TM V2

White Paper

|  |  |  |  |
| --- | --- | --- | --- |
| 版本 | 日期 | 作者 | 内容 |
| V1.0 | 16-06-02 | Ivan Gan | 初次发行 |
| V1.1 | 16-06-02 | Ivan Gan | 增加消息格式作为文稿第一部分 |
| V1.2 | 16-06-15 | Ivan Gan | 增加系统架构和各文件说明，重新编排消息 |
| V1.3 | 16-06-17 | Ivan Gan | 增加dylib制作说明供Lua加载调用 |
| V1.4 | 16-06-29 | Ivan Gan | 增加常用State Machine控制说明 |
| V1.5 | 16-08-10 | Ivan Gan | 增加产品条码处理流程说明 |
| V1.6 | 16-08-30 | Ivan Gan | 增加startup.lua说明 |
|  |  |  |  |
|  |  |  |  |

目录

TM V2 1

第一部分：架构 3

最小系统 4

终端运行 4

文件布局 5

基本说明 6

StartUp.lua 7

项目修改 8

第二部分：内容说明 8

Test Plan 8

变量申明 9

zmqport.json 9

LuaDriver/Driver/utils 10

libGlobal.dylib（节选） 11

第三部分：ZMQ消息格式 11

Sequencer Publish 11

Sequencer RPC 13

Engine Publish 15

Engine RPC 16

State Machine 16

State Machine Publish 16

State Machine RPC 17

RPC链接 18

第四部分：消息实例 18

消息序列图 18

启动消息 19

测试中消息 19

第五部分：State Machine 23

状态图 23

控制框架 23

脚本 24

第六部分：条码处理 24

附录：dylib制作 24

# 第一部分：架构



系统中，所有模块将以独立进程运行，待测物(UUT / DUT)为N个时，将会有N个sequecer进程，N个test\_engine进程，一个State Machine进程，一个logger进程。所有进程间测通讯通过ZMQ完成。

* Sequencer：负责csv的testplan的解析，向engine发送测试请求，并获取engine的应答，判断测试结果和最终测试状态的维护。
* Test\_engine：产品通讯和产品测试，接受sequencer的函数请求并通过function\_table.lua解析执行各个驱动函数，返回测试数值。
* State Machine：测试系统状态维护，如闲置状态，测试中等，同时，state machine还控制夹具动作，触发sequencer执行测试。

## 最小系统



完整的最小系统包括state machine，sequencer和test engine。

## 终端运行

下例中只启动第一通道的系统，如果通过改变，只需相应改变后面的数值(从0开始)

1. 启动test\_engine

将路径切至LuaDriver所在路径，如下：

*cd /Users/IvanGan/Documents/IAGit/TestManager/LuaDriver*

*lua test\_engine.lua -u* ***0***

1. 启动sequencer

*export PYTHONPATH=/Users/IvanGan/Documents/IAGit/TestManager/python-sequencer*

*cd /Users/IvanGan/Documents/IAGit/TestManager/python-sequencer/x527*

*python sequencer/sequencer.py -s* ***0***

1. 启动state machine

*cd /Users/IvanGan/Documents/IAGit/TestManager/LuaDriver*

*lua StateMachine/StateMachine.lua*

## 文件布局







## 基本说明

1. laucher.app将会根据TesterConfig/Config.py运行加载启动和结束进程，如下为样例，可以添加脚本修改实际所需的startup table。

*startup = {*

*"project\_name":"X527",*

*"modules":1,*

*"slots":1,*

*"processes":{*

*1:{ #must*

*"execute":"C:\\lua\\bin\\lua", #must; Absolute path*

*"path":work\_path, #must; Absolute path*

*"bin":"TestEngine\\test\_engine.lua", #must*

*"args":"-u slot", #optional*

*"sleep":3 #second; optional*

*},*

*2: {*

*"path":work\_path,*

*"bin":"sequencer",*

*"args": "-s slot -c",*

*"sleep":1, #second*

*},*

*3:{*

*"execute":"C:\\lua\\bin\\lua",*

*"path":work\_path,*

*"bin":"StateMachine\\StateMachine.lua",*

*"args":"",*

*"sleep":1 #second it can be excluded if 0*

*},*

*4:{*

*"execute":"C:\\lua\\bin\\lua",*

*"path":work\_path,*

*"bin":"Datalog\\datalog.lua",*

*"args":"",*

*"sleep":1 #second it can be excluded if 0*

*},*

*}*

*}*

*#Add your customized code here*

*def get\_config():*

*return startup*

1. 启动存在先后顺序，需先启动test engine，然后启动sequencer，sequencer在启动时会询问engine状态，如果engine没有正常启动，sequencer将会自毁。
2. test\_engine.lua 会依据入口function\_table.lua顺序加载各脚本和驱动。
   * 当在/LuaDriver/Driver/functions/下有定义calibration.lua时，engine在启动时将自动运行calibration.lua。此步可以作为硬件校准参数载入使用。
   * engine启动过程中将会自动执行reset函数，所以function\_table.lua里请定以reset函数，一般为复位夹具。
   * Test\_engine在接收到sequencer的REQ请求时，将params解析为字典作为参数{param1=PARAM1,param2=PARAM2}传入功能函数。因此功能函数书写时，只需定义一个table参数输入。
3. sequencer在接收到run的REQ时(请求内可带入产品条码，一般来自state machine，SM的内存储的产品条码一般来自GUI(RPC请参考第三部分))，将会执行回调函数start\_test，然后才执行testplan里定义的测试序列，最后执行回调函数end\_test. end\_test可以返回以逗号分隔的完整路径，Sequencer接受此路径后会pub给logger上传数据库。
4. function\_table.lua，里面必须定义start\_test, end\_test, 和reset，即使不实现任何功能。

最基本的Function\_table.lua如下：

*local function\_table = {*

*reset = require("functions.reset").reset,*

*start\_test = require("functions.callback").Test\_OnEntry,*

*end\_test = require("functions.callback").Test\_OnDone,*

*}*

*return function\_table*

## StartUp.lua

Engine最先加载执行的文件为StartUp.lua。

*local \_s = {}*

*\_s.data = nil --this should be string if there is any data*

*\_s.function\_table = nil -- engine will load this fucntion\_table, if nil, will load default "function\_table"*

*return \_s*

此表格中定义两变量，

： data为数据，在engine启动开始测试时，会将数据pub

： function\_table为engine将加载的驱动入口文件和对Sequencer的定义文件。用户可根据各工站要求加载不同的驱动

如：

*local \_s = {}*

*\_s.data = require(“calibration.lua”)*

*local station = parseStation()*

*if station = “FCT” then*

*\_s.function\_table = “function\_table\_fct”*

*else*

*\_s.function\_table = “function\_table\_ict”*

*return \_s*

## 项目修改

在根据项目要求，一般需要修改和增加的脚本或者lib分别是：

1. /LuaDriver/Driver/functions/下添加功能函数脚本
2. /LuaDriver/Driver/function\_table.lua添加函数接口申明供test\_engine.lua调用

* 在实际运用中，engine在顺序加载function\_table.lua里的脚本驱动时可能会造成lua崩溃，所以在写各脚本驱动时请保证各自的健壮性，如各脚本中使用local变量，避免使用全局变量等

1. /LuaDriver/Driver/lib/，如果必要，/functions/下的某些功能函数的驱动lib放置于此

# 第二部分：内容说明

## Test Plan

profile文件夹下放置testplan.csv

***test plan 命名规则是name\_\_version.csv.***

test plan 包含的内容有：

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **GROUP** | TID | DESCRIPTION | FUNCTION | TIMEOUT | PARAM1 | PARAM2 | UNIT | LOW | HIGH | KEY | VAL |

* GROUP:测试项目的组别
* TID:测试项目的id，需唯一性
* DESCRIPTION:测试项目描述
* FUNCTION:测试项目的功能函数，须与/LuaDriver/Driver/function\_table.lua里定义的函数一致，否则不能够被执行
* 如果被<<>>定义，Sequencer将会调用系统python路径下的fctplugin的模块内函数。
* TIMEOUT:执行功能函数的超时时间
* PARAM1:参数1，此值将会与PARAM2合成数组被sequencer通过REQ发送到test\_engine.
* PARAM2:参数2，此值将会与PARAM1合成数组作为params被sequencer通过REQ发送到test\_engine.
* Test\_engine在接收到params后，将其解析为字典作为参数{param1=PARAM1,param2=PARAM2}传入功能函数。因此功能函数书写时，只需定义一个参数输入。
* UNIT:测试项目执行结果的单位。
* LOW:测试项目的下限
* HIGH:测试项目的上限
* KEY:关键字
* VAL:关键字值，
* KEY和VAL实现测试项目的skip或条件执行。
* KEY和VAL为空时，默认为执行
* KEY定义了变量{{}}(此定义必须在PARAM2中已定义，请参考变量申明)时，sequencer将会将变量值与VAL比较，如果相等则执行测试项目，否则skip

## 变量申明

test plan里允许申明变量，且不同表示方式在PARAM1和PARAM2里出现，sequencer内处理方式各不相同。

|  |  |  |
| --- | --- | --- |
| 标示 | PARMA1 | PARAM2 |
| [[]] | 此变量将会被替换成实际值后发送到Engine。在此前，该值必须被初始化或赋值 | 期望被Engine返回的值。Sequencer将会以此变量名保存Engine返回的值 |
| {{}} | 这个此出现是为parse函数解析用，是模板文件中的关键字 | 作用和[[]]一致，同时，将会作为测试项目(data)上传pdca |
| <<>> | 不应该在此出现 | 作用和[[]]一致，同时，将会作为属性(attribute)上传pdca |

## zmqport.json

定义ZMQ端口和地址。Engine 和 sequencer等所有可能需要建立ZMQ的需从此文件解析获取端口。文件内定义为json格式。

{ "TEST\_ENGINE\_PUB": 6150, *//engine的PUB端口*

"TE\_PROXY\_PUB": 6300,

"SEQUENCER\_PORT": 6200, *//Sequencer的REP端口*

"TEST\_ENGINE\_PORT": 6100, *//engine的REP端口*

"SEQUENCER\_PUB": 6250, *//Sequencer的PUB端口*

"DEBUG\_LOGGER\_PORT": 6700,

"SEQUENCER\_PROXY\_PUB": 6350,

"PUB\_PORT": 50,

"LOGGER\_PUB": 6900, *//Logger的PUB端口*

"PUB\_CHANNEL": "101",

"SM\_PUB": 6880, *//State Machine的PUB端口*

"SM\_PORT":6480, *//State Machine的REP端口*

"SM\_HEARTBEAT":6580*//State Machine的Heart beat端口,最后一行没有逗号*

}

可以根据实际情况继续增加或者修改端口

解析：LuaDriver/Driver/ZmqPortParse.lua是此文件的解析函数，

例：

*local zmqport = require "utils.ZmqPortParse"*

*SEQUENCE\_BASE\_PORT = zmqport.zmqport("TEST\_ENGINE\_PORT")*

## LuaDriver/Driver/utils

* EnginePub.lua

通过engine的PUB端口发布消息，在其他脚本里也可调用。此处仅列举使用说明，PUB的消息具体格式，请参考Enigne PUB章节

例：

*local enginepipe = require("utils.EnginePub")*

*enginepipe.send("fixture.spam.lua > Fixture Port connect successfully")*

* sequence\_utils.lua

提供数值上下限比较，字符串搜索和比较，最常用的为单位转换

例：

*local sequence\_utils = require("utils.sequence\_utils")*

*voltage = sequence\_utils.convert\_units(3.15, “V”, "mV")*

* time\_utils.lua

提供有延时，获取绝对时间，获取时间字符串(精确至毫秒%Y-%m-%d %H:%M:%S.mmm)

* uuid.lua

uuid,唯一识别码,如0021c175-8838-4cad-ccef-6284d4080d87

例：

*local uuid = require("utils.uuid")*

*print(uuid())*

## libGlobal.dylib（节选）

常用的函数有：

进程互斥锁（通过文件锁实现）

int LockInstrument(const char \* szLockName);

int UnlockInstrument(const char \* szLockName);

int UnlockAllInstrument();

数据转换

const char \* floatConversion(float value);

float floatToNumber(char \* dataStr);

const char\* floatToFixed(double num);

double fixedToNumber(char \* dataStr);

SFC通讯

const char\* putSFC(char\* szHttp);

例：

*require "libGlobal"*

*LockInstrument(“test”)*

*UnlockInstrument(“test”)*

*floatConversion(0.93768596858663) -- 0x3F700C30*

*floatToNumber(“0x3F700C30”) -- 0.93768596858663*

# 第三部分：ZMQ消息格式

## Sequencer Publish

Sequencer,Engine,StateMachine,Logger都有通过各自的Publisher通道101广播心跳

1. 发布者(Publisher)的发布消息都是通过”101”通道发布，订阅者(Subscriber)可以设置属性获取信息：(Lua中的设置如下)

local subscriber, err = ctx:socket{zmq.SUB,subscribe = "101";connect = address;}

1. 发布者发布的消息有以下四部分内容

|  |  |  |
| --- | --- | --- |
| 内容 | 键 | 解释 |
| 时间 | 无 | 发布者发布信息的时间，格式是mm-dd\_hh:mm:ss.milisecond。如06-02\_16:08:27.163768 |
| 等级 | 无 | REPORTER:0, CRITICAL:1, INFO:2, DEBUG:3 |
| 发布者 | 无 | 发布者的身份标明 |
| 数据 | data | 发布的数据，定义为json格式数据，或FCT\_HEARTBEAT |

Lua中的publish的实现如下：

*watchdog\_zmq:send\_all{"101",time\_utils.get\_time\_string\_ms(),3,"TestEngine","FCT\_HEARTBEAT"}*

*watchdog\_zmq:send\_all{"101",time\_utils.get\_time\_string\_ms(),3,"TestEngine",’{"data": {"result": 1, "logs": ""}, "event": 1}’}*

以下将解释Sequencer的上表格中”数据”的内容(Json数据)。

Sequncer发布的消息有7⃣种，分别是SEQUENCE\_START(系列开始),SEQUNCE\_END(序列结束)，ITEM\_START(项目开始)，ITEM\_FINISH(项目结束)，ATTRIBUTE\_FOUND(属性获取，主要用于以属性上传PDCA)，REPORT\_ERROR(错误)，UOP\_DETECT(Unit of process)。事件(“event”)值对应为：

* SEQUENCE\_START = 0
* SEQUENCE\_END = 1
* ITEM\_START = 2
* ITEM\_FINISH = 3
* ATTRIBUTE\_FOUND = 4
* REPORT\_ERROR = 5
* UOP\_DETECT = 6

各自发布的数据格式如下：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 类型 | 数值 | 键 | 内容 | 举例(Json数据串) |
| SEQUENCE\_START | 0 | name | 测试testplan文稿名称 | {“data”: {“version”: “0531d 2”, “name”: “dfu”}, “event”: 0} |
| version | 测试testplan文稿版本 |
| SEQUENCE\_END | 1 | result | 最终测试结果1：pass，0：fail，-1：中止 | {"data": {"result": 1, "logs": ""}, "event": 1} |
| error | 错误信息，只有result=-1时才出现 |
| ITEM\_START | 2 | group | 测试项目的组 | {"data": {"group": "DCSD\_CHECK", "description": "BATT\_POWER\_ON", "to\_pdca": true, "high": "", "low": "", "tid": "BATT\_POWER\_ON", "unit": ""}, "event": 2} |
| tid | 测试项目的id |
| unit | 单位 |
| low | 下限 |
| high | 上限 |
| pdca | 是否上传pdca，true或false |
| ITEM\_END | 3 | tid | 测试项目id | {"data": {"tid": "BATT\_POWER\_ON", "to\_pdca": true, "result": true, "value": "--PASS--"}, "event": 3} |
| value | 测试值 |
| result | 测试结果,true, false或-1 |
| error | 错误信息,result=-1时才会出现此键 |
| pdca | 是否上传pdca |
| ATTRIBUTE\_FOUND | 4 | name | 属性名 | {"data": {"name": "BOARD\_REV", "value": "0x0C"}, "event": 4} |
| value | 属性值 |
| REPORT\_ERROR | 5 | site | 通道 | {“data”: {“error\_code”: -10005, “site”: 0, “error\_msg”: “AMIOK ERROR: class:32 id:31; message:Invalid Signature: signature could not be verified.”}, “event”: 5} |
| error\_code | 错误代码 |
| error\_msg | 错误信息 |
| UOP\_DETECT | 6 | result | UOP结果，true,false,-1 | {"data": {"error": "AMIOK ERROR: class:32 id:31; message:Invalid Signature: signature could not be verified.", "result": -1, "logs": ""}, "event": 6} |
| logs | 记录 |
| error | result=-1时才会出现 |

* **注意**

1. logger (PDCA上传)将会监听"event": 2，"event": 3，"event": 4的数据，处理上传pdca
2. Logger在监听到“event”: 4且 MLBSN出现时，将去检查AmIOk, UOP检查，如{“data”: {“name”: “MLBSN”, “value”: “FN6620500B4GQD15W”}, “event”: 4}
3. Sequencer会监听logger的"event": 5，并获取error\_code是否为-10005，以判断AmIOk

## Sequencer RPC

***协议***

请求应答消息全部将以Json格式字符串发送和接收。

请求(REQ)消息格式定义：

|  |  |
| --- | --- |
| 键 | 定义 |
| jsonrpc | rpc协议的版本 |
| id | uuid |
| function | 功能函数 |
| params | 参数 |

应答(REP)消息格式定义

|  |  |
| --- | --- |
| 键 | 定义 |
| jsonrpc | rpc协议的版本 |
| id | uuid |
| error | 仅当错误发生时，此键值才会出现，和result键不能同时出现。  此键值下有两键值，分别是  code：负整型错误代码  message：错误信息描述 |
| result | 对function执行的返回。如果function没有返回，此键值应设置为空字符串 |

*应用*

|  |  |  |  |
| --- | --- | --- | --- |
| REQ | | | REP |
| function | params | 功能 | result |
| status | “” | 询问sequencer状态 | 有三中状态  NONLOADED:没有载入testplan  READY:testplan已载入且可以开始执行  RUNNING：运行中 |
| run | “”或字典  attributes={MLBSN=s} | 顺序运行测试序列。如果sequencer启动时设置为fail继续测试，则会从头执行到最后，否则fail时将会停止 | True 或 False |
| load | Testpan路径 | 加载testplan(csv) | [testplan路径] has been loaded |
| show | 变量名称 | 查询sequencer里保存的变量值 | 变量值 |
| jump | Testplan的行号(1开始)，或group的名称 | 跳转到指定的行号或group | 返回下一步执行的列表，如“{‘FUNCTION’:relay, ‘PARAM1’: BATTERY\_POWER, …}” |
| list | 行数(默认为10行)，或all | 列举已加载的测试项目 | 列举项目清单,如列举两行，返回如下：[[0, 0, 2], [1, "{'FUNCTION': 'add', 'PARAM1': '1', 'GROUP': 'GROUP', 'DESCRIPTION': 'ADD11', 'VAL': '', 'HIGH': '', 'TIMEOUT': '', 'PARAM2': '1', 'KEY': '', 'TID': 'ADD11', 'UNIT': '', 'LOW': ''}"], [2, "{'FUNCTION': 'add', 'PARAM1': '1', 'GROUP': 'GROUP', 'DESCRIPTION': 'ADD12', 'VAL': '', 'HIGH': '', 'TIMEOUT': '', 'PARAM2': '2', 'KEY': '', 'TID': 'ADD12', 'UNIT': '', 'LOW': ''}"]] |
| next | “” | 查询下一项行号 | 下一测试项的行号 |
| step | “” | 单步执行 | 放回项目执行的结果和项目的描述清单 |
| abort | “” | 终端测试 | True 或False |
| wait | 超时时间 | 可以中间停断测试 | True 或 False |

例：

单步执行测试：

发送：

*{"id":"f143164097ff11e58c6d3c15c2dab3ba","function":"step","jsonrpc":"1.0","params":""}*

返回：

*{"jsonrpc": "1.0", "id": "f143164097ff11e58c6d3c15c2dab3ba", "result": [1, "{'FUNCTION': 'add', 'PARAM1': '1', 'GROUP': 'GROUP', 'DESCRIPTION': 'ADD11', 'VAL': '', 'HIGH': '', 'TIMEOUT': '', 'PARAM2': '1', 'KEY': '', 'TID': 'ADD11', 'UNIT': '', 'LOW': ''}"]}*

顺序执行测试：

发送：

*{"id":"f143164097ff11e58c6d3c15c2dab3ba","function":"run","jsonrpc":"1.0","params":""}*

* **注意**

State Machine在向Sequencer发送”function”:”run”请求时，如果State Machine中存在产品sn时，sn将会以params参数传给sequencer，格式如下：

*{“id”:”f143164097ff11e58c6d3c15c2dab3ba”,”function”:”run”,”jsonrpc”:”1.0”,”params”:[{“attributes”:{“MLBSN”:”FN6620500B4GQD15W”}}]}*

## Engine Publish

Engine发布的消息有心跳，以及REP\_REQ的接收或者返回的数据。

|  |  |  |
| --- | --- | --- |
| 内容 | 键 | 解释 |
| 时间 | 无 | 发布者发布信息的时间，格式是mm-dd\_hh:mm:ss.milisecond。如2016-06-02 16:08:27.163768 |
| 等级 | 无 | REPORTER:0, CRITICAL:1, INFO:2, DEBUG:3 |
| 发布者 | 无 | 发布者的身份标明 |
| 数据 | data | 发布的数据，有两种  REP或REQ的json格式数据，  或FCT\_HEARTBEAT |

例：

数据键值发布REQ接收到的请求数据：

*< Received > {"function": "add", "jsonrpc": "1.0", "params": ["1", "2"], "timeout": 3000, "id": "1bba6ccf320011e68b7e3c15c2dab3ba", "unit": ""}*

数据键值发布REP应答的数据

*< Result > {"id":"1bba6ccf320011e68b7e3c15c2dab3ba","jsonrpc":"1.0","result":3}*

## Engine RPC

请求(REQ)

向Engine发送请求，数据格式如下，信息需以json格式发送，function为function\_table里定义的key，params为函数的参数

|  |  |
| --- | --- |
| 键 | 内容 |
| jsonrpc | 版本 |
| id | id号 |
| function | 功能方法 |
| params | 参数,数组，lua脚本里的table |
| timeout | 超时时间 |
| unit | 单位，脚本将依据此单位将测试结果进行单位转换 |

例：(Sequencer发送到Engine的请求)

*{"function": "relay", "jsonrpc": "1.0", "params": ["batt"], "timeout": 3000, "id": "3130dc35289911e692b10c4de9b22822", "unit": ""}*

应答(REP)

Engine向请求者的应答信息格式一致，如下：

|  |  |
| --- | --- |
| 键 | 内容 |
| jsonrpc | 版本 |
| id | id号 |
| error | 错误信息，当错误发生时出现 |
| result | 结果，正常情况下。 |

例：(Engine应答Sequencer的请求)

*{"jsonrpc": "1.0", "id": "3130dc35289911e692b10c4de9b22822", "result": "--PASS--"}*

## State Machine

状态机负责维护测试状态及夹具控制。一般常用的状态机详见后章节[State Machine](#_第五部分：State_Machine).

其他的如乒乓结构，N\*M等，需修改此框图及脚本。

## State Machine Publish

状态机将会发布心跳和状态,两个为单独的通道。

心跳发布：

|  |  |  |
| --- | --- | --- |
| 内容 | 键 | 解释 |
| 时间 | 无 | 发布者发布信息的时间，格式是mm-dd\_hh:mm:ss.milisecond。如2016-06-02 16:08:27.163768 |
| 等级 | 无 | 3 |
| 发布者 | 无 | 发布者的身份标明 |
| 数据 | data | STATEMACHINE\_HEARTBEAT |

状态发布：

|  |  |
| --- | --- |
| 状态 | 内容 |
| SM\_IDLE | 空闲 |
| SM\_UUT\_LOADED | 产品已装载 |
| SM\_TESTING | 测试中 |
| SM\_TESTING\_DONE | 测试完成 |
| SM\_UUT\_UNLOADED | 产品取出 |
| SM\_ERROR | 错误 |

发布的数据结构是：

|  |  |  |
| --- | --- | --- |
| 键 | 内容 | 例 |
| function | 状态 | {“function”:” SM\_TESTING”} |

## State Machine RPC

请求(REQ):

state machine 的请求包含3个

1. 设置UUT条码

|  |  |
| --- | --- |
| 键 | 值 |
| function | SETUUTSN |
| uut | uut所在的通道，从0开始 |
| sn | 条码 |

例：

*{"function":"SETUUTSN","uut":1,"sn":"abcdefgh"}*

1. 设置UUT允许测试或禁止测试

|  |  |
| --- | --- |
| 键 | 值 |
| function | UUTENABLE |
| uut | uut通道，从0开始 |
| enable | 设置uut状态，1：允许测试，0：禁止测试 |

例：

*{"function":"UUTENABLE","uut":1,"enable":1}*

1. 触发开始测试

|  |  |
| --- | --- |
| 键 | 值 |
| function | START |
| times | loop测试次数，可以不存在 |
| uutinfo | 设置uut状态和uut条码  是字典的数组，字典有两个键  uut：通道号  sn：条码 |

例：

*{“function":"START"}*

*{“function”:”START”,”times":10}*

*{“function”:”START”,”uutinfo:[{"sn":"abcdefg","uut":1},{"sn":"hijklmn","uut":2}]”}*

应答(REP)

|  |  |
| --- | --- |
| 键 | 值 |
| status | ok 或 fail |

例：

*{“status":"ok"}*

## RPC链接

当前存在或可能存在的应答-请求关系有

|  |  |  |
| --- | --- | --- |
| Reply(应答) | Request(请求) | 用途 |
| State Machine | GUI | UUT使能或开始测试，测试中止 |
| State Machine | Fixture Controller | 开始测试 |
| Fixture Controller | State Machine | 夹具动作：进，出等 |
| Sequencer | State Machine | 开始测试，测试中止 |
| Engine | Sequencer | 项目测试执行 |
| Engine | Debug Tools | 调试 |
| Sequencer | Debug Tools | 调试 |

# 第四部分：消息实例

## 消息序列图



## 启动消息

需先启动Engine，再启动Sequencer。sequencer 启动时将会去询问Engine的状态。如果Enigne在询问时没有响应，Sequencer将会退出。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| NO. | Action | From/To | Type | Msg | PDCA |
| 1 | Send | Engine | Req | {“function”: “\_my\_rpc\_server\_is\_ready”, “jsonrpc”: “1.0”, “params”: [], “timeout”: 3000, “id”: “b6d77502293711e6b30e3c15c2dab3ba”, “unit”: “”} |  |
| 2. | Recv | Engine | Rep | {“id”:”b6d77502293711e6b30e3c15c2dab3ba”,”jsonrpc”:”1.0”,”result”:”—PASS—“} |  |

## 测试中消息

* 开始测试后，SEQ每个从REP-REQ接收或者发送的数据，都会PUB，格式和其他的PUB一直，此表中未列举。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| NO. | Action | From/To | Type | Msg | PDCA |
| 1 | Recv | SM | Req | {“id”:”f143164097ff11e58c6d3c15c2dab3ba”,”function”:”run”,”jsonrpc”:”1.0”,”params”:[{“attributes”:{“MLBSN”:”FN6620500B4GQD15W”}}]} |  |
| 2 | Send | SM | Rep | {“jsonrpc”: “1.0”, “id”: “f143164097ff11e58c6d3c15c2dab3ba”, “result”: true} |  |
| 3 | Send |  | Pub | 101  06-02\_16:08:27.163768  0  Sequencer\_0  {“data”: {“version”: “0531d 2”, “name”: “dfu”}, “event”: 0} | Start |
| 4 | Send |  | Pub | 101  06-02\_16:08:27.163768  0  Sequencer\_0  {“data”: {“name”: “MLBSN”, “value”: “FN6620500B4GQD15W”}, “event”: 4} | mlbsn |
| 5 | Recv | Logger | Pub | 101  06-02\_13:41:12.904660  0  logger  {“data”: {“error\_code”: -10005, “site”: 0, “error\_msg”: “AMIOK ERROR: class:32 id:31; message:Invalid Signature: signature could not be verified.”}, “event”: 5} | Amiok error |
| 5a | Send |  | Pub | 101  06-02\_13:41:12.923203  0  Sequencer\_0  {"data": {"error": "AMIOK ERROR: class:32 id:31; message:Invalid Signature: signature could not be verified.", "result": -1, "logs": ""}, "event": 6} | Amiok error |
| 6 | Send | Engine | Req | {"function": "start\_test", "jsonrpc": "1.0", "params": [], "timeout": 5000, "id": "2ffdcdb8289911e688b40c4de9b22822", "unit": ""} |  |
| 7 | Recv | Engine | Rep | {"jsonrpc": "1.0", "id": "2ffdcdb8289911e688b40c4de9b22822", "result": ""} |  |
|  |  |  |  |  |  |
| 8 | Send | Engine | Req | {"function": "relay", "jsonrpc": "1.0", "params": ["batt"], "timeout": 3000, "id": "3130dc35289911e692b10c4de9b22822", "unit": ""} |  |
| 9 | Recv | Engine | Rep | {"jsonrpc": "1.0", "id": "3130dc35289911e692b10c4de9b22822", "result": "--PASS--"} |  |
| 10 | Send |  | Pub | 101  06-02\_16:08:30.176955  3  Sequencer\_0  {'FUNCTION': 'relay', 'PARAM1': 'batt', 'GROUP': 'DCSD\_CHECK', 'DESCRIPTION': 'BATT\_POWER\_ON', 'VAL': '', 'HIGH': '', 'TIMEOUT': '', **'PARAM2': ''**, 'KEY': '', 'TID': 'BATT\_POWER\_ON', 'UNIT': '', 'LOW': ''} | item |
| 11 | Send |  | Pub | 101  06-02\_16:08:30.177080  0  Sequencer\_0  {"data": {"group": "DCSD\_CHECK", "description": "BATT\_POWER\_ON", "to\_pdca": true, "high": "", "low": "", "tid": "BATT\_POWER\_ON", "unit": ""}, "event": 2} | item |
| 12 | Send |  | Pub | 101  06-02\_16:08:30.418220  0  Sequencer\_0  {"data": {"tid": "BATT\_POWER\_ON", "to\_pdca": true, "result": true, "value": "--PASS--"}, "event": 3} |  |
|  |  |  |  |  |  |
| 13 | Send | Engine | Req | {"function": "parse", "jsonrpc": "1.0", "params": ["{{soc\_bin\_voltage}}"], "timeout": 3000, "id": "7cad200f289a11e695b90c4de9b22822", "unit": "mV"} |  |
| 14 | Recv | Engine | Rep | {"jsonrpc": "1.0", "id": "7cad200f289a11e695b90c4de9b22822", "result": "838"} |  |
| 15 | Send |  | Pub | 101  06-02\_16:17:46.317235  3  Sequencer\_0  {'FUNCTION': 'parse', 'PARAM1': '{{soc\_bin\_voltage}}', 'GROUP': 'INFANCY1\_Item1', 'DESCRIPTION': 'Soc\_Bin\_Voltage', 'VAL': '', 'HIGH': '', 'TIMEOUT': '', **'PARAM2': '{{soc\_bin\_voltage}}'**, 'KEY': '', 'TID': 'PARSE soc\_bin\_voltage', 'UNIT': 'mV', 'LOW': ''} | Item |
| 16 | Send |  | Pub | 101  06-02\_16:17:46.317347  0  Sequencer\_0  {"data": {"group": "INFANCY1\_Item1", "description": "Soc\_Bin\_Voltage", "to\_pdca": true, "high": "", "low": "", "tid": "PARSE soc\_bin\_voltage", "unit": "mV"}, "event": 2} | Item |
| 17 | Send |  | Pub | 101  06-02\_16:17:46.318537  0  Sequencer\_0  {"data": {"tid": "PARSE soc\_bin\_voltage", "to\_pdca": true, "result": true, "value": "838"}, "event": 3} |  |
|  |  |  |  |  |  |
| 18 | Send | Engine | Req | {"function": "parse", "jsonrpc": "1.0", "params": ["{{board\_rev}}"], "timeout": 3000, "id": "6ab8bb9e289a11e68bc30c4de9b22822", "unit": ""} |  |
| 19 | Recv | Engine | Rep | {“jsonrpc”: “1.0”, “id”: “6ab8bb9e289a11e68bc30c4de9b22822”, “result”: “0x0C”} |  |
| 20 | Send |  | Pub | 101  06-02\_16:17:16.194309  3  Sequencer\_0  {'FUNCTION': 'parse', 'PARAM1': '{{board\_rev}}', 'GROUP': 'DUT\_SW\_Item', 'DESCRIPTION': 'Board Revision', 'VAL': '', 'HIGH': '', 'TIMEOUT': '', 'PARAM2': '<<BOARD\_REV>>', 'KEY': '', 'TID': 'PARSE board\_rev', 'UNIT': '', 'LOW': ''} | Attribute |
| 21 | Send |  | Pub | 101  06-02\_16:17:16.194401  0  Sequencer\_0  {"data": {"group": "DUT\_SW\_Item", "description": "Board Revision", "to\_pdca": true, "high": "", "low": "", "tid": "PARSE board\_rev", "unit": ""}, "event": 2} |  |
| 22 | Send |  | Pub | 101  06-02\_16:17:16.195439  0  Sequencer\_0  {"data": {"name": "BOARD\_REV", "value": "0x0C"}, "event": 4} |  |
| 21 | Send |  | Pub | 101  06-02\_16:17:21.195555  0  Sequencer\_0  {"data": {"tid": "PARSE board\_rev", "to\_pdca": true, "result": true, "value": "0x0C"}, "event": 3} |  |
|  |  |  |  |  |  |
| 23 |  |  |  | {"function": "end\_test", "jsonrpc": "1.0", "params": [1], "timeout": 5000, "id": "362bdae3293511e69da23c15c2dab3ba", "unit": ""} |  |
| 24 |  |  |  | {"jsonrpc": "1.0", "id": "362bdae3293511e69da23c15c2dab3ba", "result": ""} |  |
| 25 |  |  |  | 101  06-03\_10:45:20.119489  0  Sequencer\_0  {"data": {"result": 1, "logs": ""}, "event": 1} | End  1:Pass  0:Fail |
|  |  |  |  |  |  |

# 第五部分：State Machine

以下罗列的为常用的夹具控制及状态维护。

## 状态图

常见的状态机的架构如下图，实现测试状态的维护和测试夹具进出下压上弹得控制。



## 控制框架

控制方式：

1. 夹具发出定义的字符后触发State Machine 向Sequencer发出开始测试请求,次模式一般为作业者按夹具按钮，夹具动作完成后触发测试
2. TM界面Start 按钮，向State Machine发送请求后，State Machine向夹具控制进程发送夹具动作请求，后再向Sequencer发送开始测试请求。

****

## 脚本

SipFixture.lua : 创建夹具串口连接，并建立zeq REP, REQ及PUB。

CreatSipFixture()：创建对象

InitialSerialPort(“/dev/cu.usbserial-F-CTR”,” 115200,n,8,1”);初始化串口

InitialZmqPort(“tcp://127.0.0.1:6471”, “tcp://127.0.0.1:6470”);建立REP/PUB

InitialReqPort(“tcp://127.0.0.1:6480”,"start","{\"function\":\"START\"}");建立REQ,设置监听字符，当接收到监听字符后向REQ对应的REP发送数据

Fixture.lua： 创建REQ，并发送命令

# 第六部分：条码处理



条码的来源可以由扫描枪(Scanner)读取产品条码，或者有网络获取，或者通过命令由产品读取。

1. Scanner / SFC ：当条码扫描完成后将发送通知(Notification)由GUI更新界面，并同时通过zmq的REQ请求State Machine更新产品条码和状态。开始触发测试(界面点击start或者夹具到位出发)时，State Machine将通过REQ发送命令并随带条码请求到Sequencer请求开始测试。
2. DUT读取：此处以条码作为测试项目，Sequencer通过ZMQ 的REQ 发送产品命令到Engine，Engine读取到产品条码(MLBSN)后返回。

# 附录：dylib制作

**摘要**

本文为使用Xcode生成dylib，可以在lua中使用的操作过程

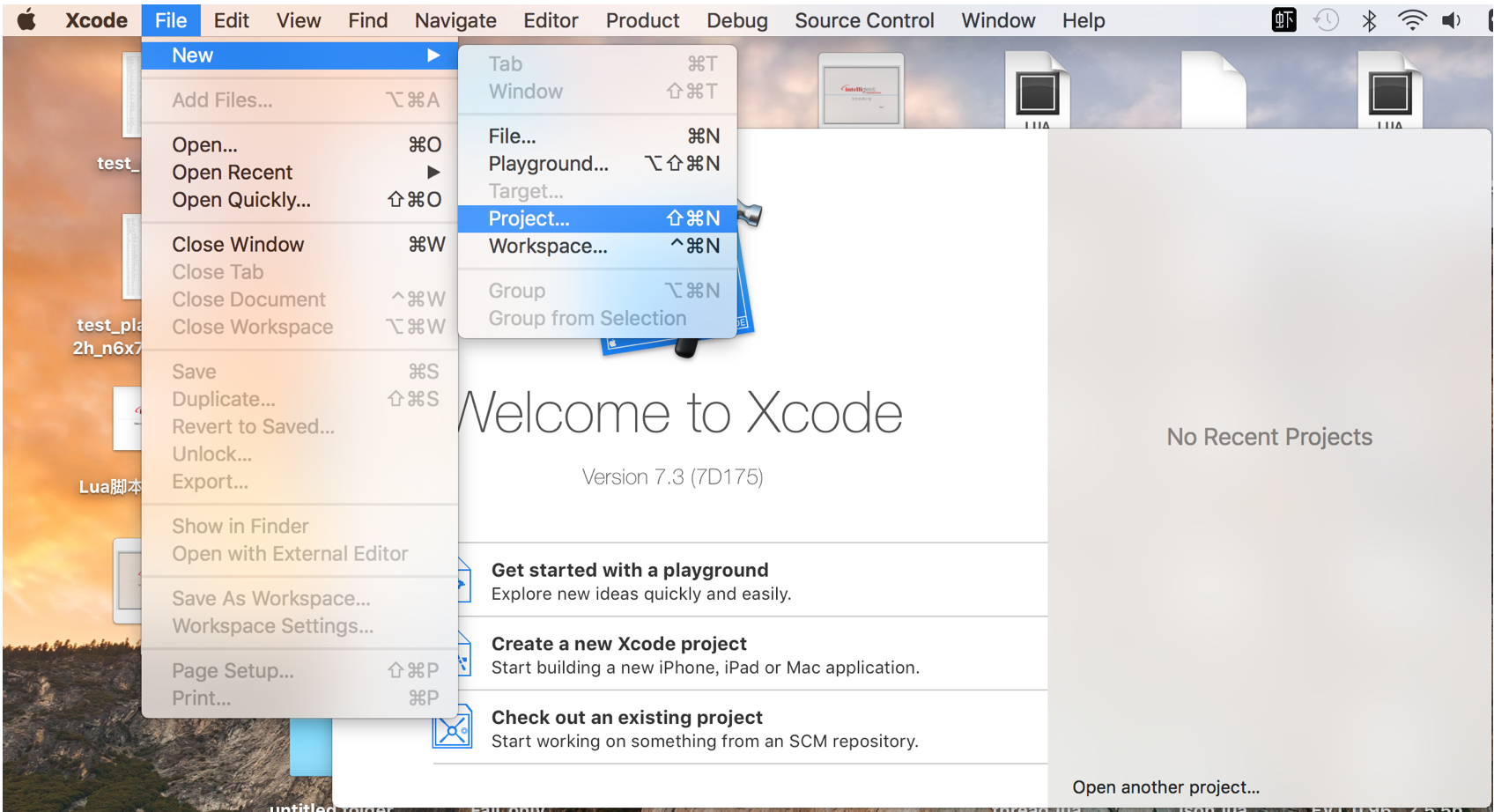
**准备事项**

电脑需已安装：

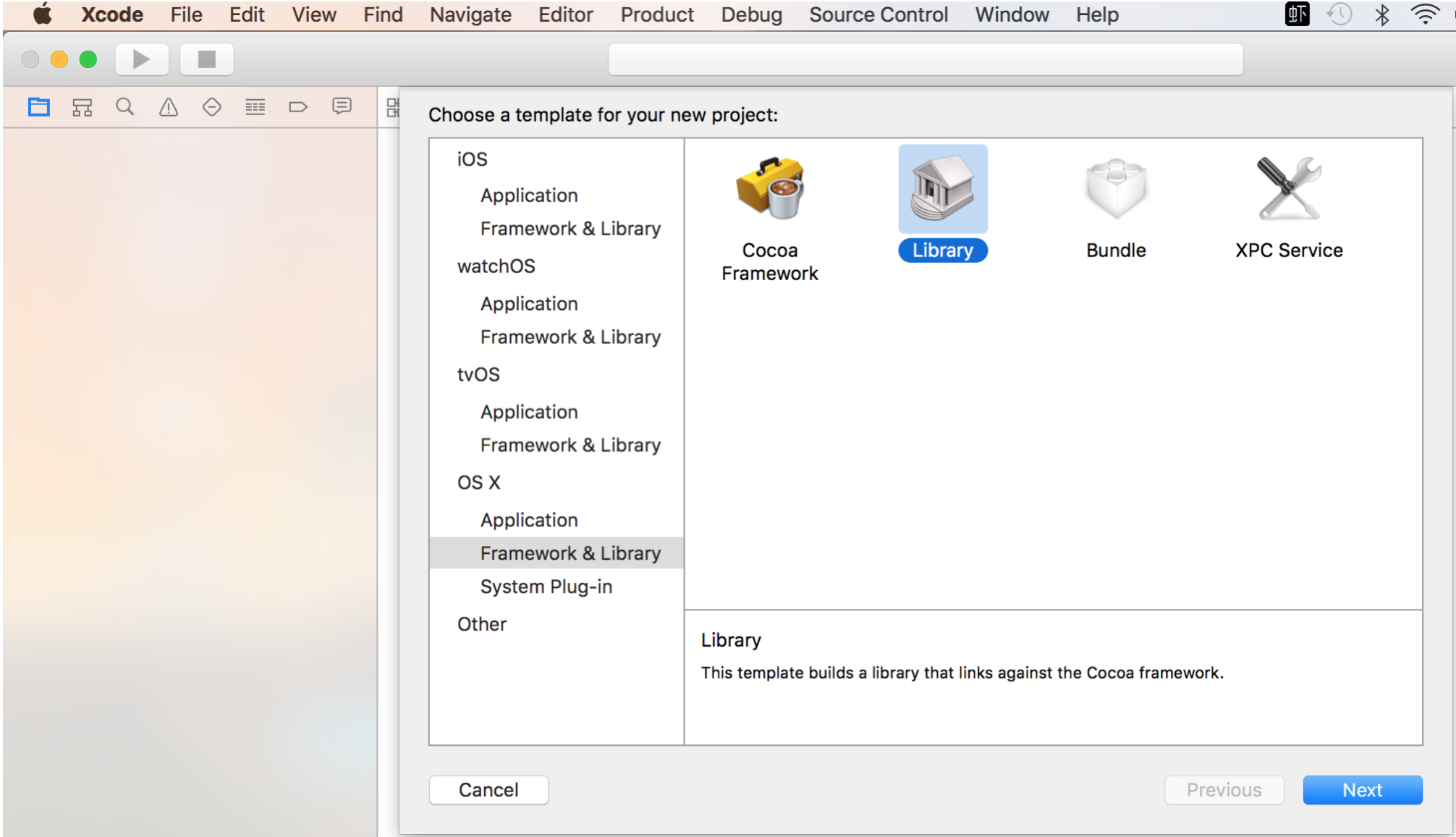
1. Xcode (5.1以上)
2. Lua (5.1.4), lua 已安装lfs

**制作**

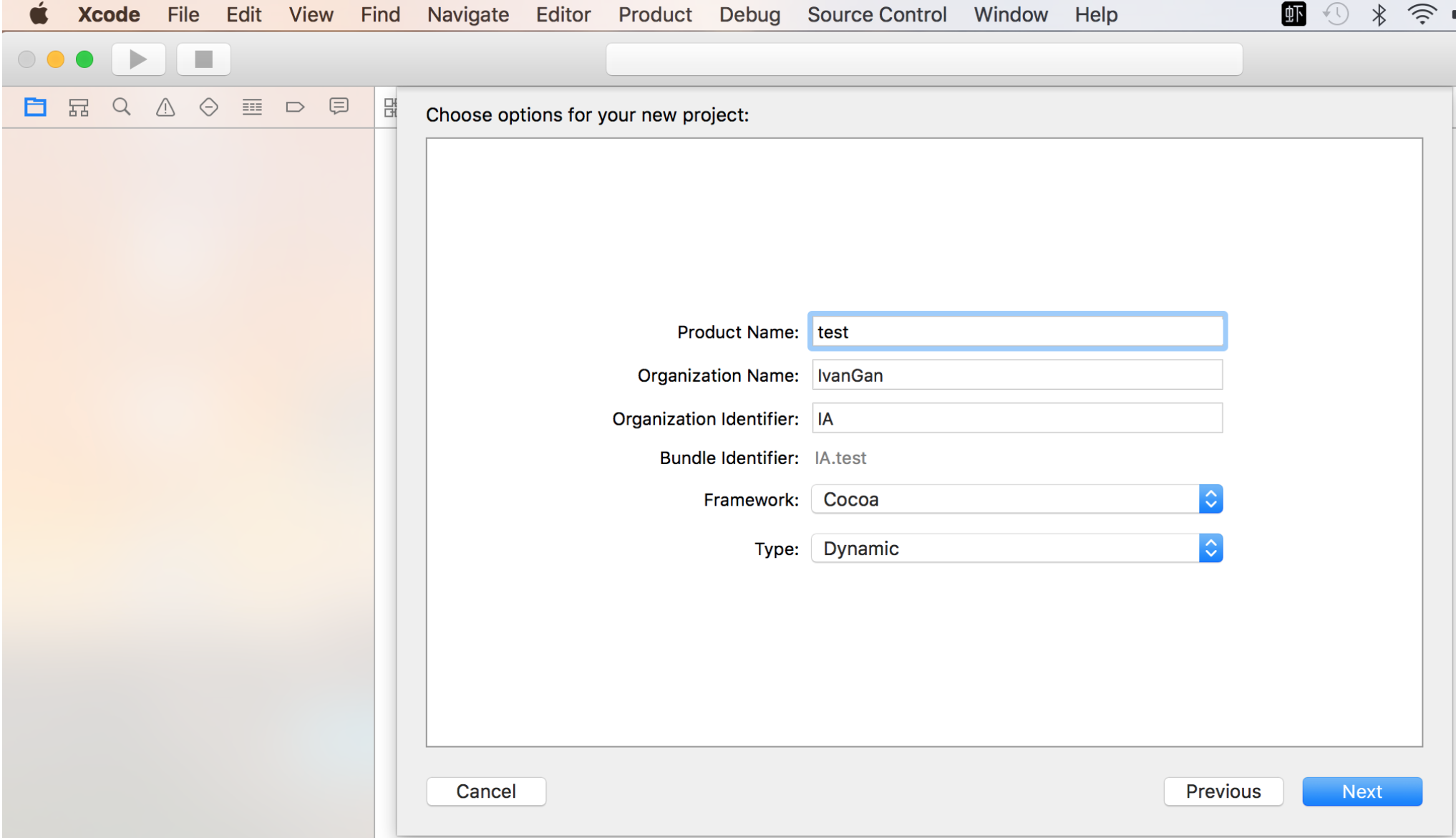
1. 打开Xcode,新建工程



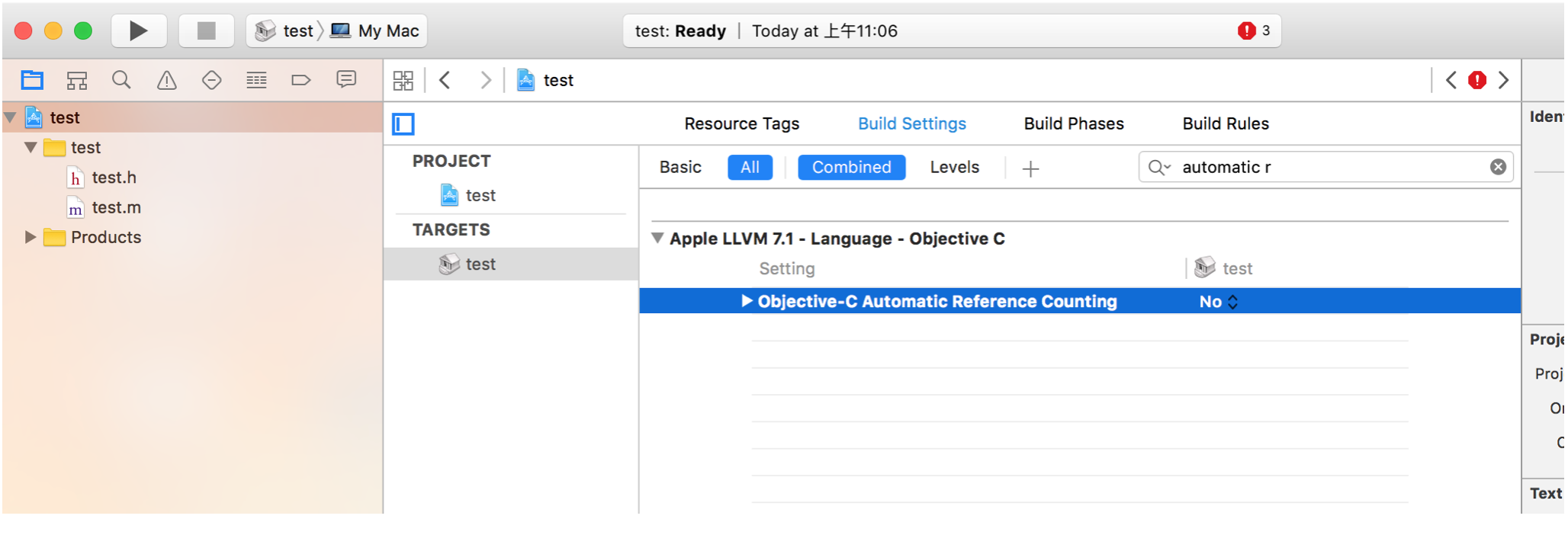
1. 选择 OS X -> Framework & Library -> Library, 点击Next



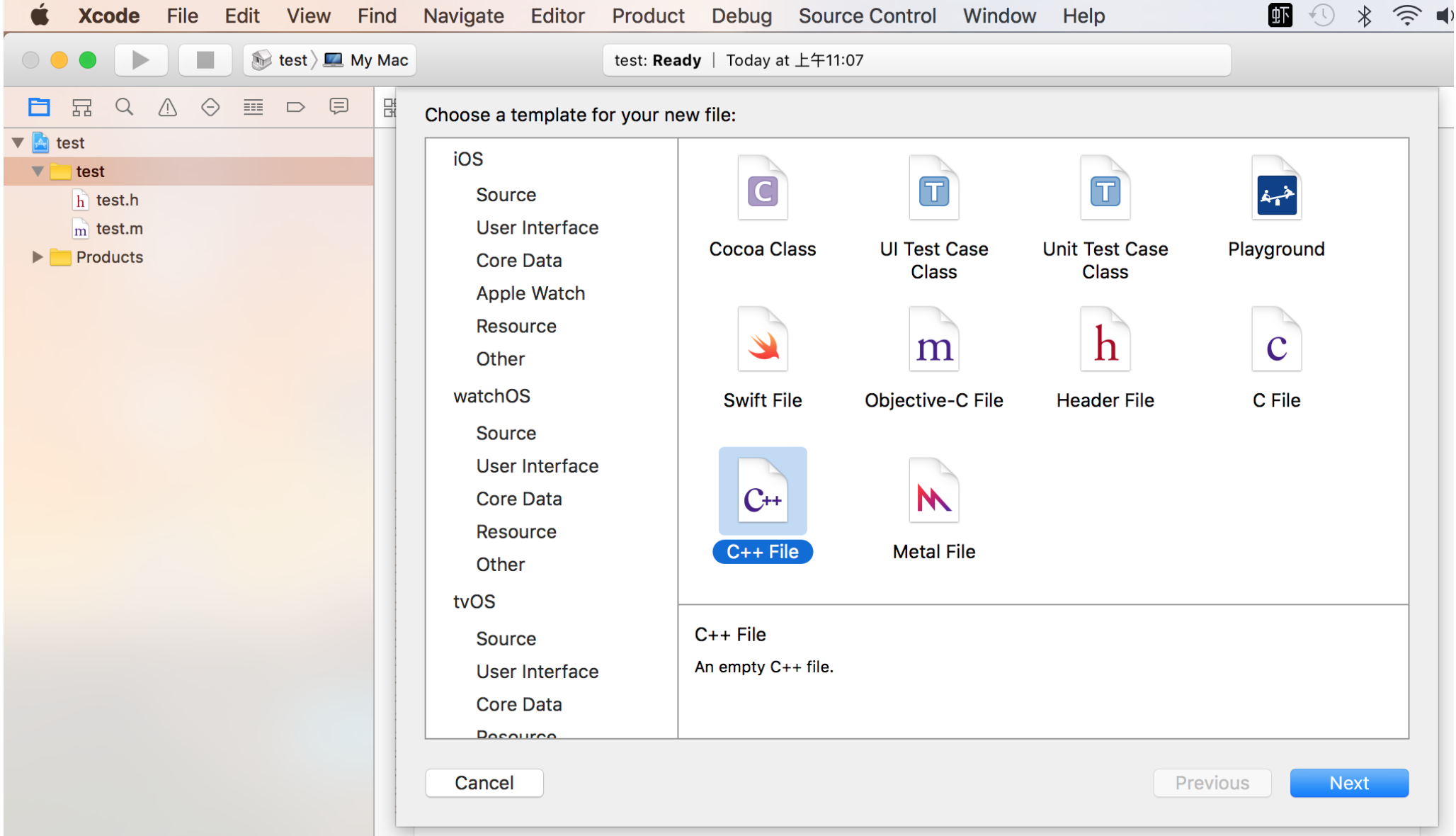
1. 输入lib名称，本例中命名为test，类型选择Dynamic, 点击Next



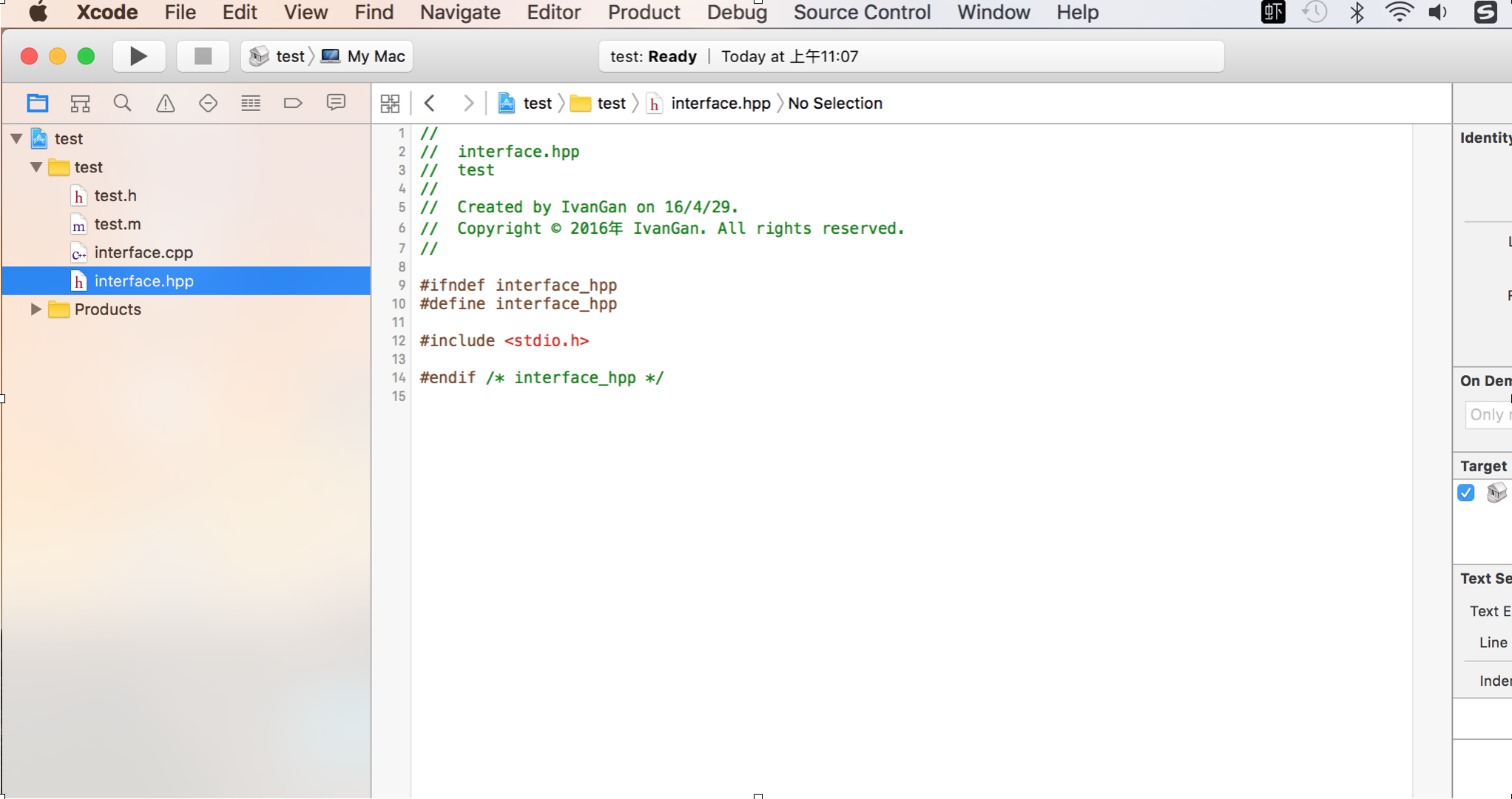
1. 选择保存路径，点击 create
2. 设置ARC，系统默认为YES，修改为NO，关闭



1. 添加接口文件，点击菜单File -> New -> File…
2. 选择C++文件类型



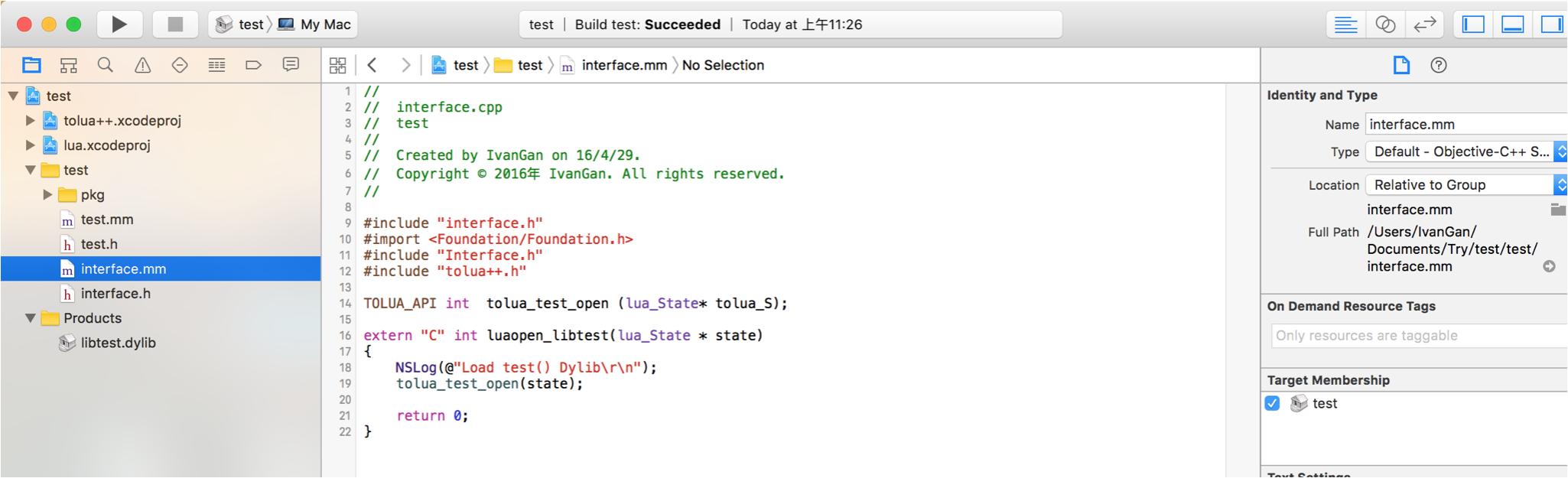
1. 本例中命名为Interface, 用户可自行修改。选择保存路径，点击Create
2. 创建完成后如下图所示



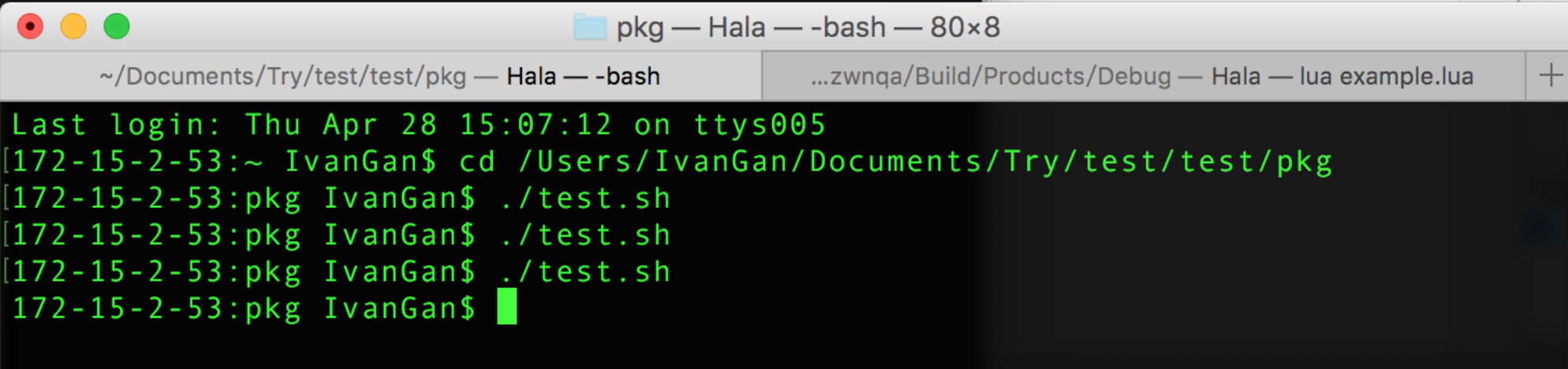
1. 添加代码，用户可根据自己实际修改代码 ( 代码在文本最后 )

本例子中只实现数据的设置和读取。(接口函数定义必须为C/C++，实现可以有Object-C)

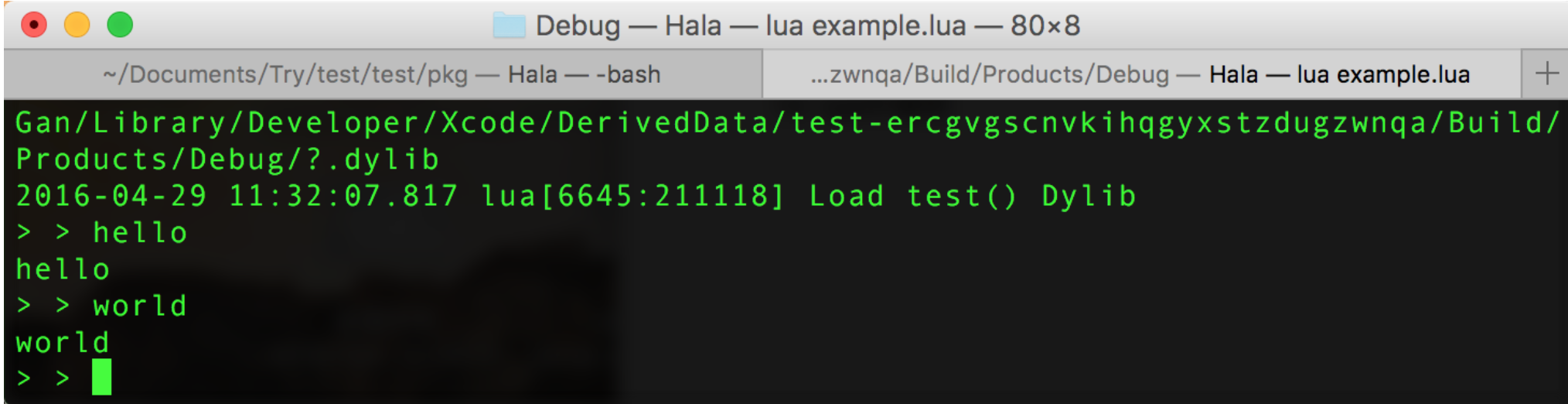
1. 修改各文件后缀名。(.mm可以同时支持C++和Object-C)



1. 添加工程lua 和tolua++, 添加pkg和sh脚本，pkg选择文件类型为C++header
2. cd到文件所在文件夹，终端运行sh脚本生成C++对lua的接口test\_lua.mm



1. lua中调用，运行example.lua

****

**代码**

*\*\*\*\*\* test.h*

*class CTest*

*{*

*public:*

*CTest();*

*~CTest();*

*void setData(const char \* str);*

*const char \* getData();*

*private:*

*NSMutableString \* data;*

*};*

*\*\*\*\*\* test.mm*

*CTest::CTest()*

*{*

*data = [[NSMutableString alloc]init];*

*}*

*CTest::~CTest()*

*{*

*[data release];*

*}*

*void CTest::setData(const char \* str)*

*{*

*[data setString:[NSString stringWithUTF8String:str]];*

*}*

*const char \* CTest::getData()*

*{*

*return [data UTF8String];*

*}*

*\*\*\*\*\* interface.h*

*注： luaopen\_libtest 应该和 lib 名字一致*

*#include "lua.hpp"*

*#define DL\_EXPORT \_\_attribute\_\_((visibility("default")))*

*#include <stdio.h>*

*extern "C" int luaopen\_libtest(lua\_State \* state);*

*\*\*\*\*\* interface.mm*

*#import <Foundation/Foundation.h>*

*#include "tolua++.h"*

*TOLUA\_API int tolua\_test\_open (lua\_State\* tolua\_S);*

*extern "C" int luaopen\_libtest(lua\_State \* state)*

*{*

*NSLog(@"Load test() Dylib\r\n");*

*tolua\_test\_open(state);*

*return 0;*

*}*

*\*\*\*\*\* test.sh*

*tolua=/usr/local/lib/tolua++*

*$tolua -o test\_lua.mm test.pkg*

*\*\*\*\*\* test.pkg*

*$#include "test.h"*

*class CTest*

*{*

*public:*

*CTest();*

*~CTest();*

*void setData(const char \* str);*

*const char \* getData();*

*};*

*\*\*\*\*\* example.lua*

*require "lfs"*

*package.cpath = package.cpath ..";" .. lfs.currentdir().."/?.dylib"*

*print(package.cpath)*

*require "libtest"*

*test = CTest:new()*

*while true do*

*print("> > ")*

*t = io.read()*

*if(#t>0) then*

*test:setData(t)*

*print(test:getData())*

*end*

*end*