Project information

Interested project: POST Code Standardization (a.k.a. Fix POST code handling)
Estimated project size: large ~350 hours

Based on Martin Roth "coreboot postcode cleanup" proposal.

Description of the project

Problem description

The current state of the coreboot postcodes is disorganized and inefficient. The `post_code()` function is frequently called with hardcoded values throughout the codebase, making it difficult to create and maintain a proper postcode table. Additionally, the `post_codes.h` file is disorganized, with values added without consideration for contiguity or order. Furthermore, there is no proper documentation of the POST codes, and the codes are duplicated between stages, which increases code size and reduces readability. This project aims to address these issues by organizing the postcodes in a more structured manner, documenting them properly, and unifying the code that sends POST codes to I/O port 80.

Current situation

POST code macros are defined in the next locations:

- `src/commonlib/include/commonlib/console/post_codes.h` - a central point for POST codes used throughout coreboot
- `src/soc/amd/common/psp_verstage/include/psp_verstage.h` - amd-specific, POST codes over eSPI (`PSP_POSTCODES_ON_ESPI`)
- `src/soc/amd/common/block/include/amdblocks/post_codes.h` - amd-specific
- `src/soc/intel/common/block/include/intelblocks/post_codes.h` - intel-specific
- `src/include/cpu/intel/post_codes.h` - intel-specific

Potential problem: How to find all these and future files automatically? Wildcard? Special suffix like `filename.postcodes.h`?

`post_code()` is defined in the two places:

- `src/include/cpu/x86/post_code.h` - used during ASM bootblock stage, send POST codes only to the `CONFIG_POST_IO_PORT` port.
  As far as this stage is written in ASM, there is one peculiarity. The `post_code()` function affects the EAX register, so we should save it or be sure that it's not used before emit the POST Code.

- `src/console/post.c` - the main function, able to redirect POST codes. For example, to CMOS and to console.

Requirements & Limitations

Before discussing any ideas on standardization of POST codes, it is important to understand the limitations and requirements. POST codes are a method of debugging that can be useful for non-technical users or in manufacturing. However, they may not be necessary for developers who have console output (with one nuance, but about it a bit later).

A common method for sending POST codes is through the use of an I/O port (0x80 port being a frequently used for this purpose). The size of an I/O port depends on the hardware architecture of the system and it can be 8/16/32-bit. If you try to send 16 bits of data to an 8-bit port, the upper 8 bits of the data will be ignored and only the least significant byte would be sent. Therefore, if one tries to write 0x1234, only 0x34 will be sent. Furthermore, the ports that the card looks at and how many bytes it looks at change from one device to another. *NOTE*: in some cases, a word write to 0x80 is equivalent to writing a singly byte to 0x80 and then 0x81.

One of the primary ways of obtaining POST codes is by using special hardware called POST cards, which come in different formats. I dare to divide them into next three groups:

1. Shows only the current POST code. Unable to process postcode context.
2. Shows both the current and previous POST codes. Able to process 2 postcode context.
3. Can save a history of POST codes => we can obtain any POST code that system sends us. Able to process any postcode context.

A context of a POST code refers to a situation where in order to understand the current POST code, it is necessary to analyze the previous one. This means that the interpretation of a particular POST code may depend on the sequence of codes that have been generated up to that point. Supporting all three types of POST cards is essential for our project (we follow the rule "as many as possible").

It can be safely assumed that the I/O port and the POST card must be capable of handling at least one-byte POST codes. However, the limitation of having only 255 values is not sufficient to cover all source code (generic + specific). It is not necessary to consider whether the I/O port or POST card can support more than 8-bit POST codes since our POST code table should be limited to 1 byte. Trying to include all custom components that can vary from platform to platform in this range would be difficult. Instead, I suggest to focus on the possibility of supporting a context of POST codes for them. Unfortunately, not all POST cards are capable of maintaining a history of previous POST codes. But at this moment, we already have a console to debug them.

Other known limitations:

- Some cheaper POST cards may not look at enough data lines and therefore cannot properly handle the situation where the same POST codes are sent one after another.
  For example, 0x40 followed by 0x40 will display 0x40, 0xFF (on 2-byte POST cards)

and 0xFF (on 1-byte POST cards).

- No need to use other than the debug postcodes in loops. Spewing out postcodes isn't helpful.

Implementation plan

To address the issues outlined in the problem description and the additional information provided, this project will be divided into the following tasks:

1. Create a clear and organized **postcode table** that includes separate blocks for each stage of the coreboot boot process (bootblock, verstage, romstage, postcar, ramstage, and smm) and different sections of the codebase (SoC/CPU/Chipset, Mainboard, and Drivers). Ensure that postcodes within each block are contiguous and ordered properly to allow for easy identification and debugging.

   Convenient place for all POST codes already exists and would be the same - `src/commonlib/include/commonlib/console/post_codes.h`.

   Template name for generic post codes: `POST_{STAGE}_SHORTDESCRIPTION`.
   For example, `POST_BOOTBLOCK_RESET_VECTOR` / `POST_RAMSTAGE_BS_DEV_INIT`.

2. Define a set of **driver-specific** postcodes that are unique to each driver. These postcodes should have identifiable identifiers to show what section of the codebase they're coming from.

   Template name for post code:
   `POST_{STAGE}_{SECTION}_SHORTDESCRIPTION`.
   For example, `POST_BOOTBLOCK_SOC_SET_DEF_MTRR_TYPE`.

3. Define a set of **debug-specific** postcodes that may be used in any area, but may not be enabled by default, but only by user request. These debug postcodes should have identifiable identifiers to show what section of the codebase they're coming from.

   Template name for post code: `POST_DEBUG_SHORTDESCRIPTION`.

4. Remap all unhardcoded values (those that use macros) to their value according to the new postcode table. Unused values will be integrated into the code or abandoned.

5. Review all instances of hardcoded values in the codebase and replace them with calls to the new postcode table.

   For task 4 and 5 would be a really useful script to summarize post code usages that it's already written by Nicholas Chin.

6. Document all POST codes thoroughly to ensure proper usage and understanding by developers.

    a. The file `Documents/POSTCODES` will become the key documentation. The current information will be replaced with the actual one. All postcode ranges with description will be there.
    b. Headers that declares POST code macros must be documented using Doxygen style comments.

    Question: Should the header file that declares generic POST codes contains the description for POST codes? The `Documents/POSTCODES` file seems to be more appropriate place. Duplication in two places is NOT a variant.

7. Test the functionality of the new postcode table and POST code functionality thoroughly. Make any necessary revisions and ensure that the final codebase is properly documented and maintainable.

As I mentioned before, I don't have any real hardware that supports the coreboot. All my patches will be tested on i440fx and Q35 QEMU machines. However, it is possible that I will ll have obtained hardware that supports coreboot by the summer.

Driver/SuperIO/Debug IDs

To help identify which device is producing postcodes, we need to create a way to obtain Driver/SuperIO/Debug IDs. For this purpose, a special postcode would be emitted by the driver, superIO, or debug section, preceded by the actual ID code.

The ID code would be a single byte, but a POST code for IDs that require multiple bytes would be reserved. A table of such IDs will be maintained in the documentation. The allocation of IDs for specific sections will be done manually by the developers. The driver IDs and debug IDs would not share the same table and would be stored separately.

For example, the driver emits a postcode with the byte sequence: `0xF1 0x01`. These postcodes are emitted immediately which means that if you have a POST card that shows only the current postcode, you will see `0x01`. This at least indicates that coreboot has stopped at some driver entry point. (that's the reason why we use "preceded" and not "followed" byte technique) To determine which one exactly, you need a POST card with a two-byte display, which will show you `0xF1`. And at this point, you should take a look at driver table and find the `0xF1` entry. Usually, after these initial two postcodes, postcodes from the standard range for driver are emitted (see table below) sequentially, not simultaneously. For example, `0xF1 0x01` `0xd0` `0xd1` `0xd3`. The meaning of `0xd0`, `0xd1`, `0xd3` depends on the specific driver.

This approach would allow us to uniquely identify the area with a tool, making it easier to locate and address issues.

Tools / Scripts

Decode tool

The decode tool is an essential component of the proposed project. It will solve the problem of understanding the POST code output generated during the boot process of a computer system. The tool will be able to parse the POST code output and translate it into human-readable descriptions, making it easier to debug any issues that may arise during boot.

The decode tool is useful because it will allow users to quickly and easily identify any issues that may be preventing the system from booting properly. It will save time and effort that would otherwise be spent manually analyzing the POST code output, which can be time-consuming and error-prone.

The main functionality of the decode tool will be to read the binary image or standard input from POST port and analyze it. To be able to analyze driver-specific IDs, coreboot config should be passed to the tool.

Potential features:

- Filtering options: filter out certain types of POST codes or focus only on a particular section of the boot process.

Initial version is already published, you can take a look at
https://review.coreboot.org/c/coreboot/+/73978

Linter

To avoid issues with unauthorized/unregulated POST codes, changes should be made to existing linter tests. The updated tests will implement the following improvements:

- The use of bare number as an argument to the `post_code()` function should be **prohibited**.
- Only macros that are defined in POST headers are **allowed**.

In all other cases, the linter will give an error. As a result, potential issues with POST codes will be caught by the linter.

Additional Improvements

- Postcodes should be able to be included in the timestamp table so we can grab timestamps for individual postcodes.
- Specific Postcodes should have associated text names, similar to the timestamps, and should have a decode tool.

Proposed postcode ranges / table

The beginning of the table is the place where items that will not be expanded should be located.

Code range	Description
0x00	N/A - used to clear I/O port
0x01	Driver/SuperIO ID, preceded by ID byte (serves as driver entry)
0x02	Debug ID, preceded by ID byte
0x03 - 0x0f	RESERVED; potential ID, follow by N-number ID byte
0x10 - 0x1f	Debug postcodes - never enabled by default.
0x21 - 0x27	general coreboot startup code & common (before console)
0x28 - 0x2b	general coreboot console
0x2c - 0x2f	RESERVED
0x30 - 0x3f	general coreboot bootblock
0x40 - 0x4f	general coreboot romstage
0x50 - 0x53	general coreboot postcar
0x54 - 0x5f	general coreboot ramstage
0x60 - 0x6f	coreboot ramstage hardwaremain boot stages
0x70 - 0x7e	Vendorcode calls: FSP, AGESA, etc. entry & exit.
0x7f	coreboot identifier - used when returning from FSP, AGESA, etc
0x80 - 0x87	general coreboot SMM
0x88 - 0x8f	end of coreboot code - resume and the like.
0x90 - 0x97	ACPI postcodes (NOTE: potential removal)
0x98 - 0x9f	Unregulated - may be used anywhere for any purpose. (NOTE: potential removal)
0xa0 - 0xbf	SOC specific - Each SoC/CPU/Chipset can use as it chooses. POST codes for CAR mode should be defined there
0xc0 - 0xcf	Mainboard specific - Each mainboard can use as it chooses

0xd0 - 0xdf	Driver/SuperIO specific - Can use as the device chooses
0xe0 - 0xe9	RESERVED for expansion
0xf0 - 0xfd	Errors
0xfe	coreboot finished - jump to payload
0xff	N/A - some POST cards use it to indicate that they are powered.

Please, do not consider this table as the final version. It is based on a number of assumptions that may not be entirely correct, and therefore requires further analysis and confirmation.