMP user-side sample application

- To invoke the application standalone, enter the following commands:

```
$ ./heapbufmpapp_debug 1 0 ./heapbufmp_omapl1xx_dsp.x674
```

Alternatively:

- To invoke the application with procmgrapp, enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/heapbufmp_omapl1xx_dsp.x674 0
```

- Run the heapbufmpapp sample application

```
$ ./heapbufmpapp_debug
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/heapbufmp_omapl1xx_dsp.x674 1
```

Note: Procedure for running **heapbufmpapp_release** sample application is same as above.

Running the HeapMemMP sample application

HeapMemMP kernel-side sample application

- To invoke the application enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/heapmemmp_omapl1xx_dsp.x674 0
```

- Run the heapmemmpapp sample application by inserting its kernel module

```
$ insmod heapmemmpapp.ko
```

- For clean up:

```
$ rmmod heapmemmpapp
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/heapmemmp_omapl1xx_dsp.x674 1
```

HeapMemMP user-side sample application

- To invoke the application standalone, enter the following commands:

```
$ ./heapmemmpapp_debug 1 0 ./heapmemmp_omapl1xx_dsp.x674
```

Alternatively:

- To invoke the application with procmgrapp, enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/heapmemmp_omapl1xx_dsp.x674 0
```

- Run the heapmemmpapp sample application

```
$ ./heapmemmpapp_debug
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/heapmemmp_omapl1xx_dsp.x674 1
```

Note: Procedure for running **heapmemmpapp_release** sample application is same as above.

Running the FrameQ sample application

FrameQ kernel-side sample application

- To invoke the application enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/frameq_omapl1xx_dsp.x674 0
```

- Run the frameqapp sample application by inserting its kernel module

```
$ insmod frameqapp.ko
```

- For clean up:

```
$ rmmod frameqapp
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/frameq_omapl1xx_dsp.x674 1
```

FrameQ user-side sample application

- To invoke the application standalone, enter the following commands:

```
$ ./frameqapp_debug 1 0 ./frameq_omapl1xx_dsp.x674
```

Alternatively:

- To invoke the application with procmgrapp, enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/frameq_omapl1xx_dsp.x674 0
```

- Run the frameqapp sample application

```
$ ./frameqapp_debug
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/frameq_omapl1xx_dsp.x674 1
```

Note: Procedure for running **frameqapp_release** sample application is same as above.

Running the RingIO sample application

RingIO kernel-side sample application

- To invoke the application enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/ringio_omapl1xx_dsp.x674 0
```

- Run the ringioapp sample application by inserting its kernel module

```
$ insmod ringioapp.ko
```

- For clean up:

```
$ rmmod ringioapp
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/ringio_omapl1xx_dsp.x674 1
```

RingIO user-side sample application

- To invoke the application standalone, enter the following commands:

```
$ ./ringioapp_debug 1 0 ./ringio_omapl1xx_dsp.x674
```

Alternatively:

- To invoke the application with procmgrapp, enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/ringio_omapl1xx_dsp.x674 0
```

- Run the ringioapp sample application

```
$ ./ringioapp_debug
```

- For clean up:
 - Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/ringio_omapl1xx_dsp.x674 1
```

Note: Procedure for running **ringioapp_release** sample application is same as above.

Running the RingIO GPP sample application

RingIO GPP kernel-side sample application

- To invoke the application enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/ringiogpp_omapl1xx_dsp.x674 0
```

- Run the ringioapp sample application by inserting its kernel module

```
$ insmod ringiogppapp.ko
```

- For clean up:

```
$ rmmod ringiogppapp
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/ringiogpp_omapl1xx_dsp.x674 1
```

RingIO GPP user-side sample application

- To invoke the application standalone, enter the following commands:

```
$ ./ringiogppapp_debug 1 0 ./ringiogpp_omapl1xx_dsp.x674
```

Alternatively:

- To invoke the application with procmgrapp, enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/ringiogpp_omapl1xx_dsp.x674 0
```

- Run the ringiogppapp sample application

```
$ ./ringiogppapp_debug
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/ringiogpp_omapl1xx_dsp.x674 1
```

Note: Procedure for running **ringiogppapp_release** sample application is same as above.

Running the SharedRegion sample application

SharedRegion kernel-side sample application

- To invoke the application enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/sharedregion_omapl1xx_dsp.x674 0
```

- Run the sharedregionapp sample application by inserting its kernel module

```
$ insmod sharedregionapp.ko SHAREDMEM=0xC3C00000
```

- For clean up:

```
$ rmmod sharedregionapp
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/sharedregion_omapl1xx_dsp.x674 1
```

Note: If your memory map is different, you may need to specify a different physical address value for SHAREDMEM.

SharedRegion user-side sample application

- To invoke the application standalone, enter the following commands:

```
$ ./sharedregionapp_debug 0xC3C00000 1 0  
./sharedregion_omapl1xx_dsp.x674
```

Alternatively:

- To invoke the application with procmgrapp, enter the following commands:
 - Execute the following to load & start the slave processor. You can replace path and executables as per your setup:

```
$ procmgrapp_debug 0 /opt/syslink/sharedregion_omapl1xx_dsp.x674 0
```

- Run the sharedregionapp sample application

```
$ ./sharedregionapp_debug 0xC3C00000
```

- Execute the following to stop the slave processor.

```
$ procmgrapp_debug 0 /opt/syslink/sharedregion_omapl1xx_dsp.x674 1
```

Note: If your memory map is different, you may need to specify a different physical address value as the first parameter.

Note: Procedure for running **sharedregionapp_release** sample application is same as above.

Article Sources and Contributors

SysLink 02.00.00.68 beta1 InstallGuide *Source:* <http://ap-fpdsp-swapps.dal.design.ti.com/index.php?oldid=90343> *Contributors:* MugdhaKamoolkar

SysLink 02.00.00.68 beta1 InstallGuide Linux TI81XX *Source:* <http://ap-fpdsp-swapps.dal.design.ti.com/index.php?oldid=90711> *Contributors:* DeepaliUppal, Goutamkumar, MugdhaKamoolkar, Ravindranath Andela

SysLink 02.00.00.68 beta1 InstallGuide Linux OMAP3530 *Source:* <http://ap-fpdsp-swapps.dal.design.ti.com/index.php?oldid=90718> *Contributors:* Goutamkumar, MugdhaKamoolkar

SysLink 02.00.00.68 beta1 InstallGuide Linux OMAPL1XX *Source:* <http://ap-fpdsp-swapps.dal.design.ti.com/index.php?oldid=90720> *Contributors:* Goutamkumar, MugdhaKamoolkar