实验 1 AutoSar 开源软件环境搭建和编译实验

实验目的：

了解 AutoSar 软件结构，掌握 Linux 平台下 Autosar 环境的搭建和开源代码编译。通过搭建过程熟悉硬件模拟器 Qemu 和 Scons 命令的使用。

实验内容：

1. Linux 虚拟机配置

2. AutoSar 开源软件下载

3. Qemu 的运行与使用

4.基于 Scons 编译 AutoSar 源码

预备知识：

1. Linux、Git 基本命令

实验步骤：

1. 虚拟机环境配置

虚拟机使用 Ubuntu18.04 版本，可在 Windows 系统中安装 Virtual Box 或 VMware，或者直接使用 Ubuntu18.04 系统，并通过 apt install git 命令配置好 Git.

2. 开源代码下载与查看

2.1 项目介绍

AUTOSAR(Automotive Open System Architecture, AUTOSAR)是汽车和软件行业领先公司的全球合作联盟为智能移动开发建立的标准化软件框架以及开放的 E/E 系统软件架构。本实验 AS 代码包含了 AUTOSAR 的工具链、域控模拟器 qemu 环境、仪表模拟，发包工具、ArcticCore 源码，各种 RTOS 等，是一个整合型的开源汽车软件开发平台。目前开源的版本较少，有代表性的是 parai.wang 编写的开源项目 ，源码地址为 https://github.com/autoas/as。

本实验的开源地址为 https://github.com/thatway1989/as，项目中涉及到的 ArcticCore 的源码版本较老，最新的 ArcticCore 源码为：https://github.com/openAUTOSAR/classic-platform。感兴趣的同学可以自行下载和编译。

2.2 项目下载

```
git clone https://github.com/thatway1989/as
```

2.3 源码结构分析

汽车框架演进分为三步：分布式（ECU）→集中式（域控制器）→中央式（硬件虚拟化+SOA）。

分布式架构在传统机械设备上引入了电子电路，使用 **ECU（Electronic Control Unit, ECU）**电子控制器单元来控制汽车的行驶状态以及实现各种功能，是一块独立的电路板。但随着汽车智能化的发展，ECU 的数量增多，出现电路板功能冗余、总线长度长成本高、ECU 交互功能复杂等问题。于是很多功能相似、分离的 ECU 被整合到一个性能更强的处理器硬件平台上，这就是汽车域控制器 **DCU（Domain Control Unit, DCU）**。域控制器的出现是汽车 EE 架构从 ECU 分布式 EE 架构演进到域集中式 EE 架构的一个重要标志。

域控制器的具体实现需要**电路板+代码**，因此在 AS 平台中，通过 qemu 虚拟机模拟了域控制器的硬件，而在硬件上跑的代码即为 AUTOSAR CP 代码。

AS 的代码结构如下

```
├── build
|   └── posix #编译后产生的临时目录
├── com
|   ├── as.application #编译的板卡类型相关配置
|   ├── as.infrastructure #Arccore开源代码
|   ├── as.tool #studio、Asone等工具
|   ├── as.virtual
|   └── SConscript #scons 脚本文件
├── Console.bat #在windows下运行的脚本文件，用于准备运行环境
├── Kconfig #kcongig文件
├── kernel.bin #编译后产生的bin文件，由autosar的代码编译而来
├── README.md #说明文件，还包括作者的一些参考网址记录
├── release
|   ├── asboot #bootloader
|   ├── ascore #示例应用程序的核心
|   ├── askar #一种RTOS
|   ├── aslinux #
|   ├── asminix #
|   ├── asslave #
|   ├── download #从网络上下载的一些工具，例如编译器、lwip组件、库文件等
|   ├── README.md #说明文件
|   ├── SConscript #scons脚本
|   └── ucos_ii #一种RTOS
├── SConscript #scons 脚本
├── SConstruct #scons 脚本
└── TINIX.IMG #编译出来的目标文件
```

其中，Com 文件主要包括

1) as.application/SConscript：主要包括三类文件

```
objs += SConscript('board.%s/SConscript'%(BOARD))
objs += SConscript('common/SConscript')
objs += SConscript('swc/SConscript')
```

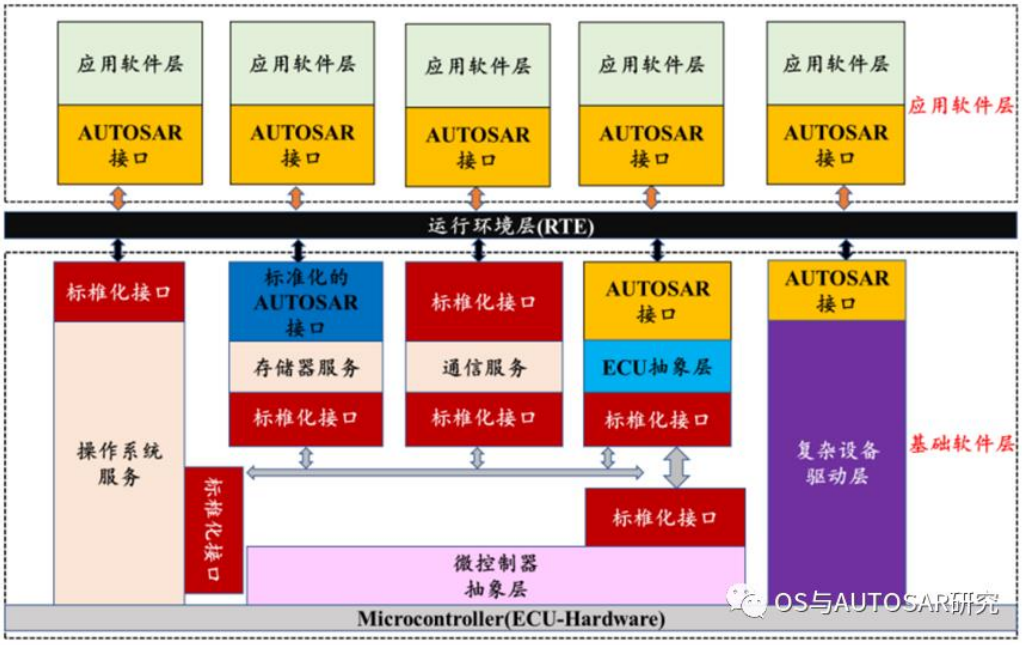
- as.application/board.%s 平台对应 BOARD 参数，如 x86，stm32，mpc56 等。
- as.application/common 下主要是配置文件、rte 文件、测试文件

2) as.infrastructure：包括 arccore 的源码

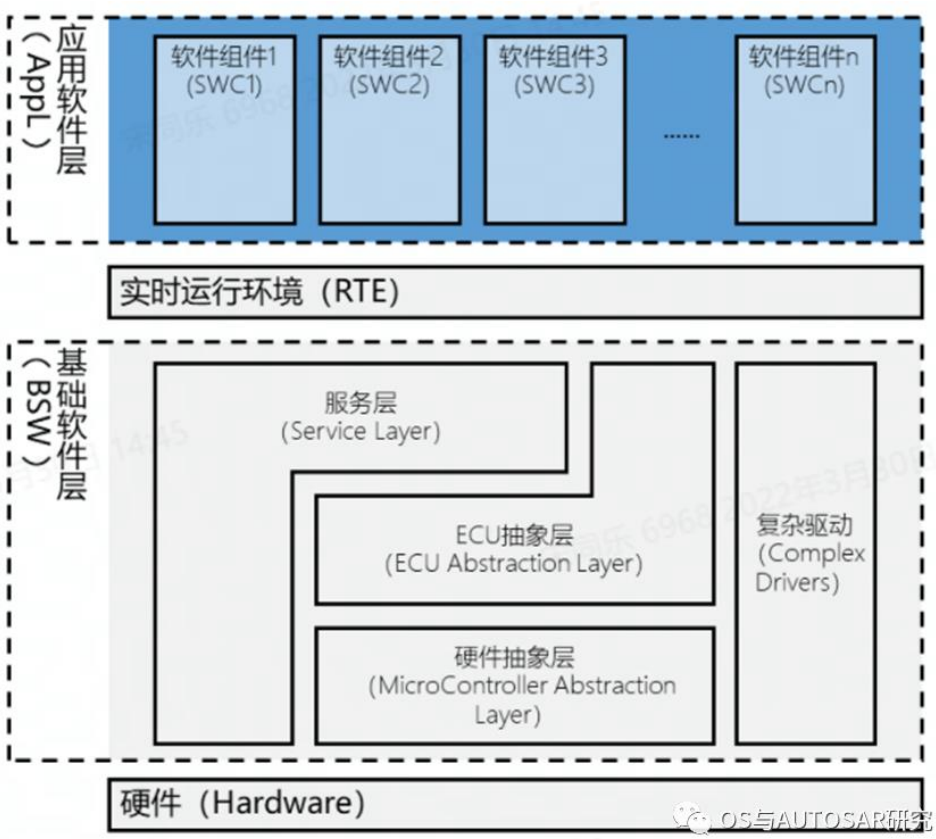
3) as.tool：代码结构如下图

```
├── as.one.py #AsOne工具
├── cancasexl.access #
├── config.infrastructure.gui #仪表盘UI
├── config.infrastructure.system #工具脚本的集中存放地方，例如studio、AsOne等
├── Kconfig
├── kconfig-frontends #Kconfig工具
├── lua #can、lin驱动相关的一些代码
├── py.can.database.access #
├── qemu #qemu虚拟机相关的一些驱动
└── SConscript
```

单独看各个文件的作用会比较抽象，但结合域控制器的架构就会比较清楚。域控制器 DCU 主要完成了抽象复用的功能。在汽车系统中，**上层的应用软件**会根据需求不断变化，**最下层的底层硬件**也会不断更换，而中间不变的这一层就可以提取出来，提取出的部分如图中除应用软件层和 Microcontroller 之外的部分所示：



根据上述抽取内容，就可得出 AUTOSAR 的分层结构如下所示：



- SWC（Software Component）：as 中代码路径：as/com/as.application/common/test
  与车辆应用相关，多个 SWC 组成具体的应用软件层（Application Software Layer，ASW）。
- RTE（Runtime Environment）：as 中代码路径：as/com/as.application/common/rte
  RTE 作为应用软件层与基础软件层交互的桥梁，为软硬件分离提供了可能。RTE 可以实现软件组件间、基础软件间以及软件组件与基础软件之间的通信。RTE 封装了基础软件层的通信和服务，为应用层软件组件提供了标准化的基础软件和通信接口，使得应用层可以通过 RTE 接口函数调用基础软件的服务。此外，RTE 抽象了 ECU 之间的通信，即 RTE 通过使用标准化的接口将其统一为软件组件之间的通信。由于 RTE 的实现与具体 ECU 相关，所以必须为每个 ECU 分别实现。
- BSW（Basic SoftwareLayer）：as 中代码路径：as/com/as.infrastructure
  BSW 又可分为四层，即服务层（Services Layer）、ECU 抽象层（ECU Abstraction Layer）、微控制器抽象层（Microcontroller Abstraction Layer，MCAL）和复杂驱动（Complex Drivers）。主要用于提供基础软件服务，包括标准化的系统功能和功能接口。

Release 文件主要存放 C 和 C++发布的应用程序。

3. 编译

3.1 编译代码

编译工具需要用到 scons，Python3 等，通过以下命令安装

```
sudo apt install scons  autoconf libtool-bin python3-sip python3-sip-
dev sip-dev python3-pip flex bison gperf \
```

```
libncurses-dev nasm gnome-terminal gcc-arm-none-eabi libreadline-
dev python3-pyqt5  libcurl4-openssl-dev libgtk-3-dev pkg-
config libglib2.0-dev
```

编译命令如下，通过 export 命令设置 BOARD 与 RELEASE 参数，本实验模拟的域控制器为 x86 平台，仅供演示。

```
1. cd as
2. scons
3. export BOARD=x86
4. export RELEASE=ascore
5. scons
```

编译**成功**后提示

```
cp kernel.bin to TINIX.IMG:/kernel.bin
scons: done building targets.
zhouyu@zhouyu-virtual-machine:~/as$
```

3.2 编译过程解释

SCons 是一个开放源码、以 Python 语言编码的自动化构建工具，可用来替代 make 编写复杂的 makefile。并且 scons 是跨平台的，scons 脚本写好后，可以在 Linux 和 Windows 下编译。

SCons 具体语法可参考官方提供文档：https://scons.org/doc/production/PDF/scons-user.pdf

在本实验中，当输入 scons 命令后，命令行窗口会打印出以下信息，展示系统的编译过程

```
zhouyu@zhouyu-virtual-machine:~/as$ export RELEASE=ascore
zhouyu@zhouyu-virtual-machine:~/as$ scons
scons: Reading SConscript files ...
INFO: if the downloading of toolchain is too slow,Ctrl+C and then run below comm
and to set up the toolchain
 mkdir -p /home/zhouyu/as/release/download/i686-elf-tools-linux/bin
 cd /home/zhouyu/as/release/download/i686-elf-tools-linux/bin
 ln -fs /usr/bin/gcc i686-elf-gcc
 ln -fs /usr/bin/ld i686-elf-ld
 echo native > ../../i686-elf-tools-linux.zip
 cd -
INFO: set LWIP_DEBUG=yes to enable debug
scons: done reading SConscript files.
scons: Building targets ...
scons: building associated VariantDir targets: build/posix/x86/ascore
SHLINK build/posix/x86/ascore/release/ascore/elf/dll/libsample.so
scons: done building targets.
```

1) SConstruct

执行 scons 命令时，系统首先会执行根目录下的 **as/SConstruct** 文件，整体执行流程为 PrepareEnv 构造环境 asenv → Conscript 获取要编译文件的集合 → Building 进行文件编译

① 环境变量设置

首先给 sys 添加环境变量

```
1. studio=os.path.abspath('./com/as.tool/config.infrastructure.sys
   tem/')
2. sys.path.append(studio)
```

构造环境 asenv。

```
1. from building import *
```

```
2. asenv = PrepareEnv()
```

from building 是 python 语法从 building 这个包里面加载类，这个包的位置在./com/as.tool/config.infrastructure.system/building.py，所以 PrepareEnv 函数就是在这个 building.py 里面声明的。进入这个函数

```
1. asenv=Environment(TOOLS=['ar', 'as','gcc','g++','gnulink'])
2. os.environ['ASROOT'] = ASROOT
3. asenv['ASROOT'] = ASROOT
4.
5.
6. BOARD = os.getenv('BOARD')
7. if(BOARD not in board_list):
8.       print('Error: no BOARD  specified!')
9.       help()
10.      exit(-1)
11.
12.
13.PrepareBuilding(asenv)
14.return asenv
```

函数首先构造环境 asenv，然后将 ASROOT 变量加入，同时把 ASROOT 也加入至 os.environ 里面。之后的代码检查环境变量中是否存在 BOARD 的设置。函数的最后获取环境变量并返回 asenv。

② 执行 SConscipt 函数，读取 objs

```
objs = SConscript('SConscript',variant_dir=BDIR, duplicate=0)
```

③ 执行编译

```
Building(target,objs)
```

Building 函数中，首先对 objs 中要编译的文件，进行了分类：arxml、xml、py、dts、其他。

2) Sconscript 生成 objs

打开根目录下的 **as/SConscript** 文件，内容如下

```
1. from building import *
2.
3. objs = SConscript('com/SConscript')
4. objs += SConscript('release/SConscript')
5.
6. Return('objs')
```

可以看出，该脚本通过递归调用 SConscript 的方式，将**要编译的文件**加入至 objs 中。对 Building 中不同分类的文件，编译的过程也不相同。

① arxml 生成 LCfg 文件

这个流程也是 **xml 生成 c 源码**的过程，Building 函数中，如下对 arxml 进行了处理

```
1. if( ((not os.path.exists(cfgdone))and (not GetOption('clean')))
2.       or forceGen ):
3.   MKDir(cfgdir)
4.   RMFile(cfgdone)
```

```
5.    xcc.XCC(cfgdir, env, True)
6.    if(arxml != None):
7.        arxmlR = PreProcess(cfgdir, str(arxml))
8.        for xml in xmls:
9.            MKSymlink(str(xml),'%s/%s'%(cfgdir,os.path.basename(
   str(xml))))
10.       xcc.XCC(cfgdir)
11.        argen.ArGen.ArGenMain(arxmlR,cfgdir)
12.   MKFile(cfgdone,  SHA256(glob.glob('%s/*xml'%(cfgdir))))
```

先判断 cfgdone：as/build/posix/x86/ascore/config/config.done 是否存在，若不存在，则按上述代码进行处理：

- 新建文件夹 cfgdir：as/build/posix/x86/ascore/config（生成代码位置）
- xcc.XCC(cfgdir, env, True)函数在 xcc.py 中定义：完成根据 env 里面 MOUDLES 生成宏、根据 env 里面 CONFIGS 生成宏、执行两个可执行文件。上述过程可理解为 python 代码生成.h 代码的过程。
- PreProcess(cfgdir,str(arxml))函数中 arxdml 打印出来为：com/as.application/common/autosar.arxml （xml 文件位置）
- argen.ArGen.ArGenMain 函数在 argen/ArGen.py 中定义：完成找一个简单的模块，对这个文件进行 xml 解析

② DTS、OFS、SWCS 编译

本实验中没有到 dts、ofs 和 swcs 文件，所以 BuildDTS()、BuildOFS()与 BuildingSWCS()函数均没有文件要处理。系统定义默认编译为可执行文件。

3) 生成 TINIX.IMG

Scons 读取完脚本获取 objs 后，就将编译目标文件，改过程由编译器 gcc 完成

```
1. scons: done reading SConscript files.
2. scons: Building targets ...
3. scons: building associated VariantDir targets: build/posix/x86/
   ascore
```

系统会用 gcc 编译器对 objs 里面出现的 c 文件进行编译成 o 文件，最后进行链接为 x86 目标文件。

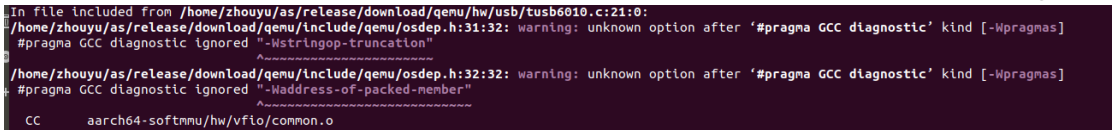
最终生成的 TINIX.IMG 有三部分：boot.bin、loader.bin、x86→kernel.bin

4. AutoSar 代码运行

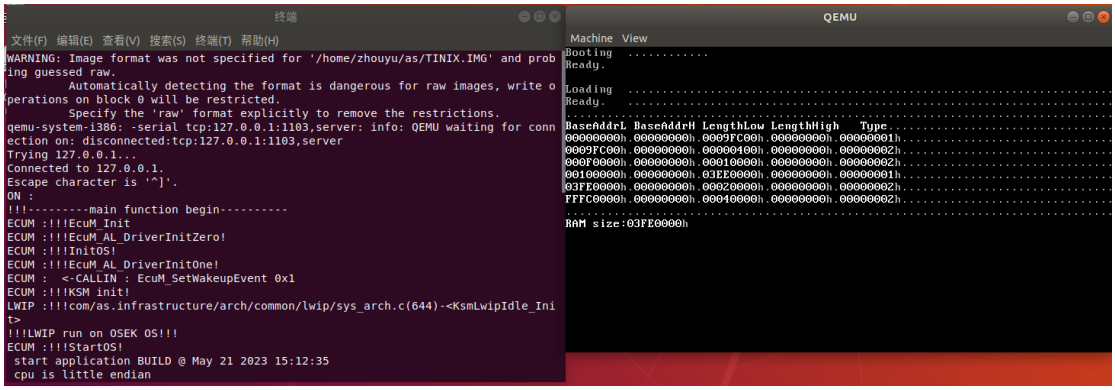
编译完成后就可以运行 qemu 虚拟机加载 AUTOSAR 开源软件。执行命令：

```
1. scons run
```

第一次运行会先**下载编译 Qemu 虚拟机软件**，耗费时间比较长。并会持续打印 warning 信息

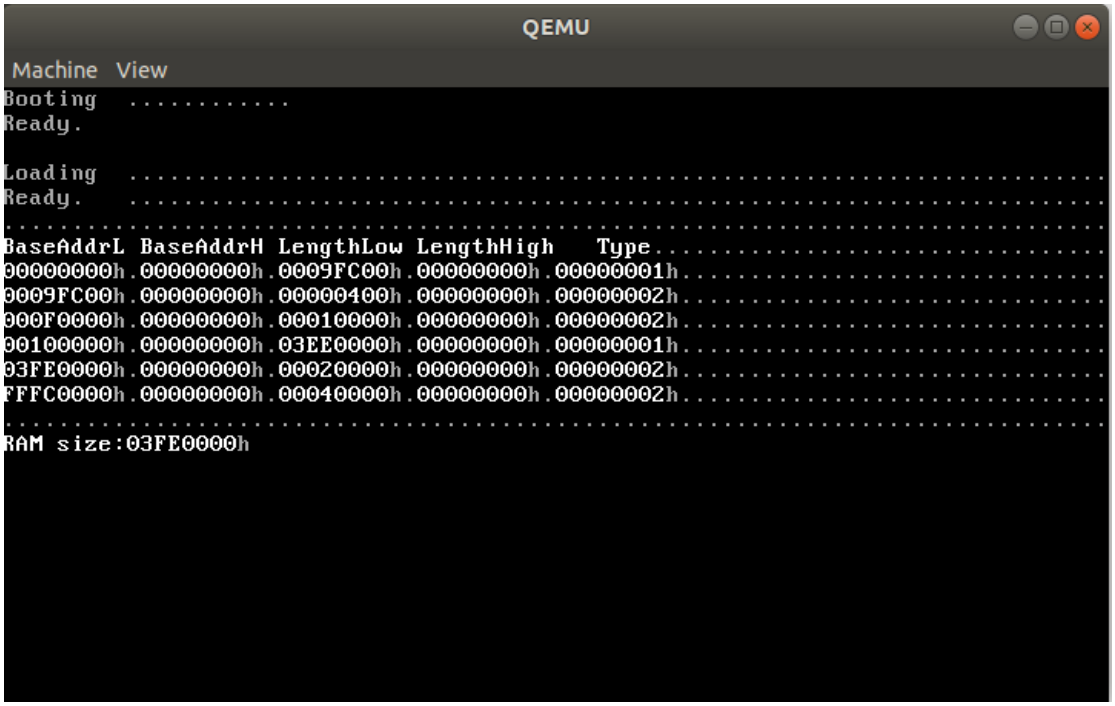


当运行成功后，会出现以下界面

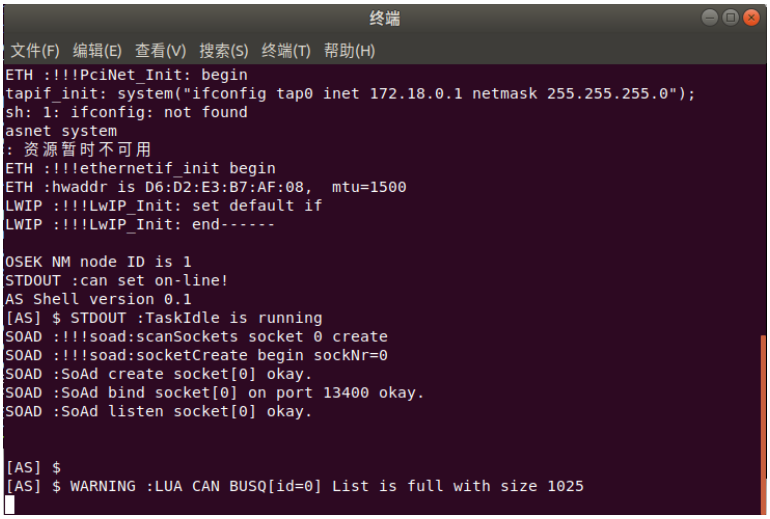


5. 结果分析

Qemu 是纯软件实现的虚拟化模拟器，可以模拟大多数硬件设备，Qemu 充当 Linux 虚拟机认与 AS 代码之间的处理器角色，虚拟机将命令传递给 Qemu 模拟出的硬件，Qemu 将这些指令转译给真正的硬件。



运行后的 Qemu 界面如图所示，此界面不显示多余的信息，具体命令使用与 log 查看还需要到 telnet 界面进行。



在 telnet 界面中，可以使用 help 命令查看使用方式

```
help
help
List of commands:
can               - can info / can write hth canid#data_in_hex_string
lssg              - lssg
wrsg              - wrsg sid value [gid]
ifconfig          - ifconfig
ps                - ps <task/alarm/counter>
kill              - kill pid [sig]
ls                - ls [path]
cd                - cd path
pwd               - pwd
mkdir             - mkdir path
rm                - rm path
cp                - cp file path
cat               - cat file
hexdump           - hexdump file [-s offset -n size]
dll               - dll [path]
help              - help <cmd>
free              - free

exit(0)
[AS] $
```

Ps 命令可以查看进程信息，如 BswService 进程等。

```
[AS] $
[AS] $ ps
ps
Name            State      Prio IPrio RPrio  StackBase  StackSize   Used       Event(set/wait)   Act/ActSum parent
                                                                               list/entry
TaskApp         SUSPENDED  37   37    37     0x00C71BC0 0x00001000  9%(0x0160) 00000000/00000000   0/58740
TaskShell       READY      33   33    33     0x00C72BE0 0x00001000 20%(0x032C) 00000001/00000000   1/7       <-RunningVar
TaskNmInd       WAITING    39   39    39     0x00C73C00 0x00001000  5%(0x00C0) 00000000/0000000F   1/1
TaskLwip        WAITING    41   41    41     0x00C74C20 0x00001000 11%(0x01B0) 00000000/00400000   1/292853
TaskPosix       WAITING    33   33    33     0x00C75C40 0x00001000  4%(0x00A0) 00000000/FFFFFFFF   1/1
TaskCanIf       SUSPENDED  41   41    41     0x00C76C60 0x00001000  9%(0x0160) null                0/6763
TaskIdle        READY       0   32     0     0x00C77C60 0x00001000 13%(0x0210) null                1/1
SchM_Startup    SUSPENDED  39   39    39     0x00C78C60 0x00001000 27%(0x0430) null                0/1
SchM_BswService SUSPENDED  40   40    40     0x00C79C60 0x00001000 19%(0x02F0) 00000000/00000000   0/29374

Name            Status Value      Period     Counter
AlarmApp        start  293735     5          OsClock(293732)
Alarm_Lwip      start  293733     0          OsClock(293732)
Alarm_SIGALRM   stop   0          0          OsClock(293732)
Alarm_BswService start  293742     10         OsClock(293732)

Name
OsClock         Alarm_Lwip(293734) -> AlarmApp(293735) -> Alarm_BswService(293742) ->
exit(0)
```

实验报告

用自己的话给出上述开源 AutoSar 软件的代码框架与代码组成分析，给出基于 SCons 的编译过程理解，附上最终实验运行结果。

报告以“**Automobile_lab1_学号_姓名**”格式命名，每人 1 份，提交至研究生信息系统，截至时间为 2024 年 6 月 12 日。