RODAC

SPARKLETRON

November 2, 2024

Jay Convertino

PRE-ALPHA

Contents

1 Usage

1.1 Introduction

This manual describes how to use the RODAC (Retro Only Device Application Creation) system for development. Items such as how to build included apps, what the structure of the system looks like, and how to create your own app is included. The final section are links to doxygen generated documentation about the drivers used by this system.

RODAC is a multisystem development kit that targets retro systems with a z80 CPU and TMS based video display processors. Various sound and other IO peripheral are supported. RODAC is made to be a easy to use by creating a modern directory structure and library system for the code. All code, outside of the C runtime is written in C. This makes porting between systems easier and the code readable. Applications can be made to target all or just a few of the systems architectures by including only the makefiles for the those systems. This is due to each architecture using the same file structure.

Overall the system makes adding new architecture targets, applications, and drivers easy. Allowing other systems to be added quickly and effortlessly. Inline assembler is avoided so future iterations may support other CPU targets, though this is a distant goal. RODAC is a development kit for retro systems that makes development about the application, not the processes of trying to build it.

1.2 Dependencies

The following are the dependencies needed to build the applications targeting various retro systems.

- sdcc 4.X.X
- python 3.X
- make

1.3 Building

Makefiles are used to execute all builds. All sources will rebuild when make because the libraries must be updated. If this didn't happen the new memory map setup in the defines.h would not be applied. Each application has its makefile located in its root folder. To run a build you must run, in the target apps root folder, the following command.

```
$ make SYSTEM
```

Where SYSTEM is the target you would like to build for. All will do nothing but toss an error telling you the same. Currently the targets are **coleco, coleco_sgm, msx, sg1000**. All targets have been tested for coleco based systems. The others are not tested on real hardware at the moment.

1.3.1 hello_world

Hello World is a simple application that prints all of the characters from the TMS memory to screen. It also prints hello world in the center of the screen and scrolls it horizontally. This is done in the TMS txt mode with 40 columns and no sprites. This application has been tested in emulation on all available systems. It also generates a single annoying constant tone, as a really poor sound test. To build this run the following for the Colecovision in the root of the apps/hello_world folder.

```
$ make coleco
```

1.3.2 multicart

Multicart creates a non-scrolling list of ROMs in alphabetical order. The number of ROMs is limited by the target flash size and the number of lines (21) on screen. The generation of the header that contains the list of ROMs is automatically updated with new rom using a python script, rom_header_gen.py. The full ROM is also auto generated by a python script rom_file_gen.py. Currently these default to the roms folder located in the apps/multicart folder root. This can be changed in the make file via the ROM_ variables. There are dummy ROMs with random data for testing of the system. This will work in an emulator up to the point of bank switching ROMs, since the PIC and the logic with it is not emulated. This is currently only targeted and tested on the Colecovision. To build for the Colecovision you would run the following in the apps/multicart folder.

```
$ make coleco
```

1.4 Directory Guide

Below highlights important folders from the root of RODAC.

1. **docs** Contains all documentation related to this project.
   - **arch** Contains all architecture docs related to retro systems.
   - **manual** Contains user manual and wiki that are generated from the same latex source.
2. **apps** Contains source code in C for the applications to run on the target architecture.
   - **hello_world** Example hello world application. Targets all architectures.
   - **mutlicart** Example multicart application, written for the coleco only.
3. **drivers** Contains all source code related to the project.

- **gisnd** Simple driver for the GI AY-3-8910 sound chip and its variants.
- **sn76489** driver for the TI SN76489 sound chip.
- **tms99XX** driver for all TMS99XX and TMS9XXX video chips.

2 Creating new apps, drivers, or systems.

My goal is to create an easy way to add new parts to this system. The best way to start is to copy and pasta something that is close and edit it. This applies to Applications, Architecture, and Drivers. The following subsections explain each part when it comes to the files and structure.

2.1 Application Creation

The structure of the apps should be consistent. All apps are located in the apps folder. The base folder should be a recognizable name. For this examples we will call it example. The easy way to get started is the to copy and rename the hello_world example.

```
$ cp -r hello_world example
```

The above command will copy and rename the folder to example. This assumes hello_world has not been built. This folder will contain a src directory, a makefile, and a readme. I recommend a readme so the user can quickly find out what the program does. You now have a starting point for creating your application.

Makefile kicks off the main build and contains the system targets available. All builds call sdcc as the compiler and use CFLAGS and LFLAGS to set various flags. These flags also include variables that are set by the system include makefiles. This makefile will call all other makefiles used for driver, and architecture builds.

```
ifeq (coleco_sgm,$(MAKECMDGOALS))
include ../../arch/coleco_sgm.mk
LIBPATH += ../../drivers/gisnd
LIBPATH += ../../drivers/sn76489
BIN     := $(addprefix $(MAKECMDGOALS)_, $(BIN))
DIROBJ  := $(addprefix $(MAKECMDGOALS)_, $(DIROBJ))
endif
```

The above is an example of add a architecure target. In this case if the make variable MAKECMDGOALS is equal to coleco_sgm, the the makefile for it is included along with driver libraries. This will also prefix all the outputs for the application with the contents of MAKECMDGOALS. Alterations such as changing coleco_sgm to your new architecture, or adding or removing drivers is done in this block.

A few last notes, all source files will be added from the source directory (src). If you need to include headers, I recommend putting them in there own folder and adding a flag to the end of CFLAGS. The following is an example of using a variable to add a header directory.

```
-I$(HEADER_DIR)
```

The source is located in the src directory of the application folder, in this case named example. This can be any name you wish, but I recommend using the same name as the your application. There should be only one file, the main executable. Any utilities, drivers, and other items should be external and called in as libraries that are built by the makefile. This helps make core reusable and less congested. Things such as a game engine would be a separate folder in the root of the RODAC directory. Future examples will be added, though none are done at this moment in pre-alpha.

2.2 Architecture Creation

The structure of the architecture of each system is consistent. All architectures are located in the arch folder in the root of RODAC. Each one has its own makefile.mk with the name SYSTEM.mk. Where SYSTEM is the target. For instance coleco.mk is for the colecovision. The base header (base.h) file is for all of the architecture targets and holds defines and functions that are used for all. In the architecture folder named after the target, Colecovision the name would be coleco, contain the header file defines.h and the source files. The defines.h contains the port locations and memory map locations that are used by drivers. The base.c file is the actual functions definitions of the declared functions in base.h. The makefile is a universal build for the target system library. To create a new architecture i t is recommended to copy an existing system and rename it.

```
$ cp -r coleco example
$ cp coleco.mk example.mk
```

The above commands will copy and rename the folder and all its contents to example, and copy the makefile to a new file named example.mk.

Makefile.mk are the various makefiles in the arch folder named for their architecture target. In this case we are looking at coleco.mk. In coleco.mk the system is setup by variables, they are shown below.

```
SYS_DEFINE := _COLECO
ARCH    := z80

BASELIBPATH := ../../arch/
BASELIBARCH := coleco
BASELIBNAME := $(BASELIBPATH)$(BASELIBARCH)/base.lib

CODELOC     := 0x8100
DATALOC     := 0x7000
IRAMSIZE    := 1024
```

The above variables control the base system information, they are the following.

- **SYS_DEFINE** A define used in various other C files to switch out code. Should only be used in applications sparingly.
- **ARCH** Defines the CPU target, currently only z80 is supported.
- **BASELIBPATH** Path to this arch folder, relative to the apps folder.
- **BASELIBARCH** Name for the architecture. This should be the system name and match the folder and makefile.mk name.
- **BASELIBNAME** Complete path and name for the base library file.
- **CODELOC** Start of the C code location. This is after the crt0.s.
- **DATALOC** Start of the RAM location of the system.
- **IRAMSIZE** Number of bytes of RAM the system has.

Base header is simply a collection of functions and defines for all systems. It will also include the defines from the target architecture. When implementing your own base.c, these functions must be supplied. They are as follows.

- **__delay_us** A function that does a busy loop delay in microseconds. This has to be tuned to each system.
- **set_vdp_irq_callback** A function that sets the VDP IRQ callback. This is usually connected to the NMI interrupt.
- **set_spin_irq_callback** This is coleco specific. This function is for the interrupt generated by the spin controller. Needs to removed in the future.
- **getControllerOne** Base function for getting controller one 8 bit output. This is bare bones at the moments and defines.h contains the button bit defines.
- **getControllerTwo** Same as getControllerOne.

Defines header contains all the defines for the system. At this moment this document will not list each one, but will talk about the general idea. The defines included contain button bit locations for the getController functions. Another set of defines, that end in _ADDR are all port addresses of the devices connected to the z80 processor. Variables ending in _PORT are the z80 io port locations (0 to 255) of the devices to access. Then there are two defines called ei() (enable interrupt) and di() (disable interrupt). This was done to match the original xc8-cc PIC library calls that the driver libraries where ripped from.

Source file base is the source file that implements the base.h functions. It contains the function pointers that end up linking to crt0.s for IRQ calls.

C runtime file is the set of assembly instructions that setup the system in question and executes the main C function. This is written on a system by system basis. Differences such as SGM vs non-SGM Colecovision consoles are dealt with here. Where stack pointers should start, and how to initialize data regions.

2.3 Driver Creation

The structure of the drivers is consistent across all drivers. They contain a header that is doxygen commented, makefile, and src directory containing the source file. A license can also be added if need be, along with additional sources such as headers and such. There are no files in the drivers folder, just folders that contains the driver folders with the files for the target driver. The output is follows the name of the folder in the drivers folder and is case sensitive. Easy way of starting creation of your own driver is to copy one close to what you're targeting. In this case I will add a new sound chip, example.

```
$ cp -r gisnd example
```

The above commands will copy and rename the folder and all its contents to example. One item to keep in mind is any ports you read and write from for the driver are defined in the base system architecture in defines.h . This is handled by the compiler in the linking stage.

Makefile should require no changes from driver to driver. It will locate all files in the src folder that are C source files. All header files are expected to be in the root of the folder along side the makefile. The names of all libraries are expected to be lower case and the same as the folder name for the src and header files using similar naming.

Header will contain all public functions that can be called by the application. All of the doxygen comments are also located in this file. Any doxygen specific settings are in the config file dox.cfg.

3 Architecture

This will be an in depth look at how the makefile, engines, utilities, drivers, and architecture are all put together. Along with being a general roadmap. Since this is pre-alpha, well its simply a TODO.

4 Driver Documentation

4.1 TMS99XX Doxygen

If viewing the PDF version, please see tms99XX.pdf If viewing in html follow the link, TMS99XX HTML Doxygen, for the HTML version of the document.

4.2 SN76489 Doxygen

If viewing the PDF version, please see sn76489.pdf If viewing in html follow the link, SN76489 HTML Doxygen, for the HTML version of the document.

4.3 GISND Doxygen

If viewing the PDF version, please see gisnd.pdf If viewing in html follow the link, GISND HTML Doxygen, for the HTML version of the document.