February 2021

AMP with Ac6 tools

Introduction

In recent years, the trends for embedded multicore systems are toward heterogeneous asymmetric multiprocessors system on chips (HMPSoCs) which combine two groups of cores where each has a different instruction set architecture (ISA); we can find, for example, one group with microprocessors and another with microcontrollers. This technology has many advantages since we can dedicate some cores for specific functions, in addition to installing different operating systems on each core.

Furthermore, several studies have shown that using different cores with dissimilar hardware characteristics on the same processor has many advantages; it brings better performances and lowers the power consumption compared to homogeneous cores. However, Asymmetric Multi-Processor (AMP) systems present many challenges to operating systems that should consider the shared hardware by implementing capable synchronization algorithms. Moreover, schedulers must distribute processing load on each core proportionally to its computing efficiency.

SoC architects, are creating mixed types of processing cores for complex systems to perform sophisticated operations in an effective way, for example, we can find a SoC with a quad-core ARM Cortex-A53, two ARM Cortex-R5 and a Field-Programmable Gate Array (FPGA) or another SoC with an ARM cortex-A7 and an ARM Cortex-M4. The microcontrollers typically offer less processing power, but also consume less power and dissipate less heat than the microprocessors.

These systems, in comparison with classical Symmetric Multi-Processing systems (SMP), present multiple challenges especially in the development phase.

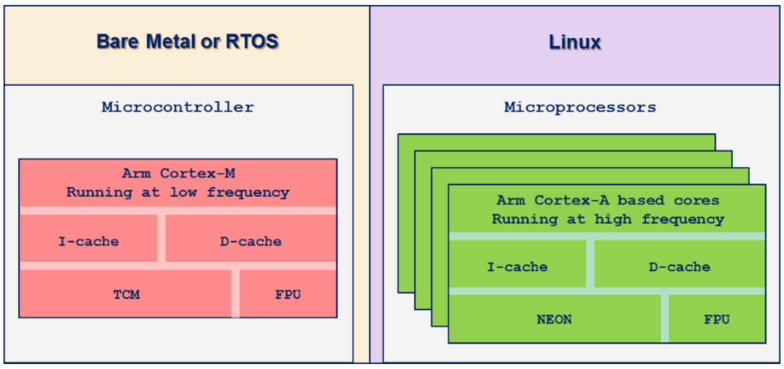


Figure 1-Software used on AMP cores

Embedded systems developers usually work with microcontrollers on bare metal or using a real time operating system (RTOS). With these architectures, they need to be familiar with embedded Linux as it is implemented most of the time on the microprocessors. So now the developer needs to get used to the Linux environment in addition to shared resources management and AMP communication.

System Workbench for Linux (SW4Linux) revolutionize the embedded Linux implementation and development. It introduces a simple way to get started with Linux and create advanced systems. It is the simplest way for developing all aspects. It is also compatible with all the eclipse-based microcontrollers development tools, enhancing them with Linux development features; when combining them together in one environment we create a perfect unified development tool for HMPSoCs.

AMP development tools

AMP system has several development components; we can split them in two sides: microcontroller side and microprocessor side. We have shared resources in between. The usual tools provided for these systems are: a tool for microcontroller development, a tool to handle shared resources and peripherals (generating intermediate files) and a Linux distribution builder based on Yocto.

The tools provided for microcontrollers development usually work very well and the tool that manages the peripherals is one of a kind, it generates the firmware in addition to configuring the system and the peripherals. However, as some of these peripherals are shared between the two sides, Linux should be aware of these configurations; in the best case you might only need to compile a device tree and to reconfigure the kernel. Nevertheless, in other cases, you might need to rewrite some kernel modules and drivers.

These two things for a non-Linux expert might be a blocking point: Firstly, they require some knowledge to get the right code and the right configuration, then the right command to compile and deploy it. Secondly, for kernel modules development, you need to understand the Linux code structure and to use some special techniques and files to get started with the development. Usually, a text editor is the only graphical user interface tool available as there is not any intuitive integrated development tool (IDE) for kernel development.

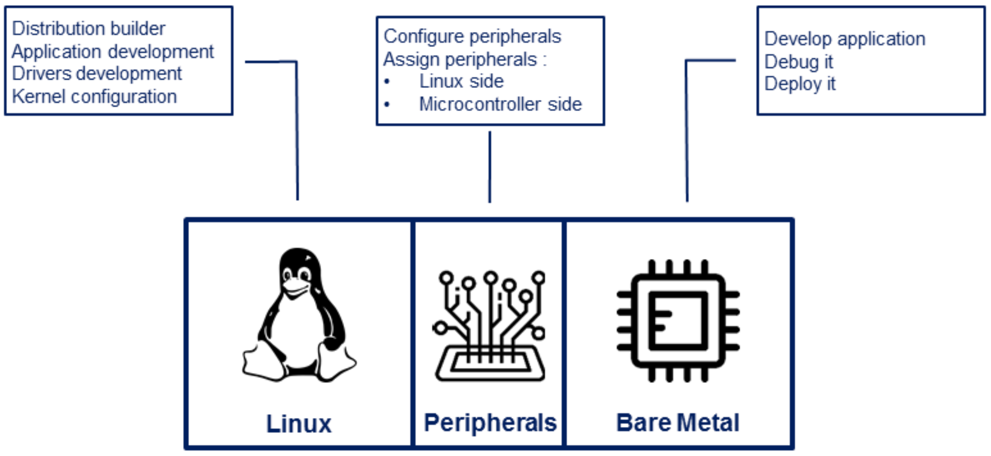


Figure 2-Needed tools for HMPSoCs

On the other side, to use the microprocessors, you need to create a custom Linux distribution. Yocto is usually the build system that allows creating and customizing it. However, a developer needs to learn how use it, and to get used to these concepts; this is a very complex step especially for an embedded system engineer that is used to develop systems for microcontrollers.

Once we have our distribution, we still need to develop our custom Linux application and for this part, there is no standard way to do it.

Embedded Linux Solutions

To run a Linux distribution on the microprocessors, we have two options, either use a prebuilt distribution or create our custom one; nowadays the software designers tend to implement a custom distribution, as this will let them have more control on the system and only provide what they specifically need.

Generating a distribution is not an easy step, it can be created manually, but in this case, it is time consuming, less portable, hard to maintain and can't be used to create complex systems; that is why it is not the recommend way to go. It is better to use a Linux build system that automates the build of all the component from the bootloader and kernel to the root file system. The most popular build systems for embedded Linux are Buildroot and Yocto, open-source projects, but our choice is usually limited by the support of the embedded board, Buildroot support less boards and Yocto is usually the reference build system provided by suppliers.

These two build systems, like most of other Linux tools, are created by and for "geeks"; they lack a very important feature: a graphical user interface. Buildroot is based on makefiles and some configuration files that need specific variables.

Yocto might look like a great solution when creating Linux distributions; it is a very powerful tool, regularly updated and maintained and it provides all the needed features to generate a fully working Linux system. Everything is represented by recipe files, which are written in a special scripting language "Bitbake" a python-shell mix. They contain many variables that control the build and sometimes you need to define some functions and tasks to tell the build system how to properly configure and build the package.

Yocto generates an SDK that contains the needed tools to develop user-space application for the built Linux system, but Yocto, itself is not a development tool. Although there are some plugins that can be added to some IDEs, it is not intuitive and most of the time the developers has to use some text editor for recipe or code development and command line to build and debug the application.

System Workbench for Linux

SW4Linux was created to resolve the complexity of classical build systems tools and to provide more functionalities for developers. It is a user-friendly eclipse-based IDE that provides a graphical user interface Linux build system for embedded systems; it lets you, not only easily build your embedded Linux distribution, but also develop, deploy and debug your application in a single environment.

In figure 3, you can see the difference between two definitions of a same package, on the left side, the SW4Linux view, on the right side the yocto recipe. The advantage, besides displaying the information with an intuitive interface, is that when there are some fields that need to be filled you do not need to guess which Yocto variable has to be defined or in which format.

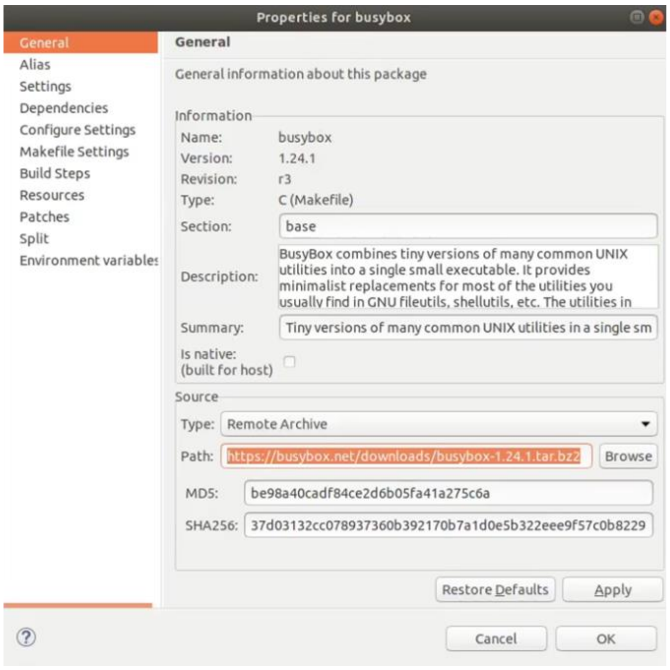
Each element of SW4Linux is basically an eclipse project. The most important one is called the platform project, which defines the board, Kernel and lists all related packages. We provide default platforms for many boards from popular SoC vendors.

Based on the default platform you can import custom packages from an online archive, local archive or folder, git or a C/C++ eclipse project. When customizing the configuration of a package, you have all the options available displayed in dialogs, helping you select an accurate configuration in the simplest way.

A developer encounters two challenges when working on embedded Linux. The first one is building the device tree, especially complicated for a beginner, as it becomes used and modified a lot during the development process; SW4Linux compiles it in just a few clicks. The second is deploying and debugging the application, SW4Linux provides a debug environment tailored for each platform.

SW4Linux is not only made for beginners in Linux development, but also for experts that work on modules and driver's development. Our tool support module development by creating module projects referring to a kernel project to build and test them in a few steps with all the help of the powerful Eclipse environment.

It offers easy to use development experience, in order to avoid losing time to learn the new scripting languages required for the traditional build systems. In addition, you can gain a lot of time by not having to re-configure the environment to build Linux Kernel related elements, which are specific for each system. Additionally, it provides functionalities that you cannot find in any other IDE.



General

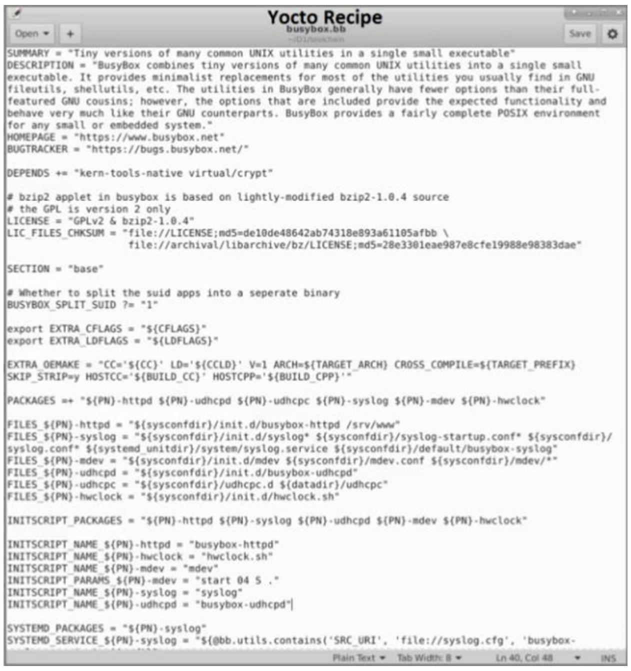
General information about this package

Information

Name: busybox
Version: 1.24.1
Revision: r3
Type: C (Makefile)
Section: base
Description: BusyBox combines tiny versions of many common UNIX utilities into a single small executable. It provides minimalist replacements for most of the utilities you usually find in GNU fileutils, shellutils, etc. The utilities in
Summary: Tiny versions of many common UNIX utilities in a single sm
Is native: (built for host) ☐

Source

Type: Remote Archive
Path: https://busybox.net/downloads/busybox-1.24.1.tar.bz2
MD5: be98a40cadf84ce2d6b05fa41a275c6a
SHA256: 37d03132cc078937360b392170b7a1d0e5b322eee9f57c0b8229



```
SUMMARY = "Tiny versions of many common UNIX utilities in a single small executable"
DESCRIPTION = "BusyBox combines tiny versions of many common UNIX utilities into a single small
executable. It provides minimalist replacements for most of the utilities you usually find in GNU
fileutils, shellutils, etc. The utilities in BusyBox generally have fewer options than their full-
featured GNU cousins; however, the options that are included provide the expected functionality and
behave very much like their GNU counterparts. BusyBox provides a fairly complete POSIX environment
for any small or embedded system."
HOMEPAGE = "https://www.busybox.net"
BUGTRACKER = "https://bugs.busybox.net/"

DEPENDS += "kern-tools-native virtual/crypt"

# bzip2 applet in busybox is based on lightly-modified bzip2-1.0.4 source
# the GPL is version 2 only
LICENSE = "GPLv2 & bzip2-1.0.4"
LIC_FILES_CHKSUM = "file://LICENSE;md5=de10de48642ab74318e893a61105afbb \
                    file://archival/libarchive/bz/LICENSE;md5=28e3301eae987e8cfe19988e98383dae"

SECTION = "base"

# Whether to split the suid apps into a seperate binary
BUSYBOX_SPLIT_SUID ?= "1"

export EXTRA_CFLAGS = "${CFLAGS}"
export EXTRA_LDFLAGS = "${LDFLAGS}"

EXTRA_OEMAKE = "CC='${CC}' LD='${CCLD}' V=1 ARCH=${TARGET_ARCH} CROSS_COMPILE=${TARGET_PREFIX}
SKIP_STRIP=y HOSTCC='${BUILD_CC}' HOSTCPP='${BUILD_CPP}'"

PACKAGES =+ "${PN}-httpd ${PN}-udhcpd ${PN}-udhcpc ${PN}-syslog ${PN}-mdev ${PN}-hwclock"

FILES_${PN}-httpd = "${sysconfdir}/init.d/busybox-httpd /srv/www"
FILES_${PN}-syslog = "${sysconfdir}/init.d/syslog* ${sysconfdir}/syslog-startup.conf* ${sysconfdir}/
syslog.conf* ${systemd_unitdir}/system/syslog.service ${sysconfdir}/default/busybox-syslog"
FILES_${PN}-mdev = "${sysconfdir}/init.d/mdev ${sysconfdir}/mdev.conf ${sysconfdir}/mdev/*"
FILES_${PN}-udhcpd = "${sysconfdir}/init.d/busybox-udhcpd"
FILES_${PN}-udhcpc = "${sysconfdir}/udhcpc.d ${datadir}/udhcpc"
FILES_${PN}-hwclock = "${sysconfdir}/init.d/hwclock.sh"

INITSCRIPT_PACKAGES = "${PN}-httpd ${PN}-syslog ${PN}-udhcpd ${PN}-mdev ${PN}-hwclock"

INITSCRIPT_NAME_${PN}-httpd = "busybox-httpd"
INITSCRIPT_NAME_${PN}-hwclock = "hwclock.sh"
INITSCRIPT_NAME_${PN}-mdev = "mdev"
INITSCRIPT_PARAMS_${PN}-mdev = "start 04 S ."
INITSCRIPT_NAME_${PN}-syslog = "syslog"
INITSCRIPT_NAME_${PN}-udhcpd = "busybox-udhcpd"

SYSTEMD_PACKAGES = "${PN}-syslog"
SYSTEMD_SERVICE_${PN}-syslog = "${@bb.utils.contains('SRC_URI', 'file://syslog.cfg', 'busybox-
```

Figure 3-Comparison of the busybox package definition, SW4Linux (left) vs Yocto (right)

Several types of programs must meet timing constraints, especially in the embedded systems domain. Therefore, it is necessary to measure the execution time of some block of code. Ac6 tools provide a feature to simplify the execution time measurement for a function written in C or C++. The tool adds a new feature that duplicates the working project and injects some code to allow the measurement, based on custom or predefined measurement methods, it will configure the system, do the measurement and the output data will be recovered automatically using the debugger.

It is also possible to provide a function entry values sets, and it will perform the measurement for all the entry data and repeated several times. This feature works with the Linux side of the application as well as the microcontroller side.

Unified IDE for AMP systems

An ideal tool for HMPSoCs is a unified IDE that allows everything to be done in one easy to use environment. SW4Linux is the perfect solution, which adds to the already existing tools, all the missing features. When merged, we create the IDE that can handle the development, and build of all the aspects of theses architectures. Most of the chipmakers build their development environment on top of eclipse; SW4Linux can be easily installed as a plugin to add its features.

As shown in the figure 4, it adds the missing pieces, the first is to generate the Linux distribution so it can be implemented on the microprocessors, the second is to provide development and debug tools that work with the generated distribution.

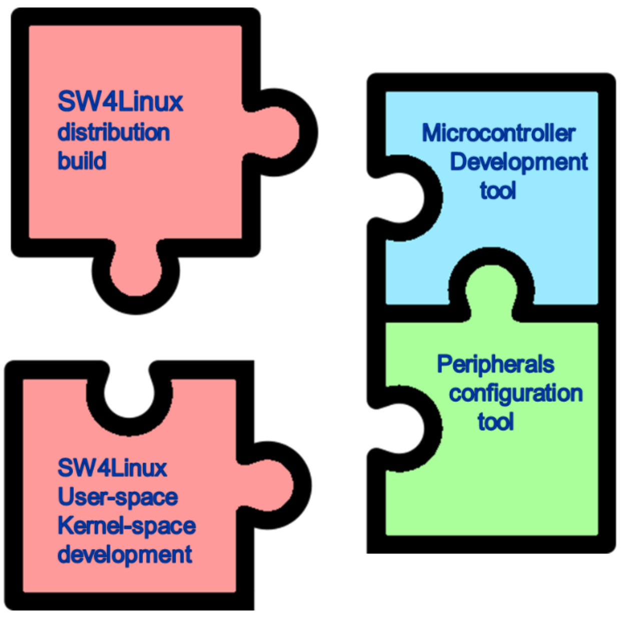
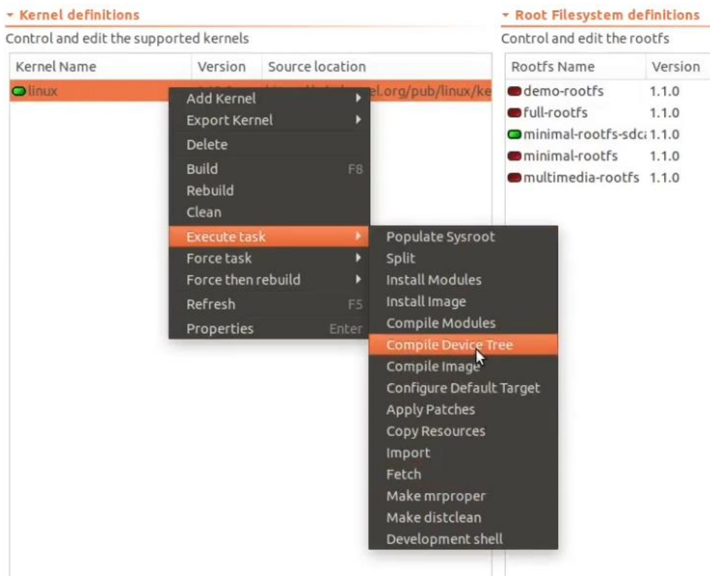

SW4Linux distribution build
Microcontroller Development tool
Peripherals configuration tool
SW4Linux User-space Kernel-space development

Figure 4-SW4Linux added features

Linux peripherals are managed using device trees; the device tree source (dts) is modified regularly during the development phase; as peripherals are reconfigured repeatedly in embedded systems, the dts needs to be recompiled using the device tree compiler and usually depends on other source files inside the kernel's source code. First, it must be copied inside the kernel build tree and then compiled using a command that needs specific arguments. These steps are automatically handled by SW4Linux, and the device tree can be compiled in a few clicks: when a device tree is detected in the workspace, it is possible to export it to a kernel build tree; after that, you just select the kernel project, then compile the device tree by executing the device tree compilation task as seen in figure 5.


Kernel definitions

Control and edit the supported kernels

Kernel Name	Version	Source location
linux		el.org/pub/linux/ke

Root Filesystem definitions

Control and edit the rootfs

Rootfs Name	Version
demo-rootfs	1.1.0
full-rootfs	1.1.0
minimal-rootfs-sdc	1.1.0
minimal-rootfs	1.1.0
multimedia-rootfs	1.1.0

Figure 5-Compiling a device tree

Furthermore, developing embedded Linux applications usually require accessing the hardware, which is only possible from kernel space code, using either an in-tree or an out of tree driver module. The latter is the classical way to do this or at least to develop the code before integrating it in the main kernel code. Developing kernel space application was usually restricted to Linux experts; they use a text editor and sophisticated makefiles.

SW4Linux introduces a new strategy for kernel module development, build and debug. It has features that allow any embedded developer to write kernel code starting from a preconfigured project that is linked to some kernel project, like device trees, making building it one click away. Furthermore, it has a code generator, based on values provided in dialogs, that can generate the driver's skeleton including most of the needed code structure.

This feature lets developing complex drivers easily and lets non-Linux experts doing advanced Linux development; most important it saves a lot of time.

AMP development

Developing a Linux side application for the target is similar to developing a microcontroller application.

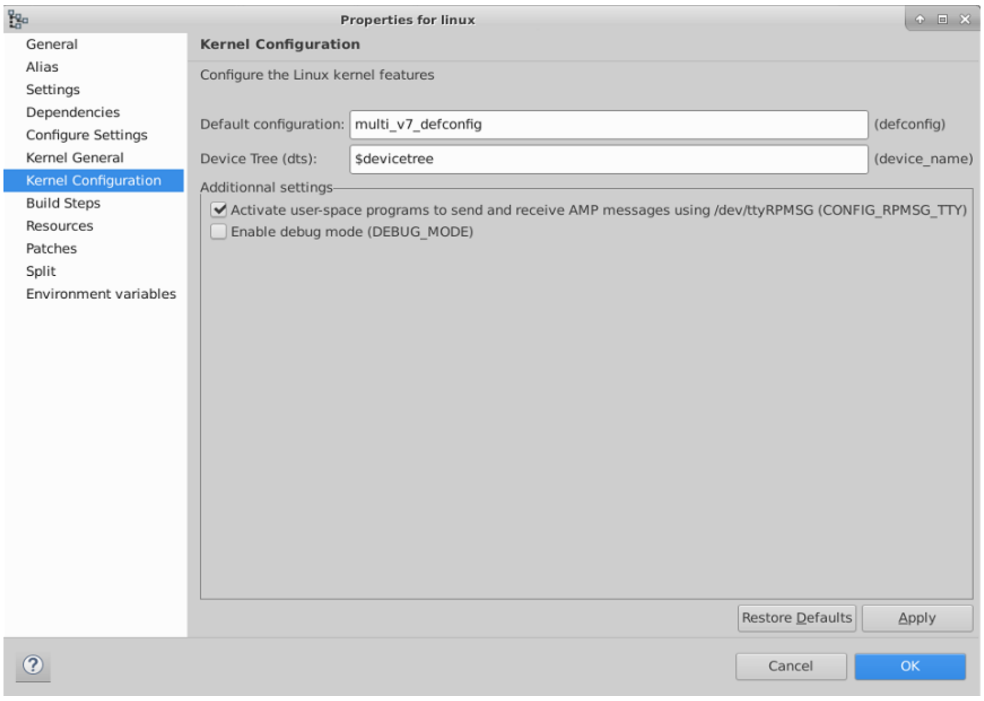
Firstly, we create a cross compiler eclipse project based on the default option available in CDT eclipse environment; without the need to fill in the information about the compiler or the compiler path. All we have to do is simply add the project to the working platform, and then project will automatically be configured to make it compatible with the generated distribution. The compiler and the libraries are detected, and all needed information is added to the project.

Building the cross Linux application is like any other eclipse project, press the hammer icon or right click on the project and build it. It is the same way as you can build the microcontroller's project; it is also possible to right click on the project then build.

Communication between heterogeneous cores is necessary; this is done through an AMP messaging protocol called OpenAMP (or an alternative).

On the microcontroller, the library is usually part of the microcontroller's firmware; the protocol's library just should be included in the project.

On the Linux side, before starting the development, it is important to make sure that the protocol's driver is activated in the kernel. SW4Linux provides a feature to activate and deactivate kernel options with check boxes, based on kernel configuration fragments; for AMP boards there is rpmsg which is the remote messaging driver. The following figure shows the rpmsg driver enabled.



Kernel Configuration

Configure the Linux kernel features

Default configuration: multi_v7_defconfig (defconfig)

Device Tree (dts): $devicetree (device_name)

Additionnal settings

[✓] Activate user-space programs to send and receive AMP messages using /dev/ttyRPMSG (CONFIG_RPMSG_TTY)

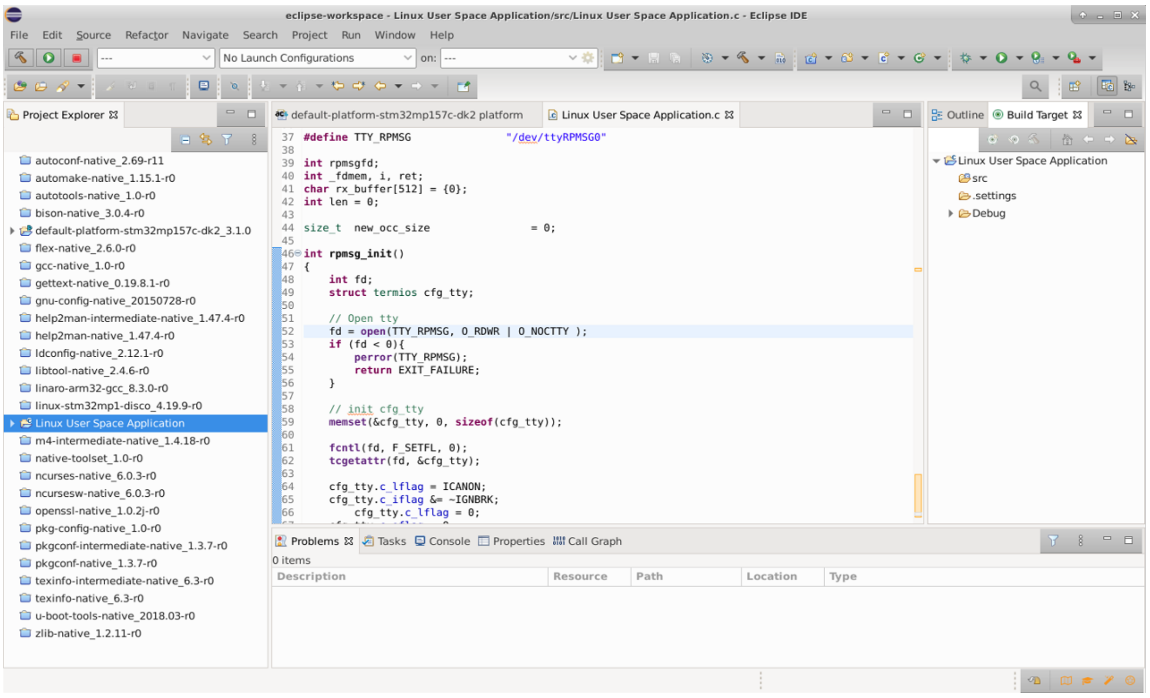
[ ] Enable debug mode (DEBUG_MODE)

Figure 6-SW4Linux - Kernel Configuration menu

The AMP communication driver, like any other driver, is accessed through a /dev special file (/dev/ttyRPMSG); this driver makes it possible for user-space programs to send and receive AMP messages by reading or writing on a virtual tty device, using standard write and read system calls.

On the microcontroller's side specific functions of the OpenAMP library will allow to receive these messages, and send answers or requests to the Linux side. This library typically is available either for bare metal programming or, if the real-time critical part of the system is more complex, using a real-time operating system like FreeRTOS.

On the Linux side, SW4Linux makes Linux application's development very simple and it looks like the development of a microcontroller application as seen in figure 7.


eclipse-workspace - Linux User Space Application/src/Linux User Space Application.c - Eclipse IDE
File Edit Source Refactor Navigate Search Project Run Window Help
No Launch Configurations
on:
Project Explorer
autoconf-native_2.69-r11
automake-native_1.15.1-r0
autotools-native_1.0-r0
bison-native_3.0.4-r0
default-platform-stm32mp157c-dk2_3.1.0
flex-native_2.6.0-r0
gcc-native_1.0-r0
gettext-native_0.19.8.1-r0
gnu-config-native_20150728-r0
help2man-intermediate-native_1.47.4-r0
help2man-native_1.47.4-r0
ldconfig-native_2.12.1-r0
libtool-native_2.4.6-r0
linaro-arm32-gcc_8.3.0-r0
linux-stm32mp1-disco_4.19.9-r0
Linux User Space Application
m4-intermediate-native_1.4.18-r0
native-toolset_1.0-r0
ncurses-native_6.0.3-r0
ncursesw-native_6.0.3-r0
openssl-native_1.0.2j-r0
pkg-config-native_1.0-r0
pkgconf-intermediate-native_1.3.7-r0
pkgconf-native_1.3.7-r0
texinfo-intermediate-native_6.3-r0
texinfo-native_6.3-r0
u-boot-tools-native_2018.03-r0
zlib-native_1.2.11-r0
default-platform-stm32mp157c-dk2 platform
Linux User Space Application.c
Outline
Build Target
Linux User Space Application
src
.settings
Debug
Problems
Tasks
Console
Properties
Call Graph
0 items
Description
Resource
Path
Location
Type

```c
#define TTY_RPMSG                "/dev/ttyRPMSG0"

int rpmsgfd;
int _fdmem, i, ret;
char rx_buffer[512] = {0};
int len = 0;

size_t  new_occ_size              = 0;

int rpmsg_init()
{
    int fd;
    struct termios cfg_tty;

    // Open tty
    fd = open(TTY_RPMSG, O_RDWR | O_NOCTTY );
    if (fd < 0){
        perror(TTY_RPMSG);
        return EXIT_FAILURE;
    }

    // init cfg_tty
    memset(&cfg_tty, 0, sizeof(cfg_tty));

    fcntl(fd, F_SETFL, 0);
    tcgetattr(fd, &cfg_tty);

    cfg_tty.c_lflag = ICANON;
    cfg_tty.c_iflag &= ~IGNBRK;
        cfg_tty.c_lflag = 0;
```

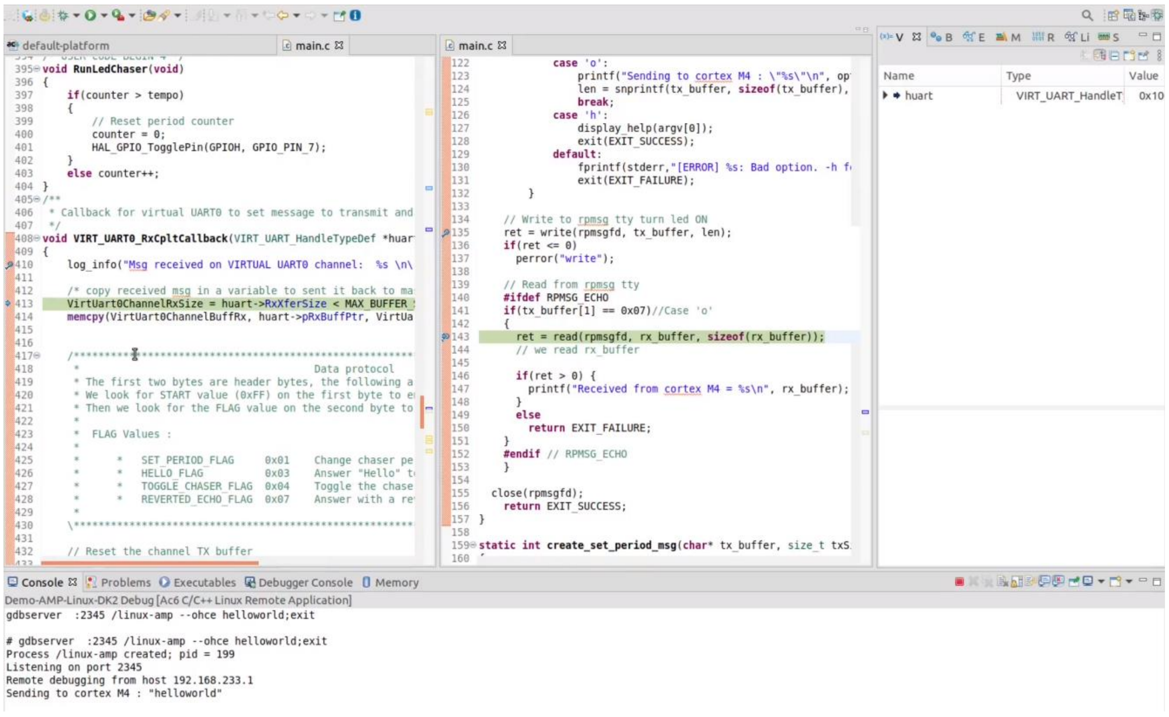
Figure 7-Linux user space application

AMP simultaneous debugging

The project used for developing the Linux side application, can be deployed, tested or debugged, there is a new Run/Debug option available, the "AC6 C/C++ Remote Debugging"; it will detect automatically the working platform and provide the compatible settings and debugger. The only information that need to be filled manually is the target's IP address; SW4Linux needs it to find the target, transfer, install and execute the program, in addition to staring the debugging server. All of this happens in a transparent manner to the developer, he will only find the debugging session has started and there is a breakpoint at the entry of the main function.

For the microcontroller side, it will be based on the microcontroller specific development tool, each provider has a different debugging setting, all we have to do is to launch the debugging session.

Using these simple steps, we will have a fully working system, ready to be used for developing the AMP applications, on Linux the remote debugging session can be launched in a few clicks and the microcontroller application's debugging session, depending on the tools, will be launched in parallel.



```
void RunLedChaser(void)
{
    if(counter > tempo)
    {
        // Reset period counter
        counter = 0;
        HAL_GPIO_TogglePin(GPIOH, GPIO_PIN_7);
    }
    else counter++;
}
/**
 * Callback for virtual UART0 to set message to transmit and
 */
void VIRT_UART0_RxCpltCallback(VIRT_UART_HandleTypeDef *huar
{
    log_info("Msg received on VIRTUAL UART0 channel:  %s \n\

    /* copy received msg in a variable to sent it back to ma
    VirtUart0ChannelRxSize = huart->RxXferSize < MAX_BUFFER_
    memcpy(VirtUart0ChannelBuffRx, huart->pRxBuffPtr, VirtUa


    /*********************************************************
     *                                     Data protocol
     * The first two bytes are header bytes, the following a
     * We look for START value (0xFF) on the first byte to e
     * Then we look for the FLAG value on the second byte to
     *
     *  FLAG Values :
     *
     *      *   SET_PERIOD_FLAG     0x01    Change chaser pe
     *      *   HELLO_FLAG          0x03    Answer "Hello" t
     *      *   TOGGLE_CHASER_FLAG  0x04    Toggle the chase
     *      *   REVERTED_ECHO_FLAG  0x07    Answer with a re
     *
    \*********************************************************

    // Reset the channel TX buffer
```

```
            case 'o':
                printf("Sending to cortex M4 : \"%s\"\n", op
                len = snprintf(tx_buffer, sizeof(tx_buffer),
                break;
            case 'h':
                display_help(argv[0]);
                exit(EXIT_SUCCESS);
            default:
                fprintf(stderr,"[ERROR] %s: Bad option. -h f
                exit(EXIT_FAILURE);
        }

    // Write to rpmsg tty turn led ON
    ret = write(rpmsgfd, tx_buffer, len);
    if(ret <= 0)
      perror("write");

    // Read from rpmsg tty
    #ifdef RPMSG_ECHO
    if(tx_buffer[1] == 0x07)//Case 'o'
    {
      ret = read(rpmsgfd, rx_buffer, sizeof(rx_buffer));
      // we read rx_buffer

      if(ret > 0) {
        printf("Received from cortex M4 = %s\n", rx_buffer);
      }
      else
        return EXIT_FAILURE;
    }
    #endif // RPMSG_ECHO
    }

  close(rpmsgfd);
    return EXIT_SUCCESS;
}

static int create_set_period_msg(char* tx_buffer, size_t txS
```

Name	Type	Value
huart	VIRT_UART_HandleT	0x10

Demo-AMP-Linux-DK2 Debug [Ac6 C/C++ Linux Remote Application]

```
gdbserver  :2345 /linux-amp --ohce helloworld;exit

# gdbserver  :2345 /linux-amp --ohce helloworld;exit
Process /linux-amp created; pid = 199
Listening on port 2345
Remote debugging from host 192.168.233.1
Sending to cortex M4 : "helloworld"
```

Figure 8-Two simultaneous debugging sessions on heterogeneous cores

In figure 8 above, we can see a screenshot of this unified IDE: on the left side we have the microcontroller debugging session running with the application stopped on a breakpoint; on the right side we find the Ac6 Linux remote debugging session with the code blocked on a breakpoint as well. In this specific application we have used the OpenAMP protocol that provides the communication mechanism between the AMP cores and we have set a breakpoint when sending or receiving messages.

Conclusion

As AMP heterogeneous SoCs are getting more popular embedded Linux is used more than ever; so being familiar with both microcontroller and microprocessor sides is a must. Developers have to create applications for two types of cores that require different programming skills. On the microcontroller side, development environments are getting more intuitive and usually we have several options. On the other side, most of the development tools for Linux are command line and requires Linux experts.

SW4Linux solve that problem, it introduces an intuitive eclipse-based build system, where it is possible to execute most of the operations in a few clicks. Furthermore, it introduces a lot of development and debugging features, for both user-space and kernel-space applications and brings advanced Linux development to all embedded system developers.

The most important advantage for AMP systems, is that it is now possible to merge all development tools in one unified IDE, which can be used to configure and develop the whole system in a single environment.