



T-Head 曳影 1520

# Yocto 用户指南

版本序号	V1.0.0
保密等级	保密
日期	Sept-16-2022

**Copyright © 2022 T-HEAD Semiconductor Co., Ltd. All rights reserved.**

This document is the property of T-HEAD Semiconductor Co., Ltd. This document may only be distributed to: (i) a T-HEAD party having a legitimate business need for the information contained herein, or (ii) a non-T-HEAD party having a legitimate business need for the information contained herein. No license, expressed or implied, under any patent, copyright or trade secret right is granted or implied by the conveyance of this document. No part of this document may be reproduced, transmitted, transcribed, stored in a retrieval system, translated into any language or computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual, or otherwise without the prior written permission of T-HEAD Semiconductor Co., Ltd.

**Trademarks and Permissions**

The T-HEAD Logo and all other trademarks indicated as such herein are trademarks of T-HEAD Semiconductor Co., Ltd. All other products or service names are the property of their respective owners.

**Notice**

The purchased products, services and features are stipulated by the contract made between T-HEAD and the customer. All or part of the products, services and features described in this document may not be within the purchase scope or the usage scope. Unless otherwise specified in the contract, all statements, information, and recommendations in this document are provided "AS IS" without warranties, guarantees or representations of any kind, either express or implied.

The information in this document is subject to change without notice. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, express or implied.

**Copyright © 2022 平头哥上海半导体技术有限公司，保留所有权利。**

本文档的所有权及知识产权归属于平头哥半导体有限公司及其关联公司(下称“平头哥”)。本文档仅能分派给：(i)拥有合法雇佣关系，并需要本文档的信息的平头哥员工，或(ii)非平头哥组织但拥有合法合作关系，并且其需要本文档的信息的合作方。对于本文档，未经平头哥半导体有限公司明示同意，则不能使用该文档。在未经平头哥半导体有限公司的书面许可的情形下，不得复制本文档的任何部分，传播、转录、储存在检索系统中或翻译成任何语言或计算机语言。

**商标申明**

平头哥的 LOGO 和其它所有商标归平头哥半导体有限公司及其关联公司所有，未经平头哥半导体有限公司的书面同意，任何法律实体不得使用平头哥的商标或者商业标识。

**注意**

您购买的产品、服务或特性等应受平头哥商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，平头哥对本文档内容不做任何明示或默示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。平头哥半导体有限公司不对任何第三方使用本文档产生的损失承担任何法律责任。

**平头哥上海半导体技术有限公司 T-HEAD Semiconductor Co., LTD**

地址：中国(上海)自由贸易试验区上科路 366 号、川和路 55 弄 2 号 5 层

网址：[www.t-head.cn](http://www.t-head.cn)

## 版本历史

版本	说明	作者	日期
V1.0.0	初稿	T-Head	Sept-16-2022

# 目录

---

版本历史.....	1
目录.....	2
图表目录.....	4
术语与缩略语.....	5
1 概述.....	6
2 编译环境.....	7
2.1 docker 环境.....	7
2.1.1 安装 docker.....	7
2.1.2 下载 dockerfile.....	7
2.1.3 构建 image.....	8
2.1.4 启动 docker.....	8
2.1.5 登录 docker.....	9
2.1.6 镜像迁移.....	10
2.2 Ubuntu 环境.....	11
3 编译.....	15
3.1 组件.....	15
3.1.1 源码目录.....	15
3.2 Linux 内核.....	16
3.2.1 源码路径.....	16
3.2.2 构建 Linux 内核.....	17
3.2.3 清除 Linux 内核.....	17
3.2.4 menuconfig.....	18
3.3 U-Boot.....	18

---

3.3.1 源码路径.....	18
3.3.2 构建 U-Boot.....	18
3.3.3 清除 U-Boot.....	18
4 添加组件.....	19
4.1 查看组件 .....	19
4.2 增加组件 .....	19
4.3 进入组件目录.....	20
4.4 构建 SYSROOTS SDK.....	20
4.5 Makefile 例子 .....	21
4.6 Cmake 例子 .....	23
5 其他 .....	25

---

# 图表目录

---

## 术语与缩略语

缩略语	英文全名	中文解释

# 1 概述

---

本文将主要介绍 Yocto 的基本用法，指导用户基于 Yocto 完成日常开发。

## 2 编译环境

Yocto 编译环境使用 Ubuntu 18.04，推荐使用 Linux + docker 的方式部署，因特殊原因无法使用 docker 的，请参考 Ubuntu 环境章节。

### 2.1 docker 环境

请先安装 Linux 基础 OS，用户根据自己的需要选择 Ubuntu、Centos 等 Linux 发行版本，具体安装方法本文不做详细介绍。安装完成后，在此 OS 的基础上继续安装 docker，然后构建 docker image，将相关的开发环境构建在 docker 里面，后续的开发都基于 docker 内的环境进行，具体的构建方式如下。

#### 2.1.1 安装 docker

使用官方安装脚本自动安装。

```
▼ Bash | 复制代码  
1 curl -fsSL https://get.docker.com | bash -s docker --mirror Aliyun
```

#### 2.1.2 下载 dockerfile

点击下载 [linux-dev-master.7z](#) 解压后进入 linux-dev-master 目录，打开 Dockerfile，修改用户名和 ID，“your the same user name as your host” 修改成用户 host os 的用户名：

```
▼ Bash | 复制代码  
1 ENV DOCKER_USER2 "your the same user name as your host"
```

“your user id” 的值记录在 /etc/passwd，打开后搜索自己的用户名所在的行，anonuid 字段的值即是对应的 ID。

```
▼ Bash | 复制代码  
1 ENV USER2_ID "your user id"
```

### 2.1.3 构建 image

运行以下命令构建自己的环境。

```
docker build -t linux-dev-base:base 。
```

这个 docker 镜像可以编译 thead 发布的 buildroot、yocto 等 Linux SDK。默认密码为 123。

查看构建的 dokcer image，正常情况下可以看到如下结果：

```
▼ Bash | 复制代码  
1 @:~$ docker images  
2 REPOSITORY      TAG          IMAGE ID      CREATED        SIZE  
3 linux-dev-base  base        32207ad97b7a  12 minutes ago 2.13GB  
4 ubuntu          18.04       886eca19e611  2 weeks ago   63.1MB
```

### 2.1.4 启动 docker

docker 启动可以使用 docker run 命令。

```
▼ Bash | 复制代码  
1 docker run -u thead -dt --name linux-dev-{your_name} -v  
  {your_lock_home}:{your_home} linux-dev-base:base /bin/bash
```

**注意：**可以通过-v 选项挂载 host 的一个或多个目录，起到类似共享文件的作用，其中：your\_name: docker container 名称，起一个自己的名字，不要和别人重名；your\_lock\_home: host 本地路径；your\_home: 本地路径在 docker 里的 mount 路径。

查询启动的 docker container:

```
▼ Bash | 复制代码  
1 docker ps |grep linux-dev-base
```

正常情况下能看到：

```
▼ Bash | 复制代码  
1 @~$ docker ps |grep linux-dev-base  
2 CONTAINER ID IMAGE COMMAND CREATED  
STATUS PORTS NAMES  
3 017e0217cab0 linux-dev-base:base "/bin/bash" 3 minutes ago Up 3  
minutes 22/tcp linux-dev
```

## 2.1.5 登录 docker

执行如下命令登录 docker。

```
▼ Bash | 复制代码  
1 docker exec -it linux-dev-{your_name} /bin/bash
```

此时已 ssh 登录到 docker 里的 Ubuntu 18.04，运行以下命令查询系统版本：

```
▼ Bash | 复制代码  
1 thead@017e0217cab0:/$ lsb_release -a No LSB modules are available.  
Distributor ID: Ubuntu Description: Ubuntu 18.04.6 LTS Release:  
18.04 Codename: bionic
```

在启动 docker 一节中，对 host 本地 \$HOME 目录和 docker guest \$HOME 做了映射，因此在 docker 的 home/thead 目录下能看 host \$HOME 目录下的所有内容，并且所有在 docker 该目录下的文件与 host 完全同步，docker 关闭或删除都不受影响。

## 2.1.6 镜像迁移

个别环境下，因网络受限等原因，无法从零开始构建完整 docker image，此时可以通过 docker save 和 docker load 命令完成镜像迁移的过程，即在一台网络不受限制的 PC 先构建好 image，再迁移到目标机器上。

具体步骤为先将镜像保存为压缩包，然后在其他位置再加载压缩包。

在网络不受限制的 host 上执行以下命令生成 docker image：

```
▼ Bash | 复制代码  
1 cd linux-dev-master docker build -t linux-dev-base:base .
```

查看生成的 docker image，确认 image 生成。

```
▼ Bash | 复制代码  
1 docker images linux-dev-base:base
```

开始保存 image。

```
▼ Bash | 复制代码  
1 docker save linux-dev-base:base | gzip >linux-dev-base:base.tar.gz
```

这步后可以在命令执行的当前目录下生成 linux-dev-base:base.tar.gz。

将 tar 包拷贝 linux-dev-base:base.tar.gz 到目标机上，然后用 docker load 命令导入 image。

```
▼ Bash | 复制代码  
1 docker load -i linux-dev-base\:base.tar.gz
```

此时在迁移目标机上就可以看到 image 了。

```
docker images linux-dev-base:base
```

迁移完毕后，就可以启动 docker container 了，比如：

Bash |  复制代码

```
1 sudo docker run --network=host -u $user -dt --name $user_docker-ubuntu18
-v /home/$user:/home/$user linux-dev-base:base /bin/bash
```

再执行：

Bash |  复制代码

```
1 sudo docker exec -it $user_docker-ubuntu18 /bin/bash
```

## 2.2 Ubuntu 环境

本章介绍不基于 docker 环境的编译环境搭建方法。

首先安装 Ubuntu 18.04，安装方法请参考 Ubuntu 官方网站，不再赘述。

安装编译 Yocto 所需的依赖包

```
Bash | 复制代码
1  sudo apt-get update
2  sudo apt-get install -y --assume-yes --no-install-recommends apt-utils
3  sudo apt-get install -y --assume-yes --no-install-recommends automake
   \
4   autotools-dev \
5   axel \
6   bash \
7   bash-completion \
8   bc \
9   bison \
10  build-essential \
11  busybox \
12  ccache \
13  chrpath \
14  cpio \
15  curl \
16  debianutils \
17  device-tree-compiler \
18  diffstat \
19  exuberant-ctags \
20  fakeroot \
21  file \
22  flex \
23  g++ \
24  g++-multilib \
25  gawk \
26  gcc \
27  gcc-multilib \
28  git \
29  gnupg \
30  gperf \
31  iputils-ping \
32  less \
33  libglib2.0-0 \
34  libncurses-dev \
35  libncurses5-dev \
36  libssl-dev \
37  locales \
38  lsb-release \
39  netcat-openbsd \
40  nfs-common \
41  openssh-server \
42  pkg-config \
43  procps \
44  pv \
```

```
45     python-pip \  
46     python3-pip \  
47     rsync \  
48     scons \  
49     socat \  
50     sshfs \  
51     sudo \  
52     texinfo \  
53     tig \  
54     tmux \  
55     tree \  
56     tzdata \  
57     unzip \  
58     vim \  
59     wget \  
60     x11-apps \  
61     xz-utils \  
62     zip \  
63     zlib1g-dev \  
64     zsh \  
65     libevent-dev \  
66     libgnutls-dane0 \  
67     libgnutls-openssl27 \  
68     libgnutls28-dev \  
69     libgnutlsxx28 \  
70     libidn2-dev \  
71     libjsoncpp-dev \  
72     liblog4cpp5-dev \  
73     libopus-dev \  
74     libunistring-dev \  
75     libz-dev \  
76     nettle-dev \  
77     lib32ncurses5-dev \  
78     lib32z-dev \  
79     libaio-dev \  
80     libattr1-dev \  
81     libbluetooth-dev \  
82     libbrlapi-dev \  
83     libc6-dev-i386 \  
84     libcap-dev \  
85     libgl1-mesa-dev \  
86     libgl2.0-dev \  
87     liblzo2-dev \  
88     libnuma-dev \  
89     libpixmap-1-dev \  
90     libsnappy-dev \  
91     libssl-dev \  
92     libvdeplug-dev \  

```

```
93     libx11-dev \  
94     libxml2-utils \  
95     openjdk-8-jdk \  
96     x11proto-core-dev \  
97     xsltproc \  
98     libclang-dev \  
99     cmake \  
100    libvulkan-dev \  
101    gtk-update-icon-cache \  
102    module-init-tools \  

```

## 3 编译

本章节介绍 Yocto 使用过程中会用到的常用技巧，如需系统性了解 Yocto 知识可以参考如下手册：

- SDK 提供的中文手册《[Yocto 开发手册](#)》
- 英文版《[BitBake 用户手册](#)》
- 英文版《[Yocto 工程参考手册](#)》

### 3.1 组件

Yocto 以 package 为单位管理海量的开源软件组件，如需编译某个 package，要编译某个 package，方法如下：

```
▼ Bash | 复制代码  
1 bitbake "package-name"
```

每个 package 都会在自己的 recipes 文件中定义支持哪些 task，列出本 package 支持的所有 task 以及 help 信息：

```
▼ Bash | 复制代码  
1 bitbake "package-name" -c listtasks
```

#### 3.1.1 源码目录

Yocto 集成了大量的开源 package，这些 package 编译的时候的工作目录通常在以下目录：

- tmp-glibc/work/riscv64-oe-linux
- tmp-glibc/work/\${MACHINE}

以 gnome-shell 为例，目录结果通常如下：

```
▼ Bash | 复制代码
1  docker-ubuntu18:thead-build/light-fm/tmp-glibc/work/riscv64-oe-
   linux/gnome-shell$ tree -L 2
2
3  └─ 3.34.5-r0
4     │   └─ xxx.patch
5     │   └─ build
6     └─ debugsources.list
```

其中

- gnome-shell-3.34.5: 源码目录
- image: 编译输出
- build: 编译目录

查询 package 目录的方法:

```
▼ Bash | 复制代码
1  bitbake -e your_package _name | grep ^S=
```

## 3.2 Linux 内核

### 3.2.1 源码路径

Yocto 在编译的时候将源码下载到如下路径下:

```
▼ Bash | 复制代码
1  tmp-glibc/work/d1-oe-linux/linux-thead
```

以 5.10 内核为例, 编译过一次内核后, 会看到如下目录结构, 其中:

- linux-5.10.y: 内核源码, 带 git 信息
- linux-light\_fm\_linux-standard-build: 编译中间文件等

- temp: 编译过程中日志
- image: 安装到文件系统的文件

```
▼ Bash | 复制代码
```

```
1  └─ 5.10.y-r0
2     └─ defconfig
3     └─ deploy-debs
4     └─ deploy-linux-thead
5     └─ image
6     └─ license-destdir
7     └─ linux-5.10.y
8     └─ linux-light_fm_linux-standard-build
9     └─ package
10    └─ packages-split
11    └─ pkgdata
12    └─ pkgdata-pdata-input
13    └─ pkgdata-sysroot
```

### 3.2.2 构建 Linux 内核

一般修改 Linux 内核源码后，只需要执行该命令即可。

```
▼ Bash | 复制代码
```

```
1  $ bitbake linux-thead -C compile
```

### 3.2.3 清除 Linux 内核

该命令会清除整个 Linux 内核的构建目录，通常不需要执行。

```
▼ Bash | 复制代码
```

```
1  $ bitbake linux-thead -c clean
```

**注意：**clean 会清除 temp/work 下的内核源码。

### 3.2.4 menuconfig

```
▼ Bash | 复制代码  
1 $ bitbake linux-thead -c menuconfig
```

更多命令可通过 `bitbake linux-thead -c listtasks` 查看。

## 3.3 U-Boot

### 3.3.1 源码路径

Yocto 在编译的时候将源码下载到如下路径下，其中倒数第二级目录名称为 U-Boot 版本号。

```
▼ Bash | 复制代码  
1 tmp-glibc/work/d1-oe-linux/u-boot/1_2018.05-r0/git
```

### 3.3.2 构建 U-Boot

一般修改 U-Boot 内核源码后，只需要执行该命令即可。

```
▼ Bash | 复制代码  
1 $ bitbake u-boot -C compile
```

### 3.3.3 清除 U-Boot

该命令会清除整个 U-Boot 内核的构建目录，通常不需要执行。

```
▼ Bash | 复制代码  
1 $ bitbake u-boot -c clean
```

更多命令可通过 `bitbake u-boot -c listtasks` 查看。

## 3.4 构建镜像

构建命令如下：

```
MACHINE=light-a-public-release bitbake light-fm-image-linux
```

MACHINE 支持列表：

light-a-public-release: 曳影 1520 EVB-A 验证板

light-b-public: 曳影 1520 EVB-B 验证板

light-beagle: beagle-board 开发板

## 3.5 镜像打包

预发布的镜像包含了 light-a-public-release, light-b-public, light-beagle 的多种镜像，为了简化烧写镜像包的打包流程，可以用如下仓库的脚本完成相关工作。

```
cd ~/xuantie-yocto/build/light-fm/sdk
```

```
./sdk.sh
```

对于一般开发者，如果用于镜像烧写，可以减少一些打包行为，只需执行：`./sdk.sh no-deb no-tarball`

# 4 添加组件

## 4.1 查看组件

如查看名字中带有“perf”的组件是否存在：

```
▼ Bash | 复制代码  
1 $ bitbake-layers show-recipes *perf*
```

## 4.2 增加组件

比如要在镜像中增加 perf 命令，方法如下：

第一步：在 image 对应的 bb 文件中增加配置。

例如在 `d1-image-miniapp-dev image` 中增加 `perf` 工具，则：

```
Bash | 复制代码
1  $ vi meta-d1/recipes-core/images/d1-image-miniapp-dev.bb
2
3  IMAGE_INSTALL += " perf "
```

第二步：执行“构建镜像”命令，方法见本文 3.4 章节。

第三步：执行“打包”命令，方法见本文上 3.5 章节。

## 4.3 进入组件目录

增加组件后，如果需要查看组件源码，可以通过如下命令进入组件的构建目录：

```
Bash | 复制代码
1  $ bitbake perf -c devshell
```

## 4.4 构建 SYSROOTS SDK

通过本章节构建出来的 SDK，内包含工具链，是用来开发 Linux 应用程序，比如开发运行在 Linux 上的 `helloworld`，就需要这个 SDK。该 SDK 在提供的 SDK 压缩包中已经存在，一般用户直接使用即可，不需要操作本章节步骤。

由于 Yocto 版本有缺陷，构建 SDK 前需要先删除 `coreutils`，构建完成后再把其改回来。在 `openembedded-core` 目录下，修改如下（第 9 行）：

```
▼ Bash | 复制代码

1  diff --git a/meta/recipes-core/meta/target-sdk-provides-dummy.bb
    b/meta/recipes-core/meta/target-sdk-provides-dummy.bb
2  index e3beeb796c..765611ffda 100644
3  --- a/meta/recipes-core/meta/target-sdk-provides-dummy.bb
4  +++ b/meta/recipes-core/meta/target-sdk-provides-dummy.bb
5  @@ -4,7 +4,6 @@ DUMMYPROVIDES_PACKAGES = "\
6     busybox \
7     busybox-dev \
8     busybox-src \
9  -   coreutils \
10    coreutils-dev \
11    coreutils-src \
12    bash \
```

例如，构建 d1-image-miniapp-dev sysroots SDK：

```
▼ Bash | 复制代码

1  $ bitbake d1-image-miniapp-dev -c populate_sdk
```

构建完成后，生成的 SDK 在如下目录，使用方法可见 D1 Linux 快速上手手册。

```
▼ Bash | 复制代码

1  thead-build/d1-miniapp/tmp-glibc/deploy/sdk/
```

## 4.5 Makefile 例子

使用 `tree` 可以看到，其有一个 `bb` 文件，然后其中还有一个目录放着 `Makefile` 与 `source code`：

```
▼ Bash | 复制代码
1 meta-thead/recipes-examples/hellomakefile$ tree
2 .
3 |— hello
4 |   |— helloYocto.c
5 |   |— makefile
6 |   |— zlibtest.c
7 |— hello.bb
8 |— README.md
9
10 1 directory, 5 files
```

其中的 bb 文件内容如下：

```
▼ Bash | 复制代码
1 DESCRIPTION = "Hello World and Zlib test"
2 DEPENDS = "zlib"
3 SECTION = "libs"
4 LICENSE = "MIT"
5 PV = "3"
6 PR = "r0"
7
8 SRC_URI = " \
9     file://helloYocto.c \
10    file://zlibtest.c \
11    file://makefile \
```

可以看到，bb 文件中指定了下面几个变量的值：

- SRC\_URI：定义源码。
- LIC\_FILES\_CHKSUM：这个是 checksum，如果是基于像 git 与 svn 的版本管理 source，则不需要定义。
- FILES\_\${PN}：PN 是 Package number，指代软件版本使用的 PV 与 PR 结合表示，即前面 bitbake -s 中看到的 3-r0。

除变量外，该 bb 文件重载了有两个方法：

- do\_compile
- do\_install

这两个方法，对应了 task list 中的 compile 与 install task。

添加软件包之前，确认好以下信息：

- 原始软件在哪里，git 仓库的地址、分支、revision，tar 包的地址、sum 值。
- 是否有额外的 patch、配置文件？
- 使用哪种方式编译，makefile、cmake、meson、ninja、脚本等，确保已经可以单独编译成功。
- 编译的产物是什么，要放到哪里？

## 4.6 Cmake 例子

以下是一个基于 cmake 的例子，旨在 demo 如何在 Yocto 下，如何集成基于 cmake 的 package。

```
▼ Bash | 复制代码
```

```
1 meta-thead/recipes-examples/hellocmake$ tree
2 .
3 |— files
4 |   |— A
5 |   |   |— CMakeLists.txt
6 |   |   |— main.c
7 |   |— B
8 |   |   |— add.c
9 |   |   |— add.h
10 |   |   |— CMakeLists.txt
11 |   |— C
12 |   |   |— CMakeLists.txt
13 |   |   |— sub.c
14 |   |   |— sub.h
15 |   |— CMakeLists.txt
16 |— hellocmake.bb
17 |— README.md
18
19 4 directories, 11 files
```

本例子功能如下：

B: c 语言编写的加法函数，编译成动态库，供 C 和 A 使用，即编译出动态库的例子。

C: c 语言编写的减法函数，以及调用 B 中的减法函数例子，即动态库调用动态库的例子。

A: c 语言编写的可执行程序例子，会调用 B 和 C 编译出的动态库，以及一个 ffmpeg 的例子。

编译方法：

Bash |  复制代码

```
1  biatake hellocmake
```

---

## 5 其他

---

其他关于 Yocto 的高级用法，请参考《[Yocto 使用技巧](#)》。